High Speed Switch Scheduling for Local Area Networks

Thomas E. Anderson

Computer Science Division
University of California, Berkeley

Susan S. Owicki, James B. Saxe,
and Charles P. Thacker

Systems Research Center
Digital Equipment Corporation

Abstract

Current technology trends make it possible to build communication networks that can support high-performance distributed computing. This paper describes issues in the design of a prototype switch for a point-to-point network with link speeds of up to one gigabit per second. The switch deals in fixed-length ATM-style cells, which it can process at a rate of 37 million cells per second. It provides high bandwidth and low latency for datagram traffic. In addition, it supports real-time traffic by providing bandwidth reservations with guaranteed latency bounds. The key to the switch's operation is a technique called *parallel iterative matching*, which can quickly identify a set of conflict-free cells for transmission in a time slot. Bandwidth reservations are accommodated in the switch by building a fixed schedule for transporting cells from reserved flows across the switch; parallel iterative matching can be used to fill unused slots with datagram traffic. Finally, we note that parallel iterative matching may not allocate bandwidth fairly among flows of datagram traffic. We describe a technique called *statistical matching*, which can be used to ensure fairness at the switch and also to support applications with rapidly changing needs for guaranteed bandwidth.

This paper has been abridged for publication in this conference; an unabridged version can be obtained from the Digital Equipment Corporation Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301.

1 Introduction

Over the past few years, several technology trends have converged to provide an opportunity for high performance distributed computing. Advances in laser and fiber optic technology have driven feasible link throughputs above a gigabit per second. Dynamic RAM chips have become cheap enough to be cost-effective at providing the large amounts of buffering needed at these very high link speeds. Moreover, quick routing and switching decisions are possible with current CMOS technology.

In combination, these trends make it possible to construct a practical local area network out of multiple crossbar switches interconnected by gigabit per second point-to-point fiber links. Point-to-point networks have several advantages over other alternatives [Schroeder et al. 91]. In contrast to networks like Ethernet [Metcalfe & Boggs 76] that use a broadcast physical medium, or networks like FDDI [Ame 87, Ame 88] based on a token ring, point-to-point networks offer (i) aggregate network bandwidth that can be much larger than the throughput of a single link, (ii) the ability to add throughput incrementally to match workload requirements, (iii) the potential for lower latency by shortening path lengths and eliminating the need to acquire control over the entire network in order to begin transmitting, and (iv) a more flexible approach to high availability by allowing multiple paths between hosts.

This paper studies the architectural issues in building switches for high performance point-to-point local area networks.

High performance point-to-point networks have the potential to change the nature of distributed computing. Low latency and high throughput communication allow a much closer coupling of distributed systems than has been feasible in the past: with previous generation networks, the high cost of sending messages led programmers to carefully minimize the amount of net-

work communication [Schroeder & Burrows 90]. Further, when combined with today's faster processors, faster networks can enable a new set of applications, such as desktop multimedia and using a network of workstations as a supercomputer.

A primary barrier to building high performance networks is the difficulty of high speed switching—of taking data arriving on an input link and quickly sending it out on the appropriate output link. The switching task is simplified if the data can be processed in fixed-length cells, as discussed in Section 2.3. Given fixed-length cells, switching involves at least two separate tasks:

- *scheduling* – choosing which cell to send during each time slot, when more than one cell is destined for the same output, and
- *data forwarding* – delivering the cell to the output once it has been scheduled.

Many high speed switch architectures use the same hardware for both scheduling and data forwarding; Starlite [Huang & Knauer 84], Knockout [Yeh et al. 87], and Sunshine [Giacopelli et al. 91] are just a few of the switches that take this approach. If the input and output links of a switch are connected internally by a multi-stage interconnection network, the internal network can detect and resolve conflicts between cells as they work their way through the switch.

We take a different approach. We argue that for high-speed switching, both now and in the future, switch scheduling can profitably be separated from data forwarding. By doing this, the hardware for each function can be specialized to the task. Because switch cost is dominated by the optical components needed to drive the fiber links, the added cost of separate hardware to do scheduling is justified, particularly if link utilization is improved as a result.

We observe that switch scheduling is simply an application of bipartite graph matching—each output must be paired with at most one input that has a cell destined for that output. A primary contribution of this paper is a randomized parallel algorithm for finding a maximal bipartite match at high speed. (In practice, we run the algorithm for a fixed short time; it almost always finds a maximal match.) The algorithm can be efficiently implemented in hardware for switches of moderate scale. Our work is motivated by the needs of a prototype point-to-point network we are currently building; we expect to begin using the network in early 1993. Using only off-the-shelf field programmable gate array technology [Xil 91], our switch will be able to schedule a standard 53 byte ATM cell out each link of a 16 by 16 crossbar switch in the time for one cell to arrive at a link speed of 1 gigabit per second. This requires scheduling over 37 million cells per second. Cell latency across the switch is about 2.2 microseconds in the absence of contention. The switch does not drop cells, and it preserves the order of cells sent between a pair of hosts. If implemented in custom CMOS, we expect our algorithm to scale to larger switches and faster links.

Supporting the demands of new distributed applications requires more from a network than simply high throughput or low latency. The ability to provide guaranteed throughput and bounded latency is crucial to multimedia applications [Ferrari & Verma 90]. Even for applications that do not need guarantees, predictable and fair performance is often important to higher layers of protocol software [Jain 90, Zhang 91].

Our basic scheduling algorithm does not by itself provide either fairness or guaranteed throughput. We present enhancements to our algorithm to provide these features. These enhancements pull from the bag of tricks of network and distributed system design—local decisions are more efficient if they can be made independently of global information, purely static scheduling can simplify performance analysis, and finally, randomness can de-synchronize decisions made by a large number of agents.

The remainder of the paper discusses these issues in more detail. Section 2 puts our work in context by describing related work. Section 3 presents the basic parallel scheduling algorithm. Section 4 explores static scheduling as a way of providing guaranteed bandwidth and latency. Section 5 describes a technique called *statistical switching*, that uses additional randomness in the switching algorithm to improve fairness and to support dynamic allocation of bandwidth. Section 6 provides a summary of our work.

2 Background and Related Work

Our goal is to build a local area network that supports high performance distributed computing; for this, a network must have high throughput, low latency, graceful degradation under heavy workloads, the ability to provide guaranteed performance to real-time applications, and performance that is both fair and predictable.

The network we envision consists of switches connected in an arbitrary topology. Hosts are connected to switch ports through a *controller*. Routing in the network is based on *flows*, where a flow is a stream of cells between a pair of hosts. (Our network also sup-

ports multicast flows, but we will not discuss that here.) There may be multiple flows between a given pair of hosts, for example, with different performance guarantees. Each cell is tagged with an identifier for its flow. Switches contain routing tables, built during a configuration phase, that determine how cells of each flow are routed at that switch. All cells from a flow take the same path through the network.

In this section, we place our work in context by discussing aspects of our switch's design in the context of related work. These include switch size, the switch's internal interconnection fabric, fixed-length cells vs. variable-length packets, and buffer organization.

2.1 Switch Scale

A key parameter in designing a point-to-point network is the scale of each switch. Part of the the host-to-host interconnect is provided by the fiber connections between switches and part by the silicon within each switch. In designing a network, we need to find an appropriate balance between using a large number of small switches or a small number of large switches.

At one extreme, very small switches are not cost-effective. The largest component in the cost of a local area fiber optic network comes from the optical devices in each switch that drive the fiber; these devices account for half the parts cost of our prototype switch. A smaller switch size requires the network to have a larger number of fiber connections and thus a larger number of optical devices.

On the other hand, very large switches are often inappropriate for local area networks. While it is feasible to build switches with thousands of ports, such a switch would be unduly costly for sites that have only dozens of workstations. Smaller switches allow capacity to be added incrementally at low cost; smaller switches can also improve availability by making it less expensive for the network to have redundant paths.

For these reasons, our algorithms are designed for switches of moderate scale, in the range of 16 by 16 to 64 by 64. In the near-term, we expect workstations to be unable to utilize a full gigabit per second link; in our prototype, multiple workstations can connect to the network through a single switch port.

2.2 Internal Interconnect

Once the switch size has been decided, there are several approaches to designing the internal data path needed to transport cells from the inputs to the outputs of the switch. Probably the simplest approach to transporting data across a switch is to use shared memory or a shared bus. We do not pursue these techniques here, because they do not seem feasible for even moderate-sized switches with gigabit per second links, much less for the faster link speeds of the future.

Another uncomplicated approach is to connect inputs to outputs via a crossbar, using some external logic to schedule the crossbar, i.e., to decide which cells are forwarded over the crossbar during each time slot and to set up the crossbar for those cells. In the absence of a fast algorithm, however, scheduling the crossbar quickly becomes a performance bottleneck for all but the smallest switches.

Many switch architectures call for the switch's internal interconnection to be *self-routing* [Ahmadi & Denzel 89]. The switch is organized internally as a multistage network of smaller switches arranged in a butterfly, or more generally, in a banyan [Patel 79]. Cells placed into a banyan network are automatically routed and delivered to the correct output based solely on the information in the cell header.

Unlike a crossbar, however, banyan networks suffer from *internal blocking*. A cell destined for one output can be delayed (or even dropped) because of contention at the internal switches with cells destined for other outputs. This makes it difficult to provide guaranteed performance.

Internal blocking can be avoided by observing that banyan networks are internally non-blocking if cells are sorted according to output destination before being placed into the network [Huang & Knauer 84]. Thus, a common switch design is to put a Batcher sorting network [Batcher 68] in front of a normal banyan network. As with a crossbar, a cell may be sent from any input to any output provided no two cells are destined for the same output.

Our scheduling algorithm assumes that data can be forwarded through the switch with no internal blocking; this can be implemented using either a crossbar or a batcher-banyan network. Our prototype uses a crossbar because it is simpler and has lower latency. Even though the hardware for a crossbar for an N by N switch grows as $O(N^2)$, for moderate scale switches the cost of a crossbar is small relative to the rest of cost of the switch. In our prototype, for example, the crossbar accounts for only 5% of the overall cost of the switch.

2.3 Fixed-Length Cells vs. Variable-Length Packets

Within our network, data is transmitted in fixed-length cells rather than variable-length packets. This choice has a number of advantages for switch design, despite the disadvantage that it requires the switch to make more frequent scheduling decisions. The chief gain is that performance guarantees are easier to provide when the entire crossbar is re-scheduled after every cell time slot. In addition, fixed-length cells simplify non-FIFO buffer management (discussed in the next sub-section). Packet latency is improved because small cells simulate the performance of cut-through [Kermani & Kleinrock 79] while permitting a simpler store-and-forward implementation.

Applications may still deal in variable-length packets. It is the responsibility of the sending host controller to divide packets into cells, each containing the flow id for routing; the receiving controller re-assembles the cells into packets for the host.

2.4 Buffer Organization

Even with an internally non-blocking switch, when several cells destined for the same output arrive in a time slot, at most one can actually leave the switch; the others must be buffered. There are many options for organizing the buffer pools. For example, buffers may be placed at the switch inputs or outputs; when placed at the inputs they may be strictly FIFO or allow more general access. There has been considerable research on the impact of these alternatives. In this section we review the work that is most relevant to our switch design.

The simplest approach is to maintain a FIFO queue of cells at each input. However, with FIFO queueing, when a cell at the head of an input queue is blocked, all cells behind it in the queue are prevented from being transmitted, even when the output link they need is idle. This is called *head-of-line (HOL) blocking.* Karol et al. [1987] have shown that HOL blocking limits its switch throughput to 58% of the link bandwidth, when the destinations of incoming cells are uniformly distributed among all outputs.

Unfortunately, FIFO queueing can have even worse performance under certain traffic patterns. For example, if several input ports each receive a burst of cells for the same output, cells that arrive later for other outputs will be delayed while the burst cells are forwarded sequentially through the bottleneck link. If incoming traffic is periodic, Li [1988] shows that the total switch throughput can be as small as the throughput of a single link, even for very large switches; this is called *stationary blocking.* Figure 1 illustrates this effect.[1] The small boxes represent cells arriving at each input; the number in each box corresponds to the output destination of that cell. The worst case occurs when scheduling priority rotates among inputs so that the first cell from each input is scheduled in turn. Note that if cells could be forwarded in non-FIFO order, all of the switch's links could be fully utilized in steady state.

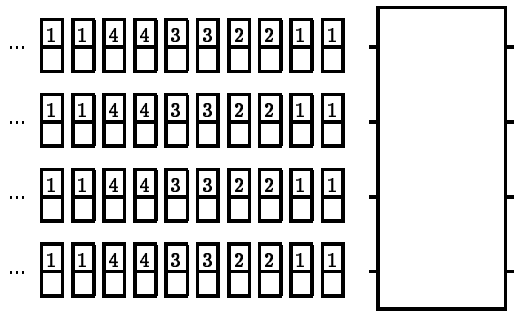

Figure 1: Performance Degradation Due to FIFO Queueing

Various approaches have been proposed for avoiding the performance problems of FIFO input buffers. One is to expand the internal switch interconnect so that it can route k cells to an output in a single time slot. This can be done in a batcher-banyan switch, for example, by replicating the banyan part of the switch k times [Huang & Knauer 84]. Since only one cell can depart from an output at each slot, buffers are required at the outputs with this technique. If more than k cells arrive during a slot for a given output, not all of them can be routed immediately. Typically, the excess cells are simply dropped. While few cells are dropped with a uniform workload [Giacopelli et al. 91], local area network traffic is rarely uniform. Instead, a common pattern is client-server communication, where a large fraction of incoming cells tend to be destined for the same output port [Owicki & Karlin 92]. Unlike previous generation networks, fiber links have very low error rates; the network we are building, for example, has a link bit error rate of 10^{-12}. Thus, loss induced by the switch architecture will be more noticeable.

[1] In this and other figures, input and output ports are shown as distinct entities. However, the ith input and the ith output are actually the two parts of a full-duplex port. The example assumes for simplicity that cells can be sent out the same port they came in on. If this is not the case, switch throughput can be as small as twice the link throughput.

Another technique, often combined with the previous one [Giacopelli et al. 91], is to shunt blocked cells into a *re-circulating queue* that feeds back into extra ports in the batcher network. The re-circulated cells are then sorted, along with incoming cells, during the next time slot. Once again, if there is too much contention for outputs, some cells will be dropped.

Our switch takes the alternative approach of using non-FIFO input buffers. Cells that cannot be routed in a slot are retained at the input, and the first cell of any flow can be selected for transmission across the switch. This avoids the cell loss problem in the schemes above, but requires a more sophisticated algorithm for scheduling the cells to be transmitted in a slot.

While there have been several proposals for switches that use non-FIFO input buffers [Karol et al. 87, Tamir & Frazier 88, Obara & Yasushi 89, Karol et al. 92], the difficulty is in devising an algorithm that is both fast enough to schedule cells at high link speeds and yet effective enough to deliver high link throughput. For example, Hui and Arthurs [1987] use the batcher network iteratively to schedule the batcher-banyan. At first, only the header for the first queued cell at each input port is sent through the batcher network; an acknowledgement is returned indicating whether the cell is blocked or can be forwarded during this time slot. An input that loses the first round of the competition sends the header for the second cell in its queue on the second round, and so on. After some number of iterations k, the winning cells, header plus data, are sent through the batcher-banyan to the outputs. Note that this reduces the impact of HOL blocking but does not eliminate it, since only the first k cells in each queue are eligible for transmission.

3 Parallel Iterative Matching

In this section, we describe our algorithm for scheduling the switch, first giving an overview and a possible hardware implementation, then discussing several issues encountered in its design.

3.1 Overview

The task of the scheduling algorithm is to determine which inputs will transmit to which outputs in a given time slot. With non-FIFO buffers, an input may transmit to any one of the outputs for which it has a cell. Thus the task comes down to matching inputs with outputs for which they have a cell, under the constraint that each input can be connected to at most one output and vice versa.

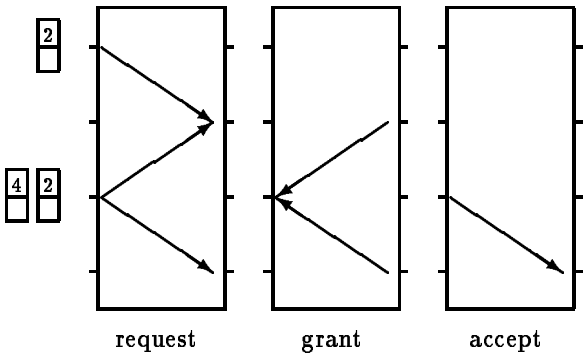

Figure 2: Parallel Iterative Matching: One Iteration

Our algorithm, *parallel iterative matching*, uses parallelism, randomness, and iteration to accomplish this task efficiently. We iterate the following three steps:

1. Each unmatched input sends a request to each output for which it has a buffered cell. This notifies an output of all its potential partners.
2. If an unmatched output receives any requests, it chooses one *randomly* to grant. The output notifies each input whether its request was granted.
3. If an input receives any grants, it chooses one to accept and notifies that output.

Figure 2 illustrates the request, grant, accept protocol. Two or more inputs can request the same output; the grant phase chooses among them. Two or more grants can be made to the same input; the accept phase chooses among them.

Note that this is a distributed algorithm—there is no centralized scheduler. Each of the steps can be performed in parallel at all of the links.

After a single iteration of steps 1–3, we have paired inputs and outputs, but inputs may remain which can be paired to other unpaired outputs. This is demonstrated in Figure 2. Two or more outputs can grant to the same input; an output whose grant is not accepted may be able to be paired with some other input.

To address this, we repeat the request, grant, accept protocol, retaining the matches made in previous iterations. As in Hui and Arthurs' algorithm, we iterate to "fill in the gaps" in the match left by previous iterations. However, there is no HOL blocking in our approach, since we can route any flow with queued cells.

Our algorithm may forward cells through the switch in an order different from the order in which they arrived. However, the switch maintains a FIFO queue

for each flow, so cells within a flow are not re-ordered. This use of FIFO buffers does not lead to HOL blocking: since all cells from a flow must be forwarded to the same output, either none of the cells of a flow are blocked or all are.

We implement this algorithm in a straightforward way, by running a wire between every input and output. Even though this implementation requires hardware that grows as $O(N^2)$ for an N by N switch, the cost of our scheduling hardware is small relative to the cost of the fiber optic switch components. The request and grant signals can be encoded by a single bit on the appropriate wire. No separate communication step is required to indicate which grants are accepted. When an input chooses to accept an output's grant, it simply continues to request that output on subsequent iterations, but drops all other requests. Once an output grants to an input, it continues to grant to the same input on subsequent iterations unless the input drops its request.

Our algorithm can be generalized to handle switches with replicated switching fabrics. For instance, consider a batcher-banyan switch with k copies of the banyan network. With such a switch, up to k cells can be delivered to a single output during one time slot. (Note that this requires buffers at the outputs, since only one cell per slot can leave the output.) In this case, we can modify parallel iterative matching to allow each output to make up to k grants in step 2. In all other ways, the algorithm remains the same. An analogous change can be made for switch fabrics that allow inputs to forward more than one cell during any time slot. For the remainder of the paper, however, we assume that each input must be paired with at most one output and vice versa.

3.2 Maximal vs. Maximum Matching

The problem of scheduling a switch, that is, determining which input and output ports should be connected during each slot, is an application of bipartite graph matching [Tarjan 83]. Switch inputs and outputs form the nodes of a bipartite graph; the edges are the connections needed by queued cells.

Bipartite graph matching has been studied extensively. There are two interesting kinds of bipartite matches. A *maximum* match is one that pairs the maximum number of inputs and outputs together; there is no other pairing that matches more inputs and outputs. A *maximal* match is one for which pairings cannot be trivially added; each node is either matched or has no edge to an unmatched node. A maximum match must of course be maximal, but the reverse is not true; there may be a way of pairing more inputs and outputs than a maximal match by completely re-shuffling which inputs and outputs are paired with one another.

We designed our switch scheduler to find a maximal match, even though link utilization would be better with a maximum match. One reason is performance—finding a maximum match for an N by N graph with M edges can take $O(N \times (N + M))$ time. Karp et al. [1990] give a randomized algorithm that comes close on average to finding a maximum match, but even that algorithm can take $O(N + M)$ time. As discussed below, our parallel algorithm finds a maximal match in logarithmic time, on average.

Another reason for using a maximal match is that maximum matching can lead to starvation. Figure 2 illustrates this possibility. Assuming a sufficient supply of incoming cells, maximum matching would always connect input 1 with output 2 and input 3 with output 4. Cells queued at input 3 destined for output 2 would never be transmitted.

In contrast, our algorithm does not incur starvation. Since every output grants randomly among requests, an input will eventually, with very high probability, receive a grant from every output it requests. Provided inputs choose among grants in a round-robin or other fair fashion, every queued cell will eventually be transmitted.

In the worst case, the number of pairings in a maximal match can be as small as 50% of the number of pairings in a maximum match. However, simulations show that if all input-output pairs are requested with equal probability p, then the expected size of the matches found by iterative matching ranges between 85% and 100% of maximum, depending on p.

3.3 Convergence

Parallel iterative matching takes a variable number of iterations to find a maximal match. It is important for performance to understand how long the algorithm will take to converge. In the worst case, it is no faster than the sequential approach. If all outputs grant to the same input on the first round, only one match will be made. If this pattern is repeated, it will take N rounds to reach a maximal match. On the other hand, in the best case, every output grants to a unique input, in which case the algorithm takes only one iteration to converge.

To achieve fast convergence, we make it unlikely that most outputs grant to the same input. This is done by granting randomly among requests in step 2. It can

be shown that the algorithm finds a maximal match in $O(\log N)$ iterations, on average, independent of the pattern of requests. The proof is given in Appendix A. The key is that each iteration resolves, on average, at least 3/4 of the remaining unresolved requests.

Our implementation uses a small fixed number of iterations rather than iterating until a maximal match is reached. This is because we have a fixed amount of time to schedule the switch—the time to receive one cell at link speed. In our current implementation using 53 byte ATM cells, off-the-shelf field programmable gate array technology, and 1.0 gigabit per second links, there is time for four iterations. Simulations of a 16 by 16 switch show that if all input-output pairs are requested with equal probability p, then four iterations achieve a maximal matching over 98% of the time, regardless of p. In other words, the constant factor in the $O(\log N)$ iterations needed to reach a maximal match is small.

A remaining implementation problem is selecting randomly among k requesting inputs. The most straightforward way to do this is to generate a random number between 1 and k. Depending on the size of the switch, more efficient implementations may be possible. For instance, in our prototype, it is possible to select more efficiently using a scheme based on tables of precomputed values.

To summarize, parallel iterative matching makes it possible for the switch to achieve a nearly maximal match in a short time, regardless of the pattern of output requests. Moreover, the hardware requirements are modest enough to make parallel iterative matching practical for high speed switching.

4 Static Scheduling

As network and processor speeds increase, new types of high-performance distributed applications become feasible. Supporting the demands of these applications requires more from a network than just high throughput or low latency. Parallel iterative matching, while fast and effective at keeping links utilized, cannot by itself provide the needed services. The remainder of this paper discusses these issues and suggests ways of augmenting our basic algorithm to address these concerns.

One important class of applications depend on real-time performance guarantees. For example, multimedia applications must display video frames at set intervals. They require that the network provide a certain minimum bandwidth and a bounded latency for cell delivery. Following the conventions of the ATM community, we will call traffic with reserved bandwidth *constant bit rate (CBR)* traffic, and refer to other traffic as *variable bit rate (VBR)* traffic. (VBR traffic is often called *datagram* traffic). Switches distinguish VBR and CBR cells based on the flow identifier in the cell header.

To ensure guaranteed performance, an application issues a request to the network to reserve a certain bandwidth and latency bound for a flow of CBR traffic [Ferrari & Verma 90]. If the request can be met without violating any existing guarantees, the network grants it and reserves the required resources on a fixed path between source and destination. The application can then transmit cells at a rate up to its requested bandwidth, and the network ensures that they are delivered on time. Applications that do not require guarantees can transmit cells with no prior arrangement in VBR flows. If the network becomes heavily loaded, VBR cells may suffer arbitrary delays. But CBR performance guarantees are met no matter how high the load of VBR traffic.

Our contribution in this area is in showing how to implement performance guarantees in a network of input-buffered switches with unsynchronized clocks. The rest of this section describes our approach to CBR traffic. We first describe the form of a bandwidth request and the criterion used to determine whether it can be accepted. We next show how a switch can be scheduled to meet bandwidth guarantees. Finally, we show that buffers for CBR traffic can be statically allocated and the latency of CBR cells can be bounded, even when network switch clock rates are unsynchronized. Our approach smoothly integrates both CBR and VBR traffic; VBR cells can consume all of the network bandwidth unused by CBR cells.

Bandwidth allocations are made on the basis of *frames* which consist of a fixed number of *slots*, where a slot is the time required to transmit one cell [Golestani 90]. An application's bandwidth request is expressed as a certain number of cells per frame. Frame boundaries are internal to the switch; they are not encoded on the link.

Frame size is a parameter of the network. A larger frame size allows for finer granularity in bandwidth allocation; we will see later that smaller frames yield lower latency. The frame size in our prototype switch is 1000 slots; a frame takes less than half a millisecond to transmit. This leads to latency bounds that seem acceptable for multi-media applications, the most likely use for CBR guarantees.

When a request is issued, network management software must determine whether it can be granted. In our approach, this is possible if there is a path from source to destination on which each link's uncommitted capacity is greater than the requested bandwidth. If network

Reservations (cells per frame)

Input	Output 1	2	3	4
1		1	1	1
2	2			
3		2		1
4	1		1	

Schedule

Slot 1	$1 \rightarrow 3$	$2 \rightarrow 1$	$3 \rightarrow 2$	
Slot 2	$1 \rightarrow 4$	$2 \rightarrow 1$	$3 \rightarrow 2$	$4 \rightarrow 3$
Slot 3	$1 \rightarrow 2$		$3 \rightarrow 4$	$4 \rightarrow 1$

Figure 3: CBR Traffic: Reservations and Schedule

Reservations (cells per frame)

Input	Output 1	2	3	4
1		1	1	1
2	2			1
3		2		1
4	1		1	

Schedule

Slot 1	$1 \rightarrow 2$	$2 \rightarrow 1$	$3 \rightarrow 4$	
Slot 2	$1 \rightarrow 4$	$2 \rightarrow 1$	$3 \rightarrow 2$	$4 \rightarrow 3$
Slot 3	$1 \rightarrow 3$	$2 \rightarrow 4$	$3 \rightarrow 2$	$4 \rightarrow 1$

Figure 4: CBR Traffic with Added Reservation

software finds such a path, it grants the request, and notifies the involved switches of the additional reservation. The application can then send up to the reserved number of cells each frame. The host controller or the first switch on the flow's path can meter the rate at which cells enter the network; if the application exceeds its reservation, the excess cells may be dropped. Alternatively, excess cells may be allowed into the network, and any switch may drop cells for a flow that exceeds its allocation of buffers.

Note that this allocation criterion allows reservation of 100% of the link bandwidth. Meeting this throughput level is straightforward with an output-buffered switch [Golestani 90, Kalmanek et al. 90], but this assumes the switch has enough internal bandwidth that it never needs to drop cells under any pattern of arriving CBR cells. With input buffering, parallel iterative matching is not capable of guaranteeing this throughput level. Instead, we exploit the Slepian-Duguid theorem [Hui 90] to build an explicit schedule of switch connections for each slot in a frame. The theorem implies that such a schedule can be found for any traffic pattern, so long as the number of cells from any input or to any output is no more than the frame size. Figure 3 provides an example of reservations and a schedule for a frame size of 3 slots.

When a new reservation is made, it may be necessary to rearrange the connections in the schedule. For instance, consider an additional reservation of one cell per frame from input 2 to output 4. Because there is no slot in which both input 2 and output 4 are free, the existing schedule must be shuffled in order to accommodate the new flow. The result is illustrated in Figure 4. Computing a new schedule may require a number of steps proportional to the size of the reservation (in cells/frame) $\times N$, for an N by N switch. However, the test for whether a switch can accommodate a new flow is much simpler; it is possible so long as the input and output link each have adequate unreserved capacity. Once a feasible path is found, the selected switches can compute their new schedules in parallel.

CBR cells are routed across the switch during scheduled slots. VBR cells are transmitted during slots not used by CBR cells. For example, in Figure 3, a VBR cell can be routed from input 2 to output 3 during the third slot. In addition, VBR cells can use an allocated slot if no cell from the scheduled flow is present at the switch.

Pre-scheduling the switch ensures that there is adequate bandwidth at each switch and link for CBR traffic. It is also necessary to have enough buffer space at each switch to hold cells until they can be transmitted; otherwise, some cells would be lost. Our switch statically allocates enough buffer space for CBR traffic. VBR cells use a different set of buffers, which are subject to flow control.

In a network where switch clock rates are synchronized, as in the telephone network, a switch needs enough buffer space at each input link for two frames worth of cells [Golestani 90, Zhang & Keshav 91]. Note that one frame of buffering is not enough, because the frame boundaries may not be the same at both switches, and because the switches can rearrange their schedules from one frame to the next.

When the switches are not synchronized, the situation becomes more complicated. We do not present the proof here, but the number of buffers required can still be bounded if the clocks on all switches and controllers tick at rates that are within a specified accuracy of a specified nominal rate, and if the host controller or the

first switch on the flow's path is constrained to admit cells into the network slightly slower than the slowest switch can process them. The exact bound is a function of network parameters: the switch and controller frame sizes, the network diameter, and the clock error limits. Four or five frames of buffers are sufficient for values of these parameters that are reasonable for local area networks.

Now let us consider latency guarantees. If switch clocks are synchronized, a cell can be delayed at most two frame times at each switch on its path [Golestani 90, Zhang & Keshav 91]. Let p be the number of hops in the cell's path, f the time to transmit a frame, and l an upper bound on link latency plus switch overhead for processing a cell. Then the total latency for a cell is less than $p(2f + l)$. When switches are not synchronized, the delay experienced at a switch may be larger than $(2f + l)$ but end-to-end delay is still bounded by $p(2f + l)$. Again, the proof is not presented here. This yields latency bounds in our prototype that are adequate for most multi-media applications. A smaller frame size would provide lower CBR latency, but as mentioned before it would entail a larger granularity in bandwidth reservations. We are considering schemes in which a large frame is subdivided into smaller frames. This would allow each application to trade off low latency guarantee vs. small granularity of allocation.

To summarize, bandwidth and latency guarantees are provided through the following mechanisms:

- Applications request bandwidth reservations in terms of slots/frame.
- The network grants a request if it can find a path on which each link has the required capacity.
- Each switch, when notified of a new reservation, builds a schedule for transmitting cells across the switch.
- Enough buffers are permanently reserved for CBR traffic to ensure that arriving cells will always find an empty buffer.
- Latency is bounded by a simple function of link latency, path length, and frame size.

5 Statistical Matching

The prototype switch we are building combines the methods described in the previous two sections to provide low latency and high throughput for VBR traffic and guaranteed performance for CBR traffic. Two issues remain: (i) parallel iterative matching does not always provide fair and predictable network performance and (ii) some applications need performance guarantees yet generate packets at a variable rate. In this section, we present a generalization of parallel iterative matching, called *statistical matching*, that addresses both these concerns; due to space constraints, we only discuss the issue of fairness here. Statistical matching works by systematically using randomness in choosing which request to grant and which grant to accept. The first version of our prototype switch does not implement statistical matching.

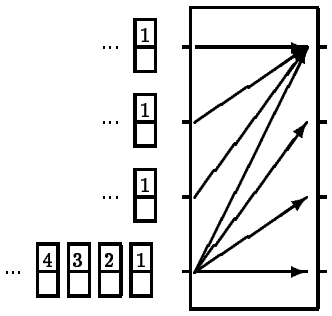

Figure 5: Unfairness with Parallel Iterative Matching

5.1 Motivation

Ramakrishnan et al. [1990] provide a formal definition of fairness in the allocation of network resources. To be fair, every user should receive an equal share of every network resource that does not have enough capacity to satisfy all user requests. If a user needs less than its equal share, the remainder should be split among the other users. One result of a fair network, then, is that users typically see graceful degradation in performance under increased load. Adding an additional user to an already crowded system will result in a relatively small decrease in everyone else's resource allocation.

Unfortunately, a network built out of switches using parallel iterative matching is not fair, for two reasons. First, to be scheduled, a queued cell needs to receive a grant from its output and to have its input accept the grant. Both the input and output ports are sources of contention; parallel iterative matching will tend to give higher throughput to input-output connections that have fewer contending connections. In Figure 5, for instance, if input 4 chooses randomly when it receives more than one grant, the connection between input 4 and output 1 will receive only one fifth the throughput of the other connections.

Second, even if a switch allocates output bandwidth

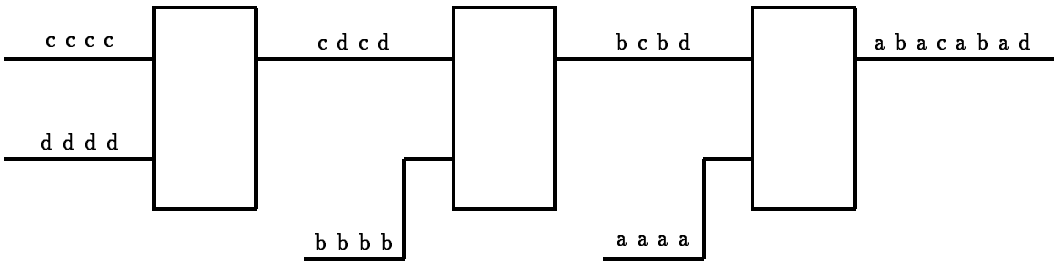

Figure 6: Unfairness with Point-to-Point Networks

equally among all requesting inputs, the switch, when embedded in a point-to-point network, will not be fair among users or flows [Demers et al. 89].[2] Depending on the topology of the network, each switch input may have a different numbers of flows. A flow reaching a bottleneck link at the end of a long chain of switches may receive an arbitrarily small portion of the link throughput, while another flow receives a much larger portion. Unfortunately, this is exactly the pattern we expect if higher level software is using a client-server model [Owicki & Karlin 92]. Figure 6 illustrates what happens when four flows share a bottleneck link. Each letter represents a cell; switches forward cells by a round-robin among input ports. To be perfectly fair, each flow should receive the same throughput on the rightmost link.

A number of approaches to fairness in point-to-point networks have been proposed, but they have been difficult to implement in high performance networks. For example, Zhang [1991] suggests a *virtual clock* algorithm. Host network software assigns each flow its fair share of the network bandwidth and notifies each switch along the path of the rate to be delivered to the flow. When a cell arrives at a switch, it is assigned a timestamp based on when it would be scheduled if the network were operating fairly; the switch gives priority to cells with earlier timestamps.

Demers et al. [89] propose a *fair queueing* scheme that does not require rates to be set by host software. Instead, switches round-robin among flows in choosing which cell to forward onto an output link. No external rate control is needed for fairness, because each flow has its own dedicated buffer space at each switch, and flow control between switches prevents buffer overflows.

Both virtual clock and fair queueing assume that it is possible for each output link to select arbitrarily among any of the cells queued for it. This is the case in an output-buffered switch. In our input-buffered switch, however, only one cell from each input can be forwarded at a time. Without centralized control of scheduling, which is incompatible with a high performance switch, it seems difficult to implement either scheme in an input-buffered switch.

[2] A "network user" may, of course, be sending more than one flow of cells through a switch, for example, to different hosts. For simplicity, though, the remainder of our discussion will assume that our target is fairness among flows as an approximation to fairness among users.

5.2 Algorithm

Statistical matching, like Slepian-Duguid, is an algorithm for delivering to each flow a pre-set portion of the link throughput. With statistical matching, up to $(1-1/e)\times(1+1/e^2)$, or 72%, of link throughput can be allocated to specific flows; any unused slot can be filled in with other traffic by parallel iterative matching. The reserved throughputs can be in any pattern, provided the sum of the throughputs at any input or output is less than 72%. Unlike Slepian-Duguid, however, adjusting throughput rates is efficient, in part because only the input and output ports used by a flow need be informed of a change in its rate.

Statistical matching is based on parallel iterative matching, but it makes more systematic use of randomness in making and accepting grants. The matching for each time slot is chosen independently, but on average, each flow is scheduled according to its pre-set throughput rate. The algorithm mirrors parallel iterative matching except that there is no request phase.

In the following algorithm, X represents the number of discrete units of allocatable bandwidth per link; $X_{i,j}$ denotes the number of units of bandwidth allocated to traffic from input i to output j. Our description of the algorithm assumes for simplicity that the bandwidth is fully allocated; a straightforward adjustment accounts for any unallocated bandwidth.

1. Each output j randomly chooses an input i to grant to, with probability proportional to its reser-

vation:

$$\Pr\{j \text{ grants to } i\} = X_{i,j}/X$$

2. Each input chooses at most one grant to accept (it may accept none) as a two-step process:

   (a) Each input i randomly chooses, for each output j that makes a grant to it, a weight $m_{i,j}$, between 0 and $X_{i,j}$, representing the number of *virtual grants* received from j. This is done in such a way as to simulate the effect of $X_{i,j}$ possible grants being generated independently, each with probability $1/X$. Specifically, the probability that $m_{i,j} = m$ for positive m is

   $$\binom{X_{i,j}}{m} \times \left(\frac{1}{X}\right)^m \times \left(\frac{X-1}{X}\right)^{X_{i,j}-m} \times \frac{X}{X_{i,j}}$$

   The remaining probability mass is assigned to $m_{i,j} = 0$. When j does not grant to i, $m_{i,j}$ is set to zero.

   (b) If any grant has positive weight, the input then randomly accepts a grant with probability proportional to its weight:

   $$\Pr\{i \text{ accepts } j\} = \frac{m_{i,j}}{\sum_k m_{i,k}}$$

   If a grant is accepted, the input randomly chooses among the flows for the connection according to their bandwidth reservations.

The effect of these steps is that each input behaves as if it were choosing uniformly among X potential grants ($X_{i,j}$ of them from output j), each made with probability $1/X$. Details of the proof are not given. As X becomes large, $(1-(1-1/X)^X)$ approaches $1-1/e$. Thus a single iteration matches input i to output j with probability $\frac{X_{i,j}}{X}(1-\frac{1}{e})$, or about 63% of $\frac{X_{i,j}}{X}$.

Better throughput can be achieved by iterating these steps twice; additional iterations beyond two yield insignificant throughput improvements. On the second iteration, outputs grant and inputs accept independently of the results of the first iteration. Connections made on the second iteration that conflict with connections made on the first iteration are discarded. Because the events "i unmatched on the first round" and "j unmatched on the first round" are either independent or positively correlated (again, the proof is omitted), the probability of both of these events occurring is at least $1/e^2$, and the overall probability of a connection between i and j after two rounds is at least $\frac{X_{i,j}}{X}(1-\frac{1}{e})(1+\frac{1}{e^2})$, or about 72% of $\frac{X_{i,j}}{X}$.

Additional iterations of parallel iterative matching can then be performed to fill in slots unused by statistical matching.

While statistical matching requires more hardware to implement than does parallel iterative matching, the hardware complexity is not as daunting as it might first appear. Steps 1 and 2a can both be implemented with simple table lookups. The table is initialized with the number of entries for each outcome proportional to its probability; a random index into the table selects the outcome. Step 2b is simply a generalization of the random choice among requests needed by parallel iterative matching; the same implementation techniques apply.

5.3 Discussion

We motivated statistical matching by suggesting that it could be used to schedule the switch fairly among competing flows. For instance, statistical matching meets many of the goals that motivated Zhang's virtual clock approach. With either approach, the switch can be set to assign equal throughput to every competing flow through a bottleneck link. Statistical matching can provide roughly equal throughput without the need for tagging individual cells with timestamps and prioritizing flows based on those timestamps, although some unfairness may be added when parallel iterative matching fills in gaps left by statistical matching. With statistical matching, as with virtual clock, a flow can temporarily send cells faster or slower than its promised rate, provided the throughput is not exceeded over the long term. Queues in the network increase if the flow sends at a faster rate; queues empty as the flow sends at a slower rate. The virtual clock approach also provides a way of monitoring whether a flow is exceeding its promised rate over the long term; there is no analogue with statistical matching.

Statistical matching can also approximate Demers et al.'s fair queueing. The switch could periodically set its rates to reflect the number of flows queued for each input/output connection; this would give equal service on average to each flow. It is a question for future research to determine how well this works in practice.

6 Summary

We have described the design of a prototype switch that can support high-performance distributed computing. Key to the switch's operation is a technique called *parallel iterative matching*, a fast algorithm for choosing a conflict-free set of cells to forward across the switch during each time slot. Our prototype switch combines

this with a mechanism to support real-time traffic even in the presence of clock skew between switches. The switch will be used as the basic component of a point-to-point local area network, providing

1. high-bandwidth,
2. low-latency for datagram traffic, so long as the network is not overloaded, and
3. bandwidth and latency guarantees for real-time traffic.

In addition, the switch's scheduling algorithm can be extended to allocate resources fairly when some part of the network is overloaded.

We believe that the availability of high performance networks with these characteristics will enable a new class of distributed applications. Networks are no longer slow, serial, highly error-prone bottlenecks where message traffic must be carefully minimized in order to get good performance. This allows distributed systems to be more closely coupled than has been possible in the past.

7 Acknowledgements

We would like to thank Mike Burrows, Hans Eberle, Mike Goguen, Domenico Ferrari, Butler Lampson, Tony Lauck, Hal Murray, Roger Needham, John Ousterhout, Tom Rodeheffer, Ed Satterthwaite, and Mike Schroeder for their helpful comments.

References

[Ahmadi & Denzel 89] Ahmadi, H. and Denzel, W. A Survey of Modern High-Performance Switching Techniques. *IEEE Journal on Selected Areas in Communications*, 7(7):1091–1103, September 1989.

[Ame 87] American National Standards Institute, Inc. *Fiber distributed data interface (FDDI). Token ring media access control (MAC). ANSI Standard X3.139*, 1987.

[Ame 88] American National Standards Institute, Inc. *Fiber distributed data interface (FDDI). Token ring physical layer protocol (PHY). ANSI Standard X3.148*, 1988.

[Batcher 68] Batcher, K. Sorting Networks and their Applications. In *AFIPS Conference Proc.*, pages 307–314, 1968.

[Demers et al. 89] Demers, A., Keshav, S., and Shenker, S. Analysis and Simulation of a Fair Queueing Algorithm. In *Proc. ACM SIGCOMM '89 Conference on Communications Architectures and Protocols*, pages 1–12, September 1989.

[Ferrari & Verma 90] Ferrari, D. and Verma, D. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE Journal on Selected Areas in Communications*, 8(3):361–379, April 1990.

[Giacopelli et al. 91] Giacopelli, J., Hickey, J., Marcus, W., Sincoskie, W., and Littlewood, M. Sunshine: A High-Performance Self-Routing Broadband Packet Switch Architecture. *IEEE Journal on Selected Areas in Communications*, 9(8):1289–1298, October 1991.

[Golestani 90] Golestani, S. Congestion-Free Transmission of Real-Time Traffic in Packet Networks. In *Proc. INFOCOM '90*, pages 527–542, June 1990.

[Huang & Knauer 84] Huang, A. and Knauer, S. Starlite: A Wideband Digital Switch. In *Proc. GLOBECOM '84*, pages 121–125, December 1984.

[Hui & Arthurs 87] Hui, J. and Arthurs, E. A Broadband Packet Switch for Integrated Transport. *IEEE Journal on Selected Areas in Communications*, 5(8):1264–1273, October 1987.

[Hui 90] Hui, J. *Switching and Traffic Theory for Integrated Broadband Networks*. Kluwer Academic Press, 1990.

[Jain 90] Jain, R. Congestion Control in Computer Networks: Issues and Trends. *IEEE Network Magazine*, pages 24–30, May 1990.

[Kalmanek et al. 90] Kalmanek, C., Kanakia, H., and Keshav, S. Rate Controlled Servers for Very High-Speed Networks. In *Proc. IEEE Global Telecommunications Conference*, pages 300.3.1–300.3.9, December 1990.

[Karol et al. 87] Karol, M., Hluchyj, M., and Morgan, S. Input Versus Output Queueing on a Space-Division Packet Switch. *IEEE Transactions on Communications*, 35(12):1347–1356, December 1987.

[Karol et al. 92] Karol, M., Eng, K., and Obara, H. Improving the Performance of Input-Queued ATM Packet Switches. In *Proc. INFOCOM '92*, pages 110–115, May 1992.

[Karp et al. 90] Karp, R., Vazirani, U., and Vazirani, V. An Optimal Algorithm for On-line Bipartite Matching. In *Proc. 22nd Annual ACM Symposium on Theory of Computing*, pages 352–358, May 1990.

[Kermani & Kleinrock 79] Kermani, P. and Kleinrock, L. Virtual Cut-through: A New Computer Communication Switching Technique . *Computer Networks*, 3:267–286, September 1979.

[Li 88] Li, S.-Y. Theory of Periodic Contention and Its Application to Packet Switching. In *Proc. INFOCOM '88*, pages 320–325, March 1988.

[Metcalfe & Boggs 76] Metcalfe, R. and Boggs, D. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[Obara & Yasushi 89] Obara, H. and Yasushi, T. An Efficient Contention Resolution Algorithm for Input Queueing ATM Cross-Connect Switches. *International Journal of Digital and Analog Cabled Systems*, 2(4):261–267, October 1989.

[Owicki & Karlin 92] Owicki, S. and Karlin, A. Factors in the Performance of the AN1 Computer Network. In *Proc. 1992 ACM SIGMETRICS and PERFORMANCE '92 Conference on Measurement and Modeling of Computer Systems*, pages 167–180, June 1992.

[Patel 79] Patel, J. Processor-Memory Interconnections for Multiprocessors. In *Proc. 6th Annual Symposium on Computer Architecture*, pages 168–177, April 1979.

[Ramakrishnan & Jain 90] Ramakrishnan, K. and Jain, R. A Binary Feedback Scheme for Congestion Avoidance in Computer Networks. *ACM Transactions on Computer Systems*, 8(2):158–181, May 1990.

[Schroeder & Burrows 90] Schroeder, M. and Burrows, M. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[Schroeder et al. 91] Schroeder, M., Birrell, A., Burrows, M., Murray, H., Needham, R., Rodeheffer, T., Satterthwaite, E., and Thacker, C. Autonet: A High-Speed Self-Configuring Local Area Network Using Point-to-Point Links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318–1335, October 1991.

[Tamir & Frazier 88] Tamir, Y. and Frazier, G. High-Performance Multi-Queue Buffers for VLSI Communication Switches. In *Proc. 15th Annual Symposium on Computer Architecture*, pages 343–354, June 1988.

[Tarjan 83] Tarjan, R. *Data Structures and Network Algorithms*. SIAM, 1983.

[Xil 91] Xilinx, Inc. *Xilinx: The Programmable Gate Array Data Book*, 1991.

[Yeh et al. 87] Yeh, Y., Hluchyj, M., and Acampora, A. The Knockout Switch: A Simple Modular Architecture for High-Performance Switching. *IEEE Journal on Selected Areas in Communications*, 5(8):1274–1283, October 1987.

[Zhang & Keshav 91] Zhang, H. and Keshav, S. Comparison of Rate-Based Service Disciplines. In *Proc. ACM SIGCOMM '91 Conference on Communications Architectures and Protocols*, pages 113–122, September 1991.

[Zhang 91] Zhang, L. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.

A Convergence of Parallel Iterative Matching

In this appendix, we consider the average convergence time of the parallel iterative matching algorithm described in Section 3, showing that it is $O(\log N)$ iterations, for an N by N switch. This bound is independent of the pattern of requests. The proof is based on the observation that each iteration reduces the number of unresolved requests by an average of at least 3/4. (A request is unresolved if neither its input nor its output port has been matched on a previous iteration.)

Consider an output Q that receives requests from n inputs. Of these n inputs, some will request and receive a grant from some output besides Q, and the rest will receive no grants from other outputs. Let k be the number that receive no other grants.

Q chooses randomly to grant to one of the n inputs. Since Q's choice is independent of the choices of the other outputs, Q will grant to one of the k inputs that receive no other grants with probability k/n. In this case, Q's grant will be accepted, and all of Q's n requests will be resolved. On the other hand, with probability $1-(k/n)$, Q will grant to an input that also receives a grant from some other output. Even if Q's grant is not accepted, all of the $n-k$ inputs that received a grant will accept a grant; thus their $n-k$ requests to Q will be resolved.

Thus with probability k/n all requests to Q are resolved, and with probability $1-(k/n)$ at most $n-k$ remain unresolved. As a result, the average number of unresolved requests to Q is at most $(1-(k/n))\times(n-k)$, which is no greater than $n/4$ for all k. Note that this implies that the expected number of unresolved requests after j iterations is at most $N^2/4^j$, because there can be at most N^2 requests.

It remains to be shown that the algorithm converges in $O(\log N)$ steps on average. Let C be the random variable whose value is the step on which the last request is resolved. Then

$$\begin{aligned} E[C] &= \sum_{i=1}^{\infty} i \ \Pr\{C=i\} \\ &= \sum_{j=1}^{\infty}\sum_{i=j}^{\infty} \Pr\{C=i\} \\ &= \sum_{j=1}^{\infty} \Pr\{C>j-1\} \\ &= \sum_{j=0}^{\infty} \Pr\{C>j\} \end{aligned} \tag{1}$$

Now let U_j be the random variable whose value is the number of unresolved requests at the end of j steps. Then

$$\begin{aligned} \Pr\{C>j\} &= \sum_{k=1}^{\infty} \Pr\{U_j=k\} \\ &\le \sum_{k=1}^{\infty} k \ \Pr\{U_j=k\} \\ &= E[U_j] \ \le \ \frac{N^2}{4^j} \end{aligned}$$

Substituting $\Pr\{C>j\} \le \min(1, N^2/4^j)$ in (1) gives

$$E(C) \ \le \ \sum_{j=0}^{\infty} \min(1, \frac{N^2}{4^j}) \ \le \ \log_2 N + \frac{4}{3}.$$