Lecture 12: Input and output queueing

Anirudh Sivaraman

2018/10/29

We'll continue our discussion of the data plane this lecture, but focus on the queueing subsystem within a router's data plane. But first, what is the source of queues within a router's data plane? Queues arise because multiple input ports on a router might want to send packets to the same output port. A number of scenarios can result in this type of multiple-input-single-output (also known as fanin/incast) behavior. One example is that of several client machines talking to a single server machine as in the case of a flash crowd to a very popular server. Another example scenario is what happens internally at a search provider like Google when a user issues a search query: the search query is received by a frontend Google server, which then delegates the query to several worker machines, where each worker machine runs the user's search query on a subset of the Internet's web pages that Google has previously crawled. These worker machines all return their query results to the frontend server at roughly the same time (because they are all given the same time budget to work on their queries). This causes a fanin or incast at the frontend server.

When multiple inputs send packets to the same output at roughly the same time, we need to deal with the fact that the input arrival rate is higher than the output departure rate—at least temporarily. The standard solution is to transmit as many packets as the output port can transmit and queue up the rest somewhere for later transmission. If the situation persists, the queue might overflow and drop packets. This is what causes TCP's AIMD algorithm to cut its window in half. In this lecture, we'll look at a few architectures for organizing these queues within a router and their associated tradeoffs. We'll finally discuss how technology trends determine the right choice of the architecture for the queueing subsystem.

1 Shared memory

The simplest architecture for a router with N input ports and N output ports is a *shared memory* architecture.[1] In a shared memory architecture, there is a common pool of memory that is shared by all input and output ports on the router.

Let's assume the smallest unit of time in the router (i.e., the time that it takes to receive the smallest size packet on an input port or send the smallest size packet on an output port) is a single tick. The time that it takes to receive the smallest size packet on an input port is typically equal to the time that it takes to transmit the smallest size packet on an output port, because it's not very useful to run the input ports and output ports at different speeds—it's like having a cable with different transmission and reception speeds.

Now, on each tick, we can receive a packet from up to N input ports and transmit a packet on up to N output ports. Two or more input ports may have a packet destined to the same output port, requiring us to queue packets somewhere as discussed before. In a shared memory architecture, these packets are queued up in a single pool of shared memory that is common to all output ports and all input ports. This pool needs to support up to N enqueues per tick and up to N dequeues per tick so that the memory is sufficiently powerful to handle the worst case of a packet on each input and output port at every time tick. One benefit of shared memory is the ability to dynamically allocate more or less of the same memory to different output ports on demand depending on how much fanin or incast is happening on a given output port.

[1] The number of input and output ports is typically the same because the input and output ports are really the input and output portions of the same physical port that you see on your router/switch. Also, it typically makes little sense to send a packet from an input port to the output port corresponding to that same input port. It's equivalent to looping back the packet on your own local machine.

2 Output queueing

The N enqueues and N dequeues per tick is quite demanding as memory requirements go. Another architecture for organizing a router's queues is output queueing. Here, instead of having one shared memory pool that is common to all inputs and all outputs, we have N separate memories, one for each output port. Each of these memories is exclusively dedicated to a particular output port alone, which means it only needs to support N enqueues and *one* dequeue per tick, which is a considerable reduction in the number of operations per tick.

The flip side of output queueing is that memory can no longer be shared across output ports. When the router hardware is designed, each output port is allocated a certain amount of memory that cannot be allocated to another output port, even if the first one is idle and the other one is filling up with packets due to an ongoing fanin. This increases the total amount of memory required to support a given queue size limit on each port. Equivalently, if we kept the memory size the same as a shared memory router, it decreases the memory that can be allocated to any one output port.

Output queueing also shares one other defect with shared memory, which is the fact that the memories need to support N enqueues per tick. In other words, the memories need to run N times faster than the links (this is called an N times speedup), which for large N can be quite challenging.[2]

3 Input queueing

The speedup requirement for output queueing led to the development of an architecture called *input queueing* where each input port had a separate memory dedicated to that input port alone. This memory would support 1 enqueue and 1 dequeue per tick. Why 1 enqueue and 1 dequeue per tick?

An input-queued architecture that supports N enqueues and 1 dequeue per tick doesn't make sense on the input port side because packets come in at most once every tick on every input. An input-queued architecture that supports 1 enqueue and N dequeues per tick does not make sense because an input port typically has a packet destined for one output port (as determined by the destination address on the packet). In other words, while it is common for an output port to get packets simultaneously from multiple input ports, it is much rarer for an input port to send the same packet to multiple output ports.[3]

3.1 Input queueing and bipartite matching

In an input-queued router, we need a way to connect up inputs to outputs so that: (1) each input is connected to at most one output (out of the outputs for which this input currently has a packet), and (2) each output is connected to at most one input (out of the inputs that currently have a packet for this output). This is called a matching of inputs to outputs and can be formulated as a bipartite matching problem from your Algorithms class. The input ports and output ports form the two sets of vertices in a bipartite graph and an edge connects an input port to an output port if that input port has a packet destined for that output port.

Solving this matching problem is hard at the speeds at which these routers run, and developing fast algorithms for bipartite matching has been at the core of the development of input-queued routers. Typically, these algorithms try to approximate a maximal match (i.e., a match that cannot be improved further by connecting any input ports to any output ports). Notably, these algorithms do not find a maximum matching (i.e., a match that has the largest number of edges) or a maximum weight matching (i.e., a match that assigns weights to each input-output edges and finds a match with the largest sum of edge weights within the match), because both are hard to do at the speeds of a router. We'll briefly talk about maximum weight matching towards the end of the lecture.

[2] Or at least was in the early 1990s. See §4 for how this situation has changed a fair bit since the mid 2000s.

[3] The single-input-multiple-output pattern does happen in the case of IP multicast, a relatively rarely used capability, where a destination address refers to multiple end hosts and hence results in multiple next hops. We won't deal with multicast in this course.

3.2 Head-of-line blocking

Probably the simplest way to find a matching is as follows. Every input port has memory that is organized as a first-in first-out queue (FIFO) with one enqueue and one dequeue per tick. At every tick, an output port looks at the head (or first) packet of each of the N input FIFOs. If more than one input port has a head packet that is destined to the same output port, the output port picks one of these head packets at random. We repeat this process at each output port.

The problem with this approach is *head-of-line blocking*, which we describe below. Let's say input ports 1, 2, and 3 all have head packets destined to output port 4. Let's say output port 4 picks input port 1 at random out of the three. Then, even if input port 2 has a packet for output port 5 (say), this packet will have to wait behind the head packet at input port 2 that is destined to output port 4. This head packet at input port 2 is less likely to be picked if there are multiple packets destined to the same output port 4. The likelihood of this occurence goes up with increasing utilization of the switch.

Let's consider a very simple model of traffic arrivals, where a packet independently arrives at every input port on every tick with the same probability p. This packet is destined to one of the output ports picked uniformly at random. Then as p increases the likelihood of head-of-line blocking increases. In fact, it can be shown mathematically that as p approaches $2 - sqrt(2)$ [4] for large values of N, the average delay incurred by a packet as a result of waiting in the queue because of head-of-line blocking increases without bound towards infinity. In other words, the system *saturates* and the queues grow unbounded when p exceeds about 58.6%, which is quite a bit below 100% (the capacity of the router). You can visualize this phenomenon yourself as part of an extra credit assignment.

Head-of-line blocking differentiates input queueing from output queueing. In output queueing, the queuing delay incurred at a particular output can be attributed solely to packets that were destined to *that output*. In input queueing with FIFO queues, on the other hand, a packet destined for one output (the packet from 2 to 4 in our example earlier) can cause queueing delay on a *different output* (the packet from 2 to 5 in our example earlier). This possibility of collateral damage makes the performance of input queueing more unpredictable than output queueing.

3.3 Virtual output queues

The reason for head-of-line blocking in our example earlier is that output port 5 cannot "see" that input port 2 has a packet for it that can be transferred on this tick. This is because each output port only looks at the head packet in each FIFO. This situation can be partly remedied by looking at the top K packets in each FIFO starting from the head. But the issues is picking the right value of K. Further, it is hard to read multiple elements from memory in the same tick, and this is especially true if the memory is organized as a FIFO.

One way to make it apparent to output port 5 that there is a packet from 2 that it could be matched with in this tick is to organize the queues at each input port a litte differently. Instead of having a single FIFO at each input port, we could have N FIFOs at each input port, each consisting of packets from that input port to a particular output port. Each of these queues on the input side (now there are N^2 of them in total)[4] is called a *virtual output queue* (VOQs) to denote the fact that it contains a FIFO for each output port.

VOQs solve the head-of-line blocking problem by making it apparent which input port has a packet for which output port, provided the output ports now inspect the head packet of all the VOQs. Viewed differently, it provides a clean way to set K, the number of packets you need to look at at each input port. K turns out to be just the N head packets corresponding to the next packet from a particular input to each of the outputs. However, we still haven't specified how the matching algorithm itself works when given access to VOQs instead of FIFOs. In other words, the FIFO algorithm picked one input port at random out of all the input ports whose head packets were destined to a particular output port. How does an algorithm using VOQs pick input ports for each output port?

[4] Strictly speaking, we don't need a queue from an input port to its corresponding output port, but it doesn't hurt to have one.

3.4 Parallel iterative matching

Probably the first well-known algorithm to exploit a VOQ structure, instead of using plain FIFOs is the Parallel Iterative Match (PIM) algorithm, developed at the Digital Equipment Corporation (DEC) and implemented in some switches that DEC sold in the 1990s.

PIM is quite straightforward to explain, and you can implement it and compare it with the FIFO-based matching solution as an extra credit assignment. It consists of three steps [3, 7]:

1. **Request phase**: Each input port sends out a request to each of the output ports for which its VOQ is not-empty, i.e., the input has at least one packet for that particular output port.

2. **Grant phase**: If an output port receives requests from multiple input ports, it picks one input port at random and grants it the request.

3. **Accept phase**: If an input port receives grants from multiple output ports, it picks one output port at random and accepts the output port's grant.

This process can be iterated multiple times (and hence the name PIM) by only considering the input ports and the output ports that have been left unmatched at the end of the previous iteration. Typically, the number of iterations is bounded to a constant number (4 was the number used in the DEC switch implementation [3]) so as to guarantee that the operation would fix within a deterministic amount of time regardless of the traffic pattern.

One iteration of PIM saturates at a p of about 63%, which is not too much better than the FIFO algorithm. However, additional iterations of PIM raise the saturation limit considerably, and empirically after 4 iterations PIM supports p all the way to just less than 100%.[5]

3.4.1 Hardware considerations: complexity of implementation

This is for informational purposes alone. You won't be tested on this subsubsection.

PIM is simple to understand, but is still a bit too complicated for dedicated router hardware, which makes up router data planes. The first complication is that PIM requires the ability to make a random choice from a time-varying option set (in both the grant and accept phases). This is challenging to implement in hardware. Second, the use of randomness in PIM leads to the possibility that a particular input port is never picked causing starvation and long queues at that input port. An algorithm called iSLIP [6] fixes both these problems with PIM, requires fewer iterations than PIM for high arrival probabilities (hence it is fast), and occupies less gate area relative to PIM when implemented in hardware. We won't describe iSLIP here, but the iSLIP paper [6] is quite readable and provides a good example of understanding just the right amount of hardware-level detail when designing new router algorithms—especially if you want the algorithm to be actually used within a production router chip.

3.4.2 Asymmetric workloads and max. weight matching

This is for informational purposes alone. You won't be tested on this subsubsection.

The maximal matching algorithms above, PIM and iSLIP, work well under symmetric workloads where traffic is spread out relatively evenly across input ports and output ports (e.g., picking an output port for each packet uniformly at random). Under asymmetic workloads, where each input doesn't send to each output uniformly at random, both PIM and iSLIP run into problems [7]. A fix to these problems requires running a maximum weight matching algorithm that takes into account either the queue size of each VOQ (which becomes the edge weight in the edge connecting the VOQ's input port to the VOQ's output port in our bipartite matching formulation) or the waiting times of each VOQ [5]. This algorithm saturates only at 100%, meaning it can handle the full range of admissable workloads even when the workload is asymmetric. Unfortunately, the algorithms for max-weight bipartite matching are too slow to run on a router, but several approximations have been developed to this as well [1].

[5] Which is the best we can hope for in any system, because there is no way to support a workload where the input exceeds the output in steady state.

4 Technology trends and routers today

Input-queued routers were the preferred architecture for building routers in the mid 1990s. This was at least partly because these routers had large amounts of memory, which were typically built out of DRAM (the same technology used for memory in your laptops/desktops). But DRAM was also slow, meaning it could not support the increased number of enqueues and dequeues (recall the speedup problem) demanded by output-queued or shared-memory routers. As a result, input-queued routers were really the only feasible technique to building routers with large memories.

Today, on the other hand, things are a bit different. A large number of routers today[6] are found in large-scale *datacenters*, i.e., private networks within large companies that are the size of several adjacent football fields.

The volume of routers in such datacenters is considerable: reliable numbers are hard to come by, but anecdotal estimates put it at a few 100K such routers in total for a company like Amazon, Google, Microsoft, or Facebook. At these volumes, reducing router cost is important. To keep costs low, router chips vendors such as Broadcom have opted for minimal router feature sets, in contrast to the routers found in ISP networks in the 1990s, which were bloated with features.

One consequence of the minimalism in routers has been the reduction of memory available for queueing packets because the amount of memory in a router chip has a significant impact on its cost. This reduction of memory has also been partly aided by the development of better end-to-end congestion control algorithms that keep queue occupancy low [2]. The net result of reduced memory is the ability to use faster memory technologies such as SRAM, instead of DRAM. Roughly speaking, the faster the memory technology, the more it costs per bit, and the more chip area it consumes per bit. Hence, the reduction in memory requirements means that it is now feasible to build queueing structures out of SRAM instead of DRAM and still be cost competitive.

The use of faster SRAM relative to slower DRAM has resulted in renewed interest in output queued and shared memory architectures—especially for routers found in data centers (the so called merchant silicon routers). This is because SRAM enables us to support more enqueues and dequeues per tick and hence makes output queueing and shared memory more viable. In other words, because the memory on routers had to shrink to cut costs, the memory technology changed, and because memory technology changed, it allows the queueing architecture to also change from input queueing to output queueing and shared memory.

This story is meant to illustrate how it is useful to pay attention to technology trends. The considerably involved (and extremely interesting) algorithms in input-queued routers only make sense if the technology trends call for an input-queued router. Otherwise, it's just much simpler to build an output-queued or shared-memory router where you don't even have to worry about developing a bipartite matching algorithm!

References

[1] Adisak Mekkittikul. Scheduling Non-Uniform Traffic In High Speed Packet Switches And Routers, 1998.

[2] Mohammad Alizadeh, Albert Greenberg, Dave Maltz, Jitu Padhye, Parveen Pate, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[3] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High-speed Switch Scheduling for Local-area Networks. *ACM Transactions on Computer Systems*, 1993.

[4] Mark J. Karol, Michael G. Hluchyj, and Samuel P. Morgan. Input Versus Output Queuing on a Space-Division Packet Switch. 1988.

[5] N. McKeown, V. Anantharam, and J. Walrand. Achieving 100% throughput in an input-queued switch. In *INFOCOM*, 1996.

[6] Technically these are switches, which are like routers, except that they forward packets based on MAC addresses instead of IP addresses. But the hardware architecture of switches and routers is quite similar when it comes to the queueing subsystem.

[6] Nick McKeown. The iSLIP Scheduling Algorithm for Input-queued Switches. *IEEE/ACM Transactions on Networking*, 1999.

[7] Nick McKeown and Thomas E. Anderson. A quantitative comparison of iterative scheduling algorithms for input-queued switches. *Computer Networks and ISDN Systems*, 1998.