47

Chapter 8

Switching

8.1 Introduction

Every day, hundreds of millions of people use the telephone network to get in touch with a billion or so telephone customers around the world. These calls travel through a vast maze of switches and transmission lines to reach across the globe. How do telephone switches handle so many simultaneous calls?

Every day, millions of people use the World Wide Web to reach tens of millions of Web sites. With a click of a mouse, a user's request for data snakes its way across the Internet and contacts a remote site. The response works its way back through a vast interconnection of routers and returns to the user. How do the routers in the Internet handle this flood of packets, forwarding each packet to its destination?

In this chapter, we will study *switching*, the process by which a network element forwards data arriving at one of its inputs to one of its outputs. We will study three kinds of switching elements: (a) telephone switches, which support the telephone network, (b) datagram switches (also called *routers*), which tie the Internet together, and (c) ATM switches, optimized to deal with small, fixed-size packets called *cells*.

8.1.1 Classification

We categorize switching elements in two ways (Table 8.1). Routers and ATM switches are both *packet* switches, that is, they switch packets that contain both data and descriptive meta-data. In contrast, telephone switches are *circuit* switches that switch voice samples that contain no meta-data.

Scanned by CamScanner

	Connectionless (datagram)	Connection-oriented (switching system = switch + switch controller)
Packet switch	Internet router	ATM switching system
Circuit switch		Telephone switching system

Table 8.1: Classification of switching elements.

We also categorize switching elements as *connectionless* or *connection-oriented*. A connectionless switching element or router reads the destination address in an incoming packet's header and looks up a routing table to determine its output interface. Each packet is *self-contained* in that it contains sufficient information to be routed to its destination. In contrast, a connection-oriented *switching system* consists of a switch and a switch controller. During connection setup, the *switch controller* uses the destination address to associate a data forwarding path in the *switch* with either a particular time slot or a virtual circuit identifier. During the data transfer phase, the switch moves cells or samples from an input to an output using this association. Note that in a connection-oriented switching element, correct switching requires preestablishment of a data forwarding association. In this sense, neither voice samples nor ATM cells are self-contained.

Besides switching packets, switching elements perform three other important functions. First, they participate in routing algorithms to build *routing tables*, which map a destination address to an output trunk. A connectionless switch consults the routing table for every incoming packet, and a connection-oriented switch controller consults this table during connection establishment.[1] We will study how routing tables are built in Chapter 11. Second, switching elements resolve contention for access to an output trunk. By intelligently *scheduling* access, a switching element can give different connections (or packets) different qualities of service. We will study scheduling in more detail in Chapter 9. Third, a switching system in a connection-oriented network participates in *admission control* to decide whether to accept an incoming call. We will study this in Chapter 14.

In this chapter, we focus on routers and the switch portion of a switching system. Unless otherwise specified, we will use the term "switch" to refer to them both.

8.1.2 Requirements

The *capacity* of a switch is the maximum rate at which it can move information, assuming all data paths are simultaneously active. The primary requirement of a switch is to maximize capacity for a given cost and a given reliability. In practice, only a fraction of paths are simultaneously active, so switches rarely achieve their rated capacity.

[1] During data transfer, an ATM switch looks up a translation table for each incoming cell to determine its output [illegible]

A circuit switch must reject a call if it does not have a path from an input to an output to carry the call, since it cannot buffer data. This is called *call blocking*. In a packet switch, data can be stored in a buffer, so call blocking is not a concern. Instead, the analog of call blocking is *packet loss*, that is, the loss of one or more packets because of a buffer overflowing when a burst of packets arrives at the switch. The second requirement for a switch is that it should minimize call blocking and packet loss.

Finally, a switch should not reorder packets. With reordering, receivers must buffer out-of-order packets, increasing both their cost for memory and the end-to-end packet delay. Consequently, ATM switches are prohibited from reordering cells. Routers are more lax in maintaining packet order, and packet reordering is common in the Internet.

8.1.3 A generic switch

A generic switch has four parts: input buffers, a port mapper, a switch fabric, and output buffers (Figure 8.1).

- Input buffers store packets or samples as they arrive on the input lines. Some switches have tiny input buffers that hold data only while it is contending for the switch fabric. Other switches have almost all their buffers at the inputs.
- The port mapper is found only on packet switches. It reads either the destination address or a virtual circuit identifier from an incoming packet's header and uses

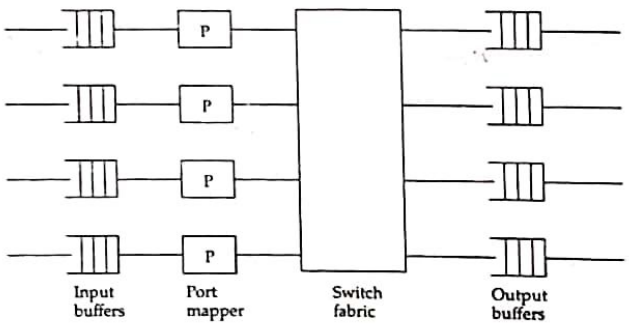

Figure 8.1: A generic switch. Data arrive on multiple trunks and are stored in an input buffer. In a packet switch, a port mapper assigns each packet to one or more outputs. A circuit switch does not need a port mapper, since the destination of a sample is known from its arrival time on a multiplexed input. A switching fabric moves data from input buffers to output buffers, which then transmit data to output trunks.

Scanned by CamScanner

a table to decide the packet's output port. A circuit switch does not need a port mapper because each time slot is automatically associated with a path from an input to an output.

- The switch fabric routes data from an input to an output. The simplest switch fabric is a processor that reads data from an input port and writes it to an output port. Switch fabrics can also be complex multiprocessor systems that simultaneously transfer thousands of packets along many parallel data paths.
- Output buffers store data as they wait for a turn on the output line. At each output port, a scheduler manages the output buffers and arbitrates access to the output line. As with input buffers, these can be small or large.

Some switches distribute, combine, or omit one or more of these functions. For example, a switch may combine the input and output buffers, distribute the port mapper among the input ports, or omit the input buffers. We will see examples of these later in the chapter.

8.1.4 Rest of the chapter

We will first study circuit switching in Section 8.2, then study packet switching, which encompasses both datagram and virtual circuit switches, in Sections 8.3–8.6. Section 8.3 is an introduction to packet switching and discusses three types of packet switches. Section 8.4 concentrates on switch fabrics for packet switches. Section 8.5 deals with the placement of buffers in packet switches, and Section 8.6 is devoted to multicast switches.

8.2 Circuit switching

A telephone circuit carries voice samples that are 8 bits long and correspond to 125 μs of sampled voice. Note that a sample does not have a header describing its source or destination: we infer this from the physical line on which it is present and the time at which it is placed on the line (i.e., its position in a *frame*). In this section, we study how to switch samples from an input of a telephone switch to an output. Before we study switches, we first make a small diversion to study the essentials of multiplexing and demultiplexing.

8.2.1 Multiplexors and demultiplexors

Most telephone trunks carry more than one voice circuit, sharing (or *multiplexing*) the trunk among samples from these circuits. At a central office, calls arriving from active local loops are combined before being placed on a shared long-distance trunk, and calls to active local loops are removed (or *demultiplexed*) from incoming trunks. Multi-

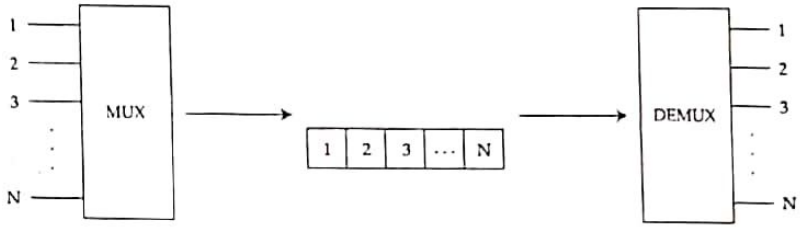

Figure 8.2: Multiplexing and demultiplexing. A multiplexer takes samples from N inputs and places them one by one on an output that is N times faster. A demultiplexer distributes data from a fast input onto N outputs, each N times slower than the input. Neither requires header information to do its task.

plexors and demultiplexors are, therefore, fundamental to the operation of the telephone network.

A synchronous multiplexor consists of N input lines and an output line that runs N times as fast (Figure 8.2). Each input line is associated with a buffer that can store at least one sample. The multiplexor visits each input buffer in turn, placing the sample on the output line. If the input lines are 64 Kbps, then a sample lasts 125 μs on the input line, but slightly less than $125/N$ μs on the output line (because framing bits take some time).

We can cascade multiplexors to carry more voice circuits on a single physical line. The T hierarchy, described in Chapter 2, is the set of rates allowed on a multiplexed line in the telephone network. For example, a T1 line carries 24 voice circuits at 1.544 Mbps, and a T3 line carries 28 T1s at 44.736 Mbps.

A demultiplexor has one input line and N output lines that run N times slower than the input line. Each output line is associated with at least 8 bits of buffering. Incoming samples are placed in an output buffer in round-robin order. From there they are transmitted on the corresponding output lines. Note that neither a multiplexor nor a demultiplexor needs addressing information. Their operation requires only accurate timing information.

Inverse multiplexing

Inverse multiplexing is the aggregation of multiple independent information channels across a network to create a single higher-rate information channel [Duncanson 95]. For example, with inverse multiplexing, we can combine three 1.5-Mbps T1 channels to create a single 4.5-Mbps channel. The incoming data stream is partitioned into multiple data streams and sent on separate connections. At the receiver, misordered data is reordered, and the separate streams are combined to form a single higher-rate stream. An *inverse multiplexor* establishes the separate channels and makes sure they stay up, establishing alternate connections if performance on an existing connection degrades. It can also dynamically change the

Scanned by CamScanner

number of streams to match the incoming data rate. Thus, from the perspective of the endpoints, an inverse multiplexor provides a single higher-rate stream, with a capacity that can dynamically adapt to the rate of the incoming data stream.

Inverse multiplexing is very useful in networks that are unable to provide individual higher-rate streams. For example, inverse multiplexors can be used to aggregate multiple telephone channels to provide a higher-speed link between datagram routers. They are commonly used to link corporate LANs over wide-area telephone circuits [Duncanson 95].

Add–drop multiplexing

A common use for multiplexors and demultiplexors is in adding or removing some calls from a trunk. Consider a DS3 trunk from New York to Chicago that passes through Harrisburg, PA. At Harrisburg, we may want to remove calls destined locally, and insert other calls that originate from Harrisburg and go to Chicago. Instead of installing a switch at Harrisburg, which is expensive, we can accomplish this task with an *add–drop multiplexor*. This demultiplexes the DS3 to DS0s, allowing some calls to be diverted, while new calls are remultiplexed and put back on the trunk.

Add–drop multiplexors for DS trunks are expensive (though less expensive than switches) because DS framing requires complete demultiplexing before calls can be added or removed. One motivation to migrate from the DS hierarchy to Synchronous Optical (SONET) links is that we can add or remove calls from a SONET trunk without demultiplexing it into DS0s, because the SONET frame header carries enough information to locate a single call. This allows us to build cheaper add–drop multiplexors for SONET [Bellamy 91] (also see Section 15.2.3).

8.2.2 Circuit switching

A circuit switch handling N voice circuits conceptually consists of N inputs and N outputs (in practice, N can be as large as 120,000). This corresponds to far fewer physical input and output lines if these lines are time-multiplexed. For example, if the inputs are DS3s that each carry 672 voice calls, the number of physical lines is reduced by a factor of 672. In this chapter, if the inputs are multiplexed, we will assume that all inputs are at the same level in the multiplexing hierarchy, and that voice samples on all inputs arrive synchronously. (In reality, small jitters in the arrival times of samples must be smoothed out by an input buffer.) The goal of a circuit switch is to get an incoming sample to the right output line and to the right slot in the output frame.

Before we study switching strategies, we distinguish between switches that connect individual telephone lines, called *line* switches, and switches that interconnect multiplexed trunks, called *transit* switches [Bellamy 91]. A line switch must connect a specific input to a specific output. In contrast, a transit switch need only connect an input to *one* of the outputs going in the right direction. For example, a transit switch in Chicago may switch some calls from a 672-call DS3 from New Jersey to another DS3 going to California. In this case, the switch need only connect an input line to one of the outputs going to California.

The difference between line switching and transit switching becomes important when considering the possibility of *blocking* in a switch. A call is blocked if no path can take it from an input to an output. At the time of call admission, the switch controller rejects blocked calls. There are two types of blocking: *internal* blocking and *output* blocking. Internal blocking occurs if the output is available, but there is no path through the switch. Output blocking occurs if two inputs compete for the same output, in which case one of them is output blocked. To return to line and transit switching, note that if an input can choose to go to any one of a set of outputs, then both internal and output blocking are reduced. Thus, a transit switch can achieve the same blocking probability as a line switch with less hardware.

The two basic strategies in circuit switching are time-division and space-division switching. We will study them next, then consider multistage switches that combine time and space switching. These topics are covered in greater detail in reference [Bellamy 91].

8.2.3 Time-division switching

A *time-slot interchanger* or *TSI* used in time-division switching has only one input and one output line, but these lines carry samples from N multiplexed voice circuits. A TSI writes incoming samples in an N-byte buffer, one byte per incoming circuit, and reads them out in a different order. The different order on the output line, when passed through a demultiplexor, results in samples being placed on different output lines. In other words, a TSI switches samples by rearranging the order of samples on a multiplexed line.

EXAMPLE 8.1

Consider a company that has two branch offices A and B, each with 24 lines. We want any telephone in A to be able to connect over a T1 line to any telephone in B. We do this by installing small switches, called *private branch exchanges* (PBXs), at sites A and B. Each PBX has 24 lines connecting it to 24 telephones and a T1 connecting it to the other PBXs (Figure 8.3a). Samples from each telephone are multiplexed in some fixed order and placed on the T1 line. Samples arriving on the T1 connection are given to a TSI, which has 24 1-byte buffers into which it cyclically writes these samples. The TSI interchanges samples before handing them to a demultiplexer.

Suppose the second telephone in A, corresponding to the second sample in the frame leaving A, wants to connect to the third telephone in B. To make this connec-

Scanned by CamScanner

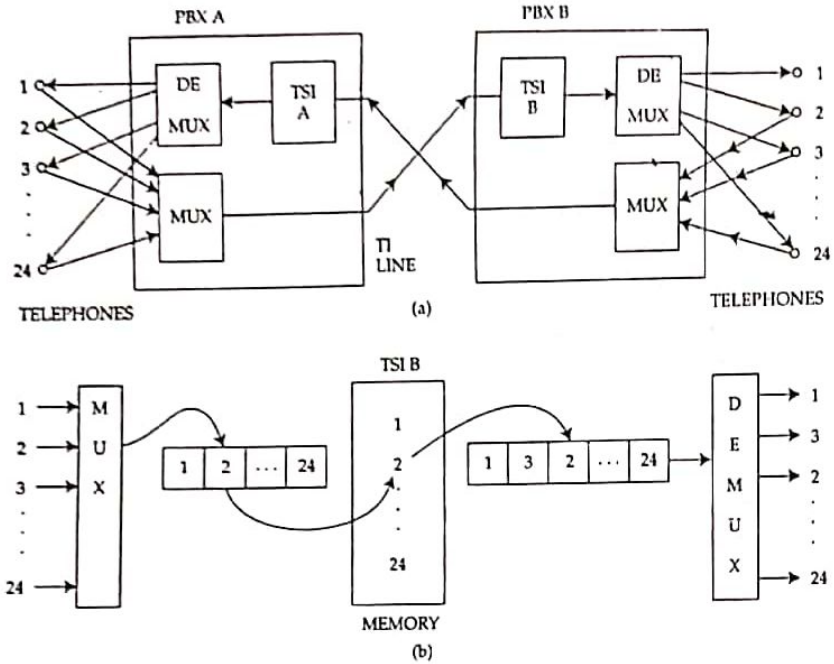

Figure 8.3: Time-division switching. A time slot interchange reorders samples on a trunk. This has the effect of switching when the trunk is demultiplexed. See Example 8.1 for details.

tion, the TSI at B places the second input sample in the third time slot on its output line (Figure 8.3b). When the output line of the TSI is demultiplexed, A's second sample is thus placed on B's third line, as desired.

If we wanted to build a large switch from a single TSI, how large a switch can we build? The limit to scaling a TSI is the time taken to read and write a sample from memory. Suppose we wanted to build a switch as large as the largest switches currently in the field, which handle around 120,000 calls. To build a TSI that can switch 120,000 circuits, we need to read and write samples from a memory 120,000 times every 125 μs. This would require each read and write operation to take on the order of half a nanosecond, which is unattainable at any price with present technology. Thus, building a switch this size with a single TSI is not practical. These days, typical DRAM access times are around 80 ns, and 40-ns memories can be bought quite cheaply. If we use a 40-ns memory, the access time is 80 times as slow as what we need for a 120,000-line switch, so the switch capacity is at most 120,000/80 = 1500 circuits, which is quite small! Thus, to build large switches, we need to turn to other techniques.

8.2.4 Space-division switching

In space-division multiplexing, each sample takes a different path through the switch, depending on its destination. We illustrate this using an example.

EXAMPLE 8.2

Suppose we wanted to use space-division switching in the PBXs of Example 8.1. We can do so by first demultiplexing the T1 line onto 24 different lines, then connecting the demultiplexor's output to the correct telephone (Figure 8.4). The sample from the second telephone in A appears at the second output line of the demultiplexor. This is connected, using the space-division switch, to the third telephone in B. Other samples can be simultaneously switched along other paths in the space-division switch.

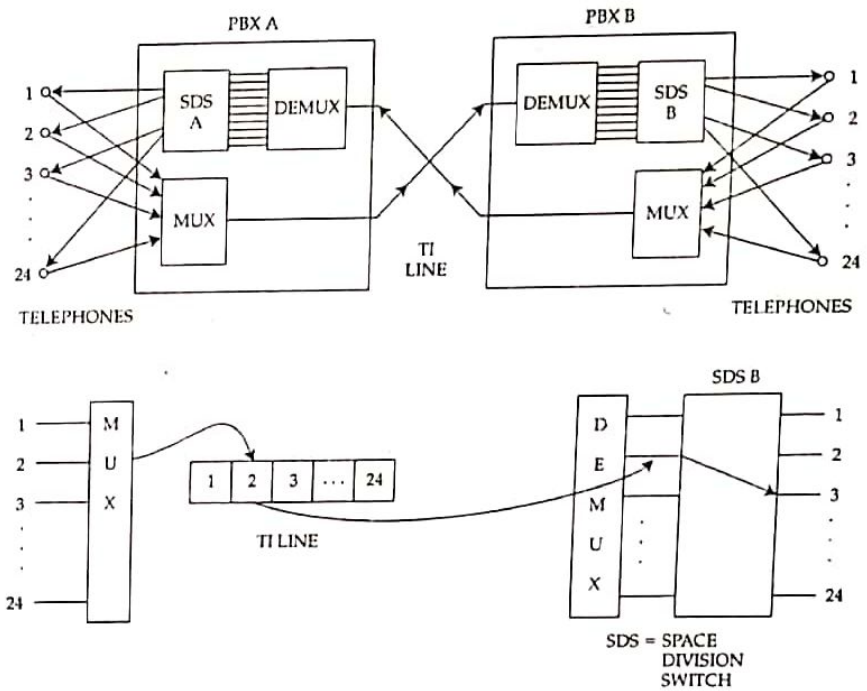

Figure 8.4: Space-division switching. Samples use spatially distinct paths in a space division switch. Thus, the space division switch in PBX B distributes the demultiplexed samples to the appropriate output ports over separate physical paths.

Scanned by CamScanner

Crossbar

The simplest space-division switch is a crossbar (Figure 8.5). Samples arrive on rows and are output along columns. Active elements called *crosspoints* are placed between input and output lines. If the crosspoint is active, data flows from the input to the output line; otherwise, the two are disconnected. (Note that a crossbar need not have the same number of input lines and output lines. An $n \times k$ crossbar has n input and k output lines.) If the input lines are not multiplexed, a crossbar always connects the same inputs to the same outputs, much like a static patch panel. However, if the input lines carry multiplexed samples, and different samples have different destinations, a crossbar requires a *schedule* that tells it which crosspoints to activate during each sample time. Depending on this schedule, samples are transferred from the selected input to the selected output.

A crossbar is *internally nonblocking* (i.e., no sample is blocked in the switch waiting for an output line). Unfortunately, an $N \times N$ crossbar uses N^2 elements and therefore is expensive for large N, such as $N = 100{,}000$ (also see the next box). A second problem with building large crossbars is that the signal distribution time; that is, the time taken for a controller to inform each crosspoint whether it should be on or off, grows large for large N. Third, a single crosspoint failure isolates an input from an output line, making a crossbar less robust to faults than the multistage crossbars we discuss next. However, crossbars are an excellent solution for building smaller (say, 8×8 or 64×64) switches.

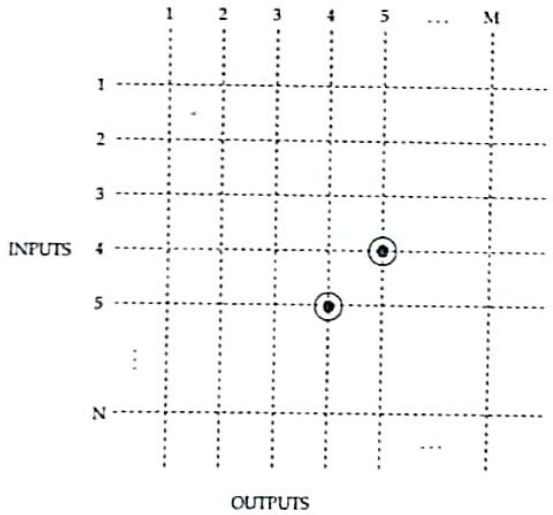

Figure 8.5: Simple crossbar. Inputs arrive along rows and are placed on columns if the corresponding crosspoint is active. A schedule sets each crosspoint for each incoming sample.

As we will see, more complex switch designs are composed from smaller switching blocks, which can be built as crossbars.

Multistage crossbar

A simple generalization of a crossbar allows us to save many crosspoints. Notice that during each switching time (the time taken to transfer one sample across the crossbar) only one crosspoint in each row or column is active. If we can somehow get an internal element to receive samples from multiple input lines, we can get rid of this inefficiency. This is done with multistage switching (Figure 8.6). Here, the inputs are broken into groups, which are internally switched by a crossbar. We allow multiple paths between input and output groups to share a center stage, thus reducing the number of crosspoints. However, at each time slot, each array in the multistage switch has to be rearranged. In a typical multistage switch, the first stage consists of N/n arrays of size $n \times k$, the second stage consists of k arrays of size $N/n \times N/n$, and the third stage consists of N/n arrays of size $k \times n$. In Figure 8.6, $N = 20$, $n = 10$, and $k = 3$.

Notice that we now have k disjoint paths from any input array to any output array. If k is small, as in Figure 8.6, it is possible that an input and output are both idle, but no path to connect the two exists (i.e., the switch can be internally blocking). How large should k be for the switch to be internally nonblocking? Clos showed that if a switch controller is willing to rearrange existing connections when a new call arrives, the condition is [Clos 53]:

$$k \geq n \tag{8.1}$$

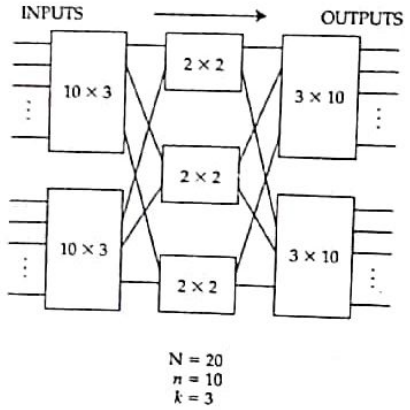

Figure 8.6: Multistage switch. The key idea is that each element in the center stage can be accessed by multiple elements in the first stage. This reduces the wasted crosspoints in a single-stage crossbar.

Scanned by CamScanner

Thus, a switch using a Clos network is also called a *rearrangably nonblocking* switch. In practice, circuit switches rarely rearrange existing connections, so this result is primarily of theoretical interest.

A Clos network of size $N \times N$ has:

$$\text{number of crosspoints} = 2N(2n - 1) + (2n - 1) \cdot (N/n)^2 \qquad (8.2)$$

which is smaller than N^2, as is the case with a crossbar. This number is minimized when $n = O(\sqrt{(N)})$.

In a single-stage switch, finding a path from an input to an output is trivial. In a multistage switch, the switch controller must find a valid path from an input to an output at the time of call admission. The controller keeps track of the inputs and outputs that are free during each time slot, and uses a search algorithm, such as depth-first-search, to find a path. This path is then stored in a switch schedule and handed to the switch for execution during actual data transfer.

Even a Clos network does not scale very well with N. For example, if we let $N = 120{,}000$, the optimal $n = 346$; then we need $k > 690$, say $k = 691$. Then, the corresponding Clos network needs 248.8 million crosspoints! To reduce this number to reasonable values, we need to combine time and space switching. Alternatively, we can choose to build a blocking switch, and accept the risk of a call being rejected due to internal blocking. A telephone company has to make the trade-off between the call blocking probability and the cost of the switch. In practice, most telephone switches do have a nonzero, but small, blocking probability. We will study how to compute these blocking probabilities in Section 14.11.2.

Are crosspoints really expensive?

Crosspoints used to be electromechanical devices that were expensive and prone to failure, and so minimizing the number of crosspoints made sense. With current VLSI designs routinely placing millions of reliable gates on a single chip, do we really need to worry about the number of crosspoints?

A typical VLSI implementation uses a set of N input buses, each attached to N output multiplexors. Each output multiplexor is programmed to select one of the N inputs. The complexity in building a VLSI crossbar is the chip area for N multiplexors. Since a multiplexor that selects one of the N inputs takes area proportional to N, an $N \times N$ crossbar requires a chip area proportional to N^2.

For large N, a crossbar's cost increases not only because of the large chip area, but also because the crossbar needs to be split across multiple packages. Moreover, the scheduler must program each of the output multiplexors individually, which takes time. For example, for a 64×64 crossbar, the scheduler must address and

individually set up 64 output multiplexors. Setting one multiplexor requires six addressing pins and one data pin. Even if a crossbar chip allocates fifty-six pins for scheduling so that a scheduler can address eight multiplexors simultaneously, setting up the crossbar takes eight clock cycles. For these reasons, a crossbar is useful only when N is small (64×64 or smaller). We need to look to other switching fabrics to reduce the number of switching elements when building large switches.

8.2.5 Time–space switching

Recall that a time-slot interchange (TSI) has a single input and a single output, and can be thought of as rearranging samples on the input line. In time-space (TS) switching, the outputs of many TSIs are fed into a space-division switch (Figure 8.7). There are two ways to view a TS switch. From one perspective, the TSI delays samples so that they arrive at the right time for the space division switch's schedule. If the time or T stage were missing, then, at a given time, we might have many inputs of the space or S stage destined for the same output, which would cause blocking. By rearranging the order in which the S stage receives samples, we gain a degree of freedom in picking the S stage schedule. This allows us to build a space-division switch with fewer crosspoints than a Clos switch, but still make it nonblocking.

EXAMPLE 8.3

Consider a 4×4 crossbar shown in Figure 8.7 that has four time-multiplexed inputs, each carrying four circuits. Thus, the switch switches 16 circuits, four at a time. Note that in the first time slot, samples from both circuit 1 and 13 arrive. If both want to use output trunk 1, one of them will be blocked. We call this output blocking. Now, suppose we insert TSIs at each input. We can rearrange the sample from 1 so that it arrives at time slot 2 instead, so that 1 and 13 no longer conflict. This is the key advantage of the TS switch: it can reduce blocking by rearranging inputs.

A second way to view TS switching is to think of the space-division switch as allowing us to swap samples between two TSI output lines. A TSI allows us to interchange the order of samples within a line, but does not allow us to place a sample on a different output line. By tying together a set of TSIs with a space-division switch, we can take a sample arriving on, say, input line 1, and transmit it on, say, line 3.

Scanned by CamScanner

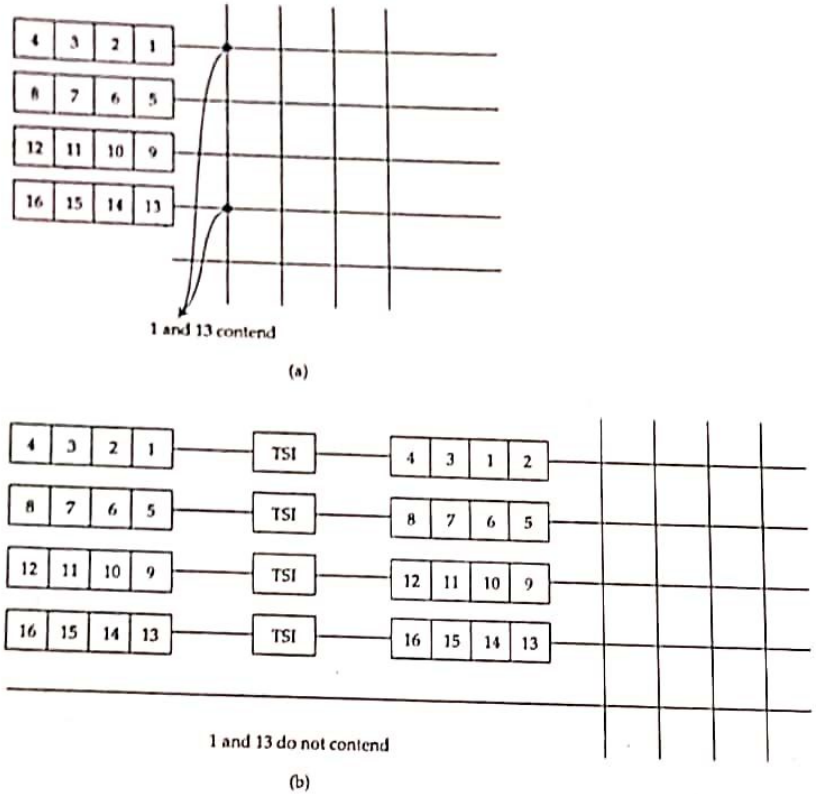

Figure 8.7: Time–space switching. Note that in (a) the samples in time slots 1 and 13 contend for output 1, so that one of them has to be buffered or dropped. With a TSI interchanging samples 1 and 2 (b), the contention disappears. Thus, the TSI allows reduced blocking in the crossbar.

8.2.6 Time–space–time switching

A time–space–time (TST) switch is similar to a three-stage crossbar, except that we replace the input and output crossbar stages with TSIs (Figure 8.8). The easiest way to understand the functioning of a TST switch is to compare it with a TS switch. A TS switch can reorder samples within an input line, and switch them to a different output line. Suppose two input lines want to send a sample to the same output line simultaneously, leading to output blocking. We could then use the TSIs at the input to rearrange the samples so that they did not suffer from output blocking. However, this rearrangement may result in samples arriving at the outputs out of sequence. By placing a second TSI at

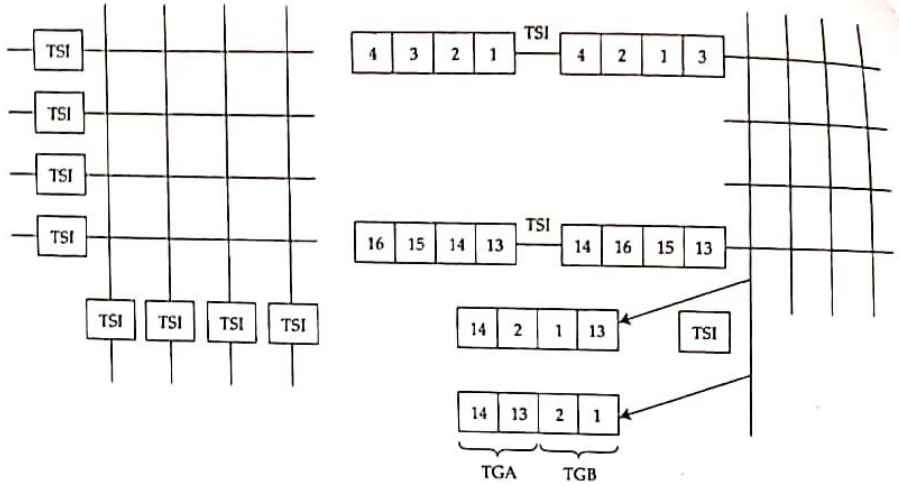

Figure 8.8: Time–space–time switching. A TST switch is similar to a TS switch, except that TSIs at the output allow samples to be rearranged at the output as well as the input. Thus, rearranging samples at inputs as well as outputs lets any free slot in the space-division switch be used for switching.

every output, we can achieve any desired sequencing on each output line. Thus, we can use any available time slot of the space-division switch to exchange samples between lines, reducing the chances of call blocking in the space-division switch.

EXAMPLE 8.4

In Figure 8.8, assume that samples 1 and 2 from input line 1 and samples 13 and 14 from input line 4 should be switched to output line 1. Further, assume that the first two samples on output 1's frame are sent on trunk group A, and the second two samples on trunk group B. We want samples 13 and 14 to be sent on trunk group A, and 1 and 2 on trunk group B. To achieve this, note first that samples 1 and 13 and samples 2 and 14 are output blocked. At the input TSIs, we rearrange them as shown to avoid output blocking. However, the order at the output line, 14, 2, 1, 13 is wrong. The TSI on the output line allows us to rearrange the order appropriately.

Many generalizations of TST switching are used in practice. For example, we can replace the S stage by a three-stage space-division switch, thus reducing the number of cross-points. This leads to a TSSST fabric, used in the EWSD switch from Siemens. The 4ESS switch from Lucent Technologies, which uses a similar principle, is a TSSSST switch with

Scanned by CamScanner

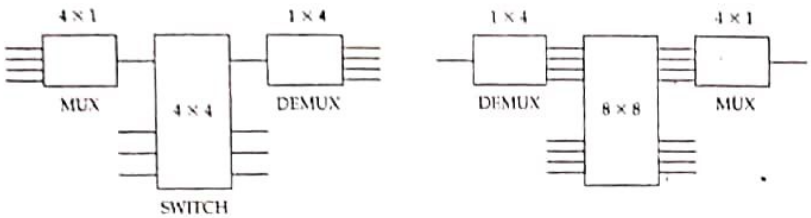

Figure 8.9: Dealing with line speed heterogeneity. If some incoming trunks are slower than others, they can be multiplexed to the standard line speed. Conversely, if some trunks are faster than others their load can be spread among multiple input ports.

four internal space-division stages and two TSIs at the ends. These complex multistage switches can handle 100,000 to 200,000 circuits and are widely deployed in the telephone network.

Dealing with line heterogeneity

Most circuit-switch designs assume that all incoming and outgoing trunks have the same rate. If this is not the case, we can use multiplexors to work around the problem. If some incoming trunks are too slow, we can multiplex them before they enter the switch. Conversely, if an input is too fast, it can be demultiplexed before being switched. Figure 8.9 illustrates multiplexor and demultiplexor placement for these solutions. In the rest of the section, we will assume that all input and output trunks have the same speed.

The true cost of telephone switching

Though space and time division switches lie at the heart of telephone switching, the real cost of a switching system is usually not in the switch, but in the associated line cards. A line card converts analog voice from the customer into digital samples. A central office needs only one switch for a hundred thousand customers, but one line card for each customer![2] Thus, even with cheap line cards, their costs dominate the cost of a central office.

Stepping back a bit further, the cost of line cards, in turn, is dwarfed by the cost of laying copper wire from a central office to customer premises. This "local loop" is responsible for nearly 80% of the cost of telephone infrastructure. Similarly, at long-distance switching centers, switching costs are dwarfed by the cost of laying long-distance transmission lines, acquiring rights of way, and the cost of writing, updating, and maintaining complex switch-control software. Thus, although clever optimizations in switching are of academic interest, in the larger scheme of

[2] To be precise, we need one analog-to-digital conversion circuit per customer. In practice, typical line cards contain 16 of these circuits.

things, saving a few thousand crosspoints is not likely to make the overall cost of providing telephone service much lower.

8.3 Packet switching

In a circuit switch, the output port of a sample is completely decided by the time at which it arrives at an input port. At every time slot, we can trace the path taken by an incoming sample to a particular output port. Thus, a circuit switch does not need a sample to carry a header to switch it. In contrast, with a packet switch, incoming packets carry a destination field that allows them to be stored and transmitted later. Consequently, unlike in a circuit switch, the destination of an incoming packet is no longer a function of its arrival time. Thus, we cannot statically allocate a path through the switch at each time slot. Instead, a port mapper examines packet headers to determine the output port for incoming packets.

In this section, we will study two types of packet switches: virtual circuit ATM switches and datagram routers. The difference between them is that ATM switches handle fixed-size packets and the port mapper decides a destination based on the packet's VCI. In contrast, a router handles variable-size packets, and the port mapper has to parse the entire destination address to decide a packet's destination. These are small enough differences that we will treat both the same, calling them both "switches." We discuss the effects of packet size on performance toward the end of this section.

Repeaters, bridges, routers, and application-level gateways

Datagram switches are given different names depending on their position in the protocol stack. *Repeaters* operate at the physical level and repeat the input signal on one or more outputs, amplifying the signal strength. They allow a link to span a longer distance. *Bridges* operate at the datalink layer to partition a shared LAN medium (such as Ethernet) into segments. A bridge forwards incoming packets by buffering them, then contending as any other station would on the destination segment. Bridges discover the end-systems on each segment by snooping on the source datalink-layer address on packets arriving on that segment. Incoming packets whose source and destination are in the same segment do not leave the segment. Carefully partitioned segments, therefore, allow network administrators to build large shared-medium LANs that still have good performance.

Routers operate at the network layer. They examine a packet's network-level address (such as an IP address) to forward it on one of their output links. Unlike bridges, which discover the network topology mainly by observing their inputs, routers do so by participating in elaborate routing protocols (described in Chap-

Scanned by CamScanner

ter 11). Routers allow LANs or smaller networks to be interconnected, to form *internetworks*. A good reference on bridges and routers is reference [Perlman 92].

An *application-level gateway* is a datagram switch that operates at the application layer. Unlike peer routers or bridges, which communicate with each other over single hops, peer application-level gateways may be separated by the diameter of the network. Application-level gateways usually translate among data representations or protocols. For example, exchanging electronic mail between the Internet and a commercial online service which has its own mail protocol requires the intervention of an application-level gateway. Such gateways typically operate on entire messages, with full knowledge of application semantics, instead of at the packet level. Common examples of application-level gateways are mail gateways; Web proxy gateways, which cache Web traffic to reduce access time to popular Web sites; and *transcoders*, which convert multimedia data from one format to another.

As we progress from repeaters to bridges, routers, and application-level gateways, we gain functionality, but lose forwarding speed. To obtain the best performance, forwarding functionality should be pushed to as low a level in the protocol stack as possible.

Port mappers in datagram routers

Before studying packet switching, we briefly examine the design of port mappers for routers. Routers examine an incoming packet's destination address and match it with a routing table to decide its destination interface number. This matching is nontrivial because the router must *efficiently* find the *best* match between the destination and all the entries in the routing table. For example, assume that packets are addressed using IP addresses (described in Section 3.3.1), and the incoming packet has a destination of 128.32.1.20. If the routing table has entries for 128.32.*, 128.32.1.*, and 128.32.1.20, the router should match only the third entry. This best-match problem is hard because a routing table may have tens of thousands of entries.

A *trie* data structure is commonly used to find the best match [Perlman 92]. We define an *alphabet* to be a finite set of elements used to form address strings. A trie is a tree where each node corresponds to a string that is defined by the path to that node from the root. The children of each node correspond to every element of the alphabet. In practice, to keep things simple and maximize access speed, each node is represented by an array that is as long as the alphabet size. An entry in the array can have one of three values:

(a) a pointer to another array, indicating that better matches exist
(b) a special symbol indicating that no better matches are known
(c) a null pointer, indicating that the longest match is to the parent of this node

In cases (a) and (b), the element is also annotated with the output interface for route to the destination associated with that node. Incoming packets are forwarded on the inter-

face number corresponding to the best match. The operation of a trie is illustrated in the next example.

EXAMPLE 8.5

Consider the trie in Figure 8.10. In the figure, the alphabet consists of the natural numbers {0, 1, 2, . . . , 255}. The root is associated with the null string and has the default route for the router (the next hop for packets whose destination is unknown). The root has 256 possible children, of which children 128, 32, and 10, corresponding to IP addresses 128.*, 32.*, and 10.*, have known routes. Thus, at the top level array, the tenth and thirty-second entries contain a special symbol annotated with the interface number for the next hop toward that destination, and the 128th entry contains a pointer to the next level of the trie. All other entries are null, corresponding to the default route. The second level of the trie, below 128, corresponds to a better match for certain addresses. Here, the trie stores pointers to lower levels at the thirty-second and fifty-fourth entries, corresponding to IP addresses 128.32.* and 128.54.*. All other entries are null. Similarly, the 128.32 level is expanded to form more detailed routes.

When an IP packet with a destination, say, 128.32.1.2, arrives at the router, it follows the trie links as far down as possible. The router first parses the incoming address into its fields, which are letters in the alphabet. Here, the string is "128, 32, 1, 2." At each level, if the corresponding element has a pointer to the next

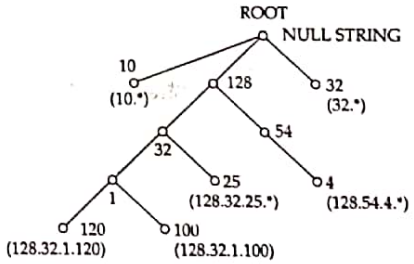

Figure 8.10: A trie. The trie data structure allows efficient search for the best match of a given string. Each node represents a partial string (and an associated output interface) and each child represents one possible single-letter extension. The leaves represent the strings stored in the trie. In this example, we show some of the strings stored by this trie, such as 128.32.1.120 and 128.54.4.*. For an incoming packet with a destination 128.32.1.2, the best match is the node 128.32.1.*, and the router forwards the packet to the destination interface associated with that node.

Scanned by CamScanner

level, we follow the pointer down. If the corresponding element has the special symbol, we know the correct interface from its annotation. If the element has a null pointer, we use the interface associated with the parent level. In this case, we follow the trie using the pointer for 128 at the top level, to 32 at the next level, and to 1 at the third level. The pointer to 2 is null, so the route is the one associated with 128.32.1.*. This algorithm results in determining the best possible match for a string with four elements with at most four matching operations. Thus, a trie efficiently solves the best-match problem.

We can improve a trie's performance in two ways. First, using the principle of locality (see Section 6.3.4), we expect that if we see a packet for a particular destination, we are likely to see packets for the same destination again soon. Thus, we store the translation from a destination to its output port for recently seen destinations in a cache, checking the cache before traversing the trie. The second trick is to optimize for the common case. It is likely that certain addresses will be looked up more often than others (for example, addresses corresponding to local destinations). We can make these addresses part of the first level of the trie, trying them before traversing the trie. This is essentially a permanent cache based on previous knowledge of the system. Depending on the workload, these two tricks can dramatically improve trie performance.

8.3.1 Blocking in packet switches

In a packet switch, as with a circuit switch, we can have both *internal* blocking and *output* blocking. A blocked packet is either dropped or buffered. In a circuit switch, if two circuits block each other, they interfere with each other once every frame time, and therefore the switch must reject one of them. In contrast, in a packet switch, even if packets a and b on connections A and B block each other, subsequent packets on A and B may not block each other. Thus, in a circuit switch, either a path from an input to an output is available at call-setup time and therefore is always available, or a path is not available at call-setup time and the call is rejected. In contrast, in a packet switch, it is nearly impossible to predict whether individual packets on a connection will be blocked. To minimize blocking loss, we can use one or more of the following techniques:

- *Overprovisioning:* By making internal links faster than inputs, the switch fabric can forward packets from more than one input in the time it takes for a single packet to arrive. Thus, for example, if the internal links are twice as fast as input links, when two packets collide, *both* contending packets can be carried, one after the other, in the switch fabric, before subsequent arrivals on those inputs.
- *Buffers:* By placing buffers at the input or in the switch fabric, we can delay packets that do not enter the switch fabric until a link is available. Of course, persistent blocking, due to overcommitment of the switch fabric, leads to packet loss. The switch controller must ensure, at the time of call admission, that the switch fabric can handle all admitted calls.

- *Backpressure:* Backpressure allows a stage in a multistage switching fabric to prevent the previous stage from sending it any more data. If an output is blocked, backpressure signals quickly propagate back to the input, forcing packet buffering and packet loss to happen only at the inputs. Thus, with backpressure, the switch fabric needs only a minimal amount of buffering.
- *Sorting and randomization:* As we will see in Section 8.4, for certain types of switch fabrics, preceding the switch fabric by a sorting or randomization stage reduces internal blocking in the fabric.
- *Parallel fabrics:* By connecting input ports to output ports with multiple parallel switching fabrics, packets can be carried from inputs to outputs in parallel. This reduces contention for the fabric, but requires faster access to output buffers. This is the spatial analog of running the fabric faster.

8.3.2 Three generations of packet switches

It is convenient to categorize packet switches into three types, which make different trade-offs between cost and performance. These are also called three switching generations, though they reflect an evolution in switching capacity rather than in technology. Thus, a first-generation switch might use more sophisticated technology than a second-generation switch. Given the same technology, however, a second-generation switch achieves a higher capacity (but only at a higher cost). Table 8.2 summarizes the characteristics of switches from the three generations.

Generation	Description	Bottleneck	Example
1	CPU transfers packets among adaptor cards	I/O bus, CPU, main memory, or host adaptor	Low-end Internet routers
2	Controllers on adaptor cards transfer packets among each other over a shared bus	Shared bus	High-end routers, low-end ATM switches
3	Switch fabric allows parallel transfers over multiple buses	Output queue controller or input-queue arbiter	High-end ATM switches

Table 8.2: The three generations of packet switches.

Scanned by CamScanner

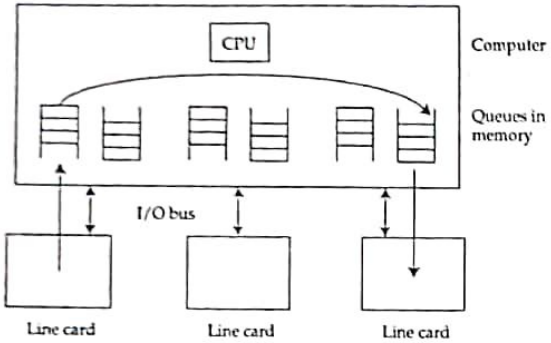

Figure 8.11: First-generation packet switch. The CPU stores packets received on line cards in its main memory, and routes them to output queues. These, in turn, contend for access to an output line card and are scheduled by a software packet scheduler.

8.3.3 First-generation switches

A first-generation packet switch is just a computer with multiple line cards (Figure 8.11). The processor periodically polls each input line card or is interrupted on packet arrival. When a packet is available, the processor reads the packet's header, determines the proper output port from a routing table, and deposits it in the appropriate output buffer. The processor also arbitrates access to the output lines. The switch buffers packets in the computer's main memory. Thus, the main cost of this switch is typically for the computer itself, followed by the cost of adaptors. Most Ethernet bridges and low-cost packet routers in the Internet are first-generation switches.

The advantage of a first-generation switch is that it is simple to build. Unfortunately, a first-generation switch usually has lower performance than a second- or third-generation switch built with the same technology (although it also costs less). The bottleneck in such a switch may be the processor, the main memory, the I/O bus, or the host-adaptor card. With recent advances in processor technology, fast processors far outstrip the I/O bus, which is usually the bottleneck. With newer I/O buses, however, in some recent designs, host-adaptors have become the bottleneck. Thus, in a first-generation switch, the bottleneck resource changes with changes in technology. A careful designer can overcome bottleneck resource limitations by employing some of the techniques discussed in Chapter 6.

EXAMPLE 8.6

Consider a first-generation packet switch built with an Intel Pentium 133-MHz processor. Assume that the mean packet size is 500 bytes, and that an interrupt takes

10 microseconds to process. Also assume that the packet-forwarding code takes 200 machine cycles independent of the packet size, and that it takes 50 ns to read or write a 4-byte word from memory.[3] (a) How fast can it switch packets if the processor is involved in data copies? (b) How fast can it switch packets if the processor only copies a 20-byte header, and the data is transferred directly from one host adaptor to another?

Solution: A data copy loop looks something like this:

```
register ← memory (read_ptr)
memory(write_ptr) ← register
read_ptr ←read_ptr + 4
write_ptr ←write_ptr + 4
counter ←counter − 1
if (counter is not 0) jump to top of loop
```

The counter is initialized to the size of the copy and read and write pointers are appropriately initialized. The loop has two memory accesses, three register operations, a test, and a branch. We assume that the register operations and test take one machine cycle each. Since the processor is a 133-MHz processor, each cycle is 7.52 ns, and so these four operations take 30.08 ns. Added to the 100 ns for the two memory accesses, the time to copy a 4-byte word is 130.08 ns.

(a) The 500-byte packet corresponds to 125 4-byte words. If the entire packet is copied, the copy takes $125 * 130.08$ ns $= 16.26\ \mu s$. The packet-forwarding code takes $200 * 7.52 = 1.504\ \mu s$. Thus, the total processing time is $17.764\ \mu s$. Added to the interrupt time, this gives us $27.764\ \mu s$ per 500-byte packet, or an effective data rate of 144.1 Mbps.

(b) If the processor only copies the header, it takes $20/4 * 130.08 = 650.4$ ns to copy the header. Processing the packet takes an addition $200 * 7.52$ ns $= 1.504\ \mu s$. Added to the interrupt time, this gives us a total of $12.15\ \mu s$ per 500-byte packet, or an effective rate of 329.21 Mbps.

These back-of-the-envelope calculations make some strong assumptions. First, we assume that the bottleneck in the system is the processor. If the host adaptors or the I/O bus cannot keep up with the processor, the performance of the switch is dictated by their capacities. Second, we assume that each packet requires an interrupt. In reality, packets arrive as bursts, so a processor pays the interrupt overhead only once for two or three packets. This increases its switching capacity. On the other hand, we assume that the processor is doing no other work. Taking into account the overhead for doing route-table computation and other processing, the rate should be lower than what we computed. With any luck, this overhead cancels

[3] We are ignoring cache effects, which can reduce the costs of accessing memory words, depending on complex parameters such as the cache line size, the read-ahead policy, and the overall system workload.

Scanned by CamScanner

the gain from amortizing interrupts over many packets, so that the calculations are likely to be at least approximately correct. This sort of back-of-the-envelope calculation, with a "fudge factor" to account for unknown effects, is how systems are built in real life.

8.3.4 Second-generation switches

In a second-generation switch, line cards have some intelligence, so that they are able to decide a packet's output port without help from a central processor (Figure 8.12). In other words, the port mapping functionality is distributed among the line cards. When a packet arrives at a line card, it matches the destination address (or VCI) with entries in a routing cache. In an ATM switch, the cache is guaranteed to contain an entry for the VCI, because it is put there by the switch controller at the time of call establishment. If a route is found, the card sends the packet over a bus to an output port. If no route is found, the card sends the packet to a control computer, which forwards the packet and installs a route in the line card. Note that if a packet is sent to a control computer (which might happen in a router, but not in an ATM switch), packets can be reordered, because packets forwarded on the fast path may be sent before packets that arrived earlier, but were forced to go through the slower path. High-end packet switches, such as the 7500 series of routers from Cisco Systems, are examples of second-generation switches. Another example is FORE Systems' ASX-200 series ATM switch.

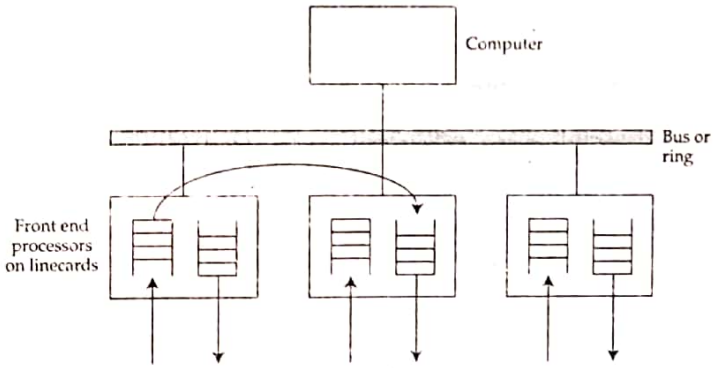

Figure 8.12: Second-generation switch. In a second-generation switch, the routing functions migrate to the front-end processors, which also provide input and output queues. The processors contend for access to a shared bus or ring, which is therefore the bottleneck in the system. If routes cannot be found, packets are handed to the control computer, which routes packets and installs a new route in the front-end processor.

An interesting variation of the slow-path/fast-path approach is used in switches from Ipsilon Networks Inc., which introduced the concept of *IP switching*, also called *cell-switched routing* [NL 96]. Given an underlying ATM network that carries IP traffic, the idea is to build an IP router by adding an adjunct processor to an ATM switch. Normally, IP datagrams arriving to the router as cells on an ATM virtual circuit are reassembled into datagrams and routed by the adjunct processor as in a first-generation datagram switch. When the adjunct detects a sequence of packets to the same destination interface on a given virtual circuit, it requests the previous router along the path (the upstream router) to send these packets on a particular VCI. If the upstream router complies, the router installs a route to that destination interface for that virtual circuit in the ATM switch. Subsequent cells arriving to the router on that virtual circuit are switched as ATM cells without intermediate reassembly. One way to view this is that packets on the slow path that place a heavy demand on switching resources are allocated a VCI in the fast path, so that subsequent packets need not use the slow path. In this sense, the incoming stream of packets constitutes implicit ATM signaling.

The line cards in a second-generation switch communicate with each other using a shared bus or a ring. The bus (or ring, as in the *Paris* switch [GCM 87]) is usually the bottleneck in the system, since it carries data from all the ports. If each input line card has bandwidth b, and there are N line cards, then the bus must have a bandwidth of Nb. Second-generation switches with 10-Gbps aggregate capacity are commercially available today.

If the network is connection oriented, we can reduce a switch's cost by placing port-mapping logic on a shared port-mapper card, instead of on every line card. At the time of call setup, the switch controller updates a routing table in the port mapper. Line cards forward all incoming packets to the port mapper, which examines the header, reads the routing table, and forwards the packet to its output line card. Since every call must be set up, every incoming packet is guaranteed an entry in the table, so there are no cache misses, and we can make the port mapper very fast. Detecting routing inconsistencies is also easier, because there is only a single copy of the routing table in the switch (besides the one in the switch controller). In this approach, every packet makes its way to the port mapper and then back to another line card, doubling the contention for the bus. We can avoid this by using two unidirectional buses, as in Datakit switches [Fraser 83].

8.3.5 Third-generation switches

In a third-generation switch, the shared bus of a second-generation switch is replaced by a *switch fabric*, which, in its most general form, is an interconnection of buses and switching elements that provides multiple parallel paths from any input to any output (Figure 8.13). When packets arrive at an input port, a port mapper module or a shared control processor tags them with the destination port ID and hands them to the switch fabric. The switch elements automatically route it to the correct output port, where the packet is queued. Thus, the switching fabric is *self-routing*. Although switch fabrics can carry variable-length packets, we will assume, unless otherwise specified, that all packets are the same size, such as, for example, ATM cells.

Scanned by CamScanner

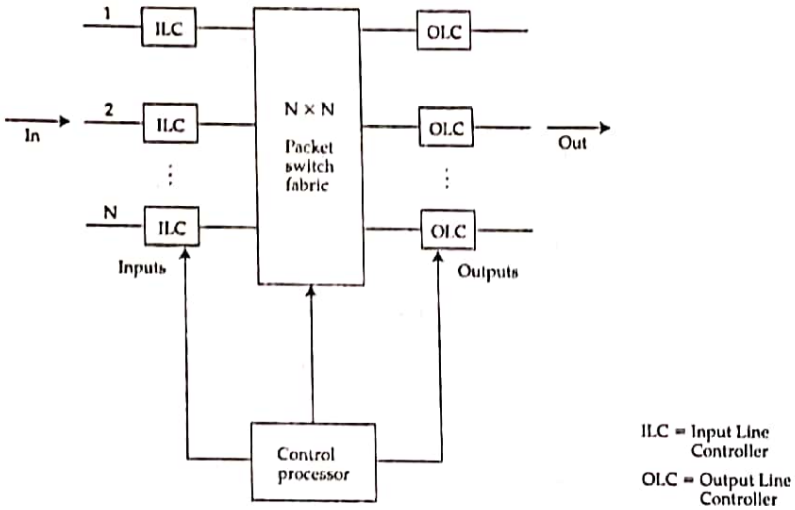

Figure 8.13: Third-generation switch. In a third-generation switch, the shared bus of a second-generation switch is replaced by a switch fabric. Packets arriving at the input ports are simultaneously introduced into the switch fabric, where they are routed in parallel to output ports.

The position of the bottleneck in a third-generation switch depends on whether it has buffers mainly at the input, or mainly at the output. If buffers are at the output, packets arriving at multiple inputs must be loaded into the output buffers rapidly to prevent packet loss. Thus, the bottleneck in the switch is the speed at which we can write packets into the output buffer. If packets are mainly at the input, the bottleneck is the speed at which an arbiter can decide which inputs should be allowed access to the switch fabric and the output line. We will study the effect of these bottlenecks on performance in more detail in Section 8.5. We now turn our attention to switch fabric designs, primarily for third-generation switches.

8.4 Switch fabrics

A switch fabric is a set of links and switching elements that transfers a packet from any input port to any output port. Many switch fabric designs have been proposed in the literature, of which we will study only a few important representatives. Variations and extensions of these designs can be found in some excellent surveys and the references therein [AD 89, DLV 89, Jacob 90, OSMKM 90, Tobagi 90]. Table 8.3 summarizes the fabrics studied in this section.

Name	Subsection
Crossbar	8.4.1
Broadcast	8.4.2
Switching element	8.4.3
Banyan	8.4.4
Sorting and merging fabrics	8.4.5
Batcher–Banyan	8.4.6

Table 8.3: Selected switch fabrics.

Why are third-generation switches rare?

Though many proposals for complicated switch fabrics have been studied on paper, few have been implemented, and fewer yet used in commercial products. There are two reasons why this happened. First, building a switch is much harder than designing it on paper. Many academic designs ignore important practical considerations such as switch cost, robustness to hardware defects and failures, need for clock synchronization, and manufacturability. These designs are doomed to failure right from the start. Even the designs that are practical and implementable have not been commercially successful because the switching capacity of second-generation switches has proven sufficient for most purposes. Only in the mid-1990s, as the demand for network bandwidth skyrocketed, have manufacturers turned to third-generation switch fabric designs. Thus, we are likely to see more switches using these designs in the years to come. In any case, though much research effort has gone into switch fabric design, many researchers now believe that the hard part in designing a switch is not the switch fabric, but deciding where to place buffers and how to schedule access to the buffers and bandwidth. We study these problems in Chapter 9.

8.4.1 Crossbar

The simplest switch fabric is a crossbar (Figure 8.5). An $N \times N$ crossbar has N input buses, N output buses, and N^2 crosspoints, which are either on or off. If the (i, j) crosspoint is on, then the ith input is connected to the jth output. A crossbar is internally nonblocking.

A crossbar needs a *switching schedule* that tells it which inputs to connect to which outputs at a given time. If packets arrive at fixed intervals, for example, if all the connections are constant bit rate, then the schedule can be computed in advance. Otherwise, the switch has to compute the schedule on-the-fly.

If a switch has N inputs and outputs, a perfectly used crossbar is N times faster than a bus-based (second-generation) switch. However, if two packets at the inputs want to

Scanned by CamScanner

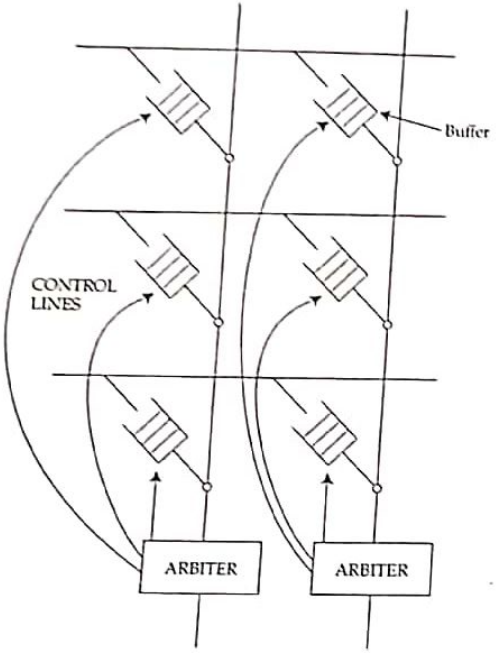

Figure 8.14: A buffered crossbar. In this variation of a crossbar, each crosspoint is associated with a buffer. Incoming packets are buffered at the crosspoint, and a per-output arbiter schedules a packet from one of the crosspoint buffers for transmission on the output trunk.

go to the same output, then they suffer from output blocking and the crossbar is not fully used. We can avoid output blocking by either running the crossbar N times faster than the inputs (which is hard to do, and expensive), or placing buffers inside the crossbar. One interesting variation is to put a buffer at every crosspoint (Figure 8.14) [Marcus 69]. An arbiter per output line then decides which of the buffered packets to serve. This is used in the Scorpio 10-Gbps ATM switch [Scorpio 95].

As with circuit-switched crossbars, packet-switched crossbars do not scale well. They are suitable for small switches, or as building blocks for larger switches.

8.4.2 Broadcast

Another simple switching fabric is based on broadcast (Figure 8.15). The switch tags a packet arriving at any input with the output port number and broadcasts it to all outputs.

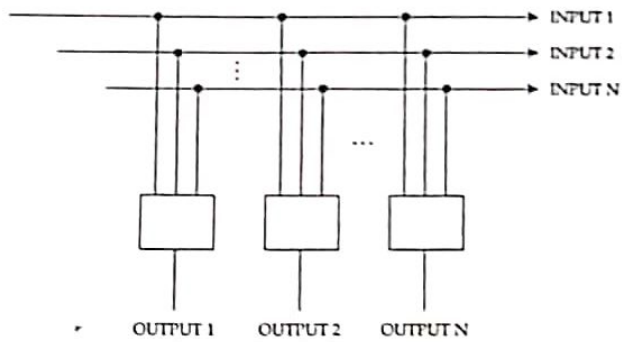

Figure 8.15: A broadcast switch. Each input is broadcast on a bus. Line cards at output ports load packets that match their port address. This configuration makes it easy to implement multicast in hardware.

Each output port stores incoming packets that match its own port address in an output queue. A scheduler resolves contention for the output trunk. In contrast to a crossbar, a broadcast switch does not have to be scheduled for each packet transmission. This comes at the expense of address matching logic at each output.

Since the switch broadcasts all packets to all output ports, a small change makes it easy to implement hardware multicast. Instead of matching port addresses, each output port matches a packet's VCI. At the time of call setup, each port is informed of the VCIs it must match, and at the time of data transfer, each port matches a packet's VCI with its list of eligible VCIs.

The problem with a broadcast switch, as with a crossbar, is that it does not scale. In an $N \times N$ switch, each output must be connected to N buses, and it needs to do an address match on packets arriving at all the inputs. For a thousand-port switch, each output needs to be connected to a thousand inputs, which becomes a physical routing problem. Moreover, address matching must be done N times faster than the speed of an input line, which becomes hard at high speeds and for large N. For example, typical input line speeds are 155-Mbps, which corresponds to 2.73 microseconds per ATM cell. For a 1000×1000 switch, the output port must do one address match every 2.73 nanoseconds! However, we can cheaply build the equivalent of an 8×8 crossbar using broadcast, then use the techniques described next to compose these 8×8 stages to build larger switches.

8.4.3 Switch fabric element

We can build a variety of complex switch fabrics using a basic fabric element consisting of two inputs, two outputs, and an optional buffer (Figure 8.16). Packets arrive simultaneously at both inputs. The element examines the packet header and uses a switching

Scanned by CamScanner

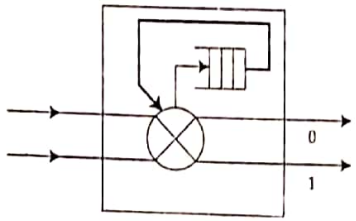

Figure 8.16: A switching-fabric element. This element is repeated many times in a switch fabric. It has two inputs and two outputs. Incoming packets are either sent to their output of choice (usually determined from a bit in the routing tag) or stored in a buffer. Stored packets contend with incoming packets at subsequent times.

rule to direct incoming packets to one or both outputs. For example, if the packet has a 1-bit header, the rule might be: forward on the upper output if the bit is 0, and on the lower output if the bit is 1. If both packets at the inputs want to go to the same output, then one of them is forwarded, and the other is either buffered (if there is space in the buffer) or dropped. We can generalize an element to N inputs and outputs, where the distribution inside an element uses a broadcast or a crossbar switch.

Switching fabrics based on switching-fabric elements share some attributes:

- For an $N \times N$ switch, if each element switches $b \times b$ lines, then the switch has $\lceil \log_b N \rceil$ stages, with $\lceil N/b \rceil$ elements per stage. For example, a 4096×4096 switch built with 8×8 blocks has four stages, with 512 elements in each stage.
- Once a packet has been labeled with the correct output port, it will automatically make its way from any input to its correct output. This is called *self-routing*.
- The fabric is recursively composed from smaller components that resemble the larger network. In other words, the topology of a fabric with bN elements is similar to that of a fabric with N elements and is built from b fabrics with N elements each.
- The fabric can be synchronous or asynchronous. In a synchronous fabric, all elements forward packets simultaneously. In an asynchronous fabric, elements forward packets as they arrive. An asynchronous fabric must have per-element buffering, but it permits variable-length packets.
- The fabrics are regular and suitable for VLSI implementation.

8.4.4 Banyan

The simplest self-routing switch fabric is the *Banyan* (Figure 8.17) [GL 73]. In a Banyan switching fabric, incoming packets are tagged with the output port expressed in binary.

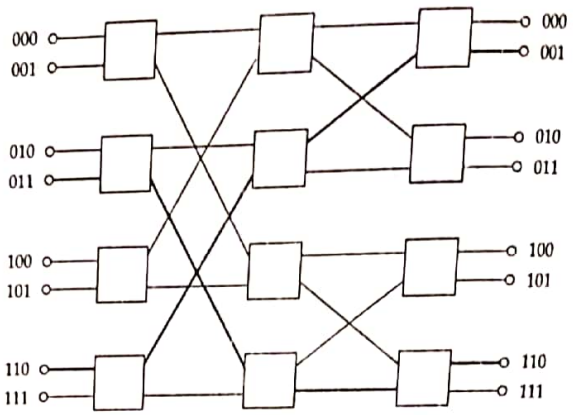

Figure 8.17: A Banyan switch fabric. A Banyan switch fabric consists of switch elements that route packets either to port 0 (their upper output) or port 1 (their lower output), depending on a specific position in the routing tag. In the figure, we see the route followed by two packets with tags 110 and 001 as they enter from ports 011 and 110 respectively. Verify that the packets would be routed to the proper output irrespective of their input port.

Each switching element at the ith layer looks at the ith bit in the header to make its decision: if the bit is 1, the packet is forwarded to the lower output, and otherwise to the upper output. A Banyan fabric with 2^n outputs needs n stages, with 2^{n-1} elements in each stage. Thus, the total number of elements is $n2^{n-1}$.

Let us see how a Banyan switch works. Number the input and output ports in binary, starting from 0, so that the uppermost input port and the uppermost output port are labeled 0. We would like a packet with a binary label to reach the corresponding output port. Note that a single switching fabric element correctly routes packets based on a single bit of their routing tag. In other words, it is a 2×2 switch. We now take two 2×2 switches and label them 0 and 1. We take two other 2×2 switches and place them before the ones labeled 0 and 1. When a packet enters a switching element in the first stage, it is sent to the 2×2 switch labeled 0 if the first bit in its routing tag is 0; otherwise, it is sent to the switch labeled 1. This combination forwards packets labeled 00, 01, 10, or 11 correctly and is a 4×4 Banyan network. We can extend this to the third stage, by labeling two 4×4 switches as 0 and 1, and preceding them with switching elements that forward packets to one of them based on the first bit of the routing tag. Note that this gives us several equivalent 8×8 Banyans, only one of which is shown in Figure 8.17.

If two incoming packets both want to go to the same output port, they collide and *block* at some internal switching element. For example, in Figure 8.17, if the top two inputs have packets destined to outputs 010 and 011, they collide at the first stage. To prevent packet loss, we can introduce packet buffers in each switching element [DJ 81, TW 83, Turner 85, Turner 88]. Thus, blocking results only in packet delay, rather than

Scanned by CamScanner

packet loss. If the same switching element repeatedly blocks, because of the packet arrival pattern (that is, it is a *hot spot*), then packet loss will eventually occur. If we want to avoid packet loss completely, the buffer size must be large enough to hold the worst-case input traffic pattern. This is a problem in traffic management, which we will discuss in Chapter 14.

A buffered banyan switch has two significant problems. First, introducing buffers in each switching element makes each element expensive. Since the elements are repeated hundreds or thousands of times, this increases the overall cost of the switch. Second, if the switch is to have zero loss, it can be shown that the utilization of a buffered switch, even with uniform random traffic, is limited to 45% when each switch element has a single buffer [Jenq 83]. Though this increases to 60% with a buffer size of 4, it is still low.

Instead of installing buffers in each switching element, an alternate proposal is to *check* whether a path is available before launching packets from the inputs [HA 87]. In this three-phase scheme, input ports send requests to the outputs during the first phase. Some requests are lost to blocking, and in the second phase, the output ports inform the winners that they can now send packets. These are then sent in the third phase. The three-phase scheme thus avoids packet loss, but require buffers at the inputs. As we will see in Section 8.5.1, if these input buffers are organized as FIFOs, then the throughput of the switch is restricted to 58%. A small variation reduces the number of phases. In this scheme, input ports send the entire packet in the first phase, but also store a copy. In the second phase, the output ports inform the winners, who then purge their copies. The losers try again in the next round.

A third way to improve Banyan performance is to use several Banyans in tandem [TKC 91]. In this design, when two packets collide in a switching element, one of them is misrouted and marked with a tag. The switch diverts untagged packets to the output port at the end of the first stage, and reintroduces misrouted packets into a second Banyan stage, where they have a second chance to reach a given destination. This process continues for several stages, until, at the end of the last stage, misrouted packets are dropped. It can be shown that for a 32×32 switch, with uniform random traffic, a tandem of nine Banyans is sufficient to reduce the packet drop probability to 10^{-9}. Of course, this performance comes at a price. First, since the switch uses several Banyans instead of just one, the hardware is much more expensive. Second, the switch can reorder packets. Finally, since outputs have to be prepared to accept packets simultaneously from any one of k stages, they have to be run k times faster than the inputs.

Finally, we can avoid the problems associated with blocking in Banyan networks by choosing the order in which packets appear at the input ports. Consider the ordered set of output port numbers of packets waiting for service at the input of a Banyan switch. For example, if input port 001 has a packet destined for output port 011, input port 010 has a packet for 010, input port 011 has no packet awaiting service, and input port 100 has a packet for output port 011, the set is {X, 011, 010, X, 011, X, X, X}, where X marks a gap. It can be shown that if this set is sorted, with gaps and duplicates removed, then a Banyan network preceded by a *perfect-shuffle* network is internally nonblocking

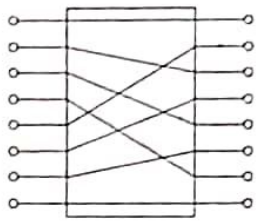

Figure 8.18: A shuffle-exchange network. If sorted inputs pass through a shuffle network and enter a Banyan network, the Banyan network suffers no collisions.

(so switch elements do not need buffers) [Lee 88] (Figure 8.18). Continuing with our example, we first sort the inputs to be {010, 011, 011, X, X, X, X, X}, then remove duplicates to form {010, 011, X, X, X, X, X, X}, then shuffle them using the network in Figure 8.18, to form {010, X, 011, X, X, X, X, X}. When this is presented to the Banyan network, it is nonblocking (which can be verified from Figure 8.17). Operationally, we achieve this by placing a *sorting*, *trap*, and *shuffle* network before a Banyan fabric. The sort network sorts the inputs in order of their output ports, the trap network removes duplicates, and the shuffle network implements a perfect shuffle. Since these networks are typically cheaper than having per-element buffering, this reduces the overall cost of the switch fabric.

8.4.5 Sorting and merging networks

Sorting networks can be built from *merging* networks, that is, networks that take two sorted lists and merge them to make a larger sorted list [Batcher 68]. We call such a sorter network a *Batcher network* after its inventor. A Batcher network that sorts N inputs needs $\lceil \log N \rceil \lceil \log N + 1/2 \rceil$ stages.

How does a sorter work? For the moment, assume we know how to merge two sorted lists to form a longer sorted list. Then a strategy to sort by merging is as follows: we divide the list of N elements into pairs, and sort each pair. This gives us $N/2$ lists, each of which is sorted. Since we assume we know how to merge sorted lists, we can combine these to form $N/4$ sorted lists, with four elements each. We repeat to form $N/8$ sorted lists, with eight elements each, and so on. In the end, we get one fully sorted list. All we need, therefore, is a way to sort two elements, and a way to merge sorted lists. The first requirement is easily met using a comparator. The second requires a merging network, which we study next.

Scanned by CamScanner

EXAMPLE 8.7

Sort the list 5, 7, 2, 3, 6, 2, 4, 5 by merging.

Solution: In the first step, we sort elements two-by-two to get four sorted lists, {5, 7}, {2, 3}, {2, 6}, and {4, 5}. In the second step, we merge adjacent lists to get two four-element lists: {2, 3, 5, 7} and {2, 4, 5, 6}. In the third step, we merge the two lists to-create a fully sorted list: {2, 2, 3, 4, 5, 5, 6, 7}.

Merging networks

A merging network of size $2N$ takes two sorted lists of size N as inputs, and creates a merged list of size $2N$ (Figure 8.19). It consists of two N-sized merging networks. One of them merges all the even elements of the two inputs and the other merges all the odd elements. The outputs of the mergers are handed to a set of 2×2 comparators. It can be proved that this structure correctly implements merging [Batcher 68].

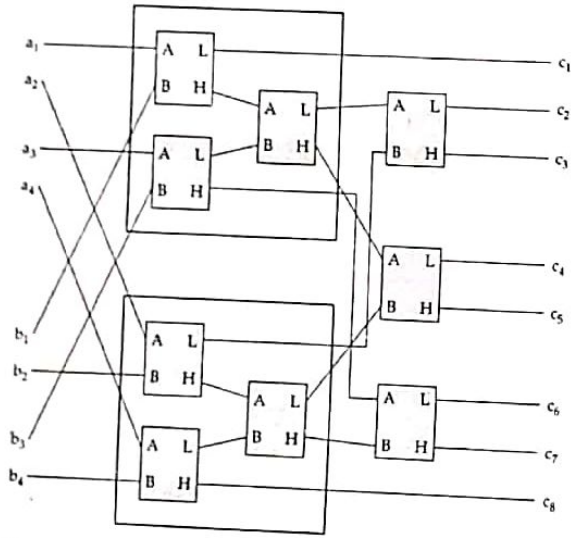

Figure 8.19: A merging network. This network takes two sorted lists and merges them to form a sorted list twice as long. The network is based on the idea that to merge two sorted lists, we need to merge their even and odd elements separately, then combine the outputs with a set of carefully placed comparators.

EXAMPLE 8.8

Merge the two lists {2, 3, 4, 7} and {2, 4, 5, 6}.

Solution: In the first stage, we merge even elements from the two lists, i.e., merge {2, 4} with {2, 5}. Recursing, we need to merge {2} with {2} and {4} with {5}, then compare them. The results of the two merges are {2, 2} and {4, 5}. Comparing the higher element of the first list with the lower element of the second list, we determine that the merged list is {2, 2, 4, 5}. We next merge {3, 7} with {4, 6}. This requires merging {3, 4} and {7, 6}. The results are {3, 4} and {6, 7}. Comparing the high and low elements, respectively, we get a merged list {3, 4, 6, 7}. Carrying out the comparisons shown in Figure 8.19, we get the final merged list {2, 2, 3, 4, 4, 5, 6, 7}.

8.4.6 Batcher–Banyan

A Batcher–Banyan network is composed from a Batcher network that sorts packets according to their output port numbers, followed by a trap network to remove duplicates, a shuffle network, and a Banyan network (Figure 8.20) [HK 84]. Since all the inputs are sorted, the Banyan is internally nonblocking. However, we still have to deal with trapped duplicates. One solution is to recirculate duplicates back to the entrance of the Batcher, so that in the next cycle they can compete with incoming packets. This is the technique used in the Starlite switch [HK 84]. The problem with this approach is that it requires at least half the Batcher's inputs to be devoted to recirculated packets. Moreover,

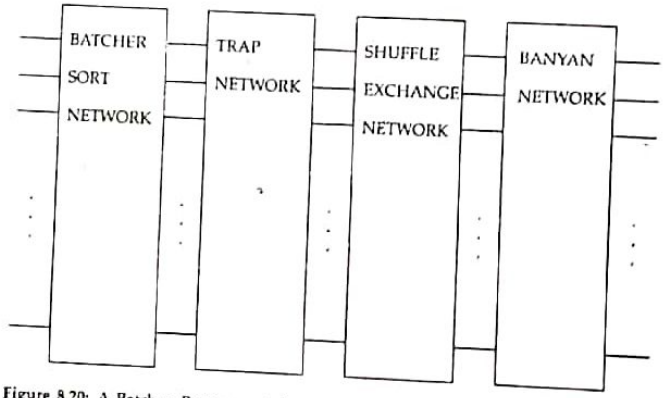

Figure 8.20: A Batcher–Banyan switch. By sorting the inputs to a Banyan switch, we avoid internal blocking. Thus, we precede the Banyan stage by a Batcher sorting stage, then use a trap network and a shuffle-exchange network to remove duplicates and gaps.

Scanned by CamScanner

the recirculated packets have to be synchronized with fresh arrivals, which is hard to do at high clock speeds [CT 92]. An alternative is to take the output of the Batcher and pass them through multiple Banyan networks in parallel, where each packet that wants to go to the same output is put on a separate Banyan (this is also called *dilation*). This way, we remove the internal blocking in the Banyan at the expense of faster output buffers. This technique is used in the Sunshine switch [GHMSL 91].

Packet size and switch performance

One motivation for small, fixed-size cells in ATM is the claim that cells make it easier to build packet switches. Is this really true? It seems that although smaller packets make it easier to provide real-time delay guarantees in large switches, a small packet size may not help in building switches. Fixed-size packets may be similarly overrated. Let us see why.

In a first-generation switch, packet size is more important than whether the packets are fixed size. If the packets are small, then the interrupt overhead is large, so we prefer large packets to small ones. Moreover, fixed-size packets do not particularly improve performance either, because the switch deals with packets individually, independent of their size.

In a second-generation switch, packets contend for bus access, and the longer the packet, the more that can be sent after a single bus contention and setup cycle. Thus, larger packets are again more efficient for second-generation switches. Besides, because packets are handled serially on the bus, fixed-size packets do not help.

In a third-generation switch, having a small packet size neither helps nor hurts, because a self-routing switch fabric is oblivious to packet size. If anything, for an input-queued switch, a small packet size makes it harder to arbitrate among packets. So, even for a third-generation switch, larger packet sizes may be better. Fixed-size packets allow a switch fabric to forward packets in synchrony. However, the switch could easily fragment packets at the entrance to the fabric and reassemble them at the exit. Or, switch elements could have sufficient intelligence to deal with variable-size packets. Or, the fabric may only switch fixed-size headers, allowing the output port to transmit data from a shared-memory buffer. It is not clear that these, or other alternatives, are too expensive.

Thus, variable-size packet switches do not seem infeasible or overly expensive. It seems fair to say that the jury is still out on whether ATM cells are necessary for fast packet switching.

8.5 Buffering

Unlike circuit switches, packet switches need buffers because their inputs are unpredictable. Even if sources agree to regulate their behavior, they can still transmit bursts at any

time. Moreover, contention for a shared output line means that even smooth inputs entering a packet network may become bursty. These bursts must then be forwarded through the switch fabric (or the shared bus) and the output trunk. If either is busy, packets from the bursts must either be buffered or dropped. Thus, to reduce packet loss, packet switches need buffers to hold packets as they wait for access to the switch fabric or the output trunk. We can place these buffers at the input port, at the output port, in the switch fabric, or in a common location shared by the input and output ports. Various combinations of these schemes are possible. In this section, we focus on the pros and cons of each choice. Note that though our discussion assumes that the switch is a third-generation switch, it applies equally to second-generation switches, where a single bus or ring replaces the switch fabric.

8.5.1 Input queuing

In a purely input-queued switch, packets are buffered at the input and released when they win access to both the switch fabric and the output trunk (Figure 8.21). An arbiter schedules access to the fabric depending on the status of the fabric and the output lines. Since packets leaving the input queue are guaranteed access to the fabric and the trunk, there is no need for an output queue.

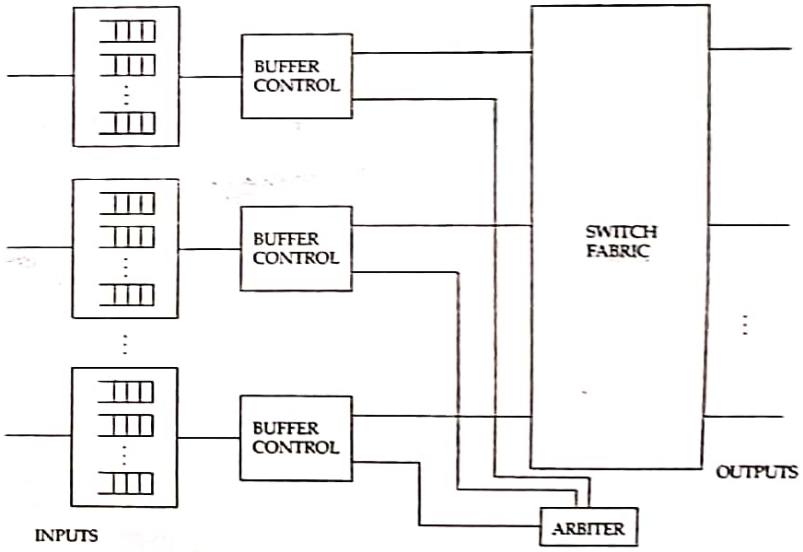

Figure 8.21: An input-queued switch. In a purely input-queued switch, all the buffering is done at the input ports. An arbiter releases packets from the input queue when a path through the switch fabric and the output trunk is available.

Scanned by CamScanner

The main advantage of an input-queued switch is that links in the switch fabric (and the input queues themselves) need to run only as fast as the input lines. For an $N \times N$ switch, the only element that needs to run N times faster than the inputs is the arbiter. We can, however, easily build fast arbiters, since they typically require only combinational logic. As we will see, with output-queued switches, the output queues need to run faster than the inputs. Thus, for a given level of technology, we can build a faster input-queued switch than an output-queued switch.

The problem with an input-queued switch is that if the input queues are served in FIFO order, and the packet at the head of the queue is blocked, it blocks others in the queue, though they might have an open path to their output. We call this *head-of-line* blocking. If packets at each input arrive randomly distributed to all outputs (uniform random traffic), then it can be shown that an input-queued switch can achieve at most ~58.6% utilization [KHM 87, HK 88]. The performance is even worse if some outputs are favored over others.

EXAMPLE 8.9

Consider a 2×2 switch serving virtual circuits A, B, and C. Cells from A and B arrive at input 1, and cells from C arrive at input 2. Suppose A and C want to go to output port 1, and B wants to go to output port 2. The accompanying table shows a particular arrival pattern.

Time	Input	Virtual circuit	Sequence number	Destined for port
0	1 2	A C	1 1	1 1
1	1 2	B C	1 2	2 1

At time 0, the arbiter must choose a cell from either A or C for output port 1. Suppose the arbiter chooses the cell from C. Then, at time 1, the input queue at port 1 has cells from A and B awaiting service, and input port 2 has a cell from C awaiting service. If the arbiter decides to choose the cell from C again, the cell from B suffers from head-of-line blocking, because it has a path to 2, but cannot use it.

We can overcome head-of-line blocking by separating the queues for each output port at each input, and selecting one packet from each queue for each output port. More precisely, let the switch have N ports. Then, at each input, we maintain N queues, one per output. At each time step, the arbiter must select at most one packet from each input port so that, of all the packets selected from all the inputs, no two packets are destined to the

same output port. The problem is hard because of the interdependencies when making this selection (this is explained at greater depth in Example 8.10).

An interesting algorithm for making this selection is a three-phase algorithm, similar to the one described in Section 8.4.4 in the context of Batcher–Banyan switch fabrics, called *Parallel Iterated Matching* [AOST 93]. In the first phase, each input port requests permission to transmit a packet from the set of destinations corresponding to packets queued at that input. The second phase (consisting of several *matching rounds*) selects a set of input ports for packet transmission in the third phase. In each matching round, an output grants permission to a randomly selected member of its set of requesting inputs. This may result in some input port getting more than one grant. In this case, the input randomly selects an output and withdraws its requests from the other output ports. The losing input ports and the losing output ports then repeat this process. It has been shown that three or more rounds of matching are usually adequate to find the best possible match, that is, a match that selects the most packets. The DEC Autonet2 switch carries out four rounds of matching in hardware before each packet transmission [AOST 93].

EXAMPLE 8.10

Consider a 4×4 switch that has input queues as shown in the table.

Input port #	Packets for output port #	Output port #	Requests from input port #
1	1, 2, 3	1	1, 2, 3
2	1, 3, 4	2	1, 4
3	1, 3	3	1, 2, 3
4	2, 4	4	2, 4

In the first round, input 1 sends requests to output ports 1, 2, and 3, input port 2 sends requests to output ports 1, 3, and 4, and so on. Each output port grants requests randomly. Suppose output ports 1, 2, and 3 all pick 1, and output port 4 picks 2. Now, input port 1 has three grants, so it may pick output port 1. Thus, output ports 2 and 3 and input ports 3 and 4 lose. In the second round, input port 3 sends a request to output port 3 (since 1 is already taken), and input port 4 sends a request to output port 2 (since 4 is already taken). These are automatically granted. Thus, in two rounds, the parallel iterated match has found the best possible match. The scheme does not guarantee fairness, since the same input may win repeatedly. This is unlikely, because of the randomness in the selection, but still possible.

Note that the arbiter, by choosing a packet for transmission, effectively allocates each connection with bandwidths and delays. It would, therefore, be useful to have the arbiter emulate a discipline such as weighted fair queuing, described in more detail in Section 9.4.4. A variation of the parallel iterated match scheme, called *fair arbitrated round robin*, can achieve nearly the

Scanned by CamScanner

same performance as a multipriority weighted fair queuing output-queue scheduler [LPR 96]. Other schemes for iterated matching are also described in reference [McKeown 95].

Arbitration schemes for input-queued switches are still on the research frontier. Since the arbiter is the bottleneck component in an input-queued switch, coming up with efficient and fair arbitration algorithms is an interesting open challenge [MAW 96].

8.5.2 Output queuing

A pure output queued switch buffers data only at its outputs (Figure 8.22). It serves packets from the output queue using one of the scheduling disciplines described in Chapter 9. Since output queues do not suffer from head-of-line blocking, we prefer output-queued switches to input-queued switches. However, in the worst case, packets at all the input ports may be destined to the same output port. If the switch has no input buffers and we want to avoid packet loss, the switch fabric must deliver N packets to a single output, and that output queue must store N packets in the time it takes for one packet to arrive at an input. This makes the switch fabric and queues more expensive than with input-queued switching. Given the same technology, building an input-queued switch is cheaper than building an output-queued switch. Alternatively, for the same cost, we can build a faster input-queued switch than we can an output-queued switch.

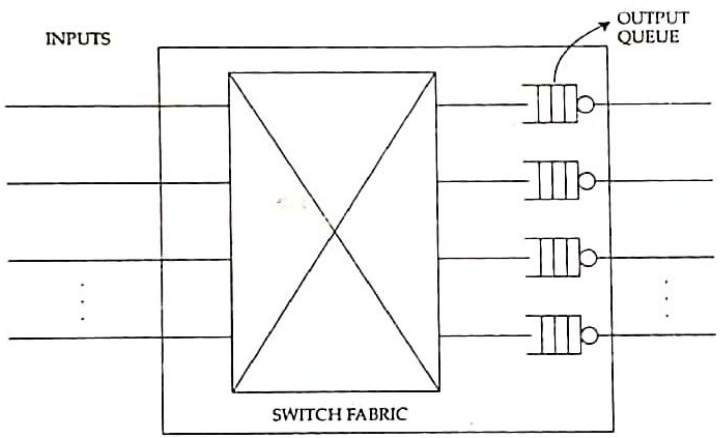

Figure 8.22: An output-queued switch. In a purely output-queued switch, buffers are found only at the output. The output buffers and the switch fabric have to run N times faster than the input trunks in order to avoid packet loss. This makes output-queued switches expensive. However, they allow fine-grained control on output scheduling.

EXAMPLE 8.11

Consider a 2×2 switch that has virtual circuits A and B arriving at input ports 1 and 2, both destined to output port 1. Suppose that cells from both A and B arrive at each time interval, and that the switch has no input queues. If the switch fabric does not serve the cells arriving at time 0 before the second set of cells arrive, then some cells will be lost. Similarly, the output queue at port 1 must be fast enough to store two cells in one cell arrival time. Otherwise, one of the two cells will be lost.

One way to reduce the cost of an output-queued switch is to use the *knockout* principle [YHA 87]. This principle states that if a switch has N inputs, it is very unlikely that *all* N packets will simultaneously arrive at the same output. Therefore, the output queue is run only S times faster, where $S < N$. It can be shown that for a variety of input distributions, $S = 8$ reduces the loss probability to less than 1 in 10^6. The knockout principle has been used in the *Knockout* switch (Figure 8.23) [YHA 87] and the *Gauss* switch [Vries 90]. The difficulty in designing the knockout circuit is to ensure that packet losses are fairly distributed among the incoming virtual circuits.

8.5.3 Shared memory

In a shared-memory switch, input and output ports share a common memory (Figure 8.24). Packets are stored in the common memory as they arrive, and the packet header is extracted and routed to the output port. When the output port scheduler schedules a packet for transmission, it removes the packet from the shared memory. Since the switch fabric only switches headers, it is easier to build. However, an $N \times N$ switch must read and write N packets in one packet arrival time. Because memory bandwidth is usually a highly constrained resource, this restricts the size of the switch.

EXAMPLE 8.12

What should be the read cycle time for a 64×64 shared-memory ATM switch, where each input runs at 155 Mbps and the memory is accessed as 4-byte words? As 53-byte words?

Solution: On a 155-Mbps trunk, a 53-byte ATM cell arrives once every 2.73 microseconds. Thus, the shared memory should be able to read 64 cells in 2.73 microseconds, or one 4-byte word every 3.05 nanoseconds. Typical dynamic RAM access times in 1996 using *Extended Data Out (EDO)* technology are 60 ns for the first word, and ~20 ns for subsequent words in the same memory line. Thus, we cannot build a 64×64 shared-memory switch with 4-byte-word accessed DRAM. Comparable static RAM access times are 2.5 to 50 ns, with the cost increasing very rapidly for

Scanned by CamScanner

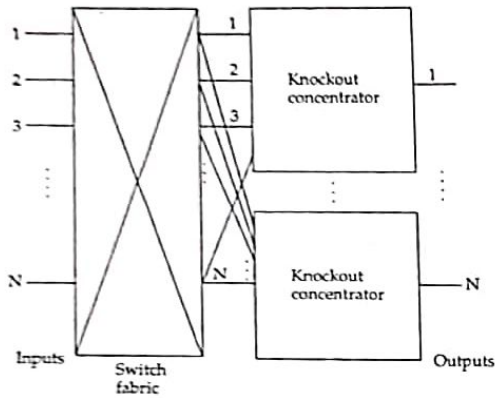

Figure 8.23: A knockout switch. The knockout principle states that an output is likely to receive packets simultaneously from only a few inputs. Thus, the output trunk runs only S times faster than the inputs, where $S < N$. If, however, more than S packets do arrive at the output, the knockout circuit eliminated some of the excess packets, fairly distributing the losses among the inputs.

shorter access times (moreover, with access times shorter than 5 ns or so, the length of the lead also adds delay, since an electromagnetic signal travels at most 30 cm in 1 ns). Therefore, while a 64 × 64 word-accessed shared memory switch is feasible with fast SRAM technology, it would be prohibitively expensive. An easier way to get fast memory access speed without paying for SRAMs is to access memory a cell at a time, instead of a word at a time. If we access 53 8-bit-wide memories simultaneously, we only need to read and write one cell every 42.9 nanoseconds. This is feasible with low-end SRAMs.

An interesting example of a shared memory switch is IDTI Inc.'s *Datapath* switch (Figure 8.25) [Kanakia 94]. The key idea in this switch is to integrate eight input and

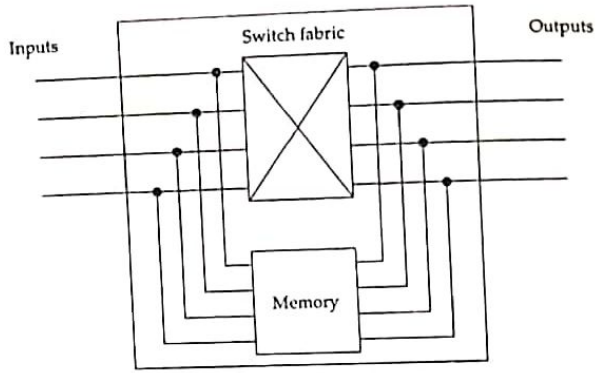

Figure 8.24: A shared memory switch. Incoming packets are stored in the memory and the packet header is routed to the output. When a packet is scheduled for transmission, the output port removes the packet from the shared memory. Note that only the header needs to be routed. Thus, packets can be variable size as long as the header size is fixed.

output port controllers and a shared memory module on a single VLSI chip. Serial inputs are parallelized in a shift register, and the entire shift register, containing an ATM cell, is read into the memory in parallel. This greatly reduces the read/write performance required from the memory (using the figures from Example 8.12, a read cycle for an 8 × 8 switch can take as long as 341.9 ns). A separate fabric-controller chip reads the packet header and assigns the packet to an output port. When the output port scheduler decides to serve a packet, it reads the entire packet in parallel into a shift register; the packet is converted to serial and transmitted. The VLSI packaging makes the chip economical: a 1.2 Gbps switch-on-a-chip costs only around $50. Datapath chips can be interconnected in various ways to form larger, buffered switch fabrics.

8.5.4 Buffered fabric

In a buffered-fabric switch, data buffers are associated with each switching element, with no input or output queues. Incoming packets are stored in the first stage's data buffers and propagate through the switch fabric, buffered in per-element buffers when a path to

Scanned by CamScanner

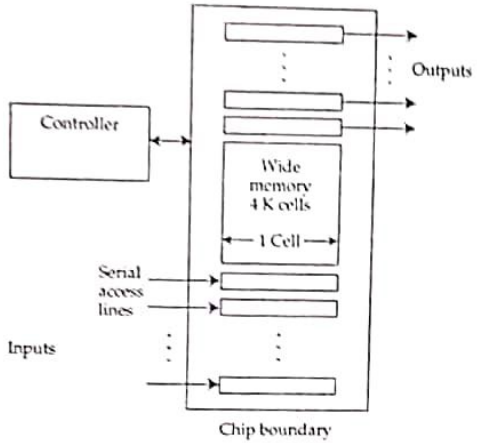

Figure 8.25: Datapath switch. A Datapath switch has eight input shift registers that shift in data from eight input lines and eight output registers that shift data out to eight output lines. An incoming ATM cell is buffered in a shift register, then written into the wide memory in a single write cycle. An adjacent controller decides the order in which cells in the memory are written to one of the output shift registers. Thus, Datapath is an 8×8 switch-on-a-chip.

the next element is unavailable. The output link scheduler schedules access to the output trunk from the last stage's buffers.

The advantage of a buffered-fabric switch is that buffers need to run only as fast as a switching element's fan-in ratio. Moreover, hardware backpressure between switching elements can be used to minimize the probability of buffer overflow. During overload, the resulting queue is distributed among several switching elements, so no queue gets too large. On the other hand, a switch that has many switching elements needs a lot of memory. Moreover, it is nearly impossible to achieve the equivalent of per-output per-VC weighted fair queuing, which is desirable for switches that provide per-connection quality of service guarantees (see Chapter 9 for details). Thus, a purely buffered-fabric switch does not seem practical. It seems more suitable when each switching element itself is a smaller switch, such as a shared-memory switch.

8.5.5 Hybrid schemes

So far, we have studied pure buffering schemes, where buffering is present only at one point in the switch. In reality, these schemes are easily combined. For example, a switch may have a few input buffers to deal with output blocking, with much larger output buffers that are scheduled according to a more sophisticated scheduling discipline. Many commercial switches use a shared bus, with each interface card providing both input and output buffering. The performance of a hybrid scheme depends on the degree to which it models one of the buffering schemes described earlier.

Transmission cost versus buffer cost

Many switch designers try to cut down on the number of switch buffers. Looking at the current (1996) cost of memory and transmission costs, this makes no sense: 80-ns DRAM memory costs about $20 a Mbyte, and long-haul transmission costs about $1000 a mile a year for DS3 trunks. If 10 Mbytes of memory increase link utilization for a 100-mile link by even 1%, the $200 it costs for memory is more than recovered in the $1000 saved per year. Thus, it is always better to have as large a memory as possible, if this results in better link utilization.

8.6 Multicasting

Thus far, we have assumed that at a switch, each incoming packet is destined for a single output. This is the case when the connection is a unicast connection. All connections in the telephone network are unicast. In the Internet and ATM networks, however, some connections are point-to-multipoint, or multipoint-to-multipoint. When a packet on a non-unicast connection arrives at a switch, it must be copied and transmitted to multiple outputs. In this section, we will study some strategies for multicast at the packet level. We will study multicast routing in Chapter 11.

We will assume that when a multicast packet arrives at a switch, the port mapper retrieves a list of output ports associated with that packet instead. The problem, then, is to copy the packet and distribute the copies to several outputs, and, for an ATM network, potentially assigning a different VCI for the packet at each output. We will study two separate problems: generating and distributing the copies, and VCI translation for the copies.

8.6.1 Generating and distributing copies

Packet copies can be generated and distributed either implicitly or explicitly. The implicit approach is suited to second-generation switches, and to broadcast-based or crossbar-based third-generation switches. An incoming multicast packet is placed on a bus and multiple output ports are simultaneously enabled. Thus, a copy of the packet is available at each output port. This approach is used in the *Paris* and *Datapath* switches, among others [GCM 87, Kanakia 94].

Scanned by CamScanner

With a Banyan-like switch fabric, a switch must explicitly create packet copies. This is done by a *copy* network [Turner 85]. The input port attaches a header to a packet containing the number of copies needed, and this modified packet is introduced into the copy network. A copy network is similar to a Banyan network, except that each switching element places a copy of the packet on *both* outputs, decrementing the packet's copy count as it does so. At the exit of the copy network, therefore, several outputs receive copies of the packet. These can then be placed into a standard switching fabric. With both the explicit and the implicit approach, multicasting increases the output blocking probability.

8.6.2 Header translation

With a unicast connection, header translation from an incoming VCI to an output VCI can be done either at the input port or the output port. With a non-unicast connection, this translation is more easily done at the output port. The translation table is identical to a unicast translation table. With this approach, multicasting requires a switch to separate the port mapping and header translation table. The first table is at the input and maps an incoming VCI to a set of output ports. The second table translates a given VCI and input port to an outgoing VCI on the current output port. Thus, a switch must perform at least two table lookups per packet.

8.7 Summary

This chapter describes techniques for building circuit and packet switches (summarized in Table 8.4). We studied techniques for switching samples within a trunk (time-slot

Circuit switching (telephone switches)	Time division (8.2.3)	
	Space division (8.2.4)	
	Combined	Time–space (8.2.5)
		Time–space–time (8.2.6)
Packet switching (routers and ATM switches)	First generation (8.3.3)	
	Second generation (8.3.4)	
	Third generation (8.3.5)	Switch fabrics
		Crossbar (8.4.1)
		Broadcast (8.4.2)
		Banyan (8.4.5)
		Batcher–Banyan (8.4.6)

Table 8.4: Summary of switching techniques.

interchanging) and among trunks (space-division switching). We also saw how to combine time and space stages to build large circuit switches.

Packet switches fall into three generations. The first generation is just a computer with multiple link interfaces, with all the switching done by the processor. These can achieve speeds up to 300 Mbps with current processors. These speeds can be multiplied by a factor of eight or more with second-generation switches, where line cards are connected by a high-speed bus. Third-generation switches give even higher performance by using multiple parallel buses. We studied switching fabrics such as the Banyan interconnection network, and several variations that increase its performance. Finally, we studied the several alternatives for buffer placement: at the input, at the output, shared, or in the switch fabric. Input-queued switches might be best suited for building fast switches, although algorithms to arbitrate contention for an output port are still an active research area.

Review questions

1. What is the primary function of a switch?
2. What does a port mapper do?
3. Why is blocking reduced in transit switches?
4. How many inputs and outputs does a TSI have?
5. How does a TSI switch?
6. What is meant by internal blocking?
7. Does a crossbar suffer from internal blocking?
8. Can a multistage network have internal blocking?
9. Can a crossbar suffer from output blocking?
10. Can a multistage network have output blocking?
11. How many TSIs does a 64×64 TST switch have?
12. Why does a TS fabric have a smaller blocking rate than an S fabric?
13. Why does a TST fabric have a smaller blocking rate than a TS fabric?
14. What is a first-generation packet switch?
15. What is usually the bottleneck in a first-generation switch?
16. What is a trie used for?
17. What is the bottleneck in a second-generation switch?
18. Where is the queuing in a second-generation switch?
19. Why does a third-generation switch provide more bandwidth than a second-generation switch?
20. Can a crossbar used a precomputed schedule with VBR connections?
21. Does a broadcast switch need a switching schedule? Why or why not?
22. Do packets destined for output ports 000 and 001 at input ports 1 and 2 of a Banyan collide?
23. Do packets destined for output ports 001 and 000 at input ports 1 and 7 of a Banyan collide?
24. How does a tandem Banyan fabric work?

Scanned by CamScanner

25. What is head-of-line blocking? What is a simple way to prevent it?
26. What problem does parallel iterated matching solve?
27. What is the fastest component in an input-queued switch?
28. What is the fastest component in an output-queued switch?
29. What is the bottleneck in a shared-memory switch?
30. Where is header translation done in a multicast ATM switch?

Exercises

8.1. Suppose the telephone network had no switches, and every user had a point-to-point link to every other user. How many lines are required for five users? Fifty users?

8.2. Assume that a router reorders packets so that a packet with sequence s may arrive after a packet with seqence $s + 2$ (but no later). If the maximum packet size is 500 bytes, how much buffering is needed at the receiver to guarantee zero loss?

8.3. How long does a 8-bit voice sample last on a T1? On a T3?

8.4. How many bytes of storage does a T3 TSI need?

8.5. (a) How would you use a TSI to build a switch that can handle 1024 individual T1 trunks? (b) How fast must the TSI read samples into its memory? Suppose you built it as a TS switch, with a 16×16 crossbar. (c) How fast do samples need to be read? (d) Describe an algorithm for a controller to find a path for a particular call from an input to an output trunk. (e) Suppose the output trunk is immediately demultiplexed to a set of DSOs. How does a TST switch help over a TS design in this situation?

8.6. Using the numbers from Example 8.5, compute the increase or decrease in packet-switching bandwidth if (a) the interrupt time is reduced from 10 μs to 5 μs, (b) the number of non-data-touching cycles is reduced from 200 to 100, (c) and the mean packet size is 53 bytes instead of 1500 bytes. (d) What conclusions can you draw about the relative effects of these perturbations?

8.7. Does the time taken to match an incoming string depend on the number of elements stored in a trie? What is the smallest and greatest number of operations taken to match an incoming string of length 4 in a trie that stores (a) 10 distinct strings, (b) 1000 distinct strings?

8.8. Suppose the Pentium packet routers in Example 8.5 were attached to a 1-Gbps bus. (a) Describe the data path from an input to an output of this second-generation switch. (b) How many processors can be attached to the bus before it is fully loaded? (c) What would happen if more processors were attached to the bus? (d) If the processors queue data FIFO, what is the data rate achievable assuming uniform random traffic?

8.9. Assume that the port-mapping information is distributed among the processors of a third-generation switch. (a) How is this state set up? (b) What happens if the switch controller reboots? (c) Propose a protocol for reestablishing state on a reboot.

8.10. Suppose you were to use the processors in Example 8.11 as input and output port controllers in a third-generation switch, built as a 64×64 crossbar. What is the best achievable throughput? What is the throughput achievable with uniform random traffic and FIFO input queuing? What is the throughput achievable in the worst case?

8.11. Draw a 16×16 Banyan switching fabric.

8.12. How many equivalent 8×8 Banyan fabrics are there?

8.13. Sort 3, 1, 9, 2, 0, 1, 3, 7 by merging.

8.14. What is the largest number of rounds that parallel iterated matching may take for a 16×16 switch?

8.15. What is the largest purely output-queued ATM switch we can build if we use 80-ns 4-byte-wide DRAM with a zero cell loss requirement? Assume input lines are 155 Mbps. How large can the switch be with cell-wide access? If we decide to run the outputs at only 0.4 N times the input lines (instead of N times faster than the input lines), how big a switch can we build?

Scanned by CamScanner