**Massachusetts Institute of Technology**
**Department of Electrical Engineering and Computer Science**

6.829 Fall 2002

# Lecture 1: Packet Switching

September 5 2002

This lecture discusses different ways of interconnecting links of the same kind to build a simple network. To do this, we will use a device called a switch, and discuss several different ways of switching to move data between networks. We will focus on "packet switching" and investigate how it works.

Before you read these notes, you should be familiar with the material in L0 (single-link communication).

## 1 Interconnections

The rather limited scope—in terms of physical distance, number of connected hosts, and amount of sustainable traffic—of single-link networks leads us to examine ways of interconnecting single-link communication media together to form larger networks of computers. In this part, we will look at a few different interconnection technologies, restricting ourselves (for the time being) to interconnecting homogeneous network links.

The key component used for such interconnections is a *switch*, which is a specialized computer running software that allows it to receive frames (of bits) (or *packets*) that arrive over links, process them, and forward them over one or more other links. Links are physically connected to switches at *attachment points* or *switch ports* (often simply called *ports*, especially when the context is unambiguous). And networks built out of such components are often called "receive-and-forward" networks.[1]

The fundamental functions performed by switches involve multiplexing and demultiplexing of data frames belonging to different computer-to-computer information transfer sessions (or "conversations"), since a given physical link will usually be shared by several concurrent sessions between different computers.

Over time, two radically different techniques have developed for doing this. The first, used by networks like the telephone network is called *circuit switching*, while the second, used by networks like the Internet, is called *packet switching*. The key difference between the two is that in circuit-switched networks, the frames do not need to carry any special information that tells the switches how to forward information, while in packet-switched networks, they do.

The transmission of information in circuit-switched networks usually occurs in two phases: first, a setup phase in which some state is configured at each switch along a path from source to destination, and second, the information transfer phase when the frames are actually sent. Of course, because the frames themselves contain no information about where they should go, the setup phase needs to take care of this.

[1]Some authors tend to call these "store-and-forward" networks, but we use a more-precise term. We will save the term "store-and-forward" for networks such as electronic mail processor networks.

A common way of implementing circuit switching is using *time-division multiplexing (TDM)*, also known as *isochronous transmission.* Here, the physical capacity of a link connected to a switch, $C$ (in bits/s), is conceptually broken into some number $N$ of virtual "channels," such that the ratio $C/N$ bits/s is sufficient for each information transfer session (such as a telephone call between two parties). Call this ratio, $R$, the bandwidth for each independent transfer session. Now, if we constrain each frame to be of some fixed size, $s$ bits, then the switch can perform time multiplexing by allocating the link's capacity in time-slots of length $s/C$ units each, and by associating the $i$th time-slice to the $i$th[2] It is easy to see that this approach provides each session with the required $R$ bits/s, since each session gets to send $s$ bits over a time period of $Ns/C$ seconds, and the ratio of the two is equal to $C/N = R$ bits/s.

Each data frame is therefore forwarded by simply using the time slot in which it arrives at the switch to decide which port it should be sent on. Thus, the state set up during the first phase has to associate one of these channels with the corresponding soon-to-follow data transfer by allocating the $i$th time-slice to the $i$th transfer. The end computers transmitting data send frames only at the specific time-slots that they have been told to do so by the setup phase.

Other ways of doing circuit switching include *wavelength division multiplexing (WDM)*, *frequency division multiplexing (FDM)*, and *code division multiplexing (CDM)*; the latter two are used in some cellular wireless networks.

Circuit switching makes sense for a network where the workload is relatively uniform, with all information transfers using the same capacity, and where each transfer uses a constant (or near-constant) bit-rate. The most compelling example of such a workload is telephony, and this is indeed how most telephone networks today are architected. (The other reason for this is historical; it was invented long before packet switching!)

However, circuit-switching tends to waste link bandwidth if the workload uses a variable bit-rate, or if the information transfer in terms of frame arrival rate at a switch is bursty. Since a large number of data transfer applications tend to be this way, we should look at other link sharing strategies. Packet-switching is a general way of getting better performance for such workloads.

## 2 Packet switching

The best way to overcome the above inefficiencies is to allow for any sender to transmit data at any time, but yet allow the link to be shared. Packet switching is a way to accomplish this, and uses a tantalizingly simple idea: add to each frame of data a little bit of information that tells the switch how to forward it. This information is added to what is usually called a *header.* There are several different forms of packet switching, which differ in the details of what information is present in the header and what information the switch needs to perform forwarding.

The "purest" form of packet switching uses a *datagram* as the unit of framing, with the header containing the *address* of the destination. This address uniquely identifies the destination of data, which each switch uses to forward the datagram. The second form of packet switching is *source routing*, where the header contains a complete sequence of switches, or complete *route* that the datagram can take to reach the destination. Each switch now has a simple forwarding decision, provided the source of the datagram provides correct information. The third form of packet switching is actually a hybrid between circuit and packet switching, and uses an idea called *virtual circuits.* Because

[2]Strictly speaking, to the $i$ *modolu* $N^{th}$ transfer.

it uses a header, we classify it as a packet switching technique, although its use of a setup phase resembles circuit switching. We now look at each of these techniques in more detail.

## 2.1 Datagram routing

In datagram routing, the sender transmits datagrams that include the address of the destination in the header; datagrams also usually include the sender's address to help the receiver send messages back to the sender. The job of the switch is to use the destination address as a key and perform a lookup on a data structure called a *routing table* (or *forwarding table*; the distinction between the two is sometimes important and will be apparent later in the course). This lookup returns an output port to forward the packet on towards the intended destination.

While forwarding is a relatively simple lookup in a data structure, the harder question is determining how the entries in the routing table are obtained. This occurs in a background process using a *routing protocol*, which is typically implemented in a distributed manner in the switches. There are several types of routing protocols possible (both in theory and practice, although some only in theory!), and we will study several in later lectures. For now, it is enough to understand that the result of running a routing protocol is to obtain routes (paths) in the network to every destination.

Switches in datagram networks that implement the functionality described in this section are often called *routers*. Forwarding and routing of packets using the Internet Protocol (IP) in the Internet is an example of datagram routing.

## 2.2 Source routing

Whereas switches implemented routing protocols to populate their routing tables in the "pure" datagram networks of the previous section, the equivalent functionality of figuring out paths could also be performed by each sender. When this is done, the network can implement *source routing*, where the sender attaches an entire (and complete) sequence of switches (or more helpfully, per-switch next-hop ports) to each packet. Now, the task of each switch is rather simple; no table lookups are needed. However, it does require each sender to participate in a routing protocol to learn the topology of the network.

People tend not to build networks solely using source routing because of the above reason, but many networks (e.g., the Internet) allow source routing to co-exist with datagram routing.

## 2.3 Virtual circuits

Virtual circuit (VC) switching is an interesting hybrid between circuit and packet switching, which combines the setup phase of circuit switching with the explicit header of packet switching. The setup phase begins with the source sending a special *signaling* message addressed to a destination, which traverses a sequence of switches on its way to the destination. Each switch associates a local *tag* (or *label*), on a per-port basis, with this signaling message and sets this tag on the message before forwarding it to the next switch. When a switch receives a signaling message on one of its input ports, it first determines what output port will take the packet to its destination. It then associates the combination of input port and incoming tag to an entry in a local table, which maps this combination to an output port and outgoing tag (unique per-output).

Data transfer does not use the destination address in the packet header, but uses these tags instead. The forwarding task at each switch now consists of a tag lookup step, which yields and output port and replacement tag, and a tag swapping step which replaces the tag in the packet header.

The reason for the replacement tag is simply to avoid confusion; if global tags were used, then each source would have to be sure that any tag it chooses is not currently being used in the network.

There are many examples of network technologies that employ virtual circuit switching, including Frame Relay and Asynchronous Transfer Mode (ATM). These networks differ in the details of the tag formats and semantics (and these tags are known by different names; e.g., in ATM, a tag is a combination of a VPI or Virtual Path Identifier and VCI or Virtual Circuit Identifier, which can be thought of as a single tag whose structure is hierarchical), and in the details of how these tags are computed. Multi-Protocol Label Switching (MPLS) is a link technology-independent approach that network switches can use to implement tag switching (we will look at this in a later lecture). The general principle in all these systems is as explained in this section.

Proponents of virtual circuit switching argue that it is advantageous over datagram routing because of several reasons, including:

- It allows routes between source and destination to be "pinned" to a fixed route, which allows network operators to provision and engineer their network for various traffic patterns.
- Tag lookups are more efficient than more-complex lookups based on various fields in the datagram header (we don't have enough context yet to understand this point, but it will become clearer mid-way through the course!).
- Because there is an explicit setup phase, applications that (think they) need resource reservation (e.g., link bandwidth, switch buffer space) can use this signaling to reserve resources.

These claims are controversial: virtual circuit switching is more complex than datagram routing, and does not handle link or route failures as naturally. The rationale and mechanism for resource reservation has been hotly debated in the community and will continue to be for some more years!

Virtual circuit technologies are common in the Internet infrastructure, and are often used to connect two IP routers in a *transport network*. Transport networks are the "link-layer" over which IP packets between two routers are communicated; examples include ATM-based link technologies.

Let us now look at a specific example of switching, using the widely deployed LAN (local-area network) switching technolgogy as an example.

## 3 An example: LAN switching

A single shared medium segment, like a single Ethernet, is limited by the number of stations it can support and by the amount of traffic that can be shared on it. To extend the reach of a single LAN segment requires some way of interconnecting many of them together. Perhaps the simplest way of extending LANs is via "LAN switching," a technique also known as "bridging" (historically). Bridges (or LAN switches; we will use the two terms interchangably) extend the reach of a single shared physical medium. They work by looking at data frames arriving on one segment, capturing them, and transmitting them on one or more other segments. We will study this in the context of a network with datagram routing.

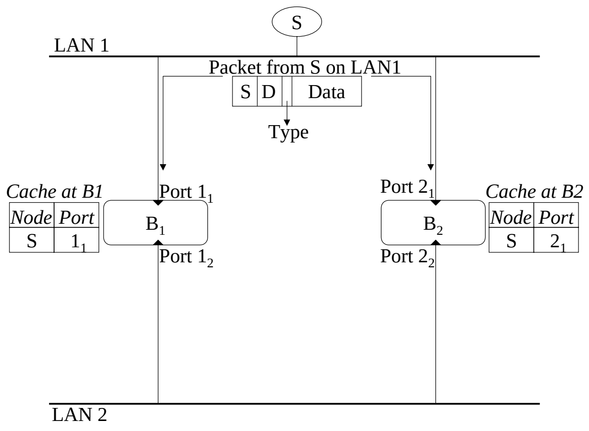

Figure 1: Learning bridges with loops. Such bridges suffer from packet duplication and proliferation.

Another reason to study bridges is because they are a great example of self-configuring ("plug-and-play") networks.

Bridges are characterized by:

1. Promiscuous receive and forward behavior with two or more ports. Each port has a LAN attached to it, which in turn could contain other bridges. In such a LAN, the aggregate capacity does not exceed the capacity of the weakest segment, since any packet transmitted on any LAN appears on all others, including the slowest one.
2. Learning, wherein they learn which stations are on which LAN segments to forward packets more efficiently.
3. Spanning tree construction, by which bridge topologies with loops can avoid packet implosions.

Bridges are *transparent* entities—they completely preserve the integrity of the packets they handle without changing them in any way. Of course, they may add a little to the delays experienced by packets and occasionally lose packets because of their store-and-forward nature, but they are transparent in terms of functionality.

## 3.1 Learning bridges

The basic idea of a learning bridge is that the bridge learns, by building a cache, which stations are downstream of which port. Then, when a packet destined to a given destination (MAC address) arrives, it knows which port to forward it on. How does it build this cache? It learns by looking at the *source* address of packets it sees. And if it doesn't have an entry for some destination, it simply floods the packet on all ports except the one the packet just arrived on. Thus, the state maintained by a learning bridge is akin to a cache, and is used only as an optimization (albeit a very useful one). Consistency of this cache is not essential for correctness.

This strategy works well, except when there are *loops* in the network topology. In fact, in the absence of loops, not only does it handle routing, but also nodes that move in the topology from time to time (mobile nodes). If a node moves to another location in the switched network, the first time it sends data, the bridges along the (new) path to its destination cache the new point of attachment of the node that moved. Indeed, a variant of this method with a few optimizations is used in various wireless LAN access points to implement link-layer mobility. The 802.11 technology deployed at LCS uses this method.

When there are loops in a switched LAN, significant problems arise. Consider the example shown in Figure 1, where bridges $B_1$ and $B_2$ have been configured to connect LAN1 and LAN2 together. (One reason for doing this would be to add redundancy to the topology for reliability.) Consider a packet from source $S$ transmitted on LAN1. Since both $B_1$ and $B_2$ see this packet, they both pick it up and learn that node $S$ is on LAN1, and add the appropriate information to their bridge caches as shown in the picture. Then, both bridges enqueue the packet to forward it on to LAN2. They compete for the channel (LAN2) according to the CSMA/CD protocol and one of them, say $B1$, wins the contention race. It forwards the packet on to LAN2, and of course this packet is seen by $B2$. $B2$ has now seen a *duplicate* packet! It really has no way of telling that it's a duplicate short of carefully looking through every bit of each enqueued packet, which would be quite inefficient. This is a direct consequence of one of the very reasons learning bridges are so attractive, their transparent behavior. Thus, the duplicate packet would continue to loop around forever (because bridges are transparent, there are no hop limit or time-to-live fields in the header).

But this isn't the worst part—packets in fact can *reproduce* over and over in some cases! To see this, add another bridge $B3$ between the two LANs. Now, each time a bridge sends one packet on to a LAN, two other bridges enqueue one packet *each* for the other LAN. It's not hard to see that this state of affairs will go on for ever and make the system unusable when bridges form loops.

There are several possible solutions to this problem, including avoiding loops by contruction, detecting loops automatically (but not working when loops are present), and making things work in the presence of loops. Clearly the last alternative is the preferred one. The trick is to find a loop-free path in the bridge topology. This is done using a distributed spanning tree algorithm.

## 3.2 The Solution: Spanning Trees

There are many distributed spanning tree algorithms. Bridges use a rooted spanning tree algorithm that generates the same tree as Dijkstra's shortest path trees. The idea is quite simple: bridges elect a root and form shortest paths to the root. The spanning tree induced by the union of these shortest paths is the final tree.

More specifically, the problem is as follows. For each bridge in the network, determine which of its ports should participate in forwarding data, and which should remain inactive, such that in the end each LAN has exactly one bridge directly connected to it on a path from the LAN to the root.

Viewing a network of LAN segments and LAN switches as a graph over which a spanning tree should be constructed is a little tricky. It turns out that the way to view this in order to satisfy the problem statement of the previous paragraph is to construct a graph by associating a node with each LAN segment and with each LAN switch. Edges in this graph emanate from each LAN switch, and connect to the LAN segment nodes they are connected to in the network topology, or to other LAN switches they are connected to. The goal is to find the subset of edges that form a tree, which span all the LAN segment nodes in the graph (notice that there may be LAN switch nodes that may be eliminated by this; this is fine, since those LAN switches are therefore redundant).

The first challenge is to achieve this using a distributed, asynchronous algorithm, rather than using a centralized controller. The goal is for each bridge to independently discover which of its ports belongs to the spanning tree, since it must forward packets along them alone. The end-result is a loop-free forwarding topology. The second challenging part is handling bridges that fail (e.g., due to manual removal or bugs), and new bridges and LAN segments that are attached to the network, without bringing the entire network to a halt.

To solve the first problem, each bridge periodically sends *configuration messages* to all other bridges on the LAN. This message includes the following information:

```
    [bridge_unique_ID] [bridge's_idea_of_root] [distance_to_root]
```

By consensus, the bridge with the smallest unique ID is picked as the root of the spanning tree. Each bridge sends in its configuration message the ID of the bridge that it thinks is the root. These messages are not propagated to the entire extended LAN, but only on each LAN segment; the destination address usually corresponds to a well-known link-layer address corresponding to "ALL-BRIDGES" and is received and processed by only the bridges on the LAN segment. Initially, each bridge advertises itself as the root, since that's the smallest ID it would have heard about thus far. The root's estimate of the distance to itself is 0.

At any point in time, a bridge hears of and builds up information corresponding to the smallest ID it has heard about and the distance to it. The distance usually corresponds to the number of other bridge ports that need to be traversed to reach the root. It stores the port on which this message arrived, the "root port" and advertises the new root on all its other ports with a metric equal to one plus the metric it has stored. Finally, it does not advertise any messages if it hears someone else advertising a better metric on the same LAN. This last step is necessary to ensure that each LAN segment has exactly one bridge that configures itself to forward traffic for it. This bridge, called the *designated bridge* for the LAN, is the one that is closest to the root of the spanning tree. In the case of ties, the bridge with smallest ID performs this task.

It is not hard to see that this procedure converges to a rooted spanning tree if nothing changes for "a while." Each bridge only forwards packets on ports chosen as part of the spanning tree. To do this, it needs to know its root port and also which of its other ports are being used by other bridges as their preferred (or *designated*) root ports. Obtaining the former is trivial. Obtaining the latter is not hard either, since being a designated bridge for a LAN corresponds to this. When two bridges are directly connected to each other, each bridge views the other as a LAN segment.

The end-result of this procedure is a loop-free topology that is the same as a shortest-paths spanning tree rooted at the bridge with smallest ID.

Notice that the periodic announcements of configuration messages handle new bridges and LAN segments being attached to the network. The spanning tree topology reconfigures without bringing the entire network to a complete standstill in most cases.

This is the basic algorithm, but it doesn't quite work when bridges fail. Failures are handled by timing information out in the absence of periodic updates. In this sense, bridges treat configuration announcements as *soft state*. The absence of a configuration message from a designated bridge or the root triggers a spanning tree recalculation. This notion of "soft state" is an important idea that we will repeatedly see in this course, and is important to robust operation in a scalable manner. Together, periodic announcements to refresh soft state information inside the network enable *eventual consistency* to a loop-free spanning tree topology using the algorithm described above.

### 3.3 Virtual LANs

As described thus far, switched LANs do not scale well to large networks. The reasons for this include the linear scaling behavior of the spanning tree algorithm, and the fact that all broadcast packets in a switched LAN must reach all nodes on all connected LAN segments. *Virtual LANs* improve the scaling properties of switched LANs by allowing a single switched LAN to be partitioned into several separate virtual ones. Each virtual LAN is assigned a "color," and packets are forwarded by a LAN switch on to another only if the color matches. Thus, the algorithms described in the previous section are implemented over each color of the LAN separately, and each port of a LAN switch is configured to some subset of all the colors in the entire system.

## 4 Summary

Switches are specialized computers used to interconnect single-link communication media to form bigger networks. There are two main forms of switching—circuit switching and packet switching. Packet switching comes in various flavors, as do switched networks themselves. We studied some features of one of them, switched LANs.

While LAN switches work well, they aren't enough to build a global network infrastructure. There are two reasons for this.

1. Lack of scalability. First, each LAN switch has to maintain per-host information in its forwarding table. In addition, the occasional flooding that's required does not scale well.

2. Inability to work across heterogeneous link technologies. One of the goals of a global network infrastructure is to work properly across a variety of link technologies (not just Ethernets).

The problem of handling the two challenges mentioned above and provide a global network infrastructure is called the *internetworking problem.* We will study ways to solve this problem in the coming lectures, starting from the next one.