Lecture 2

Gidon Rosalki

2025-10-26

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_networking`

1 Taxonomy of communication networks

Communication networks can be classified based off how the nodes exchange information.

1.1 Broadcast communication network

Information transmitted by *any* node is received by *every* other node in the network. This is usually only applicable in LANs (Local Area Networks), when using things such as WiFi, or classical Ethernet. Another such example is a lecture?
This has a few problems. Firstly, there is a very limited range. Every node that wants to listen to node a needs to be directly connected to it. Secondly, one needs to coordinate access to the shared communication medium, to nodes cannot transmit at once (not everyone can talk at the same time in a lecture). Finally, there is no privacy. Everyone can hear everything said, since you cannot say something directly to another person.

1.2 Switched communication network

Here the information is switched, and only sent to specific nodes. Last lecture we saw an example of circuit switched communication networks. There are also *packet switched* communication networks, also known as *datagram* networks. Datagrams are comprised of 2 parts:

1. Header: Instructions to the network for how to handle the packet
2. Payload (body): The actual data being transferred to the recipient

Each packet is switched independently, containing the complete header with the destination address. No resources are preallocated in advance. In this design, each packet needs to be a discrete, relatively small size. This way, if a packet is lost, relatively little data is lost. Additionally, if we were to send large packets, then it would take longer to handle them upon receipt. Smaller packets will reduce latency. However, we also do not want them to be too small, since then we would be sending large amounts of header data for insignificantly small amounts of payload data.

An important impact of using datagram communication is that we can have different packet sending patterns, they may arrive smoothly, there may be gaps as packets arrive in bursts, and so on. Due to bursts, sometimes the transient arrival rate is larger than the transmission rate, even if the long term average arrival rate is smaller than the transmission rate. How do we resolve this? One option is to drop packets. This is obviously not great, since the information is being sent for a reason, and this would just lose parts of it. A better approach is to add a buffer in which we save the excess packets, and read from it instead. Whenever new packets arrive, we add them to the end of the buffer, and read from the start of the buffer. This way, when the burst ends, we are still reading the data from the buffer, and have not lost packets. However, this has its downsides. Even with buffers (even if they should be of infinite size), packets can still be lost. Additionally, buffers can add latency to the operations, especially when the network is busy.

1.2.1 Slots per frame

Let us assume that time is divided into frames, and that frames are divided into slots. *Flows* generate packets during each frame, with a peak number of packets per frame being defined to be P. The average number of packets per frame is defined as A. A single flow must allocate P slots in order to avoid packet drops. However, this is very wasteful, since P might be much larger than A. To avoid this, many flows can exploit the "Law of Large Numbers".

Law of Large Numbers: Consider any probability distribution, and take N samples from said distribution (in this case, that is one set of packets from each flow). The theorem states that the sum of the samples is very close to $N \times A$, and gets percentage-wise closer as N increase. Therefore, sharing between many flows (high aggregation) means that you only need to allocate slightly more than the average A slots per frame.

1.3 Comparison

There are some advantages to **circuit** switching. There is a guaranteed bandwidth, with predictable communication performance, rather than “best effort” delivery, with no real guarantees. There is also a simple abstraction (for the application), with a reliable communication channel between hosts, and no worries about lost, or out of order packets. We also have simple forwarding (for the network devices) where the forwarding is based on a time slot, or frequency, with no need to inspect a packet header. Finally, there is a low overhead per packet, where we forward based off time slot or frequency, and have no headers on each packet.

There are also some disadvantages to circuit switching. There’s wasted bandwidth, since bursty traffic leads to idle connections during the silent period. It is also unable to achieve gains from “statistical multiplexing”. Connections also get refused when there are insufficient resources, resulting in blocking, and at times it cannot offer a “good enough” service to everybody. The network must store state, with network nodes storing per connection information. This means that failures are more disruptive. Additionally, we have a communication setup delay, where nothing can be sent before the connection is set up. It is also unable to avoid extra latency for small data transfers.

There is a reliability advantage to packet switching: Routers do not know about individual conversations, and when a router or link fails, it is easy to fail over to a different path. It is also more efficient, thanks to the ability to exploit statistical multiplexing (law of large numbers). Next, there is an advantage to how easy it is to deploy, it is easier for different parties to link their networks together because they are not promising to reserve resources for each other. However, packet switching must handle congestion, which is automatically handled in circuit switching. This is done through adding complexity to routers (buffering, sophisticated dropping), and so it is harder to provide good network services, since we cannot provide such good guarantees with regards to the delay, and bandwidth.

2 Modularity

2.1 Computer modularity

In computers, we partition the system into modules, and abstractions. Well defined interfaces enable flexibility, where we hide the implementation, allowing it to change freely, but limiting their scope to enable the abstraction. We may also extend the functionality of a system by adding new modules. This allows for continuing evolution, and innovation. All of this work isolates assumptions, presenting high level abstractions, making implementation easier, but this comes at the cost of performance (consider the size of an asm hello world, vs the most basic C implementation).

2.2 Network modularity

This is like computer modularity, but the implementation is distributed across many machines (routers and hosts). One must decide how to break the system into modules (layering), where the modules are implemented (on the hosts, or on the network), and where the state is stores (fate sharing).

2.2.1 Layering

From the bottom up, networking has the following tasks:

- Electrons in the wire
- Bits in the wire
- Packets in the wire
- Delivering packets across a local network (LAN) with local addresses
- Delivering packages across multiple LANs, with global addresses
- Ensuring that packets arrive (handle losses)
- Do something with the data

So how do we handle these tasks? By putting them into “layers”. Layering is a particularly strict form of modularity, where the interactions are limited to interfaces above and below.
So, if we add in the layers that we will use:

- Electrons in the wire
- **Bits in the wire (physical layer)**
- Packets in the wire
- **Delivering packets across a local network (LAN) with local addresses (link layer)**
- **Delivering packages across multiple LANs, with global addresses (network layer)**
- **Ensuring that packets arrive (handle losses) (transport layer)**
- **Do something with the data (application layer)**

In short, we have 5 layers, and the top two layers are implemented only by the hosts.

- Application (host)
- Transport (host)
- Network
- Datalink
- Physical

Peers within each layer interact, and communication goes down to the physical network, then from the network peer to peer, before finally rising up to the relevant layer. This means that when my app sends a message, it goes down to the physical layer, and to the router. The router then raises it up to the network layer, understands what there is to do, before it goes back to the physical layer, and is transmitted to the recipient, where the message is raised up to the application layer once more.

2.2.2 Protocol

The *protocol* handles all the communication, but what is a protocol? It is an agreement on how to communicate, handling the exchange of data, and coordinating sharing resources. Protocols specify *syntax* and *semantics*. The *syntax* is how the protocol is structured, handling the format, and message order, where the *semantics* are how to respond to various messages and events.
We have various examples of protocols in real life, such as how we ask questions in class, or having a conversation, or perhaps answering the phone.

Since we want many machines to work together, it is very important to **standardise** the protocol, and for everyone to follow the same protocol. Very small modifications can make very significant differences, and potentially even prevent it from working. Standards allow us to have multiple implementations of the same protocol. To create this standardisation, we have the Internet Engineering Task Force (IETF).