Section 10: Networking

Frank Austin Nothaft

July 29, 2015

Contents

1 Warmup .......................................................... 2
   1.1 Layering .................................................. 2
   1.2 Why layers? .............................................. 2

2 Problems ......................................................... 3
   2.1 Lossy Networks and the End-to-end Principle ............. 3
   2.2 RPC ....................................................... 4
1 Warmup

1.1 Layering

In class last week, we introduced the concept of a network “stack.” The stack we covered last week was a four layer stack. Can you draw out the different layers of this stack, and describe what that layer does? Can you think of common applications/technologies/protocols that map to a specific level of the stack?

From bottom to top:

1. The **physical** (also known as link) layer makes up the technologies used to actually send information between machines. Technologies like ethernet and WiFi exist at this level.\(^a\)

2. The **Internet** layer is the layer that facilitates routing between two networks. The Internet Protocol (IP) is the dominant protocol used here. This layer makes routing decisions, provides error correction/detection, and handles network fragmentation.

3. The **transport** layer provides end-to-end messaging semantics. While this is a vague description, this layer can include a variety of semantics, such as congestion control, reliability, packet order guarantees, multiplexing, etc. The best known protocol at this layer is the Transmission Control Protocol (TCP). The second most common protocol is the User Datagram Protocol (UDP), which is “cheaper” than TCP, but sacrifices reliability and other desirable semantics.

4. The **application** layer provides shared protocols for communicating between networked machines. This includes remote login protocols like `ssh`, file transfer protocols like `ftp`, request/response protocols like `http`, and so on.

\(^a\)In some stacks, such as the OSI stack, this layer can be broken into separate physical and link layers. Here, the physical layer would be the protocol that standardizes the physical media (i.e., do we send the bits over a wire or over wireless?, how many bits do we send at a time?, how are these bits signaled?) and the link layer would be the protocol we use for scheduling access to the physical link, dealing with interference, encoding packets, etc. These layers are sometimes referred to as the PHY and MAC (media access control) layers.

1.2 Why layers?

As you may have noticed, the multilayered networking stack is a stark departure from the monolithic structure of an operating system kernel\(^1\). What is the advantage of breaking networking technologies into different stacks?

Stack based approaches inherently focus on abstraction and interfaces, which decouples higher level protocols from the lower level protocols. This decoupling is nice, because it means that we don’t need separate implementations of TCP for Ethernet, WiFi, etc. When deciding between monolithic and decomposed architectures, the main tradeoff is flexibility vs. performance. In a network, flexibility is crucial: although you may know that you will always be using Ethernet (in the 90’s, you might have known that...), you don’t know what link technologies people outside of your network will be using. In an operating system, flexibility isn’t as crucial (most of us will probably never recompile our kernel), but performance is important (the performance of the OS sets the performance of all applications).

\(^1\)Many academic projects have tried to implement non-monolithic OS kernels; this body of work is largely known as “microkernels”. This work has seen a fair bit of adoption in the embedded world, but it has not been seen great industrial reception for enterprise workloads.
2 Problems

2.1 Lossy Networks and the End-to-end Principle

Typically, if we want a highly reliable connection between machines, we will use a layer 3 (transport) protocol that provides reliability (e.g., TCP). Why do we not make network reliability a function of a lower layer, such as IP?

There are a variety of reasons that we don’t push features lower down in the stack. In the context of reliability, some reasons include:

- Adding reliability in the network makes the implementation of routers much more complex. E.g., a router must store a packets until it knows that the packet has been successfully received.

- Reliable packet transfer may not be sufficient to provide application level reliability. If this is the case, then applications must then independently implement their own reliable transmission protocol, which duplicates the network functionality.

This philosophy is known as the “End-to-end” argument. Specifically, if functionality is needed for application-specific correctness, and the functionality can be fully and correctly implemented by the end hosts, we should not implement this functionality in the network.

In TCP, if we notice that a packet was dropped, what do we do? Why do we take this action? What does this assume/what is this trying to exploit?

If we see that a packet was dropped, we will retransmit the packet, and halve the TCP window size. The window size is the number of “packets-in-flight” (unacknowledged packets) that we are allowed to have at a single time. TCP does this because we assume that packets are normally dropped because of network congestion. Reducing the number of packets we can have in flight at a single time reduces our input to the network, which should reduce congestion.

Here, TCP assumes that packet losses due to error (e.g., network is interrupted, packet is corrupted) are fairly infrequent. What could happen if our physical layer was naturally lossy? For example, wireless links have packet loss rates that are orders of magnitude higher than wired links like Ethernet. How could you solve this problem? Does your solution agree with or contradict the end-to-end principle?

If our network is lossy, and TCP mistakes packet losses for congestion, it will throttle our connection to a very slow speed. One natural approach to take is to add support in the link layer for packet retries on failure. This is done in some technologies, such as WiFi. One can actually argue that this approach both agrees with and contradicts the end-to-end principle. This does add significant complexity to the link layer, but wireless links are already pretty complex! Additionally, one could argue that without link layer retransmission, the packet error rate of wireless links would render them incompatible with transport layer protocols for ensuring reliability.
2.2 RPC

Remote Procedure Call (RPC) is an API for calling functions that reside on another computer, or in another process. When we are making an RPC call, how is the call executed?

1. The client locally calls the RPC function stub.
2. The RPC stub marshalls the function arguments, and creates a message to send to the remote machine.
3. This message is sent from one machine/process to another.
4. Once received, the server stub goes through the following steps:
   (a) The received message is unpacked.
   (b) The local implementation of the function is called.
   (c) The return value of the function is marshalled.

The marshalled return value is then sent back to the client.
5. Finally, the client stub will receive the return value, unpack it, and return.

RPC is used in a variety of contexts, with varying levels of success. What are the general reasons that RPC can be difficult to use?

- RPC provides unclear error semantics. If an RPC call fails, did it fail before or after the call ran on the server? If it failed after running on the server, should we (can we?) unroll the call?
- RPC attempts to make all operations look local. While this is good if you are in a uniform, low-latency network, or making local RPC calls, this can lead to high performance variability.
- Marshalling arguments/return values is complex for complex data structures.
- Marshalled data must be decodable by both client and server platforms. What if the client and server are different OS/endianness?

What are a few examples of places where RPC is easy to use successfully?

- Systems where there is a reliable, high performance network connecting clients and hosts.
- Communication between processes on a single machine.