E2E Notes, 6.033 Spring 2020

Background

This paper was published in 1981, before the main deployment of TCP/IP (in 1982/83; there was a 1974 proposal for a TCP-like service, however), and before the Internet as we know it (1986 - NSFNET; 1990 - ARPANET decommissioned; 1995 - NSFNET decommissioned/Internet commercialized).

This is a major paper, particularly in computer networking. It is one of those classic papers that has remained worth reading, because it's still cited today, and it has informed the design of numerous systems, including the Internet. It is meant to be a design principle across systems, but it has done the best in networking.

Throughout, you'll see the common abbreviation of "end-to-end" as E2E.

Careful File Transfer

The E2E paper centers around an example of "careful file transfer". What are some threats to successful transfer completion? What approaches could you take to achieve greater reliability?

1. The file might be read incorrectly from disk, due to hardware faults. You could add a checksum on the file, check it after copy.
2. The file system software might make a mistake in buffering/copying the data at either of the hosts. You could add checksums on chunks of the file, compare to the on-disk file. An end-to-end checksums if at host B.
3. Hardware errors during buffering/copying. Again, checksums.
4. The communication system could drop a packet, or change the bits in a packet, or duplicate a packet. You could force the communication system to do reliable communication, maybe in the way TCP does.
5. Either host may crash. Perhaps you could add end-to-end ACKs? "Did you get the whole file?" "Yes I got it."

What is the main crux of the E2E argument? Is it that lower layers shouldn't provide any of these types of solutions?

No. It's that we should let the application decide what it needs, not the lower layers. The lower layers should provide some solutions for some things, but only as performance enhancements.

What do *you* feel are the most compelling reasons for this argument?

In recitation, we would've had you discuss/debate this. But here are some reasons from the staff. There is not a single correct answer here.

1. In almost every situation, the endpoints will have to perform the same sort of functionality anyway (in the file example, the endpoints have to do some sort of checksumming, because errors can happen on the local machine).
2. Other applications can suffer huge performance hits when the lower layers make decisions for them. For example, voice applications care about latency, not bit errors; correcting for bit errors will lead to unacceptably high overhead.

Just because these reasons exist does not mean that the E2E argument is "correct" in all situations.

Examples of E2E arguments

The paper gives a lot of examples, many of which are still relevant. File transfer, bit errors, security, duplicate messages, etc. But we can come up with some more modern ones. We'll hold off talking about TCP for now.

In papers we have read

UNIX: keeping the file position in the kernel. Made a choice the application (arguably) should've made themselves. Benefit: read_next_byte() is trivial to implement.

UNIX: not imposing structure on the files (file = stream of bytes instead). *Didn't* make a choice; the application was allowed to make it itself.

OS: Microkernel vs. monolithic kernel. We've actually not talked about this as an E2E argument. Stuff crammed into the kernel must be used by everyone, whether it meets their needs or not. Stuff outside the kernel can be easily tailored or replaced.

Notice that we don't always make decisions that an E2E argument would strongly support. There are good reasons to use monolithic kernels. There are always pressures to increase the function of RISC processors.

TCP/IP and the E2E argument

Consider this quote from page 5 of the paper: "Although [a packet acknowledgment] may be useful within the network as a form of congestion control, it was never found to be very helpful to applications using the ARPANET. The reason is that knowing for sure that the message was delivered to the target host is not very important."

Is that still true?

No! *So many* applications run over TCP. It's difficult to imagine this motivation.

However, not *all* applications run over TCP. TCP and IP were intentionally split into two layers so that applications could use IP directly. And now, there is also UDP (a protocol for unreliable data transport); although many applications use TCP, a non-negligible portion do use UDP.

Dave Reed (one of the authors of the paper) argued that the TCP/IP split should happen; an original proposal was to provide *only* a TCP-like interface. So the E2E argument influenced the design of TCP as being a layer above IP.

Nowadays, it can be confusing to think about TCP in conjunction with the E2E paper. We often think of TCP as a "low-level" service, since applications can just run on top of it; they don't have to reimplement reliable transport. But think of it this way: the application still has a choice about whether to run over TCP or to run over UDP (or IP directly). For instance DNS, online games, and media streaming all use UDP, at least in part.

A lot of times, overhead is an argument against doing things at lower levels. Does TCP have a lot of overhead?

In some sense, yes. It's vastly more complicated than UDP (as some evidence, the UDP RFC -- the document which details UDP's operation -- is 663 words. The TCP RFC is over 21,000). ACKs take up a surprising amount of network traffic.

Lessons

How are we supposed to use the E2E argument in designing systems? How might one think about what should go into the lower layers and what shouldn't?

- Is it effective at a lower layer?
- Is it general purpose enough that everyone above will be willing to incur the costs, even if they don't need it?
- Will every user of the capability implement exactly the same thing, or close enough that a general purpose solution will be adequate?

Often, having a piece of functionality exist solely at the endpoints gives us flexibility, openness, and simplicity. But having functionality in the “middle” of a system can give performance benefits.