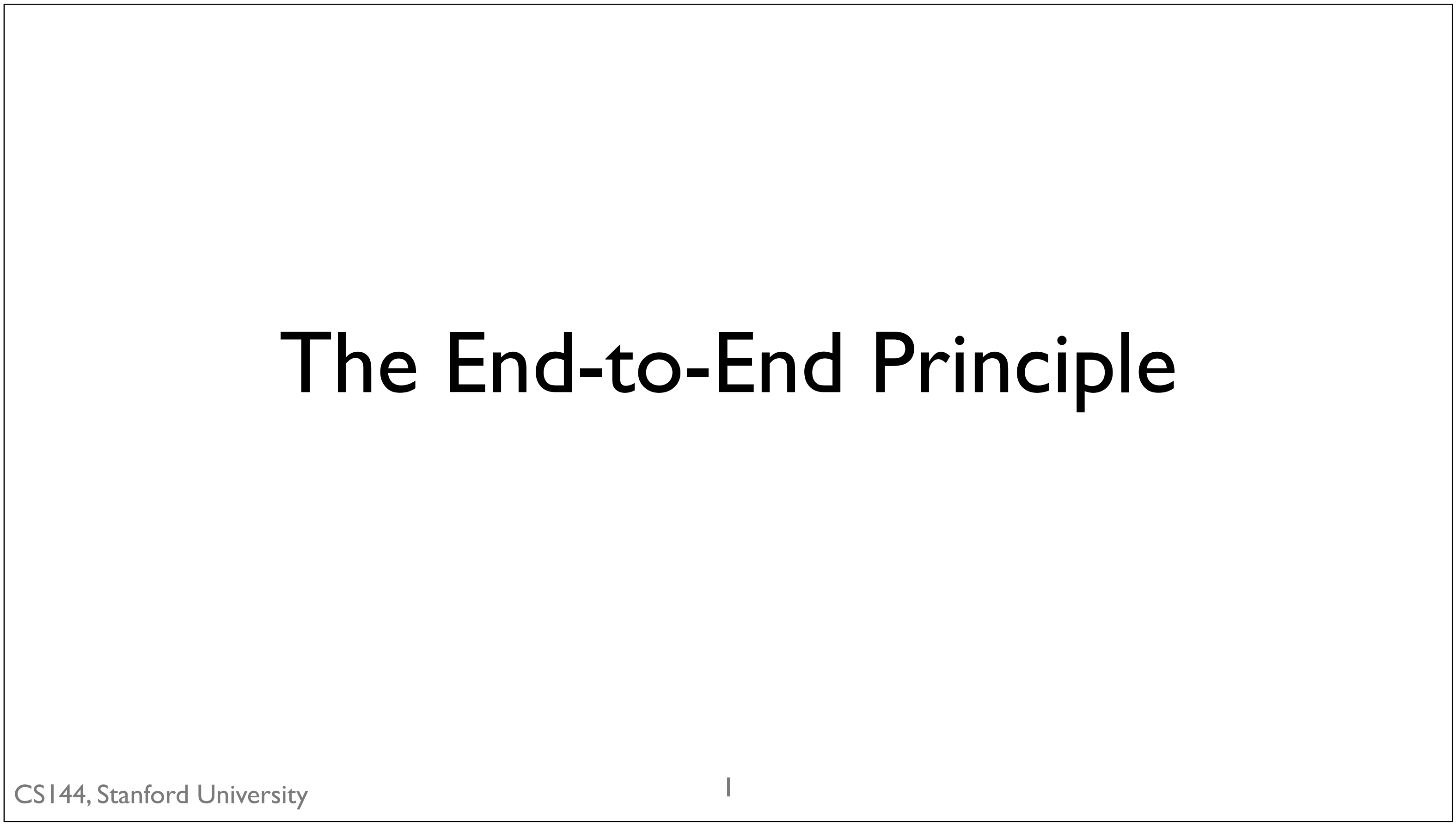
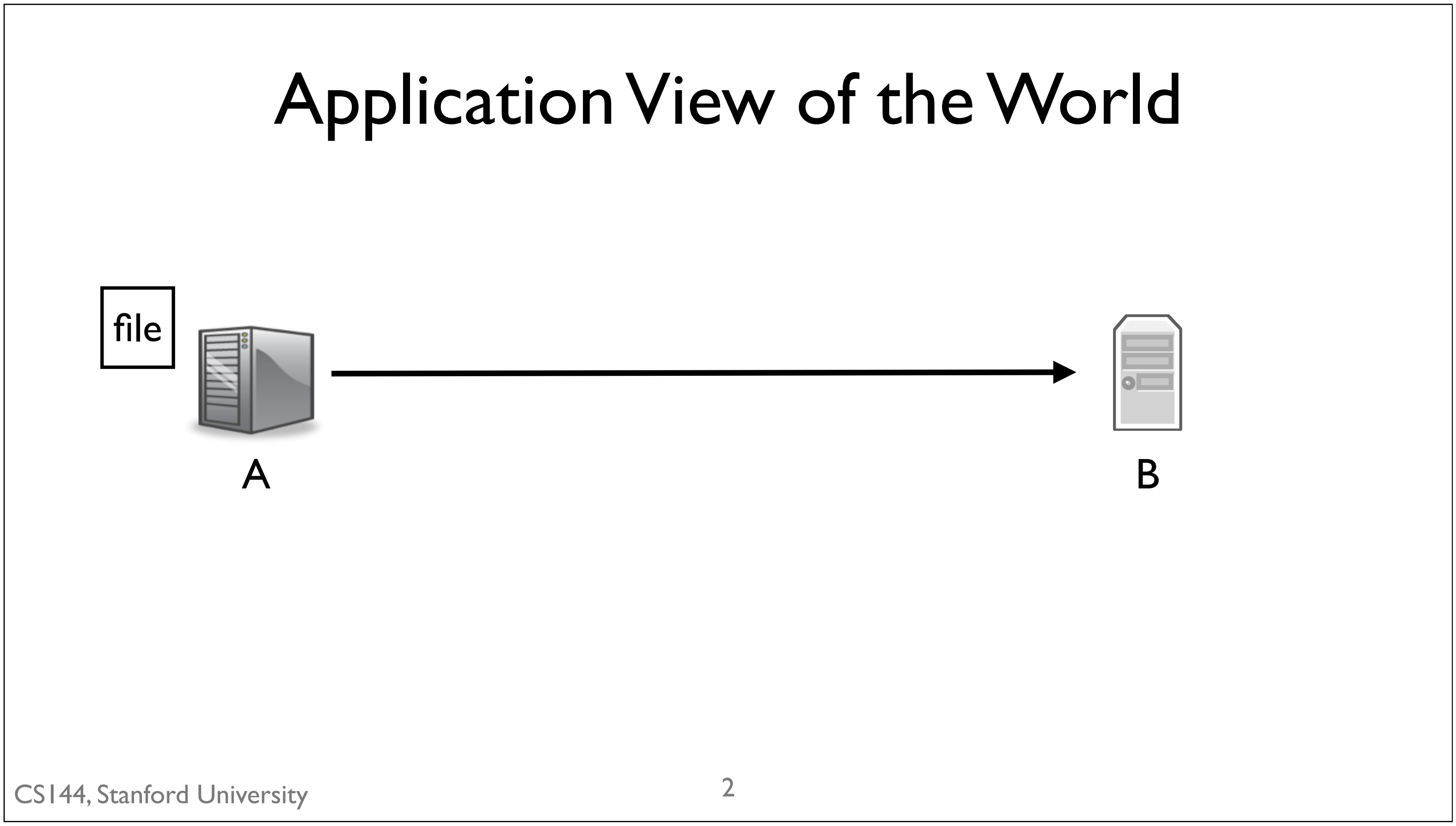


The End-to-End Principle

Nick: The end-to-end principle holds a very special place in the design of the Internet. This is because it really refers to two different principles: the first deals with correctness. If you don't follow the end-to-end principe when you design your networked system, then chances are it has a flaw and might transfer data incorrectly. The second, which we call the "strong" end-to-end principle, is much broader and general.

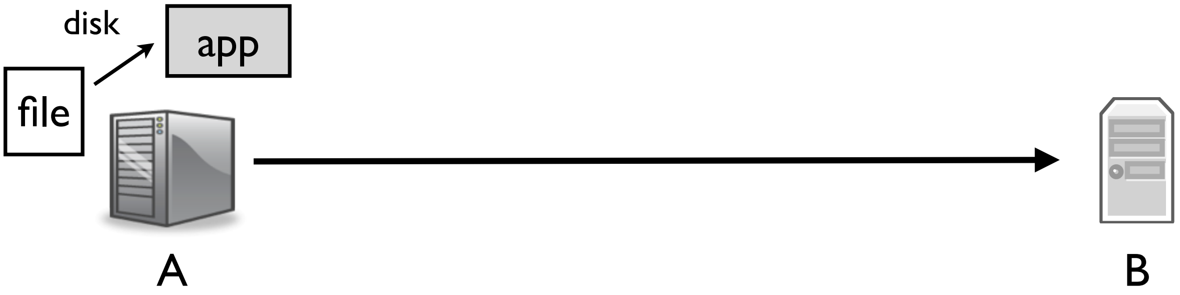


Application View of the World

Phil: So let’s say we want to transfer a file from one computer to another. Our application opens a connection between A and B. It reads a file on computer A and writes it to the TCP connection. B reads the data from the socket and writes the data to a file on computer B.

The network in this case does very little. It just forwards packets from A to B. A and B set up the connection, and the application reads and writes the data.

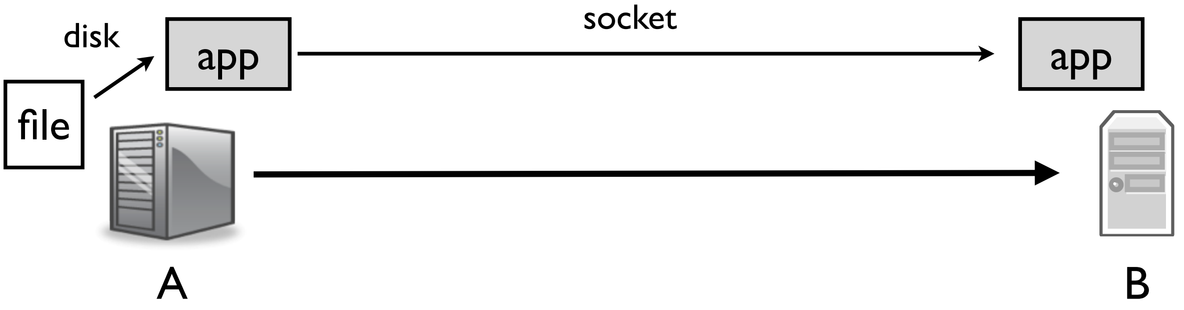
Application View of the World



Phil: So let’s say we want to transfer a file from one computer to another. Our application opens a connection between A and B. It reads a file on computer A and writes it to the TCP connection. B reads the data from the socket and writes the data to a file on computer B.

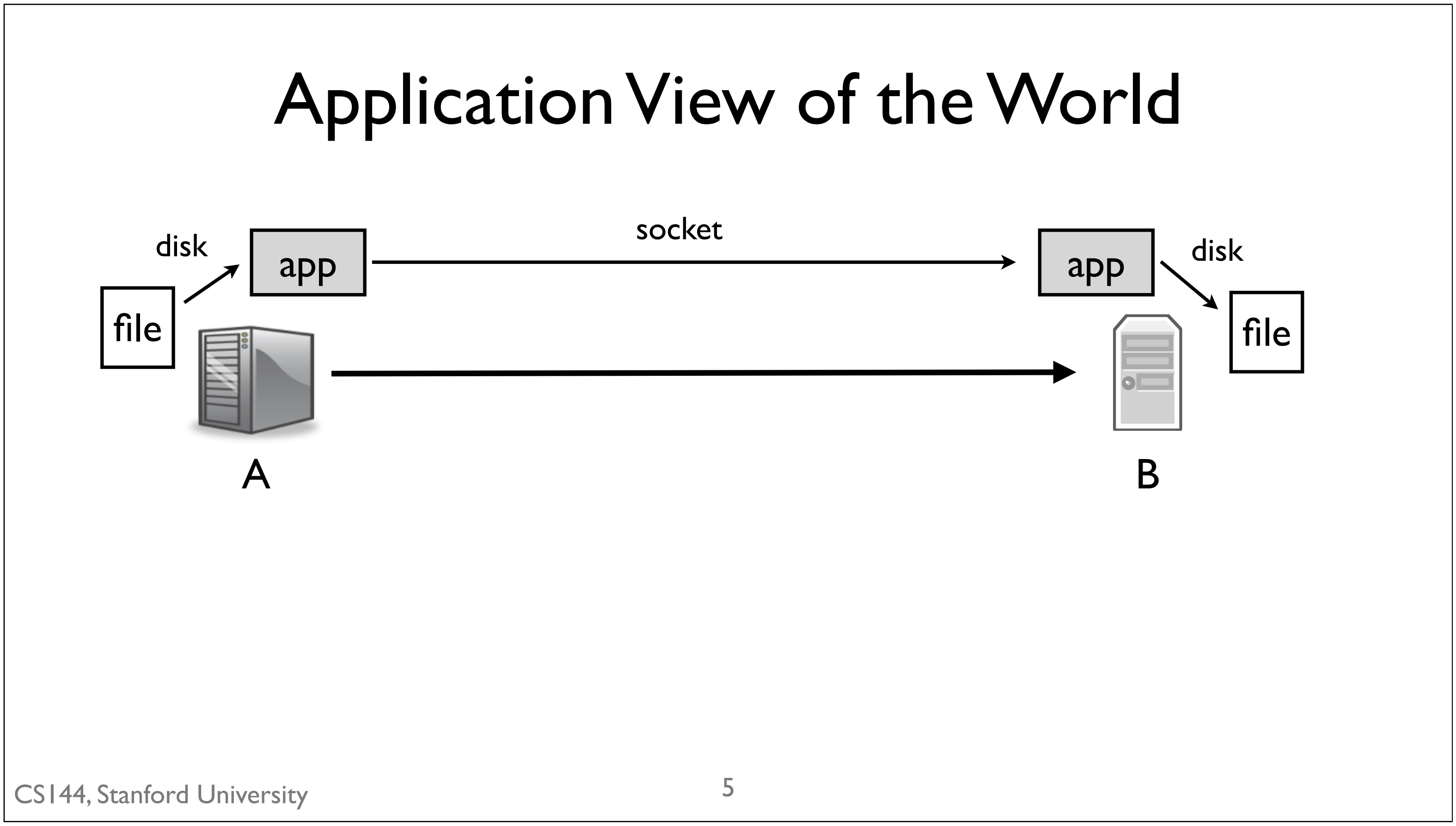
The network in this case does very little. It just forwards packets from A to B. A and B set up the connection, and the application reads and writes the data.

Application View of the World



Phil: So let’s say we want to transfer a file from one computer to another. Our application opens a connection between A and B. It reads a file on computer A and writes it to the TCP connection. B reads the data from the socket and writes the data to a file on computer B.

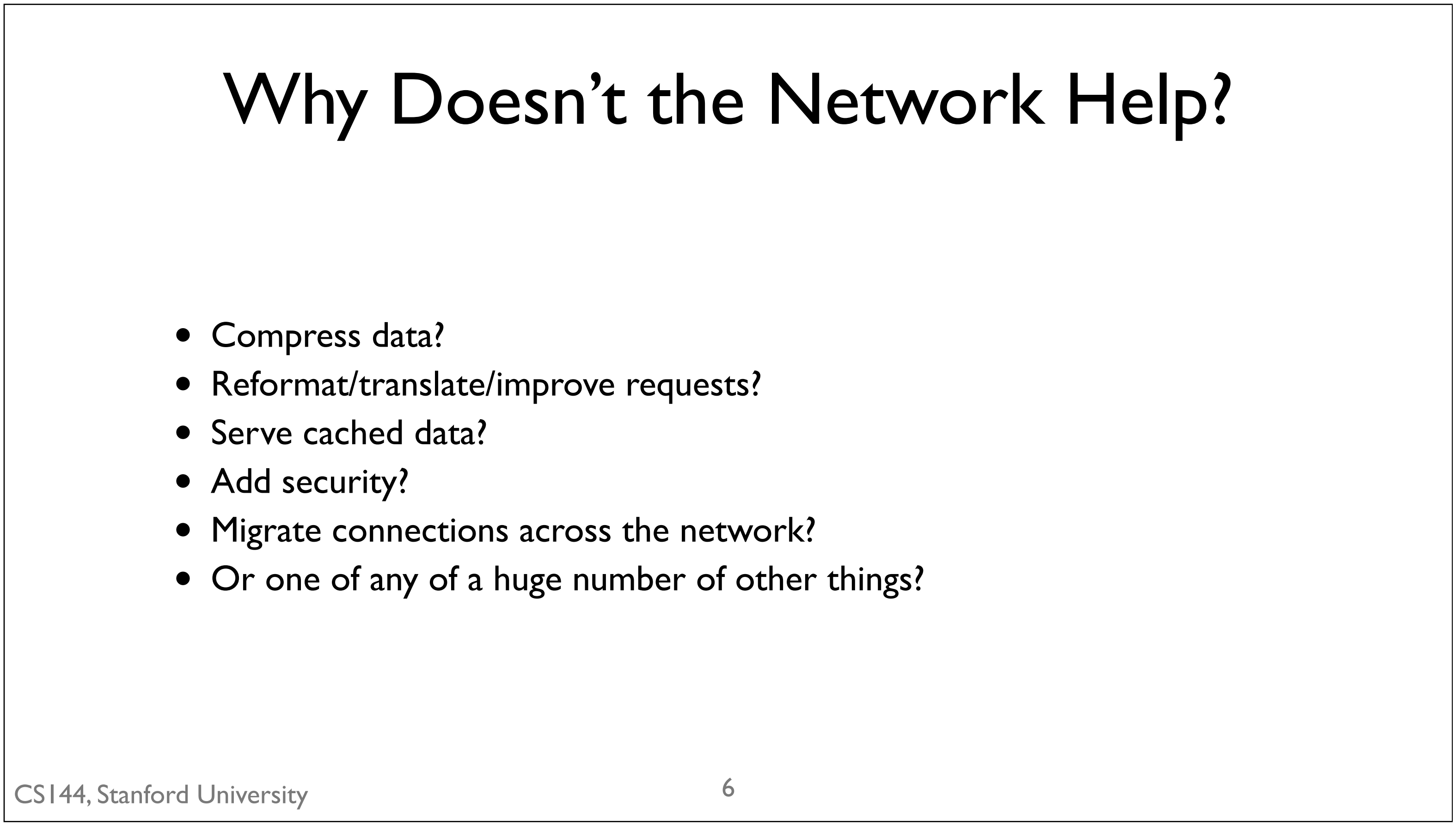
The network in this case does very little. It just forwards packets from A to B. A and B set up the connection, and the application reads and writes the data.



Application View of the World

Phil: So let’s say we want to transfer a file from one computer to another. Our application opens a connection between A and B. It reads a file on computer A and writes it to the TCP connection. B reads the data from the socket and writes the data to a file on computer B.

The network in this case does very little. It just forwards packets from A to B. A and B set up the connection, and the application reads and writes the data.



Why Doesn’t the Network Help?

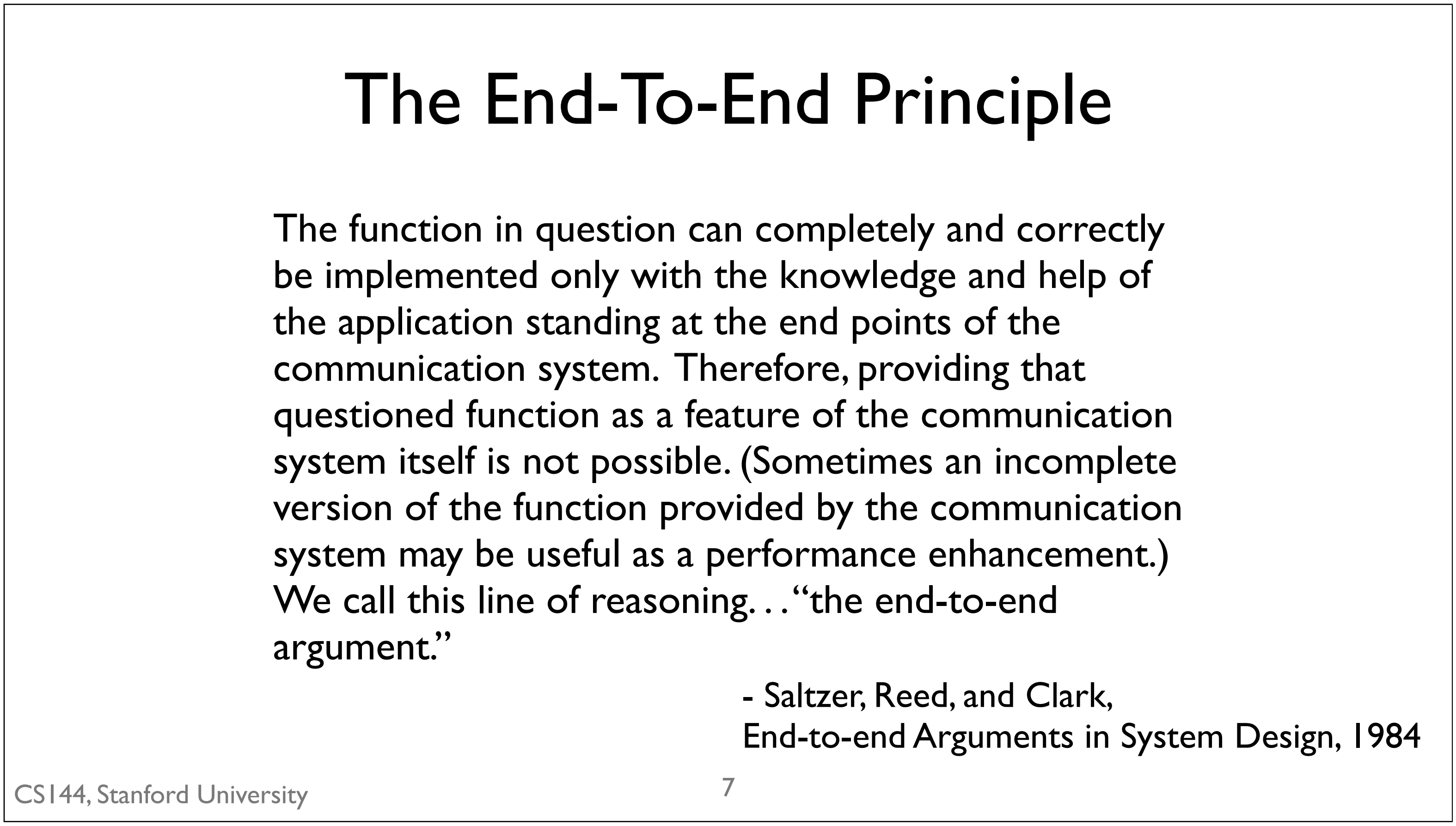
- Compress data?
- Reformat/translate/improve requests?
- Serve cached data?
- Add security?
- Migrate connections across the network?
- Or one of any of a huge number of other things?

Phil: Why doesn’t the network do more? It turns out there are a lot of things it could do to make the file transfer faster. The network could automatically compress packets between A and B. If the file is plain English text, this could reduce the transfer size tenfold. The network could reformat or improve requests. Let’s say that A wants to transfer two files to B. The network could see this and combine the two transfers into a single request. It could be that A’s file is already stored on another computer, C, that’s closer and faster to B than A is; the network could transfer the file from C rather than A.

Or the network could automatically add security, encrypting the data so bad guys can’t read the file. If the network does this for us, then we don’t have to worry about it in our application.

The network could add mobility support, so that as computer A moves through a network, routes automatically update and packets continue to flow to it. With this support, we could even possibly migrate connections across the network, moving something like a Skype video stream from our phone to our laptop.

It turns out there are *many* things the network could do to improve our application and make designing it easier. But generally speaking, it doesn’t. Why?



The End-To-End Principle

The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.) We call this line of reasoning. . . "the end-to-end argument."

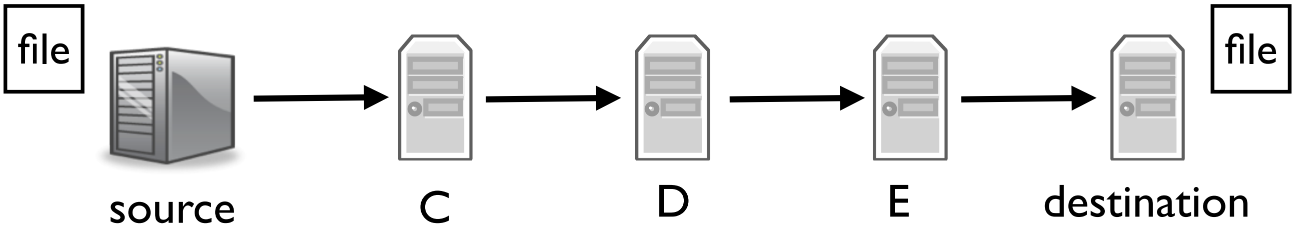
- Saltzer, Reed, and Clark,
End-to-end Arguments in System Design, 1984

Nick: The reason is the end-to-end principle. The end-to-end principle was first described by Saltzer, Reed and Clark in a 1984 paper. You'll meet David Clark later in the course when he gives a guest lecture. The end-to-end principle, as they describe it is:

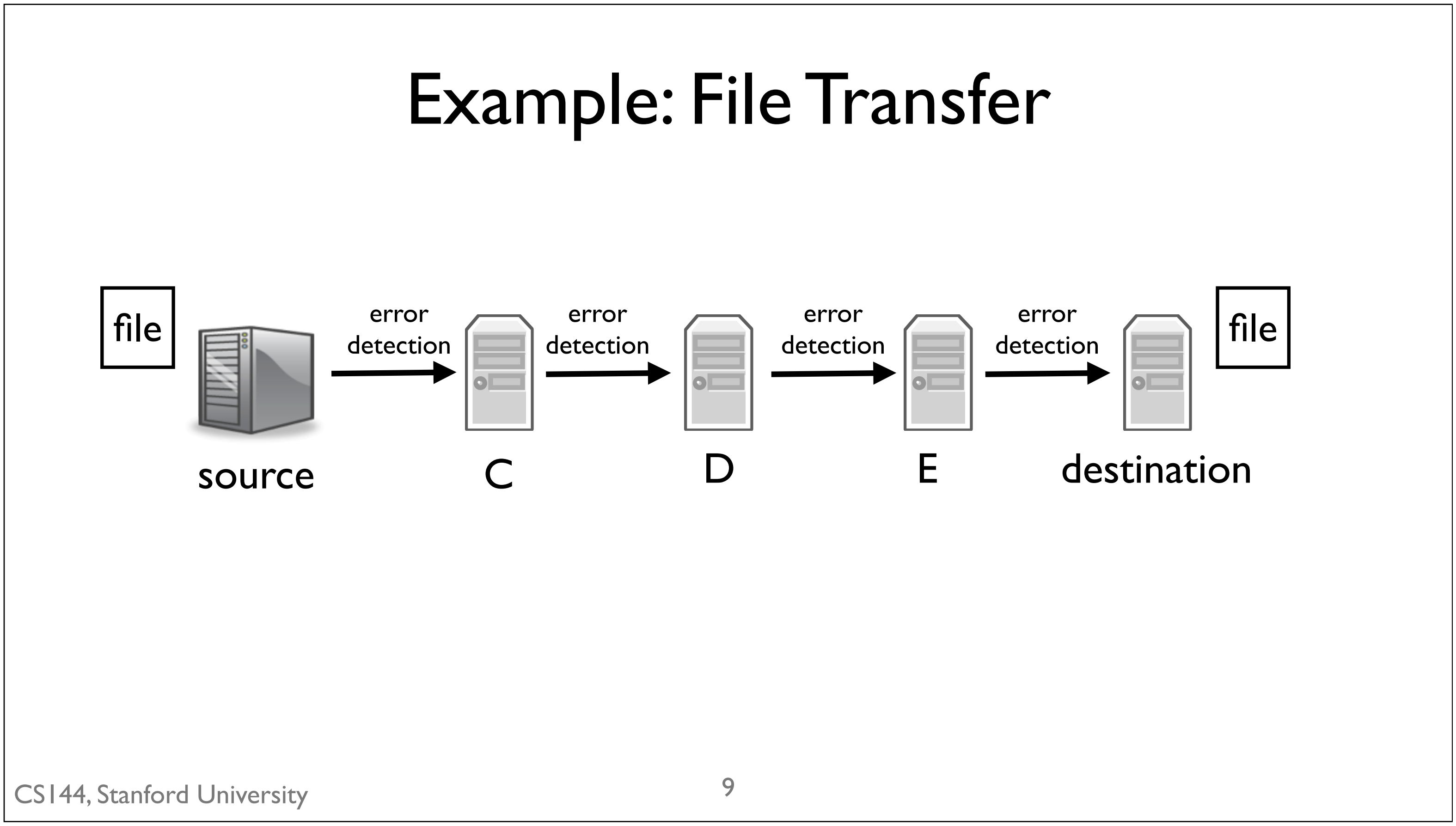
"The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.) We call this line of reasoning. . . "the end-to-end argument."

Put another way, the network could possibly do all kinds of things to help. But that's all it can do -- help. If the system is going to work correctly, then the end points need to be responsible for making sure it does. Nobody else has the information necessary to do this correctly. The network can help you, but you can't depend on it. For example, if you want to be sure your application is secure, you need to have end-to-end security implemented in the application. The network might add additional security, but end-to-end security can only be correctly done by the application itself. So making security a feature of the network so that applications don't have to worry about it is not possible.

Example: File Transfer



Nick: Let’s go back to our example of transferring a file between two computers. It was this exact problem, along with others, that led Saltzer, Clark and Reed to formulate the end-to-end argument. You want to make sure the file arrives completely and uncorrupted. The file data is going to pass through several computers between the source and the destination. So the file, coming from source, passes through computers C, D, and E before arriving at destination.



Example: File Transfer

file

error detection

error detection

error detection

error detection

file

source

C

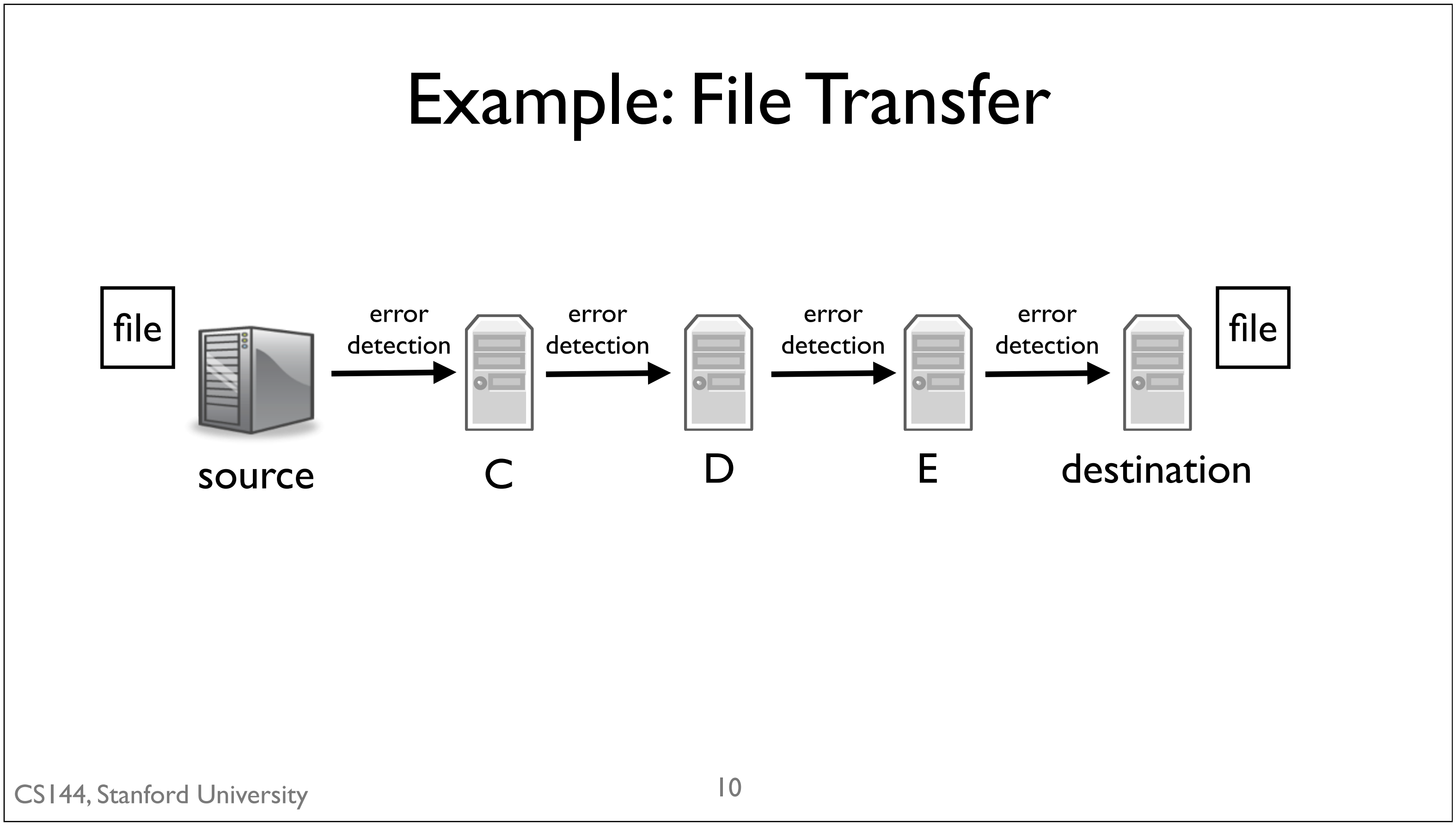
D

E

destination

Nick: Each link -- source to C, C to D, D to E, and E to destination -- has error detection. If a packet of data is corrupted in transmission, then the recipient can detect this and reject the packet. The sender will figure out the packet didn't arrive successfully, for example through TCP acknowledgments, and resend it.

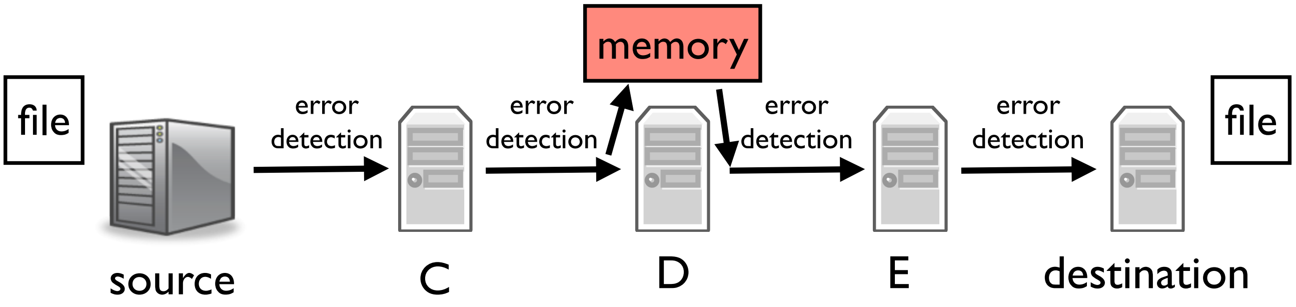
Now, one could say "Look, I know the packet won't be corrupted on any link, because I have my checks. Since it won't be corrupted on any link, it won't be corrupted. Therefore, if it arrives successfully at destination, there's no corruption, and the file has arrived successfully." This is exactly what some programmers at MIT did. Since the network provided error detection, they assumed it would detect all errors.



Example: File Transfer

Phil: This assumption turned out to be wrong, and because of this mistake the developers ended up losing a lot of their source code. This is what happened. One of the computers in the transfer path, let’s say computer D, had buggy memory, such that sometimes some bits would be flipped. D received packets of data, checked them, and found them correct.

Example: File Transfer



Phil: It would then move them into main memory, at which point they would become corrupted. It would then forward the packet, but because error detection occurs on the link, from the link's perspective the packet looked fine and it would pass E's check. The link error detection was designed for errors in *transmission*, not errors in storage.

The only way to be sure the file arrives correctly is to perform an end-to-end check. When the source sends the file, it includes some error detection information. When the destination re-assembles the file, it checks whether the file, it its entirety, has any errors. This is the only way one can be sure it arrived correctly. The network can help, but it can't be responsible for correctness.

As another concrete example, think of TCP. TCP provides a service of a reliable byte stream. But the reliability isn't perfect. There is a chance that TCP delivers some bad data to you, for example because there's a bug in your TCP stack, or some error creeps in somewhere. So while it's very *unlikely* TCP will give you corrupted data, it might, and so you need to perform an end-to-end check on the data it sends. So if you transfer a file with TCP, do an end-to-end check that it arrived successfully. BitTorrent does this, for example. It uses TCP to transfer chunks, and after each chunk is complete it checks that it arrived successfully using a hash.

Example: Link Reliability

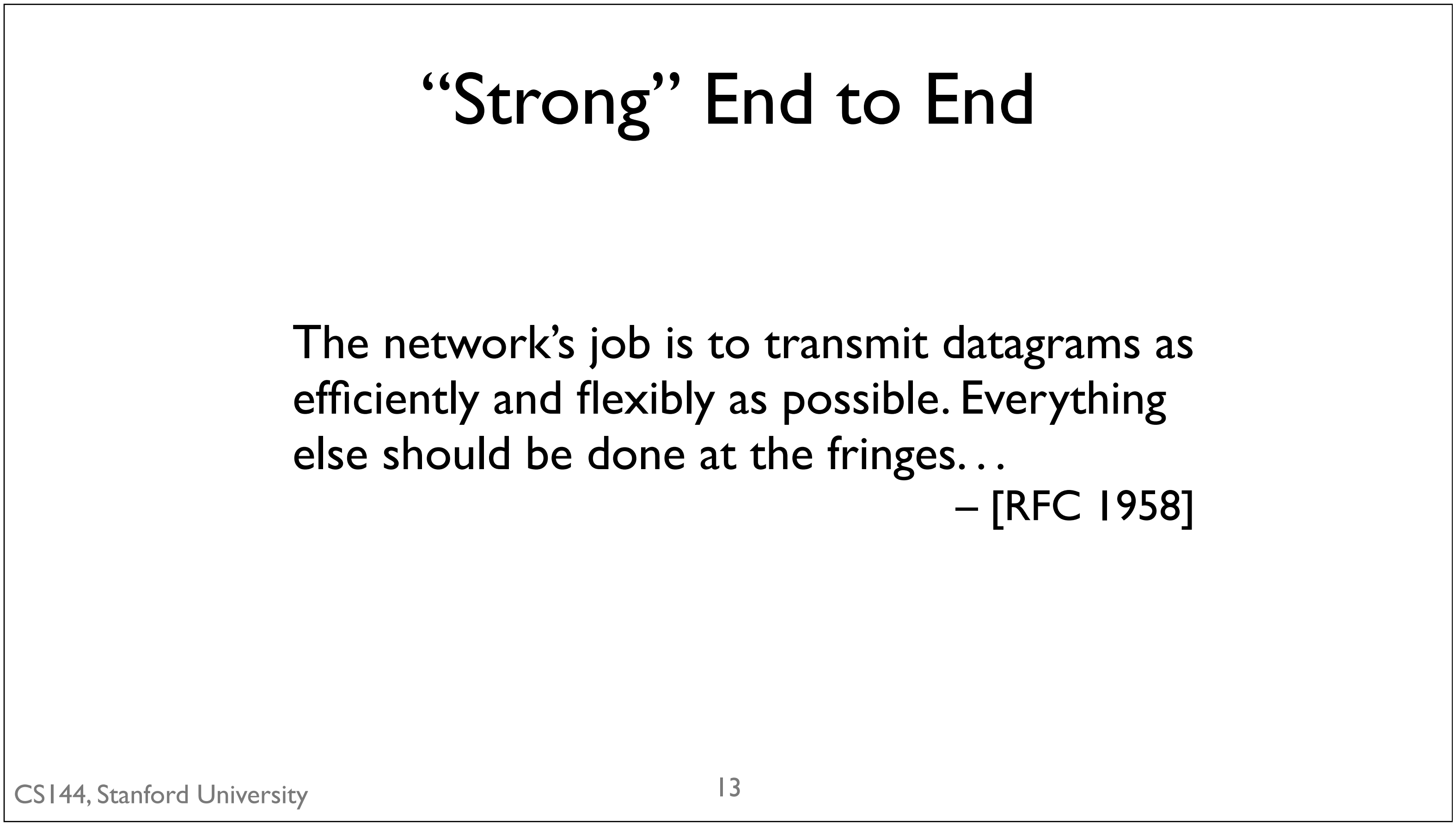
Phil: So let's go back to TCP and reliability. If you want end-to-end reliable data transfer, then you need an end-to-end reliable protocol like TCP. But following the end-to-end argument, while you must have end-to-end functionality for correctness, the network can include an incomplete version of the feature as a performance enhancement.

Wireless link layers provide such a performance enhancement. Today, wired link layers are highly, highly reliable (unless your wire or connector is bad). But wireless ones aren't, for a lot of reasons. So while usually 99.999% of packets sent on a wired link arrive successfully at the next hop, wireless links can sometimes be more like 50% or 80%.

It turns out that TCP doesn't work well when you have low reliability. So wireless link layers improve their reliability by retransmitting at the link layer. When your laptop sends a packet to an access point, if the access point receives the packet it immediately -- just a few microseconds later -- sends a link layer acknowledgement to tell your laptop the packet was received successfully. If the laptop doesn't receive a link layer acknowledgment, it retransmits. It does this several times. Using these link-layer acknowledgements can boost a poor link, with only 80% reliability, to 99% or higher. This lets TCP work much better.

TCP will work correctly -- it will reliably transfer data -- without this link layer help. But the link layer help greatly improves TCP's performance.

So that's the end-to-end principle. For something to work correctly, it has to be done end-to-end. You can do stuff in the middle to help as performance improvements, but if you don't rely on end-to-end then at some point it will break.



“Strong” End to End

> The network’s job is to transmit datagrams as efficiently and flexibly as possible. Everything else should be done at the fringes…
>
> – [RFC 1958]

Nick: There’s a second version of the end-to-end principle, described in the IETF request for comments number 1958, “The Architectural Principles of the Internet.” We call it the “strong” end-to-end principle. It says

“The network’s job is to transmit datagrams as efficiently and flexibly as possible. Everything else should be done at the fringes...”

This end-to-end principle is stronger than the first one. The first one said that you have to implement something end-to-end, at the fringes, but that you can also implement it in the middle for performance improvements. This principle says to NOT implement it in the middle. Only implement it at the fringes.

The reasoning for the strong principle is flexibility and simplicity. If the network implements a piece of functionality to try to help the endpoints, then it is assuming what the endpoints do. For example, when a wireless link layer uses retransmissions to improve reliability so TCP can work better, it’s assuming that the increased latency of the retransmissions is worth the reliability. This isn’t always true. There are protocols other than TCP, where reliability isn’t important, which might rather send a new, different, packet than retry sending an old one. But because the link layer has incorporated improved reliability, these other protocols are stuck with it. This can and does act as an impediment to innovation and progress. As layers start to add optimizations assuming what the layers above and below them do, it becomes harder and harder to redesign the layers. In the case of WiFi, it’s a link layer that assumes certain behavior at the network and transport layers. If you invent a new transport or network layer, it’s likely going to assume how WiFi behaves so it can perform well. In this way the network design becomes calcified and really hard to change.

In terms of long term design and network evolution, the strong end-to-end argument is tremendously valuable. The tension is that in terms of short term design and performance, network engineers and operators often don’t follow it. So over time the network performs better and better but becomes harder and harder to change.