Architecture: definitions

- A style and method of design and construction
- Orderly arrangement of parts
- The manner of construction of something and the disposition of its parts
- Design, the way components fits together
Architectural principles

• Many way to achieve a complex goals (such as - say - building a worldwide network of heterogeneous devices)

• **Guiding principles** are necessary, but it is difficult to get people to agree

• In this lecture we are going to discuss one of such principles: the end-to-end argument (e2e)
End-to-end arguments in system design

• 1981 paper by Jerry Saltzer, David Reed, Dave Clark

• MIT Laboratory for Computer Science

• All authors went on to make important contribution to the Internet design and architecture

• Heuristic for solving the problem of function placement among the modules of a distributed system

• Highly influential
The end-to-end argument

• Architectural principle

• Deals with *where* to place protocol functionality
  • Inside the network (i.e. in switches/routers), or
  • At the edges (endpoints)
The core principle

- If a function can only be fully and correctly implemented with the help of the applications standing at the endpoints of a communication system, providing that function as a feature of the communication system is not advisable.
Motivation

• Don't duplicate functionality at multiple levels if you have to do it at the top anyway

• Why? Two main reasons:
  • Duplicate functionality has a cost associate with it (e.g. reliable transport introduces delays and retransmissions)
  • Functionality provided at low level is forced upon all applications, even those which do not need it or would work better without it
Example 1

- File transfer application

- Components:
What could possibly go wrong?

Transmitter application

Disk storage

Network stack

Read rq

File data

Network link

File data

Network stack

Disk storage

Receiver application

Write rq

File data

Unexpected crashes

SW and/or HW malfunction

Disk malfunction

Channel errors (bit corruption, Packet loss, duplication, …)
How would you go about solving the problem?

Transmitter application

Disk storage

Network stack

Receiver application

Disk storage

Network stack

Network link

Read rq

File data

Write rq

File data

File data

File data

Disk malfunction

SW and/or HW malfunction

Unexpected crashes

Channel errors (bit corruption, Packet loss, duplication, …)
Just write correct programs!

SW and/or HW malfunction

Unexpected crashes

Redundant copies (e.g. RAID)

Disk malfunction

Network malfunction

Channel errors (bit corruption, Packet loss, duplication, …)

Packet checksums, retransmit lost packets, suppress duplicates
You could do all this… or…

Transmitter application

Disk storage → File data

Network stack

File data → Network link

Network link

Transmit hash together with file

Receiver application

File data

Disk storage

Verify that received file matches the hash

Compute file hash and store it separately from the file

Write rq

If hash does not match received file: try again!
Why is this a good idea?

- Guards against all malfunctions we have considered: errors are discovered regardless if they originate in faulty disks, SW/HW or network links.

- Simple and economical: requires no built-in redundancy and error correction in the system.

- If everything works correctly, most transfers will be completed at first attempt.

  - If a transfer require more than a couple of tries to be completed, there is probably an issue in the system.
Alternative approach: reliable network transmission

Ensure transmission errors are detected and corrected

Does not solve the problem: I still need to transmit and check the file hash to guard against all other possible errors
Another drawback of not respecting e2e

- False sense of security
- Why?
- MIT code repository example:
  - Result: OS source gradually corrupted!
Everything is relative

- Correctness checks in individual components do not solve the reliability issue...
  
  - But they may make sense as optimizations!

- Consider a file which is transmitted over 100 packets. Probability of packet error is $p$. What is the expected value $E[T]$ of the number of trials $T$ before the file is transmitted successfully, for various values of $p$?

<table>
<thead>
<tr>
<th>$p$</th>
<th>$E[T]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00001</td>
<td>~1</td>
</tr>
<tr>
<td>0.0001</td>
<td>~1</td>
</tr>
<tr>
<td>0.001</td>
<td>1.10</td>
</tr>
<tr>
<td>0.01</td>
<td>2.73</td>
</tr>
<tr>
<td>0.1</td>
<td>37648</td>
</tr>
</tbody>
</table>

\{ Acceptable \}
\{ Maybe acceptable? \}
\{ Definitely not acceptable! \}
What’s the lesson here?

• Violations of e2e, while not recommended in general, may make sense as **optimizations**

• In this example, a highly unreliable network may reduce the chance of successful transmission to negligible values

• It makes sense to introduce reliability as a network features (e.g. automatically retransmit corrupted packets)

• When is this acceptable? As always in engineering, the answer is “it depends” :-)

Delivery guarantees

• Should the network acknowledge message reception?

  • **Argument in favor:** increases reliability, message integrity etc.

  • **Argument against:** increases delay - network stack must wait for acknowledgement before further transmission

• What if app does not need it?

• Can you think of an example?
Delivery guarantees - II

• The argument against delivery guarantees in the paper is that the sender really cares about the **effect** of the packet, not its **reception**

• Distributed protocols, databases, etc. all incorporate some sort of explicit application-level commit - having a network-level acknowledgement is not enough!
The argument against delivery guarantees in the paper is that the sender really cares about the effect of the packet, not its reception.

Can you think why this consideration may be outdated in some cases?
Secure transmission of data

- Can you name a technology for true end-to-end data encryption?
  - IPsec - end-to-end-ish: encryption between IP endpoints, but not between applications
  - TLS?
Secure transmission - II

- Why is it a good idea to have e2e encryption?

- Why is it a good idea to have non-e2e encryption?

---

Google encrypts Gmail between data centers to keep the NSA out of your inbox

by Chrie Welch | Mar 30, 2014, 1:42pm EDT
Ordered message delivery

• May be difficult or impossible to provide within the network
  
  • E.g. packets may take different paths in the network due to transient routing failures
  
  • In that case, packets may arrive out of order - and there is little any entity participating in the communication can do
Duplicate message suppression

- Can you think of a reason why doing this in the network is not enough?
Nuances

- E2e specifies that functionality should be implemented at the endpoints - but it does not specify what endpoints are

- In certain cases, the choice of endpoints is non-obvious and has important consequences on the system design
What is an endpoint?

- Example: voice communication

- Where should error correction go?
What is an endpoint? - II

• This use case does not imply that you should always let the user take care of the problem :-)  

• Consider the case of a voicemail server which streams pre-recorded voice messages  
  • Error correction is probably a good idea there  

• **Lesson:** e2e cannot be applied blindly - designers must use some care in choosing the correct endpoints!
E2e in practice

• **Secure transmission**: good poster child for e2e - e.g. HTTP flows protected at application level using TLS

  • Although widespread use of VPNs is changing that

• Delivery guarantees, duplicate suppression, in-order delivery?

  • TCP provide most of those **below** the application level
E2e in practice - II

- The Internet protocol stack adopts what we may call an hybrid design
  - Delivery confirmation etc. provided at the endpoints, but below the application level
  - Turns out that those properties are pretty much necessary for many applications
  - What about applications for which packet-level errors are acceptable?
    - UDP
Not just for networking!

- E2e is often presented as a principle for distributed system design
- Which is interesting, because the paper presents several examples not related to distributed systems
- Do you remember which ones?
CPU design

- The old CISC vs RISC debate
  - CISC: complex functionality provided as processor instructions
  - RISC: processor only provides basic operation (conditional, arithmetic, etc.) - everything else build in software
OS design

- Monolithic OS vs microkernel
  - Monolithic: OS implements an extensive amount of functionality (networking, file system, etc.) in kernel space
  - Microkernel: OS provides isolation and interprocess communication, everything else is implemented outside the kernel (and does not affect its stability)

- Which one best implements the e2e idea?
Can we reformulate e2e in a non-networking specific way?

- *The correct level of abstraction at which to provide a function in a complex system is the level at which the most information about the function’s purpose is available*

- **CPU design**: complex instructions should be implemented in software, not in hardware

- **OS design**: all primitives but process isolation and IPC should be implemented by user space libraries, not the kernel

- **Compiler design**: program optimization should be performed by the compiler, not the programmer (“premature optimization is the root of all evil”, Knuth)

- **Network design**: high-level functionality should be implemented at the endpoints, not in the network core
A final consideration

• E2e is not a law - it is a heuristic

  • Heuristic: a technique serving as an aid to problem solving by experimental and especially trial-and-error method (Merriam-Webster dictionary)

• The paper we just discussed condenses lessons from years of experience in system design - but by no means this mean that this is always the correct approach

• E2e fosters simplicity and elegance of design, which are good things...

• But practical systems are often neither elegant not simple!
Occam’s razor

• Heuristic principle to help in scientific investigation (and any investigation, really)

• Often misquoted as “the simplest solution is most likely correct”

• It should actually be interpreted as “the simplest solution among all the plausible ones is most likely correct”

• Keep in mind that correctness > simplicity
Breaking e2e

Can you think of networking technologies that break e2e and yet are widely deployed?

- NAT
- Network monitoring
- In general, many of the so-called middleboxes
Other early Internet architecture goals

- D. D. Clark, “The Design Philosophy of the DARPA Internet Protocols”, SIGCOMM 1988

- Fundamental goal: connect a number of existing (and future) distinguishable, separately administered networks

- Basic design:
  - Packet-switched networks - allows multiplexed service
  - Store-and-forward gateways between component networks
Design principles: survivability

- Survivability
  - If any path exists, communication continues transparently
  - If ends die, all network state can be dropped: *fate sharing*
- Why is survivability a top concern?
Fate sharing

- Both concepts deal with the problem of state maintenance.

- Issue: oftentimes carrying a communication between two nodes requires maintaining state (think of TCP connections) that must be stored somewhere.

- In the network: reliability issue: what if network nodes (switches, routers) malfunction?

- In the host: the state is only lost if the host goes down - but this is acceptable, because implies that the state is lost together with the entity using it. This principle is called fate sharing.
Soft State

• What if I really really want to maintain some state in the network?
  • It is acceptable to maintain some communication state in the network…
  • …as long as it can be easily recreated if lost

• Idea: endpoints install state in the network; network can discard state at any time for any reason, in which case endpoints can reinstall it
  • Example: RSVP: receiver uses it to reserve resources across network; if network conditions change and state is lost, the receiver reinitialize the state
Implications

- What is the main implication of fate-sharing and soft state?

- Intermediate network links can fail without affecting communication between endpoints!

- In fact, there is no way for IP to report network malfunctions to upper layers
Design principle: hourglass design

- Modern computer networks are packet-switched and based on a stack of protocols which provide features at increasingly high level of abstraction.

- The hourglass design:

```
Does not get in the way of applications

IP provides datagram abstraction, on top of which more complex services can be build

Minimal assumptions about underlying medium
```

```
Application (high-level data stream)

Transport (reliable transmission of packets)

Network (routing across multiple networks)

Data-link (transmission of frames on link)

Physical (characteristics of physical signal)
```

```
edip email WWW phone SMTP HTTP RTP TCP UDP ethernet PPP CSMA async sonet copper fiber radio
```

IP: a datagram-based protocol

- IP implements the concept of packet-switched network - a radical departure from the (at the time) traditional circuit-switched networks:

- fixed size pipe from her to him
  - perfect for voice
  - reliable conversations (QoS)
  - provisioning, good engineering
  - dumb end points, smart network
  - evolved for 100 years (analog to digital)

Packet switching

differences:
⇒ packets as low-level component
⇒ multiple kinds of traffic
⇒ smart edges, dumb network

but:
⇒ QoS is much harder
⇒ end-points are more expensive
IP forwarding: store-and-forward

- Network elements store received packets in buffers and relay them on the appropriate output link

- Advantages:
  - Packets are self-contained units that can be manipulated by the network independently
  - Can use alternate link if standard link goes down
From NCP to TCP/IP

• Originally there was just one combined layer 3/4 protocol: NCP

• Later split into TCP/UDP and IP:
  • Varying needs in speed, latency, reliability
  • Not just bi-directional reliable connections

• TCP: provides reliable connection abstraction + congestion control
  • Retransmissions may introduce delay!

• UDP: basically a thin layer on top of IP
  • No reliability

• Useful for voice (one of the original applications of interest - why?)
Why was IP such a success?

- Clean, simple, flexible datagram abstraction
- Good, free implementation
  - BSD Unix in the mid-80s
- A good API
  - BSD socket API
  - Not perfect but “good enough”
- Compare to other APIs (e.g. file management) which are wildly different between OSes
Design principles: administrative separation

• The Internet was conceived as interconnection of pre-existent networks

• Although a clean-slate replacement with a single, consistent network paradigm was considered...
  • (Rejected as not practical)

• Because of this requirement, the Internet design trivially allows administrative separation of different segments (autonomous systems)
Other goals

• Accountability

• The original Internet was designed to provide army support even in case of total thermonuclear war

• Understandably, authentication, resource tracking etc. took a backseat!

• Basically, at network level the modern Internet still has little security (think of BGP hijacking)
Other goals - II

• Cost-effective
  • Today, quite cheap for computers/phones/smart devices
  • Effort to deploy end-host
    • In 1988, the main concern was the cost of implementing the TCP/IP stack
    • Today, mostly the cost of configuring and managing machine
      • DHCP & other autoconf protocols help…
      • …but oftentimes some manual tweaks are still required
Enjoy the weekend!

- ... and remember that the review is due on Sunday night!