Review of Networking and Design Concepts (II)

Two ways of constructing a software design:

1) *make it so simple that there are obviously no deficiencies, and*
2) *make it so complicated that there are no obvious deficiencies*

--- CAR Hoare

Based in part upon slides of Prof. Raj Jain (OSU), J. Kurose (U Mass), I. Stoica, A.Joseph (UCB)

Rensselaer Polytechnic Institute
Shivkumar Kalyanaraman

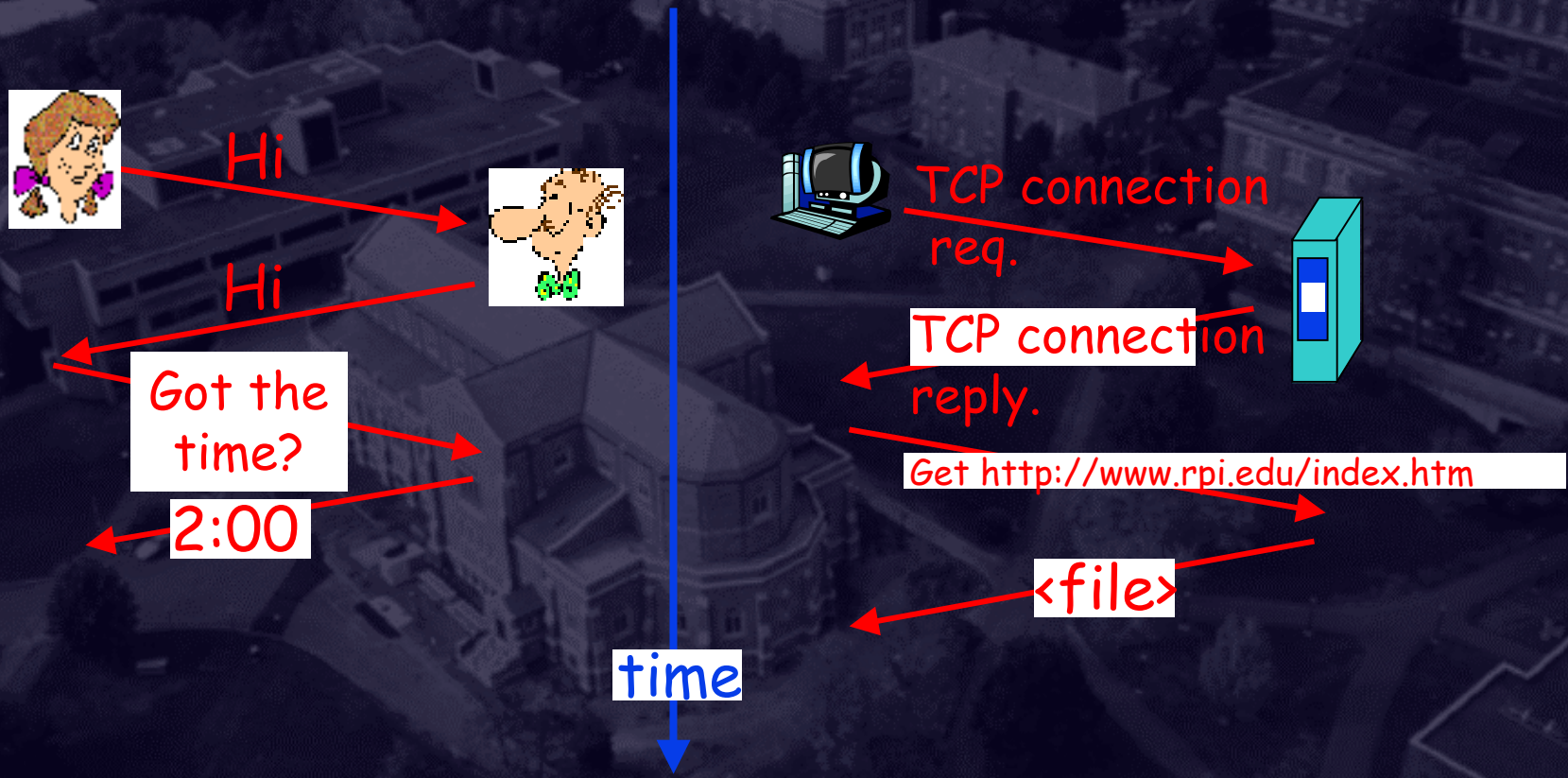
Overview

- Protocols, layering, encapsulation
- Function-placement: End-to-end principle
- Implementation: App-layer framing, ILF
- Interface design: functionality, technology, performance
- Rules of thumb in system design
- Chapter 1,2,11 in Doug Comer book
- **Reading**: Saltzer, Reed, Clark: "End-to-End arguments in System Design"
- **Reading:** Clark: "The Design Philosophy of the DARPA Internet Protocols":
- **Reading:** RFC 2775: Internet Transparency: In HTML

Protocols

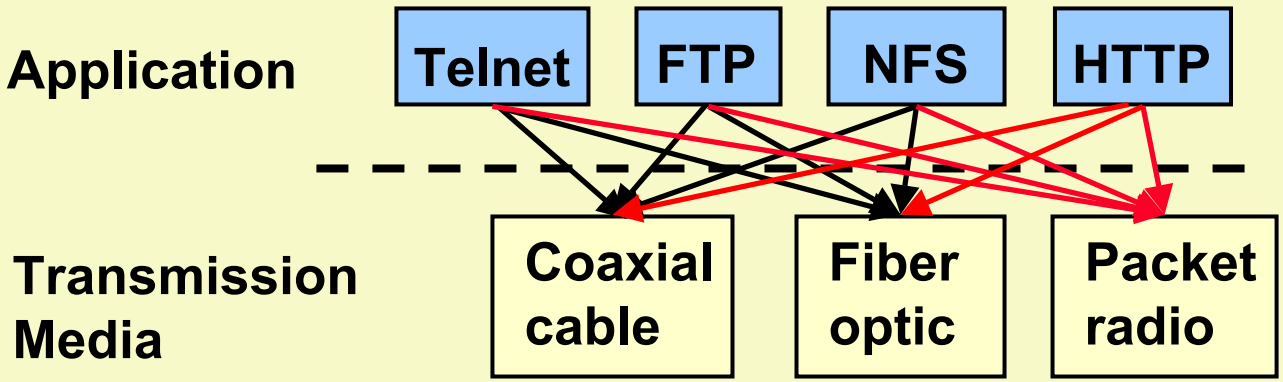
Human protocol vs Computer network protocol:

A series of functions performed at different locations.



Why Layering?

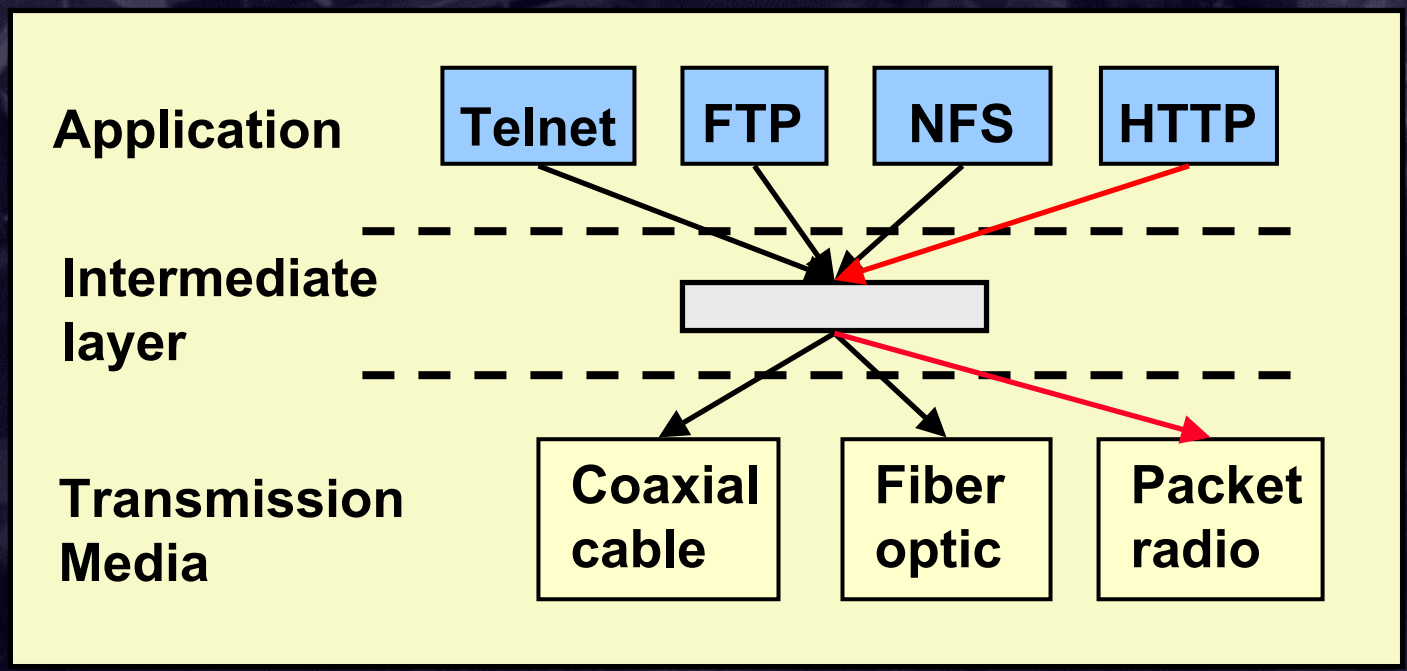
(FTP – File Transfer Protocol, NFS – Network File Transfer, HTTP – World Wide Web protocol)



- No layering: each new application has to be *re*-implemented for every network technology!

Why Layering?

- Solution: introduce an intermediate layer that provides a unique abstraction for various network technologies



What is Layering?

- A technique to organize a network system into a succession of logically distinct entities, such that the service provided by one entity is solely based on the service provided by the previous (lower level) entity

Layering

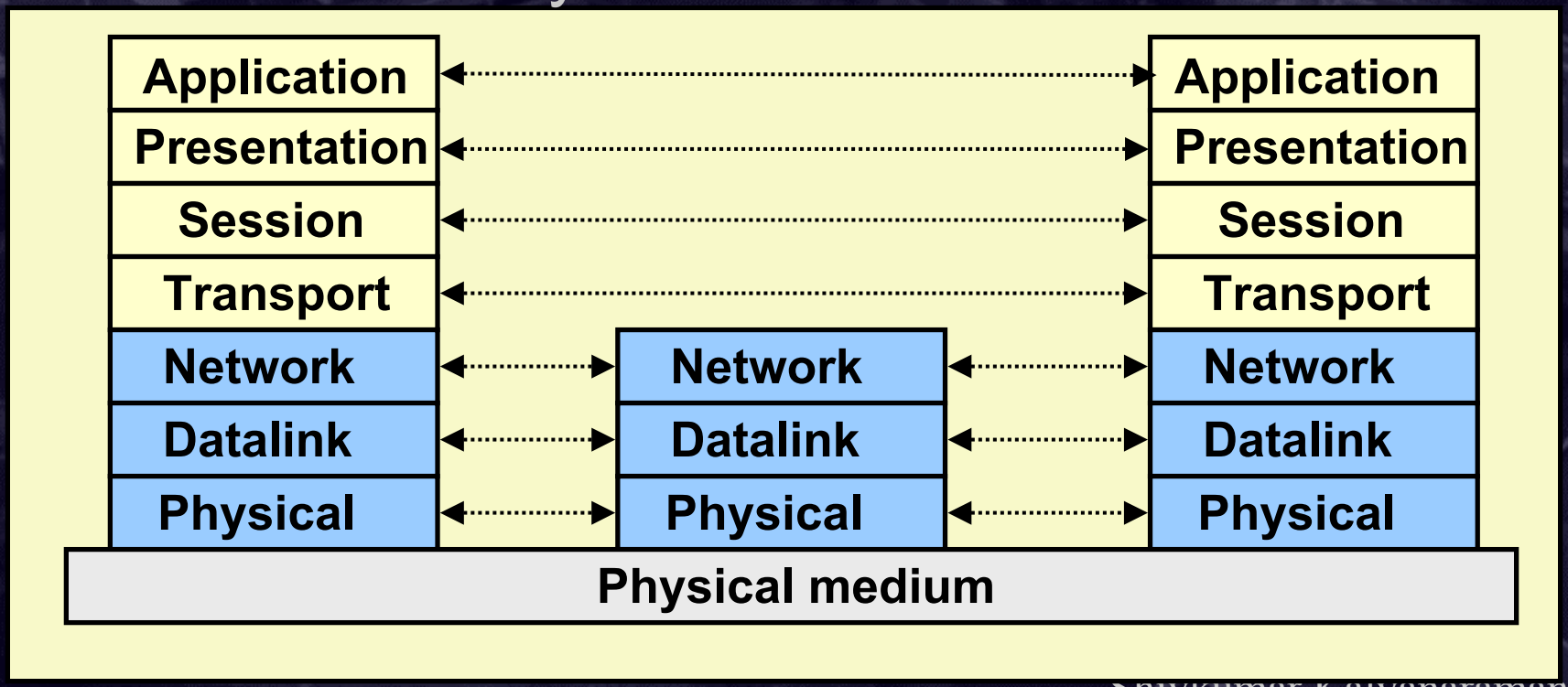
- Advantages
  - Modularity – protocols easier to manage and maintain
  - Abstract functionality –lower layers can be changed without affecting the upper layers
  - Reuse – upper layers can reuse the functionality provided by lower layers
- Disadvantages
  - Information hiding – inefficient implementations

Protocols

- Building blocks of a network architecture
- Each protocol object has two different interfaces
  - *service interface*: defines operations on this protocol
  - *peer-to-peer interface*: defines messages exchanged with peer

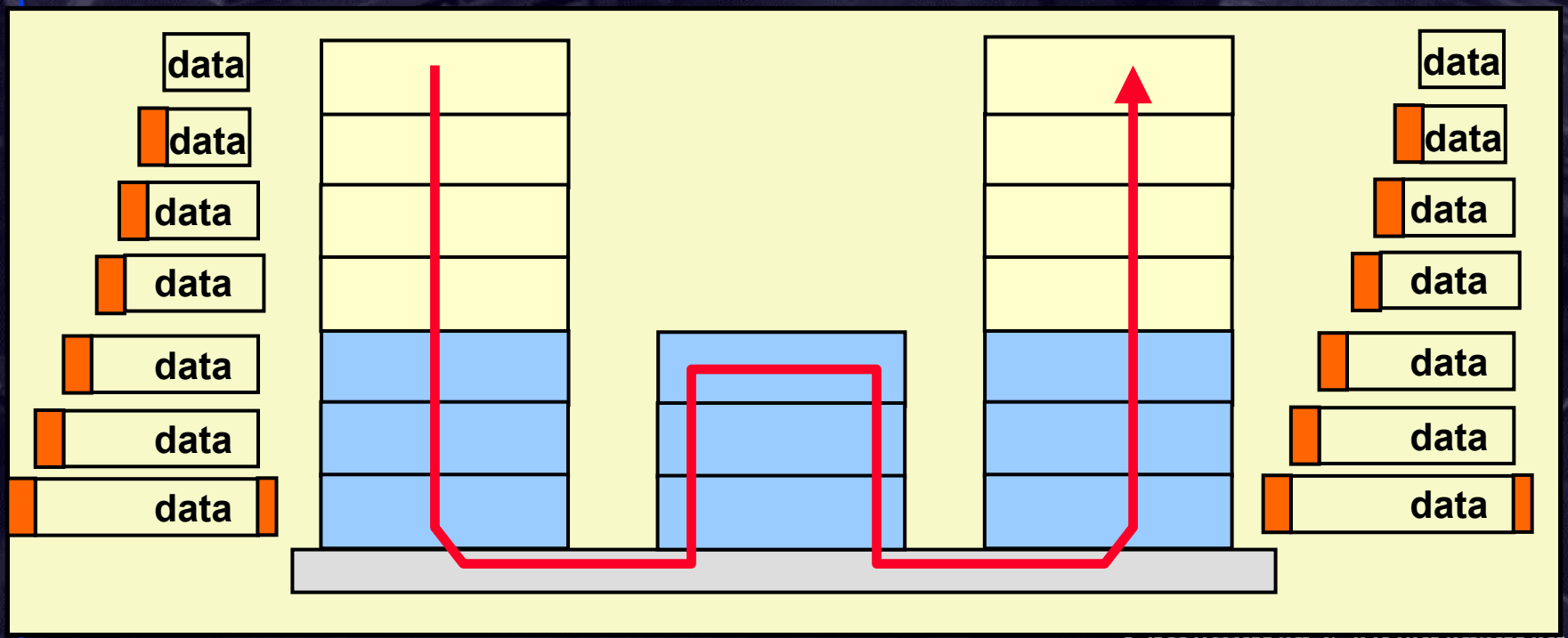
ISO OSI Reference Model

- Seven layers
  - Lower three layers are peer-to-peer
  - Next four layers are end-to-end



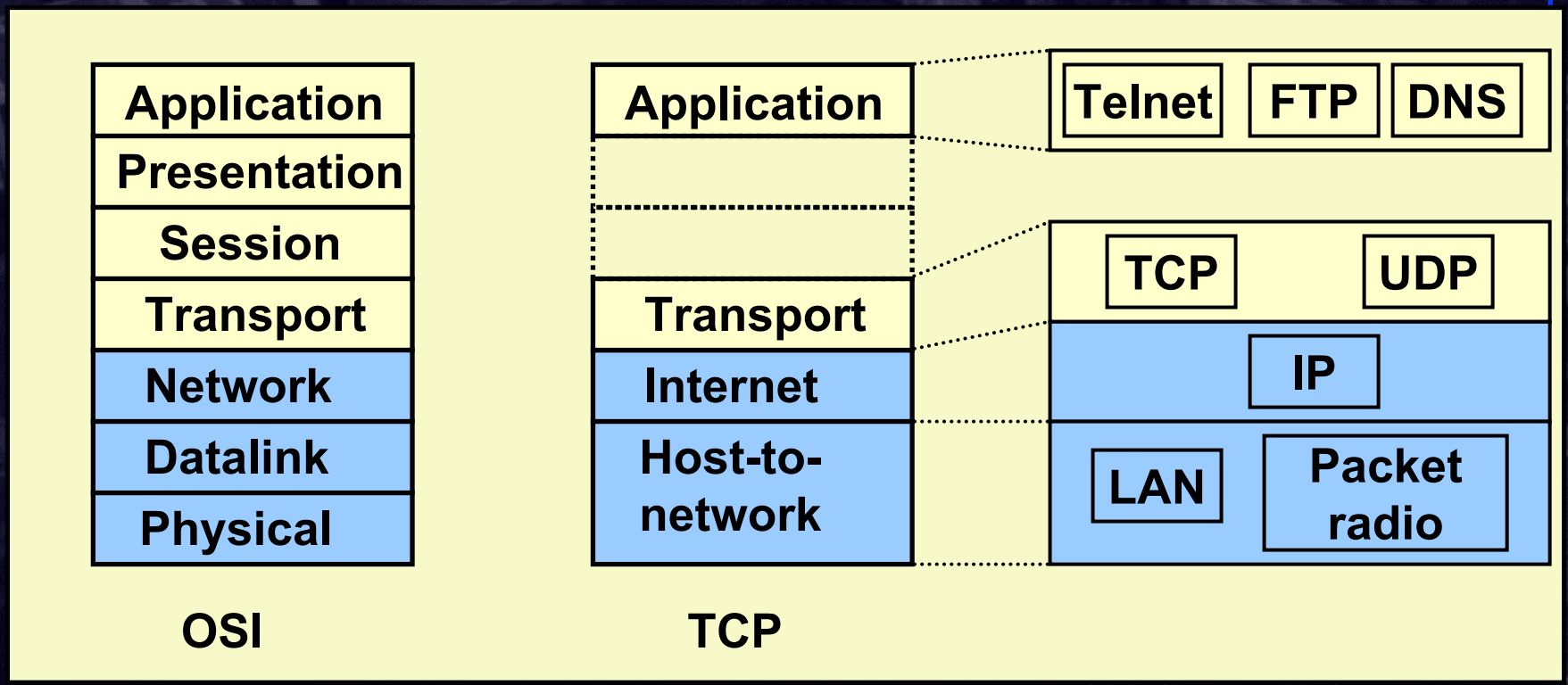
Encapsulation

- A layer can use only the service provided by the layer immediate below it
- Each layer may change and add a header to data packet



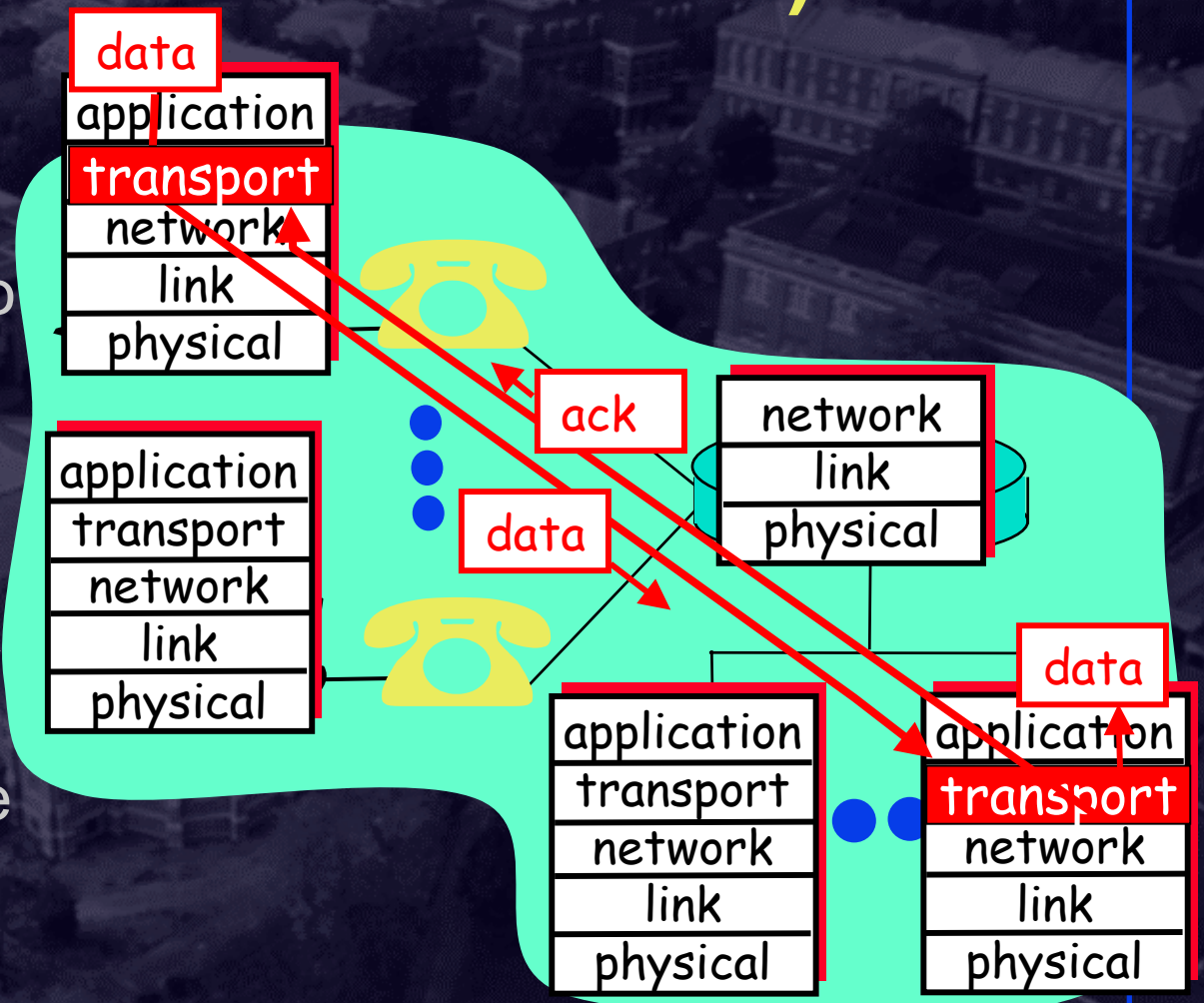
OSI vs. TCP/IP

- OSI: conceptually define services, interfaces, protocols
- Internet: provide a successful implementation



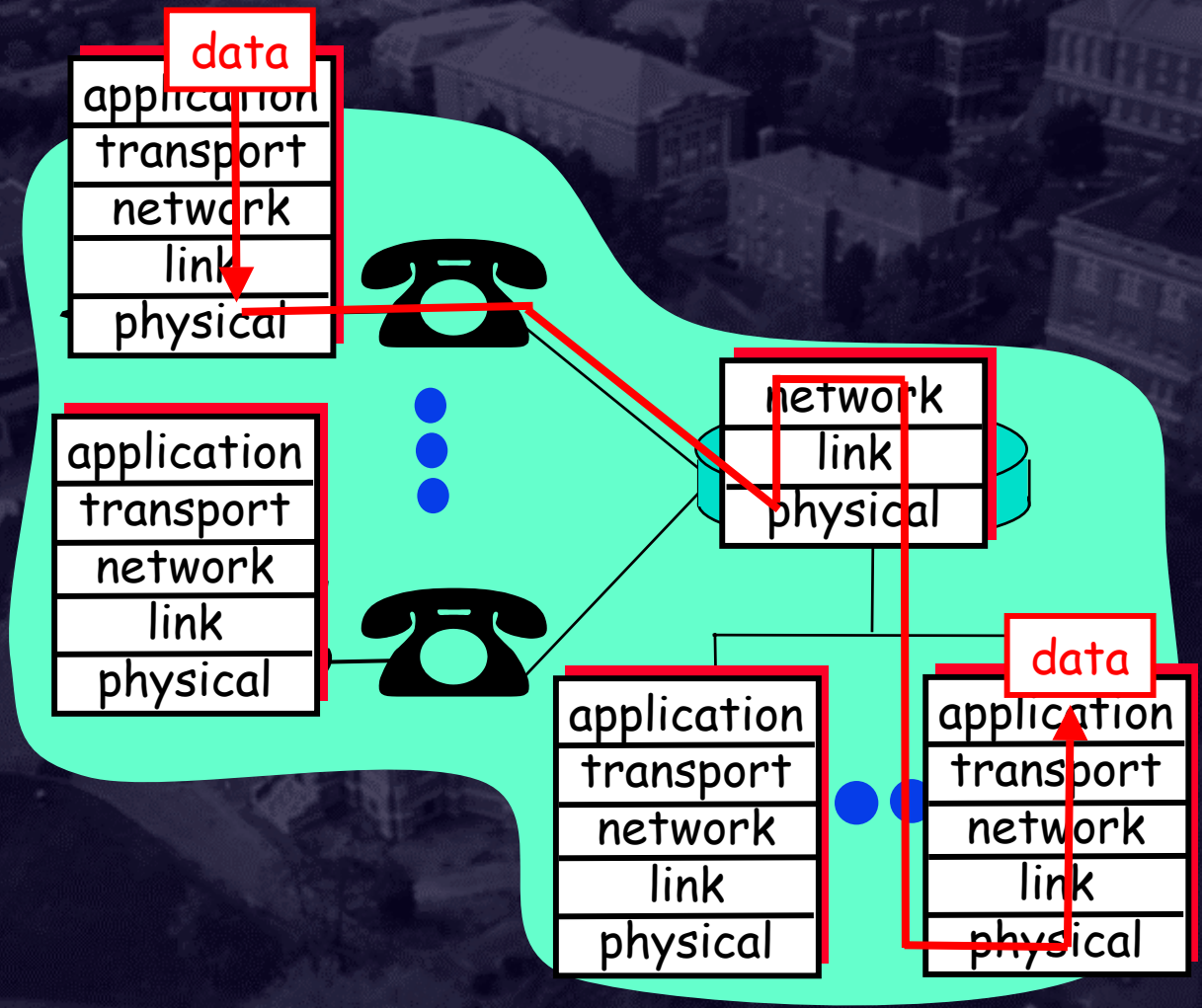
Example: Transport Protocol (Logical Communication)

- take data from app
- add addressing, reliability check info to form "datagram"
- send datagram to peer
- wait for peer to ack receipt
- analogy: post office



(Source: Kurose & Ross)

Example: Transport Protocol (Physical Communication)

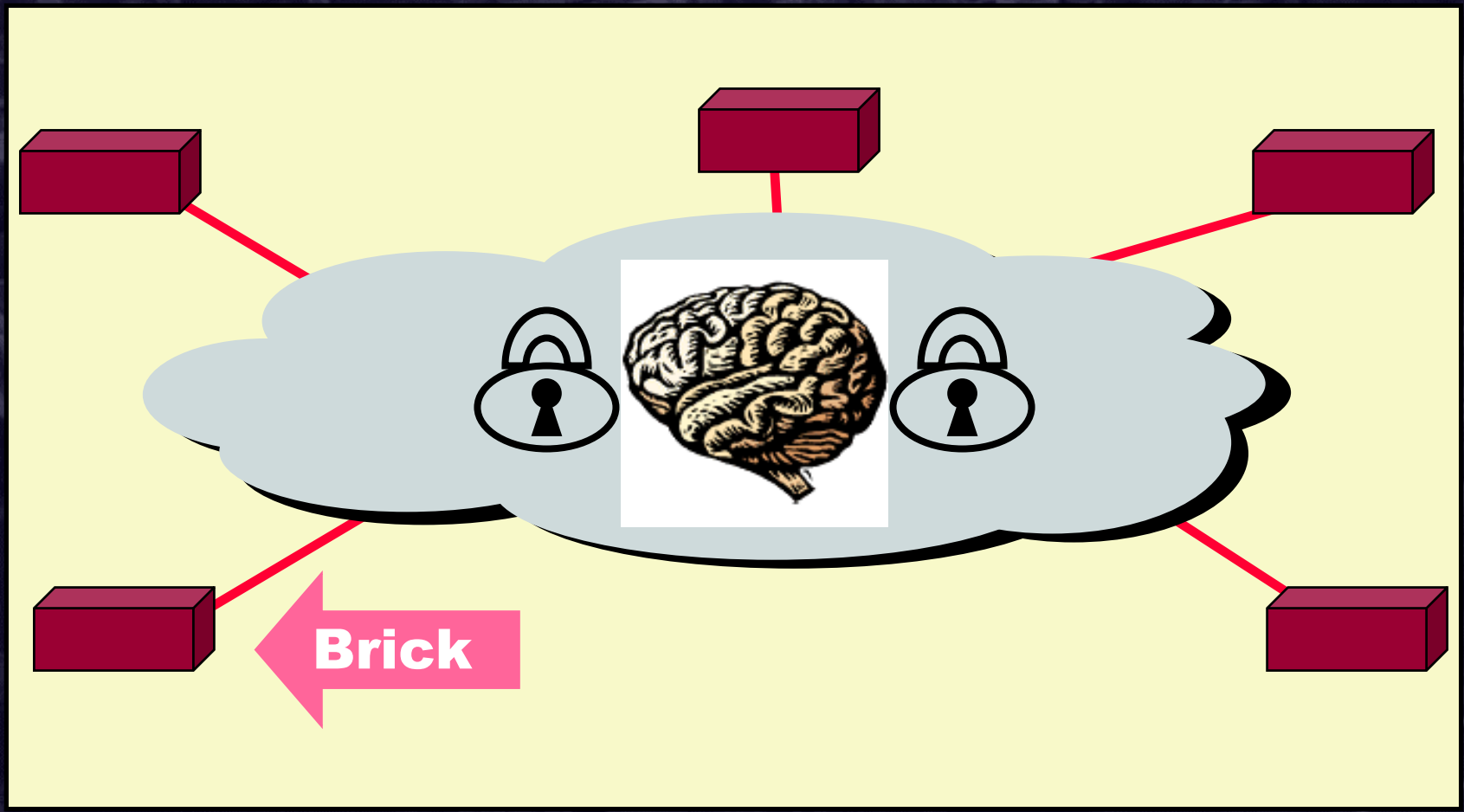


(Source: Kurose & Ross)

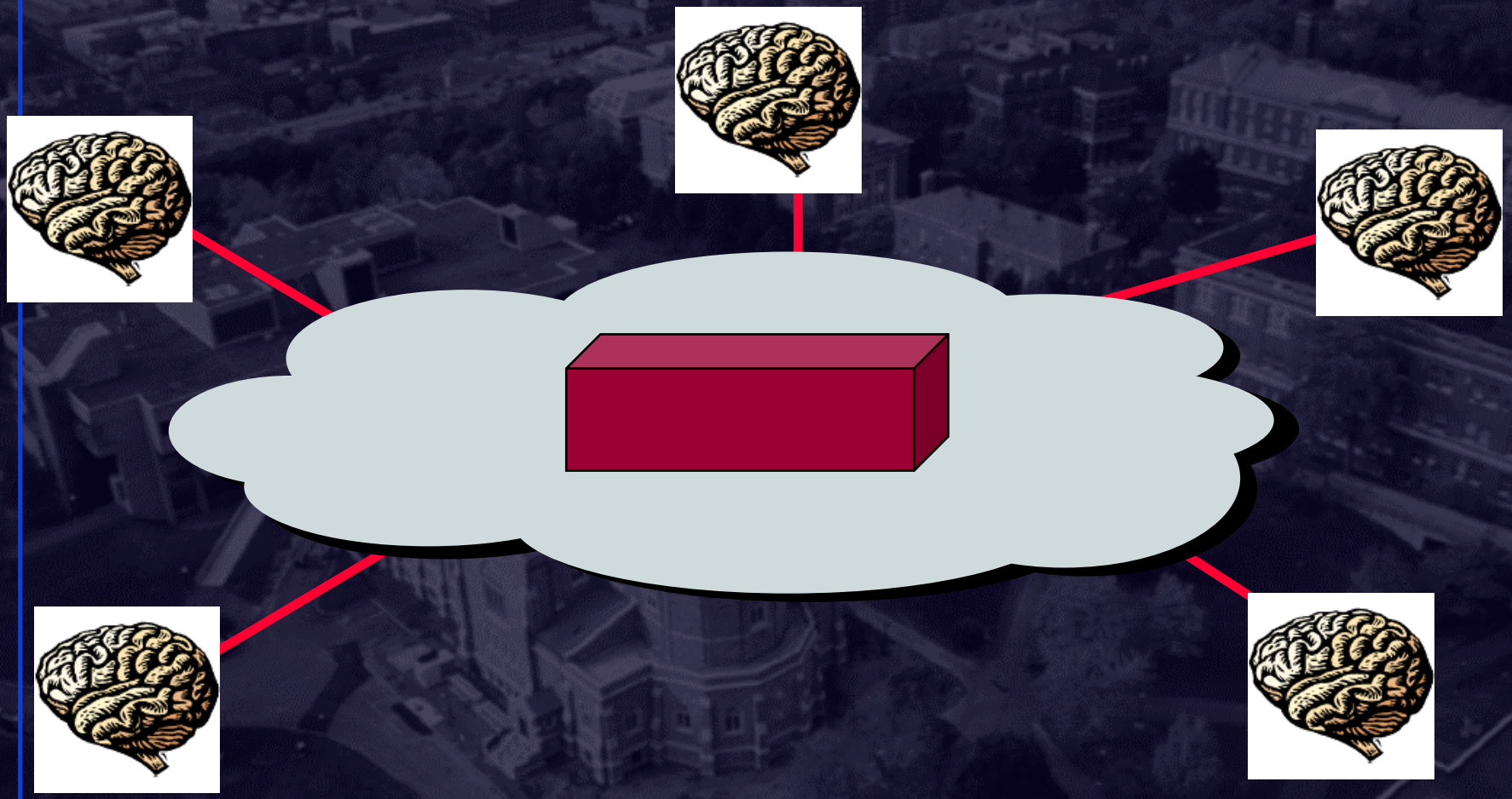
Key questions

- How to decompose the complex system functionality into protocol layers?
- What functions to be placed at which levels?
- Can a function be placed at multiple levels ?

Common View of the Telco Network



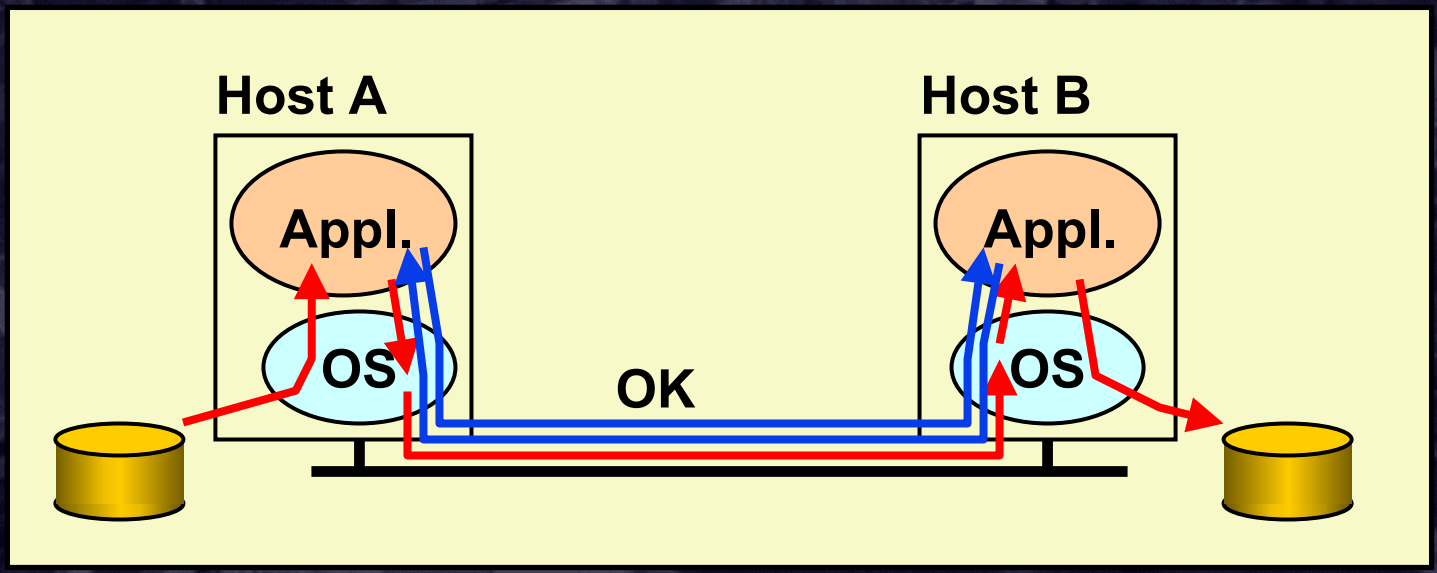
Common View of the IP Network



End-to-End Argument

- *"…functions placed at the lower levels may be redundant or of little value when compared to the cost of providing them at the lower level…"*
- *"…sometimes an incomplete version of the function provided by the communication system (lower levels) may be useful as a performance enhancement…"*
- This leads to a philosophy diametrically opposite to the telephone world which sports dumb end-systems (the telephone) and intelligent networks.

Example: Reliable File Transfer



- Solution 1: make each step reliable, and then concatenate them
- Solution 2: end-to-end check and retry

Discussion

- Solution 1 not complete
  - What happens if the sender or/and receiver misbehave?
- The receiver has to do the check anyway!
- Thus, full functionality can be entirely implemented at application layer; no need for reliability from lower layers
- Is there any need to implement reliability at lower layers?

Discussion

- Yes, but only to improve performance
- Example:
  - assume a high error rate on communication network
  - then, a reliable communication service at datalink layer might help

Trade-offs

- Application has more information about the data and the semantic of the service it requires (e.g., can check only at the end of each data unit)
- A lower layer has more information about constraints in data transmission (e.g., packet size, error rate)

- Note: these trade-offs are a direct result of layering!

Internet & End-to-End Argument

- At network layer provides one simple service: best effort datagram (packet) delivery
- Only one higher level service implemented at transport layer: reliable data delivery (TCP)
  - performance enhancement; used by a large variety of applications (Telnet, FTP, HTTP)
  - does not impact other applications (can use UDP)
- Everything else implemented at application level

Key Advantages

- The IP service can be implemented on top of a large variety of network technologies
- Does not require routers to maintain any fined grained state about traffic. Thus, network architecture is
  - Robust
  - Scalable

What is a "level" of a system?

- Protocol "layer" = "level"
- Within a single layer, closer to the core => "lower" level
  - Eg: Edge-boxes of a domain implementing functions like firewalls, address translation, QoS functions are at a "lower" level compared to other boxes in the domain
  - Core router is "lower" level compared to an "edge router"
  - In hierarchical routing, use of smaller prefixes correspond to lower levels of the system.

E2E Argument: Interpretations

- One interpretation: (limited in my opinion…)
  - A function can only be completely and correctly implemented with the knowledge and help of the applications *standing at the communication endpoints*
- Another: (more precise…)
  - a system (or subsystem level) should consider only functions that can be *completely and correctly* implemented within it.
- Alternative interpretation: (also correct …)
  - Think twice before implementing a functionality that you believe that is useful to an application at a lower layer
  - If the application can implement a functionality correctly, implement it a lower layer only as a performance enhancement

End-to-End Argument: Critical Issues

- The end-to-end principle emphasizes:
  - function placement
  - correctness
  - completeness and
  - overall system costs.
- It allows a cost-performance tradeoff
- If implementation of function in higher levels is not possible due to technological/economic reasons (eg: telephone network in early 1900s), then it may be placed at lower levels

Summary: End-to-End Arguments

- If the application can do it, don't do it at a lower layer -- anyway the application knows the best what it needs
  - add functionality in lower layers iff it is (1) used and improves performances of a large number of applications, and (2) does not hurt other applications
- Success story: Internet

Architecture vs Implementation: ALF Principle

- Architecture: decomposition into functional modules, semantics of modules and syntax used
- There should be no a priori requirement that the engineering design of a given system correspond to the architectural decomposition
  - Eg: layering may not be most effective modularity for implementation
- Summary:
  - Flexible decomposition
  - Defer engineering decisions to implementor.
  - Avoid gratuitous implementation constraints
  - Maximize engineering options for customization/optimization

Application Layer Framing (ALF)

- Several processing bottlenecks may lie at the "presentation" layer which does not really exist in the TCP/IP stack
  - These functions are absorbed partially in the transport layer and partly in the application layer.
  - Principle: the application-layer should have control of the syntax and semantics of the presentation conversions
  - Transport should provide only common functions

- Generalization of ALF: look for elegant ways to allow application visibility/participation in lower-level activities
  - Eg: QoS – carry application intelligence to the point of QoS enforcement

ILP: Integrated Layer Processing

- Motivation: ever-widening memory / CPU bottleneck
- "Integrated processing loop"
  - Loop over bytes in packet
  - Touch each byte at most once
  - Avoid multiple copies within memory
  - Massive integrated loop w/ all steps in-line
  - Trivial example: bcopy + checksum
- Architecture must minimize gratuitous precedence or ordering constraints …

Eg: Real-Time Protocol

- RTP svcs: payload type identification, sequence numbering, timestamping and delivery monitoring
- "*RTP is intended to be malleable to provide the information required by a particular application and will often be integrated into the application processing rather than being implemented as a separate layer.*"
- RTP is a protocol framework that is deliberately not complete and can be tailored modifications/additions to the headers.
  - RTP specifies only common functions for its apps
  - Avoid taking on additional functions
    - making the protocol more general or
    - Adding options requiring expensive parsing

Interface Design

- Driven by three factors:
  - ***Functionality***: what features the customer wants, and is placed at a level due to e2e principle etc
  - ***Technology***: what's possible. Building blocks and techniques
  - ***Performance***: How fast etc… User, Designer, Operator views of performance ..

- Interface design crucial because *interface outlives the technology* used to implement the interface.

Performance

- **Performance questions:**
  - Absolute: *How fast …*
  - Relative: Is A *faster than* B and *how much faster*?
- **Define system as a black box.**
  - Parameters: input; Metrics: output
- Parameters: only those the system is *sensitive to*
- Metrics: must *reflect the system design tradeoff*

Effect on Design: *Amdahl's law*

- Performance after improvement =

  Performance ***affected*** by improvement / speedup

  + *Unaffected* performance

  - ***Lesson:*** *Speedup the common case I.e. the parts that matter most* !!

- *Amdahl's law guides the definition of tradeoffs, parameters, test cases and metrics !*

Perspectives on Performance/Design

- ***Network users:*** services and performance that their *applications need*,
- ***Network designers:*** *cost-effective design*
- ***Network providers:*** system that is easy to *administer and manage*
  - Need to balance these three needs

System Design Rules of Thumb

- Design a system to ***tradeoff cheaper resources against expensive ones*** (for a gain)
- When resources are cheap and abundant, waste them. Design focuses on cutting out any expensive resource that comes in the way! (eg: parallelism)
- Filtering => Efficiency => Scalability
- Apply principles like E2E and ALF to decide on right placement of functionalities in different system levels
- Interfaces must outlive several generations of change in the components being interfaced.
  - Three factors drive interface design:
    - functionality demanded,
    - available technology,
    - performance tradeoff.

- Functionality requirements can be understood by taking different views of the system (eg: designer, implementor, operator).
  - Reduced functionality can result in cheaper, scalable, quickly engineered system
  - Placement of functionality is critical in system design
- No paradigm is going to work or functionality can be met if the available technology to implement it does not exist.
- Performance is either relative or absolute and is usually modeled at the high level as a function from system parameters (input) to system metrics (output).
  - Metrics must be design to reflect design tradeoffs.
  - Only sensitive parameters matter.
- Optimize the common case (Amdahl's law)
  - Solve 90% of the problem that matters, throw away the remaining 10% of the problem requirements!