Exactly-once Semantics

(theoretical ideal)

How long to keep old replies and sequence numbers?

Rigorous interpretation of "RPC" → forever!

Across server crashes too

- they have to be saved in non-volatile memory
- server response has to be after non-volatile write
- disk (or flash) latency on every RPC
- *clean undo of partial computations* before crash

Such an RPC would have *exactly-once* semantics

- success return → call executed exactly once
- call blocks indefinitely, no failure return

Not appropriate in many use cases

- severe performance penalty
  synchronous disk writes
- indefinite blocking
  precludes app-aware recovery
- needs transactional semantics
  all server operations

At-most-once Semantics

(practically achievable)

Avoid indefinite blocking

Declare *timeout* beyond certain delay

Such RPC has *at-most-once* semantics

- refers to worst case inference
- success $\rightarrow$ call executed exactly once
- timeout $\rightarrow$ call executed once or never

Many possible reasons for RPC timeout

1. request and retries never got to server
2. server died while working
3. network broke while server working
4. server still working
5. server replied, but reply lost
6. many server resends, but all lost
7. …

Slow Servers & Long-Running Calls

Servers don’t give progress reports

- **they reply only when work on request is complete**
- **can’t tell if server is unreachable or making progress on request**
- **complicates setting of timeout value**

Solution

- **probes to check server health during long calls**
- **server responds with *busy* if still working**
- **essentially a *keepalive* mechanism**

Orphaned Computations

Inherent danger with at-most-once semantics

- **client sends request, server starts computing**
- **network failure occurs**
- **server continues, unaware its work is useless**

  server may hold resources (e.g. locks), slowing other activity

Orphan detection **and** ***extermination*** **are difficult**

typically require application-specific recovery

"Failure" closely related to "timeout value"

- **fundamental limitation in a distributed system**
- **due to absence of** ***out-of-band error detection***

  can't tell server death from network failure

At-least-once semantics

Even simpler to implement

Requires *operation idempotency*

Example: read( ) request on locked object or read-only object

Example: current_stock_price (MSFT)

semantics of operation allows any valid price after issue of request to be returned

Implementation can be completely *stateless* on server side

even sequence numbering to detect duplication can be avoided

Choice of Semantics

Achieving exactly-once semantics

- **not provided by any real RPC package**
- **requires application-level duplicate elimination**
- **built on top of at-most-once RPC**

Most RPC packages provide at-most-once RPC

At-most-once semantics avoids

- **transactional storage**
- **non-volatile storage of replies and sequence #s**
- **indefinite storage of replies**

exactly-once > at-most-once > at-least-once

where “>” is “stronger than” and “<“ is “weaker than”

Safety and Liveness Properties

Safety property: characterizes correct functionality
(e.g., "At most one entity can execute in a critical section")

Together, safety properties confirm that *"bad things never happen"*

Liveness property: characterizes timely execution progress
(e.g., "This code is deadlock-free")
(e.g., "This code is starvation-free")

Together, liveness properties confirm that *"good things will eventually happen"*

"Exactly-once semantics" $\rightarrow$ safety property

Existence of timeout in at-most-once RPC $\rightarrow$ liveness property

Real-world design has to balance safety and liveness

Placement of Functionality

What are you promising versus what can you deliver?

aka "Protocol Layering"

What Can TCP Do For RPC?

this is what you are doing in Project-1

TCP timeout → we still have to reconnect

- new TCP connection unaware of old
  → server must keep higher level sequence #s
- server must do duplicate elimination
- orphans still possible
- *exactly-once RPC no easier with TCP*

TCP can simplify at-most-once RPC

- if timeouts rare, out-of-band handling ok (e.g. phone call)
  avoids need to implement retransmission and duplicate elimination
- absence of TCP failure ⇒ exactly once RPC
- on TCP failure, declare RPC failure (at most once RPC)
  higher level (e.g. user) invokes out of band mechanism to handle failure

TCP Hurts Best-Case RPC Performance

TCP uses independent acks in each direction

- client → server : request; server → client: ack
- server → client: reply; client → server: ack
- 4 packets for best case

But RPC on UDP only uses 2 packets: *reply is an implicit ack*

End-to-End Reasoning

Layering RPC transport on TCP rather than UDP

1. **does not simplify exactly-once implementation**
2. **worsens performance in best case**
3. **does simplify at-most once implementation**

This is an example of an *end-to-end argument*

- **most important structuring principle for distributed systems**
- **helps designer place functionality in modules**

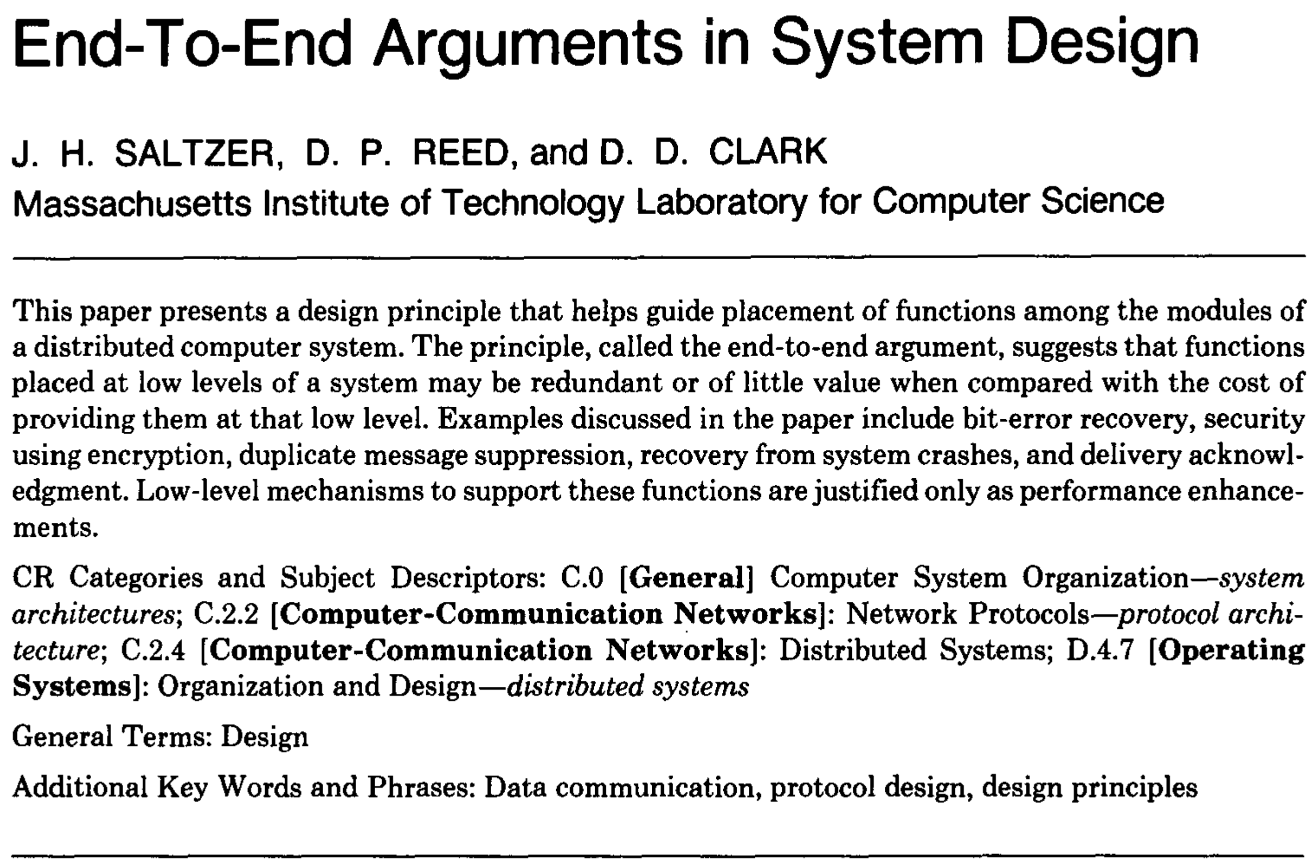
For given functionality (typically pertaining to safety)

1. ***correctness is expressed relative to two endpoints*** **(safety)**
2. ***implementation requires support of those two end points***
3. ***support below end points cannot suffice***

   at best, lower-level support may improve performance (liveness)

Optional Reading

(on course web site)



End-To-End Arguments in System Design

J. H. SALTZER, D. P. REED, and D. D. CLARK
Massachusetts Institute of Technology Laboratory for Computer Science

This paper presents a design principle that helps guide placement of functions among the modules of a distributed computer system. The principle, called the end-to-end argument, suggests that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level. Examples discussed in the paper include bit-error recovery, security using encryption, duplicate message suppression, recovery from system crashes, and delivery acknowledgment. Low-level mechanisms to support these functions are justified only as performance enhancements.

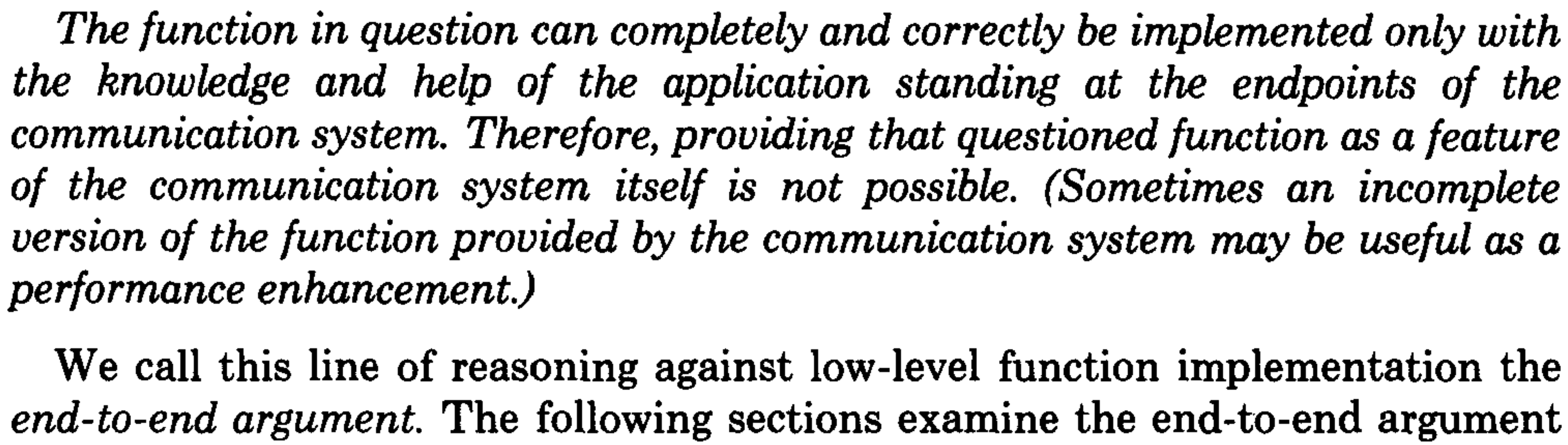
CR Categories and Subject Descriptors: C.0 [**General**] Computer System Organization—*system architectures*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Design

Additional Key Words and Phrases: Data communication, protocol design, design principles

ACM Transactions on Computer Systems,
Vol. 2, No. 4, November 1984

Statement of the Principle

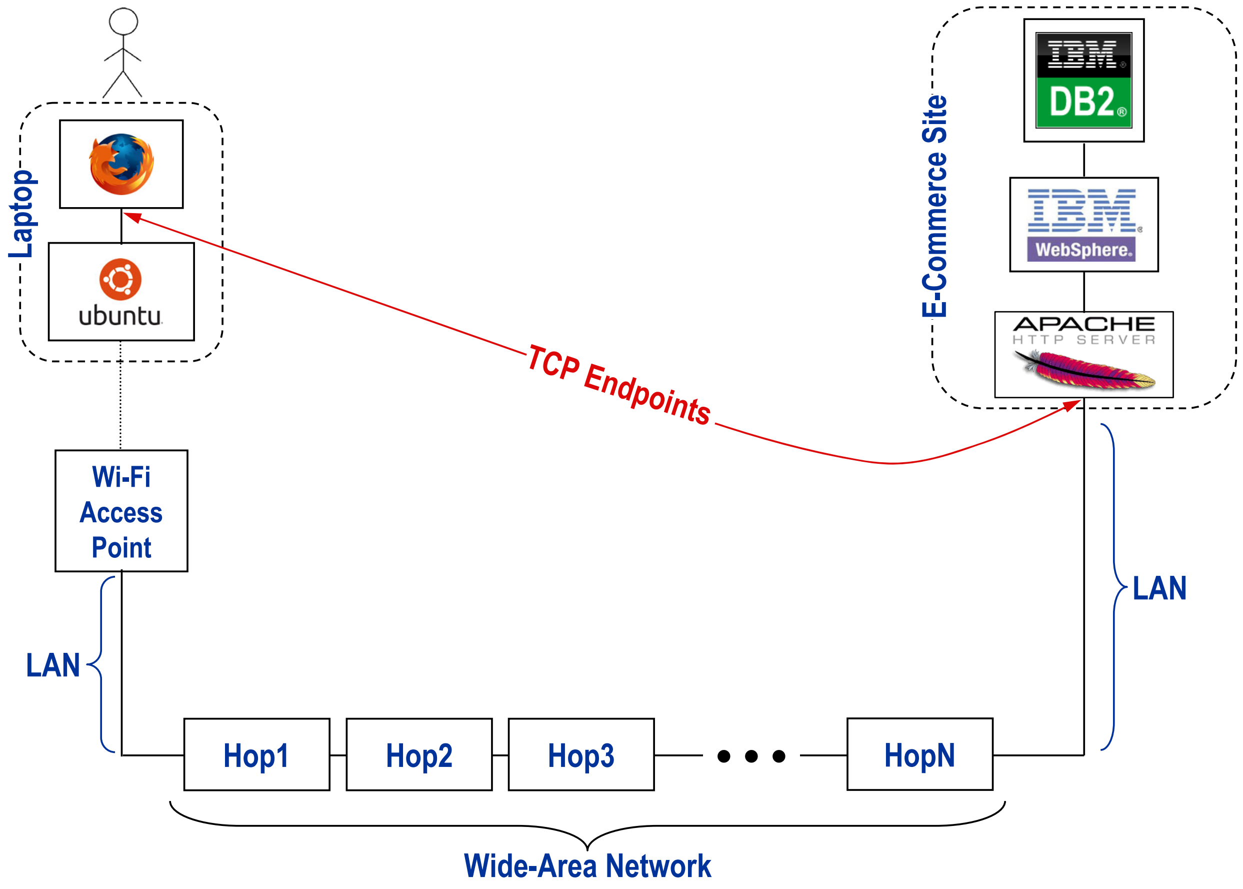


The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the endpoints of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)

We call this line of reasoning against low-level function implementation the *end-to-end argument*. The following sections examine the end-to-end argument

Answers critical question: ***Where to place a specific function in a distributed system?***

AKA: ***Who should have responsibility?***


Laptop
ubuntu
TCP Endpoints
E-Commerce Site
IBM DB2
IBM WebSphere
APACHE HTTP SERVER
Wi-Fi Access Point
LAN
LAN
Hop1
Hop2
Hop3
HopN
Wide-Area Network