THE ROLE OF BROAD ARCHITECTURAL PRINCIPLES IN SYSTEMS

CS6410

Ken Birman

Creation of a large system

- A complex undertaking
- Researchers often get to define the goals and assumptions at the same time as they architect the solution:
  - Many areas completely lack standards or prior systems
  - Many standards are completely ignored
  - Many widely adopted systems depart from the relevant standards
- Are there overarching goals that all systems share?

Candidate goals

- All systems should strive for the best possible performance given what they are trying to do
  - But of course the aspects of performance one measures will depend on the use case(s) envisioned
  - Aiming for performance in a way that ignores use cases can yield a misleading conclusion
- A system should be an “elegant” expression of the desired solutions and mechanisms
  - Puzzle: What metrics capture the notion of elegance?

First steps in the design process

- Developers often work in an iterative way
  - Identify and, if possible, separate major considerations
  - Pin down the nature of the opportunity they see, and from this refine their goals and assumptions
  - Eventually, begin to conceive of system in terms of an architectural block diagram with (more or less) well-defined components and roles for each
- Walking through the main code paths may lead to redesigns that aim at optimizing for main use cases

Critical-path driven process

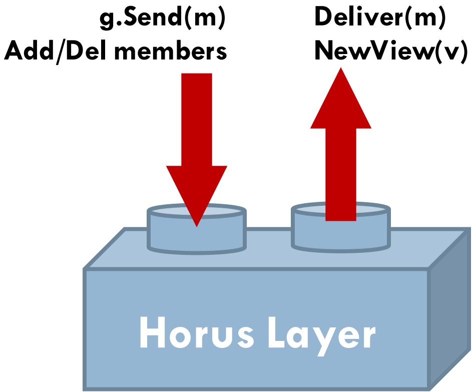
- If we can identify common patterns or use cases a-priori (or perhaps by analysis of workloads from other similar systems for which data exists)…
  - Permits us to recognize in advance that particular code paths will arise often and will really determine performance for the metrics of primary interest
  - In effect we can "distort" our design to support very short critical paths at the expense of shifting functionality elsewhere, off the critical path
- This sometimes permits us to use less optimized logic off the critical path without fear of huge performance hits

Example: Van Renesse (Horus)

- Robbert was developing a fast multicast system
  - Core functionality: Reliable multicast from some sender to some set of receivers
  - The particular system, Horus, implements the same virtual synchrony model we discussed last week
- Virtual synchrony platforms inevitably require a lot of logic to deal with complexities of the real-world
- But how much of that logic needs to be on the critical path for common operations?

Robbert adopted a layered approach

- User sees some primitive like g.SafeSend(...)
- SafeSend uses an internal infrastructure, perhaps to obtain a snapshot of the group view, with a list of current members, locking the group against membership changes until SafeSend completes
- Below this is a layer doing reliable sending, flow control and retransmission within a set of members
- Below this is a layer establishing connections
- Below this one that discovers IP addresses...

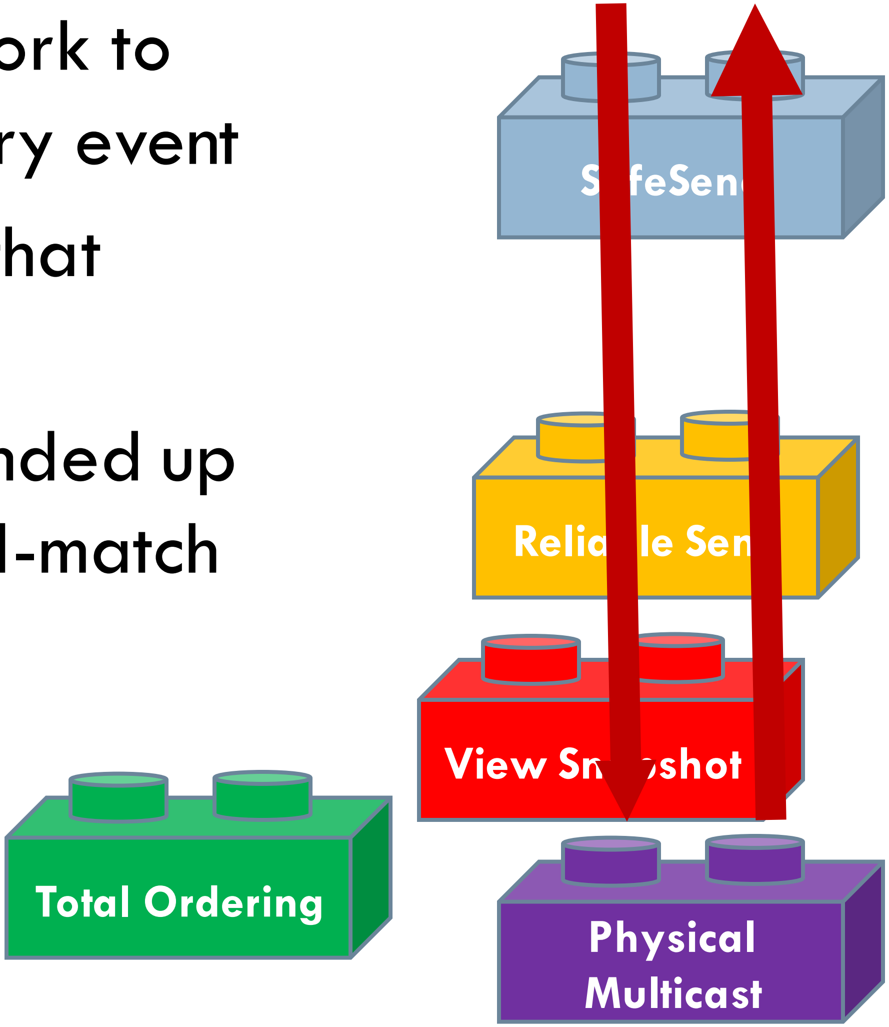


Idea: Standard layers

Like Lego^{tm} blocks
Each supports the identical interface

Horus protocol: Stack of microprotocols

- Not every layer has work to do with respect to every event
- Basic model: "events" that flow up, or down
- By standardizing he ended up with a kind of mix-and-match protocol architecture



Elegant… but what about efficiency?

- Robbert's stacks often had 15 or 20 microprotocols
  - By rearranging and changing selection he could build many kinds of higher level protocols in a standard way
  - But many microprotocols just passed certain kinds of event through, taking no action of their own

- Performance reflected very high overheads when he "microbenchmarked" his solution
  - Isolate a component, then run billions of events through

Critical path analysis

- Robbert realized that his architecture would be evaluated heavily in terms of throughput and delay
  - Delay measured from when g.SafeSend was invoked until when delivery occurs
  - Throughput: g.SafeSend completions per second
- All of that “pass-through untouched” logic on the critical path slows Horus down

Drilling down

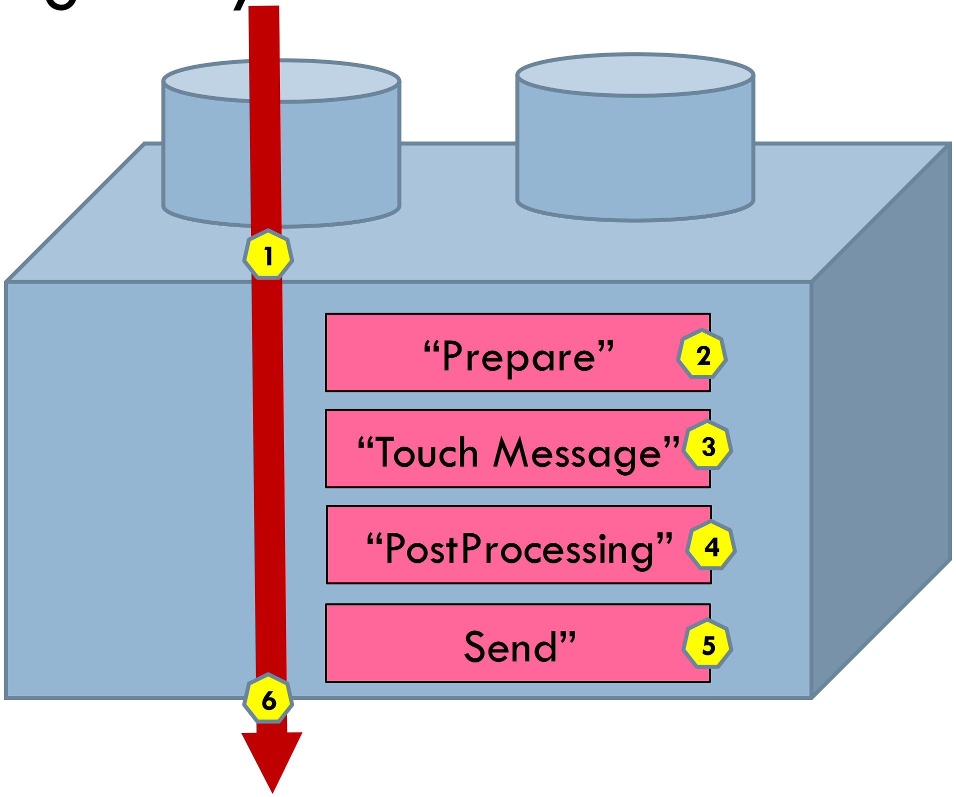
- What does the code *inside* a Horus layer do?

- Robbert had the idea of classifying the instructions into three categories

  - Logic that “could” run before ever seeing the message
  - Logic that needs to see the actual message (cares about the bytes inside, or a sequence number, etc)
  - Logic that “could” run after the message is send

Horus layers and “sub-layers”

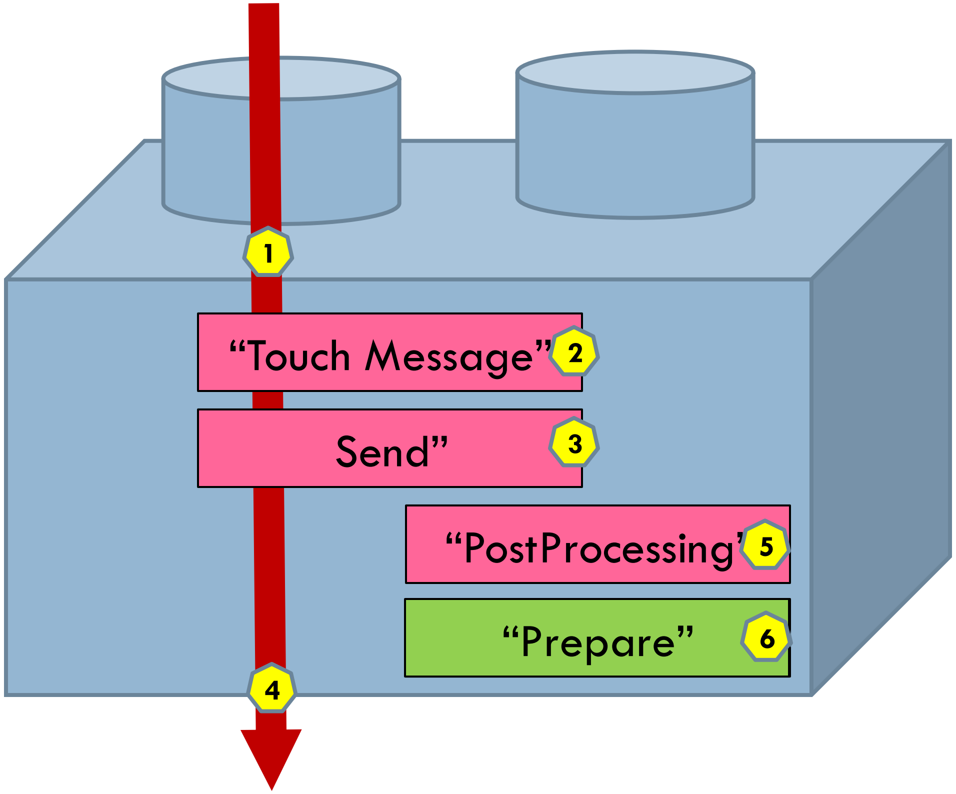
- Steps in running a layer



- Do they need to happen in this order?

Horus layers and “sub-layers”

- Send before post-processing, then prepare for next



- (Scheme makes an “optimistic” guess” that next event will be a multicast, runs “unprepare” if guess was wrong)

Success!

- Horus broke all records for multicast performance and Robbert got a great SIGCOMM publication

> **Masking the Overhead of Layering. Robbert van Renesse. Proc. of the 1996 ACM SIGCOMM Conf. Stanford University. August 1996.**

- Links nicely to today's theme: to what extent can we abstract the kind of reasoning we just saw into a set of general design principles that "anyone" could benefit from?
  - We'll look first at the Internet level, then the O/S

End-to-End arguments in System Design – Jerry H. Saltzer, David P. Reed, David D. Clark

- Authors were early MIT Internet researchers who played key roles in understanding and solving Internet challenges

  - Jerry H. Saltzer
    - A leader of Multics, key developer of the Internet, and a LAN (local area network) ring topology, project Athena
  - David P. Reed

    - Early development of TCP/IP, designer of UDP
  - David D. Clark
    - I/O of Multics, Protocol architect of Internet
      - "We reject: kings, presidents and voting.
      - We believe in: rough consensus and running code."

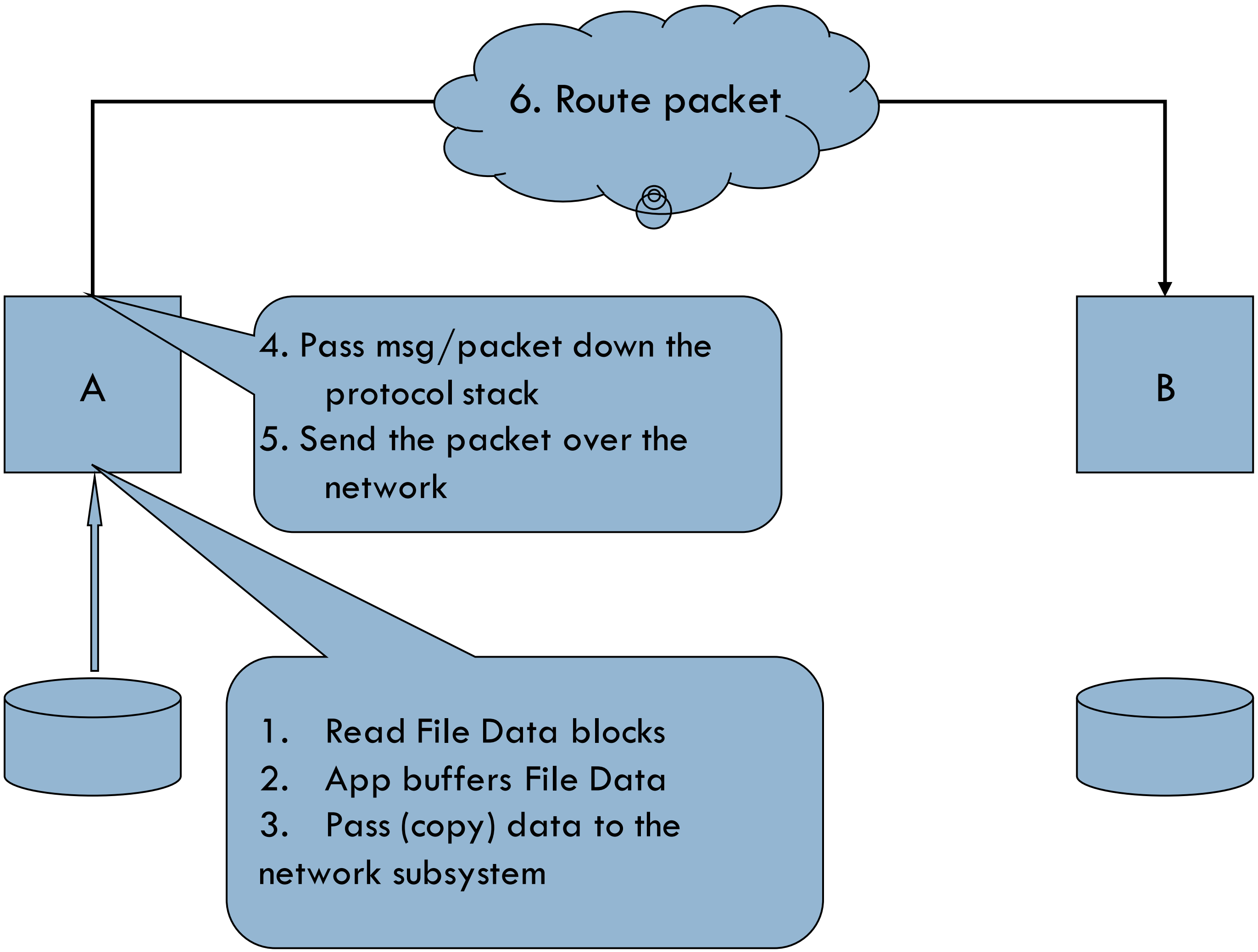
End-to-End arguments in System Design – Jerry H. Saltzer, David P. Reed, David D. Clark

- Question posed: suppose we want a functionality such as "reliability" in the Internet. Where should we place the implementation of the required logic?

- Argue for "end-to-end" solutions, if certain conditions hold
  - Can the higher layer implement the functionality it needs?
    - if yes - implement it there, the app knows its needs best

  - Implement the functionality in the lower layer only if
    - A) a large number of higher layers / applications use this functionality and implementing it at the lower layer improves the performance of many of them AND
    - B) does not hurt the remaining applications

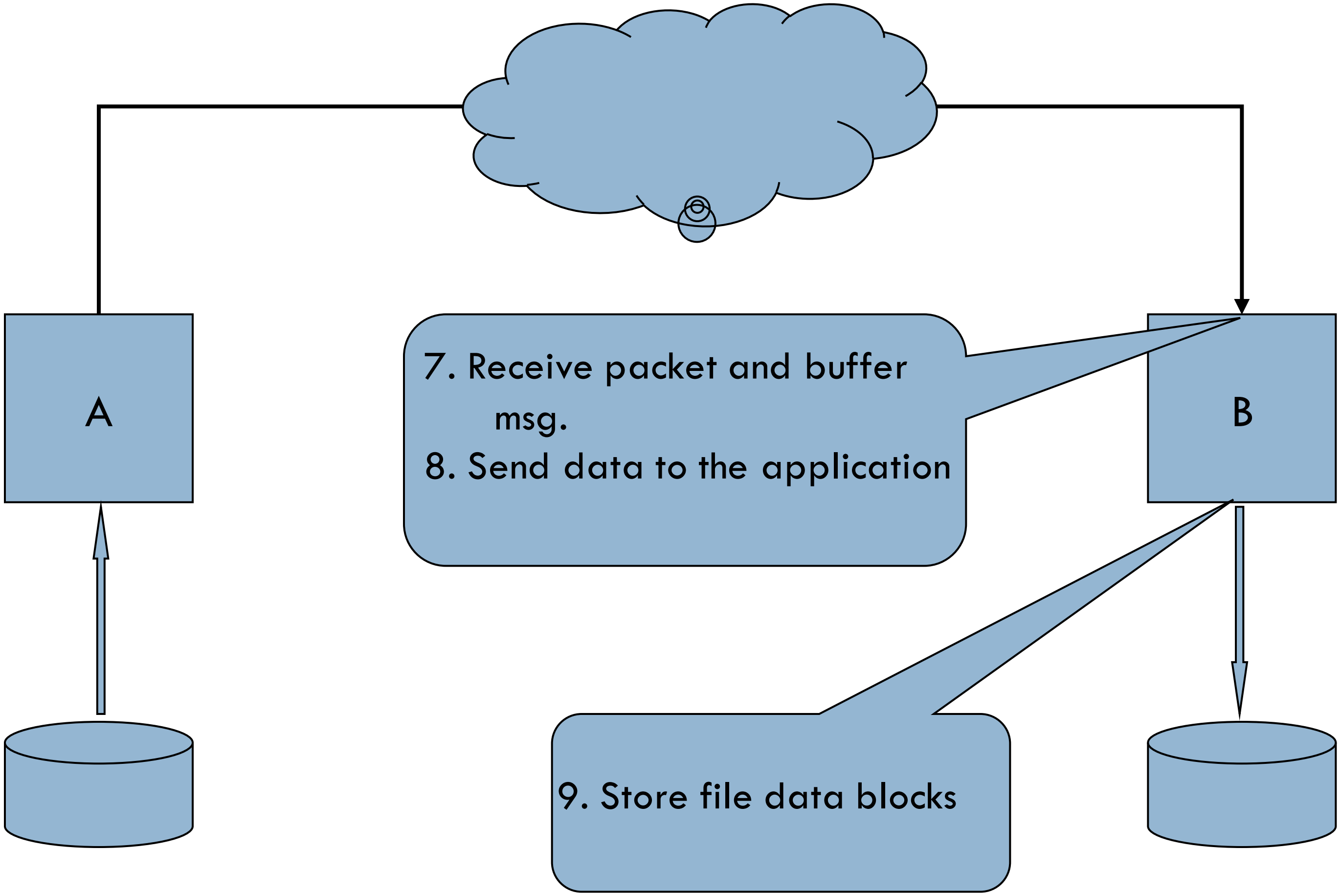
Example : File Transfer (A to B)



4. Pass msg/packet down the protocol stack
5. Send the packet over the network

1. Read File Data blocks
2. App buffers File Data
3. Pass (copy) data to the network subsystem

Example : File Transfer



7. Receive packet and buffer msg.
8. Send data to the application

9. Store file data blocks

Possible failures

- Reading and writing to disk
- Transient errors in the memory chip while buffering and copying
- *network might drop packets, modify bits, deliver duplicates*
- OS buffer overflow at the sender or the receiver
- Either of the hosts may crash

Would a reliable network help?

- Suppose we make the network reliable
  - Packet checksums, sequence numbers, retry, duplicate elimination
  - Solves only the network problem.
  - What about the other problems listed?
  - War story: Byte swapping problem while routing @ MIT

- Not sufficient and not necessary

Solutions?

- Introduce file checksums and verify once transfer completes – an end-to-end check.
  - On failure – retransmit file.

Solutions? (cont.)

- network level reliability would improve performance.
  - But this may not benefit all applications
    - Huge overhead for say Real-Time speech transmission
    - Need for optional layers

- Checksum parts of the file.

Formally stated

"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.)"

Other end-to-end requirements

- Delivery guarantees
  - Application level ACKs
    - Deliver only if action guaranteed
    - 2 phase commit
    - NACKs
- End-to-end authentication
- Duplicate msg suppression
  - Application level retry results in new n/w level packet

TCP/IP

- Internet Protocol
  - IP is a simple ("dumb"), stateless protocol that moves datagrams across the network, and
- Transmission Control Protocol
  - TCP is end-to-end.
  - It is a smart transport protocol providing error detection, retransmission, congestion control, and flow control end-to-end.
- The network
  - The network itself (the routers) needs only to support the simple, lightweight IP; the endpoints run the heavier TCP on top of it when needed.

End-to-End became a religion!

- The principle is applied throughout the Internet in a very “aggressive” way
  - Every TCP session does its own failure detection
  - Any kind of strong consistency guarantee (like the things Isis or Horus are doing) is viewed as “not part of the Internet”. Routing daemons don’t synchronize actions.
  - Accounts for one of those “forks in the road” we discussed: SIGCOMM and SOSP/NSDI have very different styles.
- Contemporary puzzle: RDMA offers reliability in hardware. Not E2E… yet RDMA is the big new thing…

Hints for Computer System Design - Butler Lampson

- Related to end-to-end argument—guidance for developer

- The paper offers a collection of experience and wisdom aimed at (operating) systems designers
  - Suggests that they be viewed as hints, not religion
  - Rules of thumb that can guide towards better solutions

Butler Lampson - Background

- Founding member of Xerox PARC (1970), DEC (1980s), MSR (current)
- ACM Turing Award (1992)

- Laser printer design
- PC (Alto is considered first actual personal computer)
- Two-phase commit protocols
- Bravo, the first WYSIWYG text formatting program
- Ethernet, the first high-speed local area network (LAN)

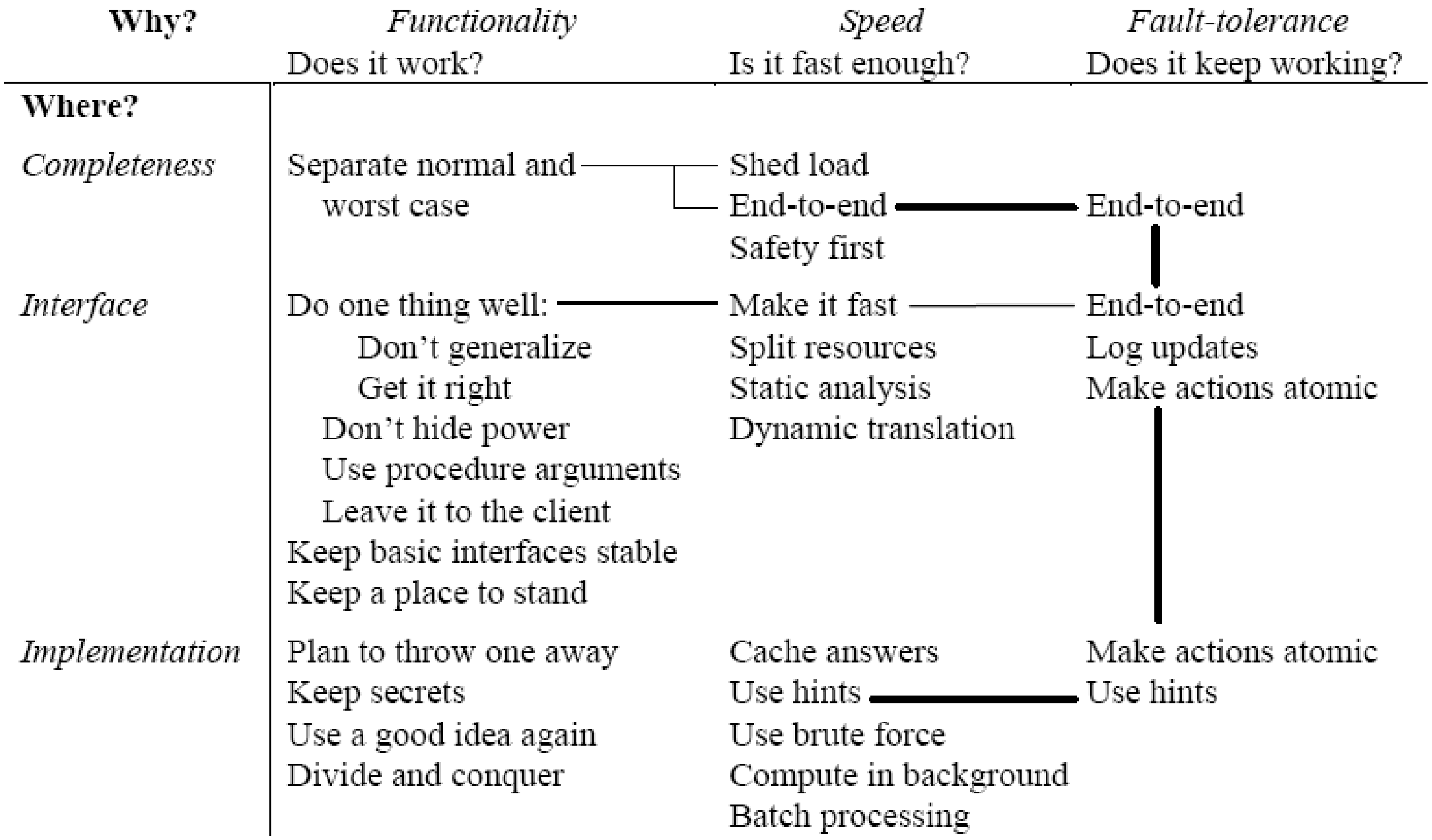
Some Projects & Collaborators

- Charles Simonyi - Bravo: WYSIWYG editor (MS Office)
- Bob Sproull - Alto operating system, Dover: laser printer, Interpress: page description language (VP Sun/Oracle)
- Mel Pirtle - 940 project, Berkeley Computer Corp.
- Peter Deutsch - 940 operating system, QSPL: system programming language (founder of Ghostscript)
- Chuck Geschke, Jim Mitchell, Ed Satterthwaite - Mesa: system programming language

Some Projects & Collaborators (cont.)

- Roy Levin - Wildflower: Star workstation prototype, Vesta: software configuration
- Andrew Birrell, Roger Needham, Mike Schroeder - Global name service and authentication
- Eric Schmidt - System models: software configuration (CEO/Chairman of Google)
- Rod Burstall - Pebble: polymorphic typed language

Hints for Computer System Design - Butler Lampson



Why?	*Functionality* Does it work?	*Speed* Is it fast enough?	*Fault-tolerance* Does it keep working?
Where?			
Completeness	Separate normal and worst case	Shed load End-to-end Safety first	End-to-end
Interface	Do one thing well: Don’t generalize Get it right Don’t hide power Use procedure arguments Leave it to the client Keep basic interfaces stable Keep a place to stand	Make it fast Split resources Static analysis Dynamic translation	End-to-end Log updates Make actions atomic
Implementation	Plan to throw one away Keep secrets Use a good idea again Divide and conquer	Cache answers Use hints Use brute force Compute in background Batch processing	Make actions atomic Use hints

Figure 1: Summary of the slogans

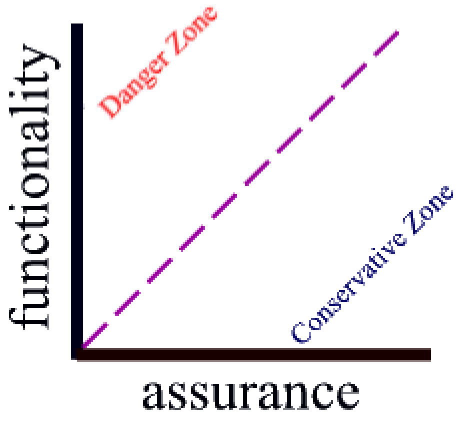
Functionality

- Interface – Contract
  - separates implementation from client using abstraction
  - Eg: File (open, read, write, close)
- Desirable properties
  - Simple
  - Complete
  - Admit small and fast impl.

Simplicity

- Interfaces
  - Avoid generalizations
    - too much = large, slow and complicated impl.
    - Can penalize normal operations
      - PL/1 generic operations across data types
  - Should have predictable (reasonable) cost.
    - eg: FindIthField [O(n)], FindNamedfield [O(n^2)]
  - Avoid features needed by only a few clients

Functionality Vs Assurance



- As a system performs more (complex interface) assurance decreases.

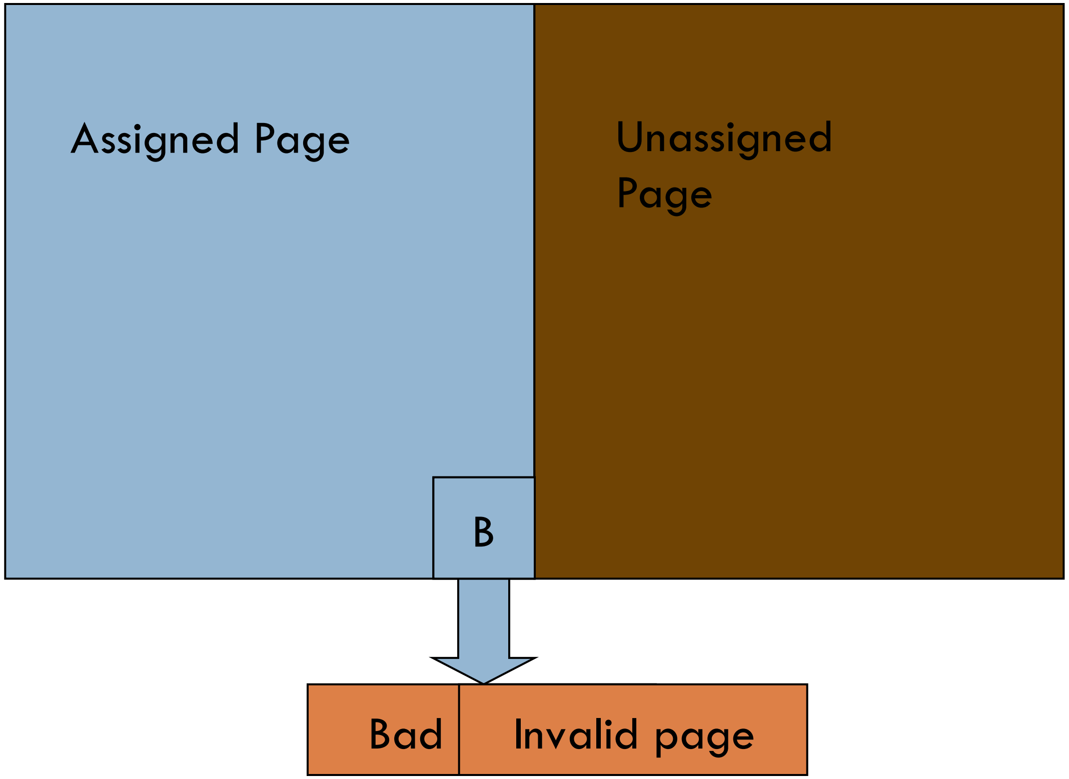
Example

- Tenex System
  - reference to an unassigned page -> trap to user program
  - *arguments to sys calls passed by reference*
  - CONNECT(string passwd) -> if passwd wrong, fails after a 3 second delay

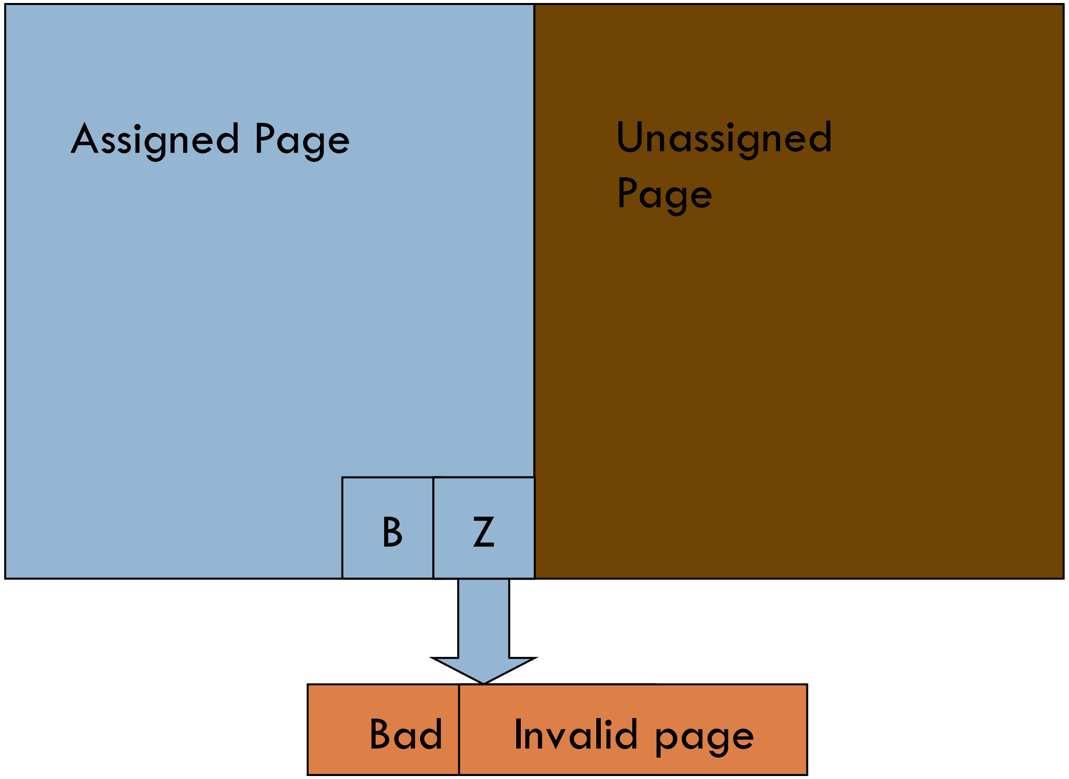
  - CONNECT

```
for i := 0 to Length(directoryPassword) do
    if directoryPassword[i] != passwordArgument[i] then
                    Wait three seconds; return BadPassword
    end if
end loop;
connect to directory; return Success
```

Breaking CONNECT(string passwd)



Breaking CONNECT(string passwd)



Worst case

128*n tries as opposed to 128^n tries

n = passwd length (bytes)

Functionality (cont.)

- basic (fast) operations rather than generic/powerful (slow) ones
  - Pay for what you want
  - RISC Vs CISC
  - Unix Pipe
    - grep –i 'spock' * | awk -F: '{print $1}' | sort | uniq | wc –l

- Use timing tools (80% of the time in 20% of code)
  - Avoid premature optimization
    - May be useless and/or expensive
  - analyze usage and optimize heavily used I/Fs

Abstractions

- Avoid abstracting-out desirable properties
  - “don't hide power”
  - Eg: Feedback for page replacement
  - How easy is it to identify desirable properties?

- Procedure arguments
  - filter procedure instead of a complex language with patterns.
    - static analysis for optimization - DB query lang
  - failure handlers
  - trust?

Continuity

- Interfaces
  - Changes should be infrequent
    - Compatibility issues
  - Backward compatibility on change
- Implementation
  - Refactor to achieve "satisfactory" (small, fast, maintainable) results
  - Use prototyping

Implementation

- Keep secrets
  - Impl. can change without changing contract
  - Client could break if it uses Impl. details
  - But secrets can be used to improve performance
    - finding the balance an art?
- Divide and conquer
- **Reuse** a good idea in different settings
  - global replication using a transactional model
    - local replication for reliably storing transactional logs.

Completeness - handling all cases

- Handle normal and worst case separately
  - normal case – speed, worst case – progress
  - Examples
    - caches
    - incremental GC
      - trace-and-sweep (unreachable circular structures)
    - piece-table in the Bravo editor
      - Compaction either at fixed intervals or on heavy fragmentation
  - “emergency supply” helps in worst-case scenarios

Speed

- Split resources in a fixed way
  - rather than share and multiplex
  - faster access, predictable allocation
  - **Safety** instead of optimality
    - over-provisioning ok, due to cheap hardware
- Use static analysis where possible
  - dynamic analysis as a fallback option
  - Eg: sequential storage and pre-fetching based on prior knowledge of how data is accessed

Speed (cont.)

- Cache answers to expensive computations
  - x, f => f(x)
  - f is functional.
- Use hints!
  - may not reflect the "truth" and so should have a quick correctness check.
  - Routing tables
  - Ethernet (CSMA/CD)

Speed (cont.)

- Brute force when in doubt
  - Prototype and test performance
  - Eg: linear search over a small search space
  - Beware of scalability!
- Background processing (interactive settings)
  - GC
  - writing out dirty pages, preparing pages for replacement.
- Shed load
  - Random Early Detection
  - Bob Morris' red button

Fault Tolerance

- End-to-end argument
  - Error recovery at the app level **essential**
  - Eg: File transfer
- Log updates
  - Replay logs to recover from a crash
  - form 1: log <name of update proc, arguments>
    - update proc must be functional
    - arguments must be values
  - form 2: log state changes.
    - idempotent (x = 10, instead of x++)
- Make actions atomic
  - Aries algorithm - Atomicity and Durability

Conclusions

- Every field develops a community intuition into the principles that lead towards “our kind of work”
  - Solutions that are esthetically pleasing and reflect sound reasoning
  - But how can one communicate esthetics? And what sorts of reasoning should be viewed as “sound”?

- For the systems area, the tensions between the hardware we work with, the problems to be solved and the fact that we create “platforms” that others will use weigh heavily into this analysis

“Second System Syndrome”

- In 2012 we are rarely the first people to build a given kind of system

- It can be hard to resist including all the usual functionality and then adding in new amazing stuff

- Lampson believes that elegance centers on *leaving things out* not including every imaginable feature!
  - Perhaps the most debated aspect of his approach
  - Think about Windows “versus” Linux (versus early Unix)

Concrete conclusions?

- Think back to the way Robbert approached Horus
  - Pose your problem in a clean way
  - Next decompose into large-scale components
  - Think about the common case that will determine performance: the critical path or the bottleneck points
  - Look for elegant ways to simultaneously offer structural clarity (like the Horus "Lego^{tm}" building blocks) and yet still offer fantastic performance
- This can guide you towards very high-impact success

Next Time

- Read and write review:
  - *The UNIX time-sharing system*, Dennis M. Ritchie and Ken Thompson. Communications of the ACM Volume 17, Issue 7, July 1974, pages 365 -- 375
  - *The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System.* M. Young, A Tavanian, R. Rashid, D. Golub, and J. Eppinger. Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (Austin, Texas, United States), ACM, 1987, pages 63--76.