The Role of Large Architectural Principles in Systems

Presented by Hakim Weatherspoon

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

Systems Research

- The study of tradeoffs
  - Functionality vs performance
  - E.g. where to place error checking
- Are there principles or rules of thumb that can help with large systems design?

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

- Background of authors at MIT

- 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."

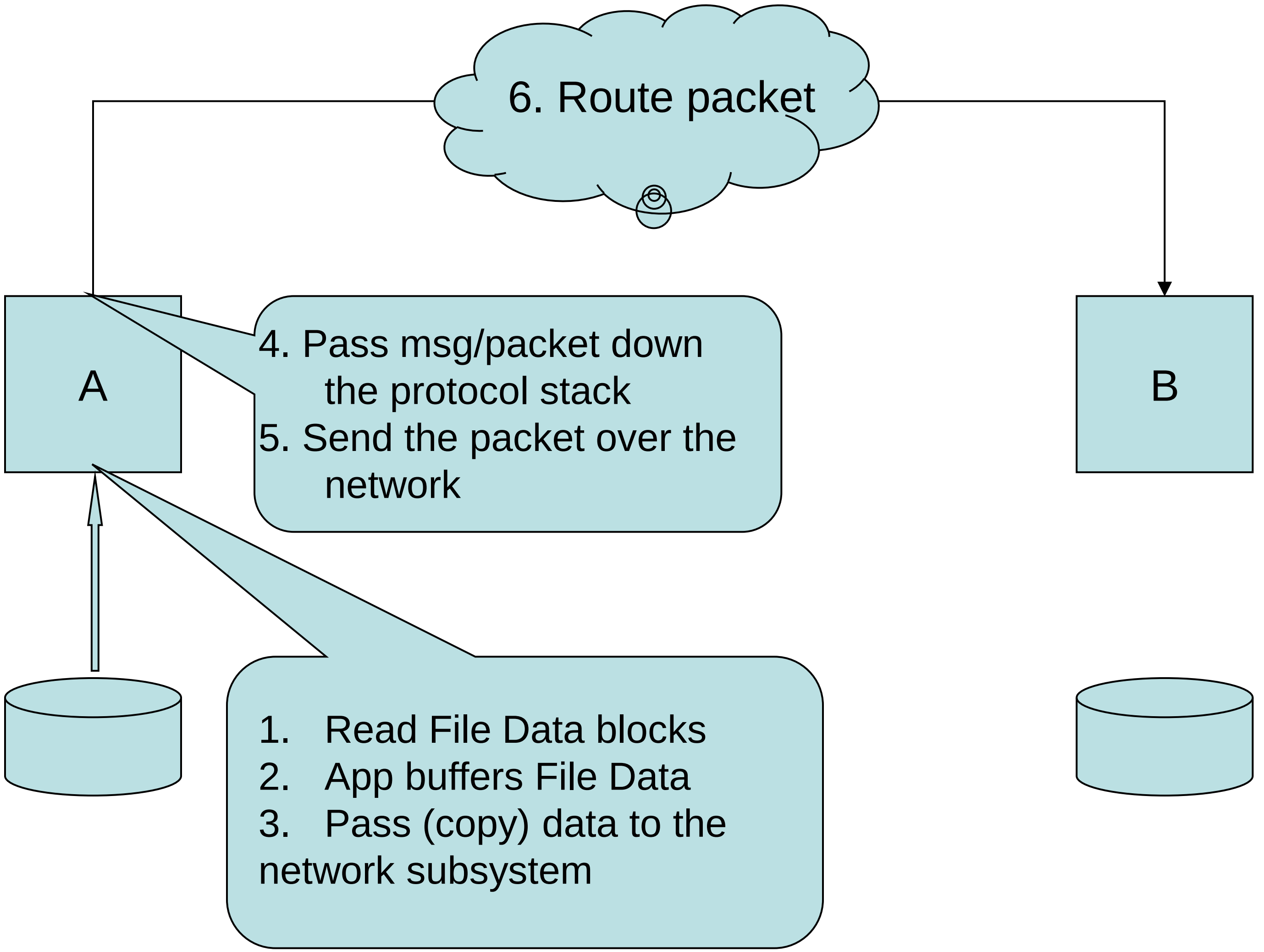
End-to-End arguments in System Design – Jerry H. Saltzer, David P. Reed, David D. Clark

- Helps guide function placement among modules of a distributed system

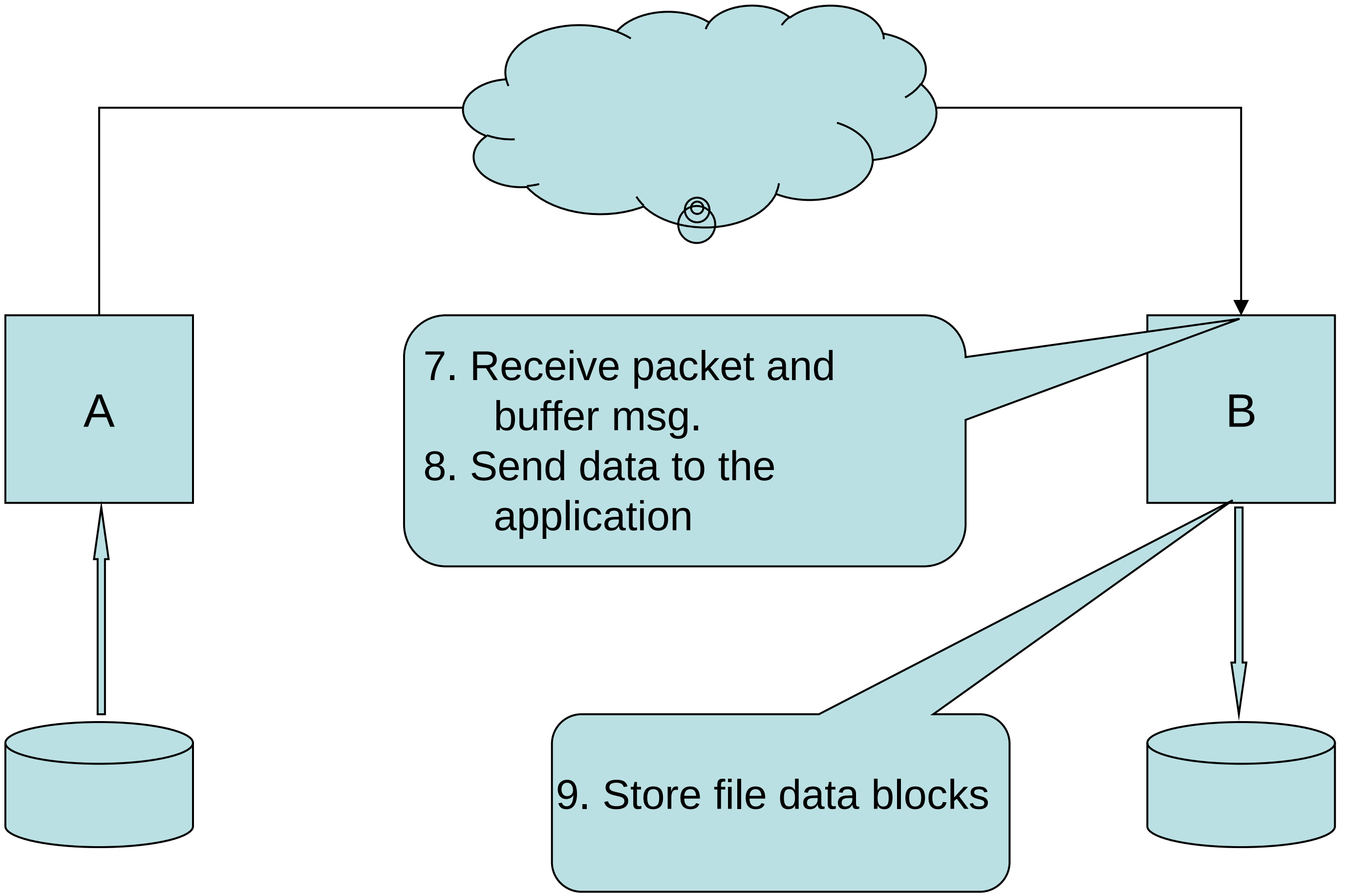
- Argument
  - can the higher layer implement the functionality it needs?
    - if yes - implement it there, the app knows it's needs best

  - 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)



Example : File Transfer



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

Solutions?

- 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 – *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

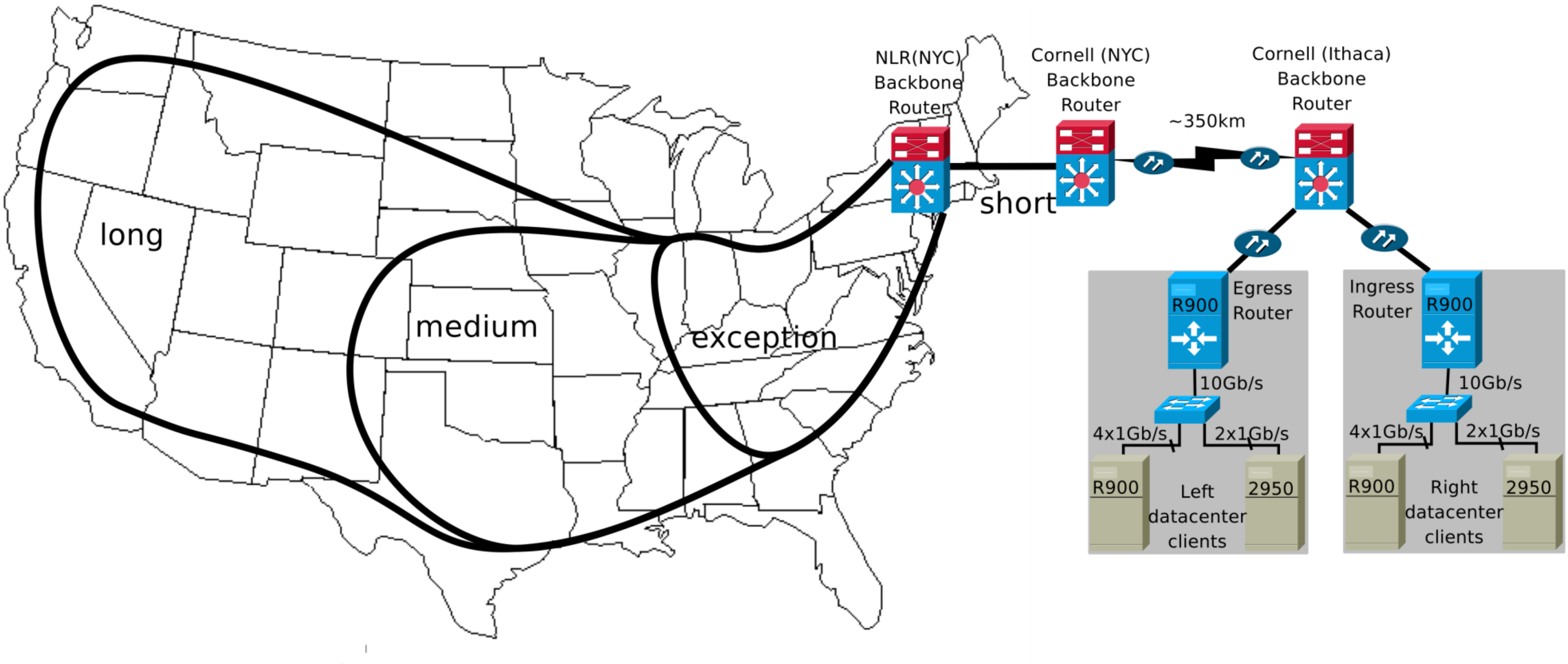
TCP/IP

- 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.

Is argument complete?

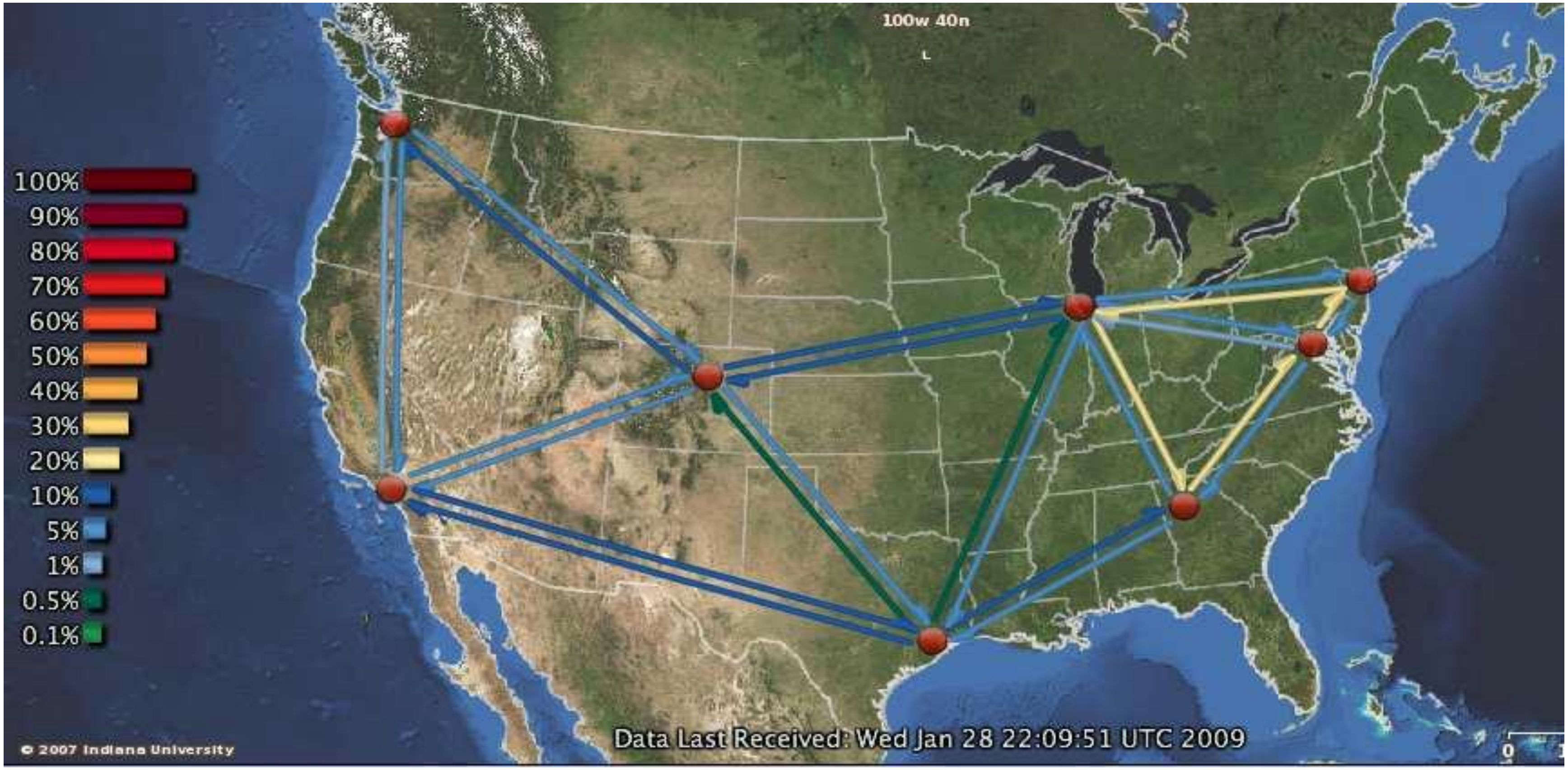
- E.g. congestion control
  - TCP leaves it to the ends
    - Should the network trust the ends?
      - RED
    - In a wireless setting
      - packet loss != congestion
- performance problems may appear in end-end systems under heavy load
- Performance enhancing Proxies

Cornell NLR Rings testbed



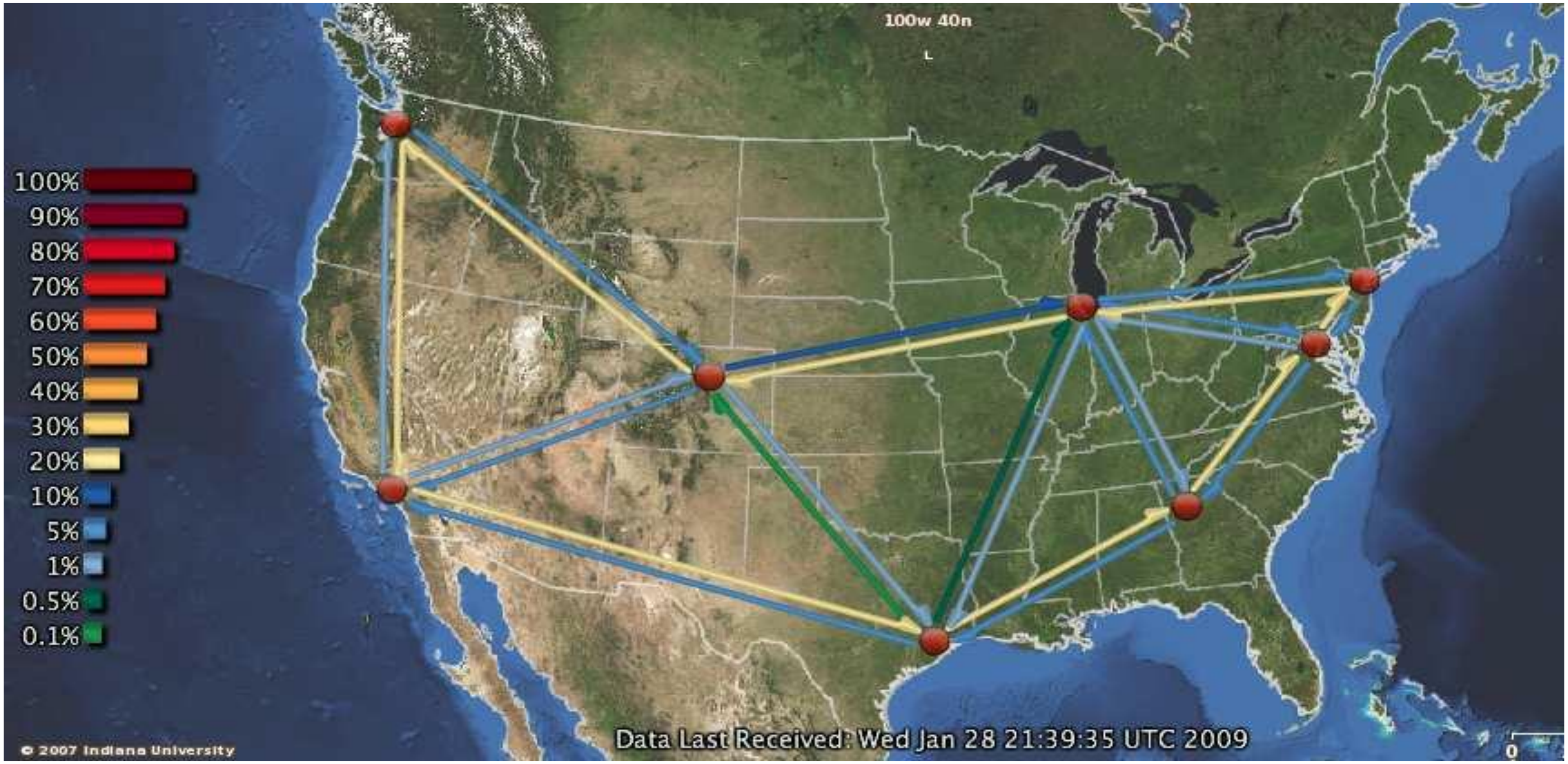
λ-network endpoints by
Hakim Weatherspoon

Cornell NLR Rings testbed



λ-network endpoints by
Hakim Weatherspoon

Cornell NLR Rings testbed



Hints for Computer System Design - Butler Lampson

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

- But a collection of experience and wisdom
  - Use a hints

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

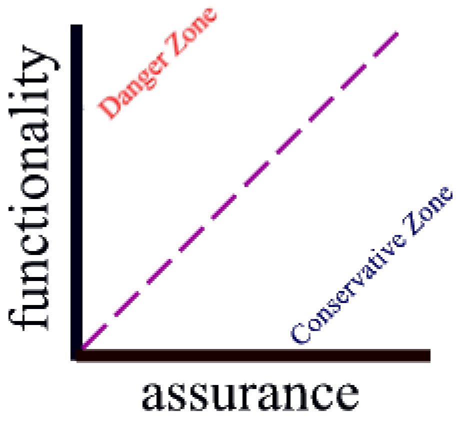
Functionality

- 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.

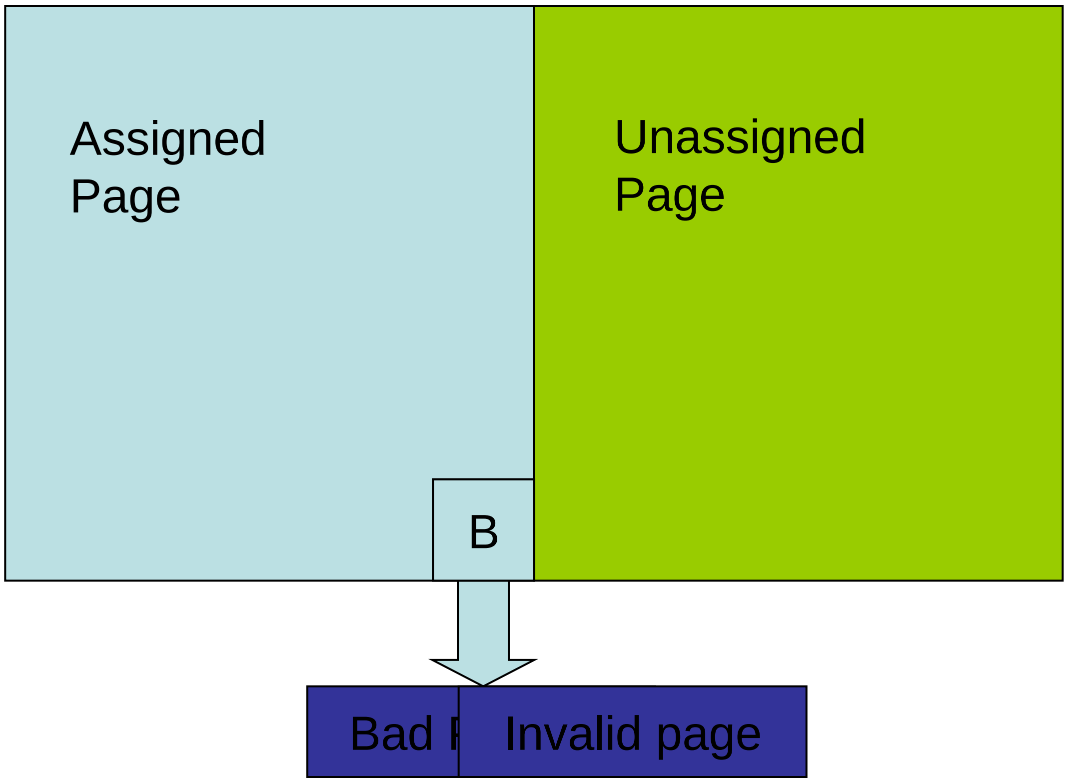
Example

- 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

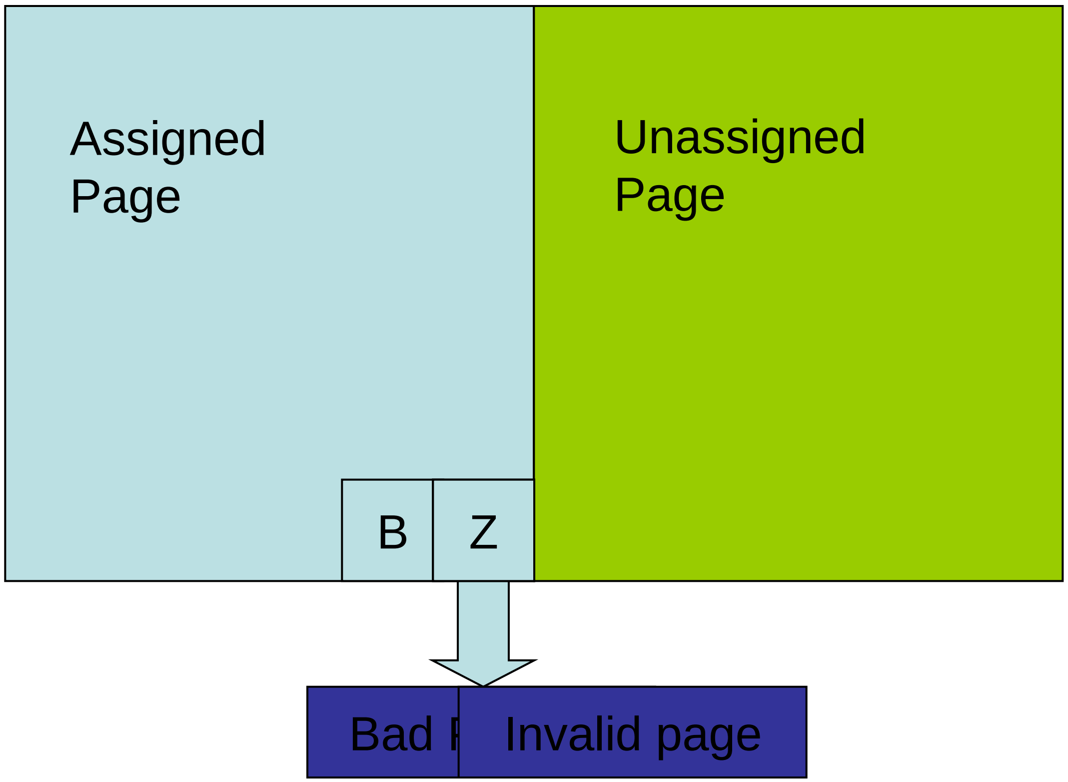
  - CONNECT

```
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

- 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

- Remember these are “hints” from a Guru
- Reuse good ideas, but not blindly.

- Your experiences

“Second System Syndrome”

- In 2013 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?

- Lessons Learned
  - 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 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 structure of the "THE"-multiprogramming system*, E.W. Dijkstra. Communications of the ACM Volume 11, Issue 5, May 1968, pages 341--346

- Do Lab 0

- Check website for updated schedule