EECS 262a
Advanced Topics in Computer Systems
Lecture 2

End-to-End / System R
January 25th, 2016

John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

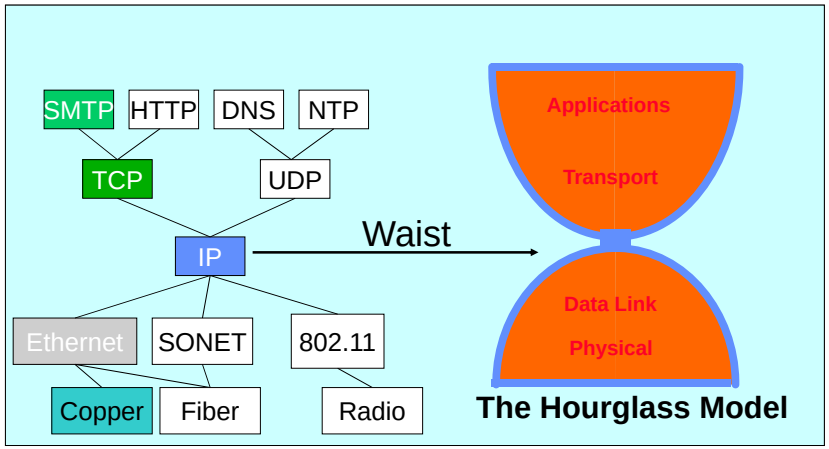
http://www.eecs.berkeley.edu/~kubitron/cs262

Today's Papers

- End-To-End Arguments in System Design
  J. H. Saltzer, D. P. Reed, D. D. Clark. Appears in *ACM Transactions on Computer Systems,*Vol 2, No. 4, November 1984, pp 277-288.
- A History and Evaluation of System R
  Donald D. Chamberlin, Morton A. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade and Robert A. Yost. Appears in *Communications of the ACM,* Vol 24, No. 10, October 1981, pp 632-646

- Both papers represented significant (controversial) paradigm shifts
- Changing the design elements
  - E2E: Where to place functionality
  - System R: Data independence from physical representation
- Contrasting top-down vs. bottom-up views

The Internet *Hourglass*



There is just one network-layer protocol, **IP**.
The "narrow waist" facilitates interoperability.

Implications of Hourglass

Single Internet-layer module (IP):

- Allows arbitrary networks to interoperate
  - Any network technology that supports IP can exchange packets
- Allows applications to function on all networks
  - Applications that can run on IP can use any network
- Supports simultaneous innovations above and below IP
  - But changing IP itself, i.e., IPv6, very involved (IPng proposed in 9/93)

Drawbacks of Layering

- Layer N may duplicate layer N-1 functionality
  - E.g., error recovery to retransmit lost data
- Layers may need same information
  - E.g., timestamps, maximum transmission unit size
- Layering can hurt performance
  - E.g., hiding details about what is really going on
- Some layers are not always cleanly separated
  - Inter-layer dependencies for performance reasons
  - Some dependencies in standards (header checksums)
- Headers start to get really big
  - Sometimes header bytes >> actual content

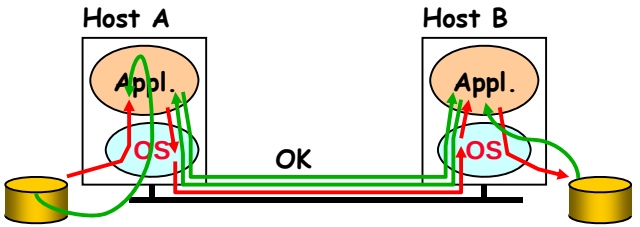
Placing Network Functionality

- Hugely influential paper: "End-to-End Arguments in System Design" by Saltzer, Reed, and Clark ('84)
- "Sacred Text" of the Internet
  - Endless disputes about what it means
  - Everyone cites it as supporting their position

Basic Observation

- Some types of network functionality can only be correctly implemented end-to-end
  - Reliability, security, etc
- Because of this, end hosts:
  - Can satisfy the requirement without network's help
  - Will/must do so, since can't *rely* on network's help
- Therefore don't go out of your way to implement them in the network

Example: Reliable File Transfer



- Solution 1: make each step reliable, and then concatenate them
- Solution 2: end-to-end check and try again if necessary

Discussion

- Solution 1 is incomplete
  - What happens if memory is corrupted?
  - Receiver has to do the check anyway!
- Solution 2 is complete
  - Full functionality can be entirely implemented at application layer with no need for reliability from lower layers
- *Is there any need to implement reliability at lower layers?*
  - Well, it could be more efficient *or more problematic*

End-to-End Principle

Implementing this functionality in the network:

- Doesn't reduce host implementation complexity
- Does increase network complexity
- Probably imposes delay and overhead on all applications, even if they don't need functionality
- However, implementing in network can enhance performance in some cases
  - E.g., very lossy links such as wireless
- It may also help mitigate denial of service and/or privacy

Conservative Interpretation of E2E

- Don't implement a function at the lower levels of the system unless it can be completely implemented at this level
- Unless you can relieve the burden from hosts, don't bother

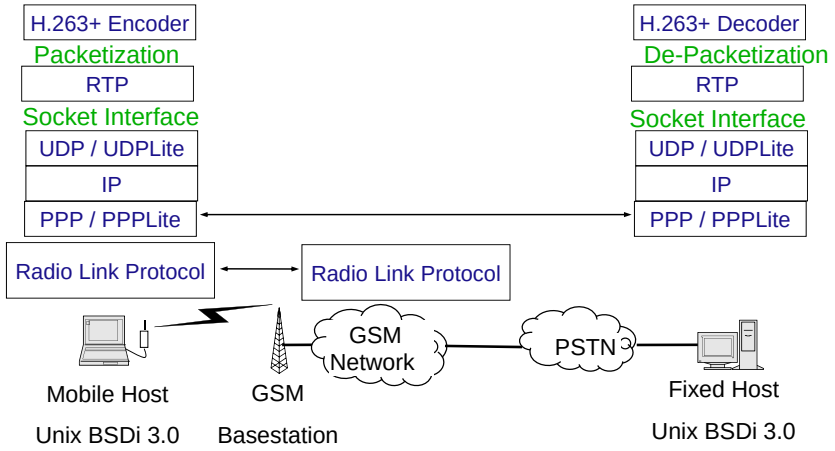
Moderate Interpretation

- Think twice before implementing functionality in the network
- If hosts can implement functionality correctly, implement it in a lower layer only as a performance enhancement
- But do so only if it does not impose burden on applications that do not require that functionality
- This is the interpretation I use

Research Example

- Goal: Flexible networking protocols in support of error resilient video codecs
- Target domain: Live video streaming over 2G GSM cellular network
- Environment: Low-bit rate video codecs that are highly tolerant of errors in the byte stream
  - H.263++: Motion vectors, prediction, error/loss concealment
- What is the role of reliability in the network?

Live Video Streaming over Cellular



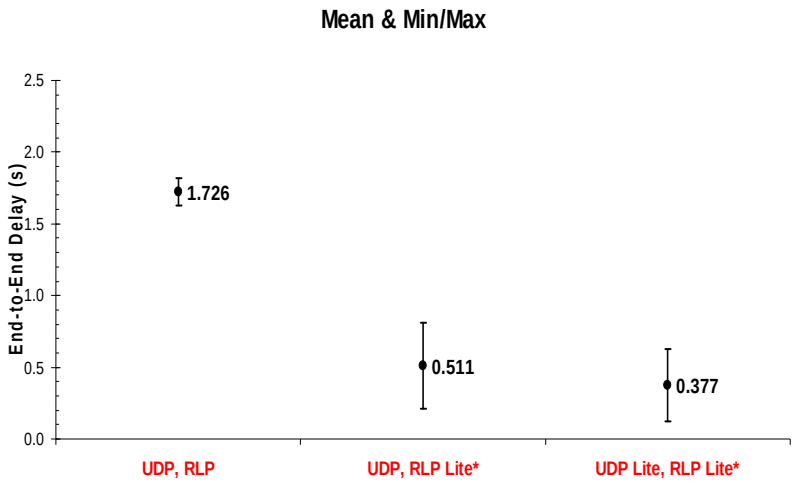
Wireless Video Streaming

- GSM Radio Link Protocol: reliable data delivery on radio link
  - Issue: reliability versus delay – do you need reliability when you have an error tolerant video codec?
- Solution #1: turn off reliability in RLP and PPP
  - Fewer video packets delivered!
  - UDP checksums caused “damaged” packets to be dropped

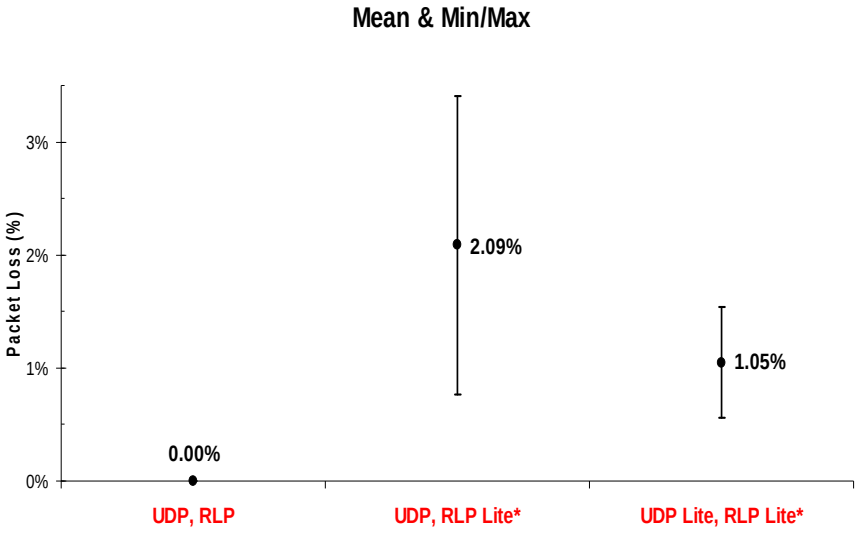
Need Variable Degrees of Reliability

- Protect headers (need for routing/delivery), not data!
- Solution #2:
  - UDP Lite (Larzon, Degemark, and Pink)
    - » Flexible checksum only protects packet headers and allows apps to receive corrupted data
  - RLP Lite / PPP Lite (new protocols)
    - » Same as UDP Lite, but for radio link / link layer
- Simulation/experiments: UDP Lite/RLP Lite/PPP Lite
  - Collected 4480 min of wireless video traces, (~4 min per video)
  - Bad channel conditions (signal strength ~2-3), BLER ~ 1.5%
  - Used simulation to repeat experiments
- Results
  - Less E2E delay, constant jitter, higher throughput, lower packet loss
    … than UDP (with or without RLP)

End to End Delay



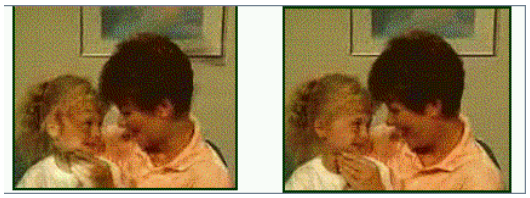
Packet Loss



Video Screenshots

Experiment

UDP UDP/RLP Lite



Simulation

UDP UDP/RLP Lite

Summary

- E2E argument encourages us to keep IP simple
- If higher layer can implement functionality correctly, implement it in a lower layer only if
  - it improves the performance significantly for application that need that functionality, and
  - it does not impose burden on applications that do not require that functionality
- Principle is broadly applicable to other systems domains
  - Storage, architecture, …

Is this a good paper?

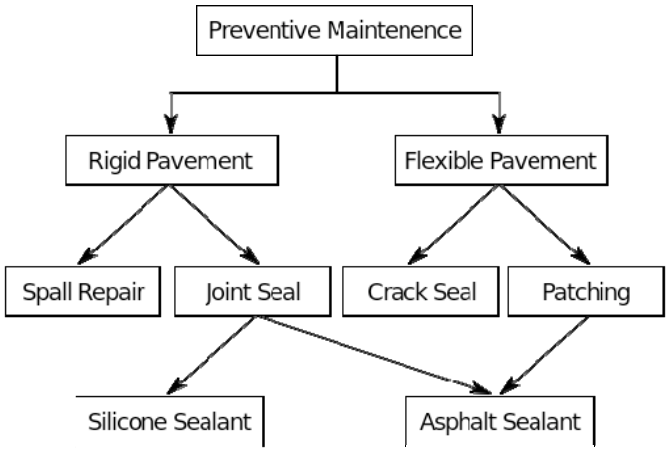
- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

BREAK

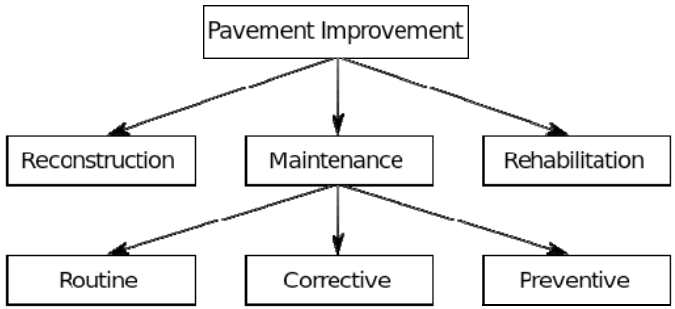
DataBase Management Systems History

- Late 60's: network (CODASYL) & hierarchical (IMS) DBMS
  - IMS built for Apollo program
  - Charles Bachman: father of CODASYL predecessor IDS (at GE in early 1960's), Turing award #8 (1973, between Dijkstra and Knuth)

Network Model Example



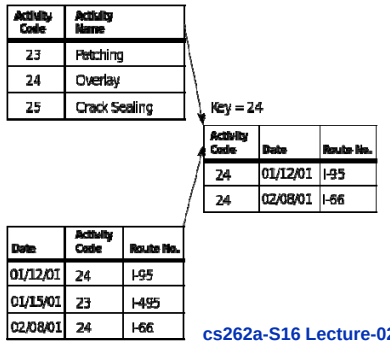
IMS Model



- IMS Example:
  - One is parent, one is child
  - Problems include redundancy and requirement of having a parent (deletion anomalies)
  - Low-level "record-at-a-time'" data manipulation language (DML), i.e., physical data structures reflected in DML (no data independence)

1970: Edgar Codd's Paper

- The most influential paper in DB research
  - Set-at-a-time DML with the key idea of "data independence"
  - Allows for schema and physical storage structures to change under the covers
  - Papadimitriou: "as clear a paradigm shift as we can hope to find in computer science"
  - Edgar F. Codd: Turing award #18 (1981, between Hoare and Cook)



Data Independence – both logical and physical

- What physical tricks could you play under the covers? Think about modern HW!
- "Hellerstein's Inequality":
  - Need data independence when *d*app/*dt* << *d*environment/*dt*
  - Other scenarios where this holds?
  - This is an early, powerful instance of two themes: *levels of indirection* and *adaptivity*

"Modern" DBMS Prototypes

- Mid 70's:
  - Wholesale adoption of Codd's vision in 2 full-function (sort of) prototypes
  - Ancestors of essentially all today's commercial systems
- Ingres and System R

Ingres : UCB 1974-77

- An early and pioneering “pickup team’”, including Stonebraker & Wong
- Begat Ingres Corp (CA), CA-Universe, Britton-Lee, Sybase, MS SQL Server, Wang’s PACE, Tandem Non-Stop SQL

System R : IBM San Jose (now Almaden)

- 15 PhDs
- Begat IBM's SQL/DS & DB2, Oracle, HP’s Allbase, Tandem Non-Stop SQL
- Jim Gray: Turing Award #22 (1998, between Englebart and Brooks)
- Lots of Berkeley folks on the System R team
  - Including Gray (1st CS PhD @ Berkeley), Bruce Lindsay, Irv Traiger, Paul McJones, Mike Blasgen, Mario Schkolnick, Bob Selinger , Bob Yost. See http://www.mcjones.org/System_R/SQL_Reunion_95/sqlr95-Prehisto.html#Index71.

Discussion

- System R arguably got more stuff “right”, though there was lots of information passing between both groups
- Both were viable starting points, proved practicality of relational approach
  - Direct example of theory --> practice!
  - ACM Software Systems award #6 shared by both
  - Stated goal of both systems was to take Codd’s theory and turn it into a workable system as fast as CODASYL but much easier to use and maintain
  - Interestingly, Stonebraker received ACM SIGMOD Innovations Award #1 (1991), Gray #2 (1992), whereas Gray got the Turing first.

Commercial RDBMS

- Early 80’s: commercialization of relational systems
  - Ellison’s Oracle beats IBM to market by reading white papers
  - IBM releases multiple RDBMSs, settles down to DB2
  - Gray (System R), Jerry Held (Ingres) and others join Tandem (Non-Stop SQL)
  - Kapali Eswaran starts EsVal, which begets HP Allbase and Cullinet
  - Relational Technology Inc (Ingres Corp), Britton-Lee/Sybase, Wang PACE grow out of Ingres group
  - CA releases CA-Universe, a commercialization of Ingres
  - Informix started by Cal alum Roger Sippl (no pedigree to research)
  - Teradata started by some Cal Tech alums, based on proprietary networking technology (no pedigree to software research, though see parallel DBMS discussion later in semester!)

The Rise of SQL

- Mid 80's: SQL becomes "intergalactic standard"
  - DB2 becomes IBM's flagship product
  - IMS "sunseted"

Today

- Network & hierarchical are legacy systems (though commonly in use!)
  - IMS still widely used in banking, airline reservations, etc. (remains a major IBM cash cow)
- Relational market commoditized
  - Microsoft, Oracle and IBM fighting over bulk of market
  - NCR Teradata, Sybase, HP Nonstop and a few others vying to survive on the fringes
  - OpenSource coming of age, including MySQL, PostgreSQL, Ingres (reborn)
  - BerkeleyDB is an embedded transactional store that is widely used as well, but now owned by Oracle
  - XML and object-oriented features have pervaded the relational products as both interfaces and data types, further complicating the "purity" of Codd's vision

Database View of Applications

- Big, complex record-keeping applications like SAP and PeopleSoft, which run *over* a DBMS
- "Enterprise applications" to keep businesses humming
  - ERP: Enterprise Resource Planning (SAP, Baan, PeopleSoft, Oracle, IBM, etc.)
  - CRM: Customer Relationship Management (E.phiphany, Siebel, Oracle, IBM, *Salesforce*, etc.)
  - SCM: Supply Chain Management (Trilogy, i2, Oracle, IBM, etc.)
  - Human Resources, Direct Marketing, Call Center, Sales Force Automation, Help Desk, Catalog Management, etc.
- Typically client-server (a Sybase "invention") with a form-based API
  - Focus on resource management secondary to focus on data management
- Traditionally, a main job of a DBMS is to make these kinds of apps easy to write

Relational System Architecture

- RDMBS are BIG, hard to modularize pieces of software
  - Many macro and micro scale system design decisions
  - We'll focus on macro design issues today (micro in future lectures)
- Disk management choices:
  - file per relation
  - big file in file system
  - raw device
- Process Model:
  - process per user
  - server
  - multi-server
- Hardware Model:
  - shared nothing
  - shared memory
  - shared disk

Relational System Architecture

- Basic modules:
  - parser
  - query rewrite
  - optimizer
  - query executor
  - access methods
  - buffer manager
  - lock manager
  - log/recovery manager

Notes on System R

- Some “systems chestnuts” seen in this paper:
  - Expect to throw out the 1st version of the system
  - Expose internals via standard external interfaces whenever possible (e.g. catalogs as tables, the /proc filesystem, etc.)
  - Optimize the fast path
  - Interpretation vs. compilation vs. intermediate “opcode” representations
  - Component failure as a common case to consider
  - Problems arising from interactions between replicated functionality (in this case, scheduling)

Some important points of discussion

- Flexibility of storage mechanisms:
  - Domains/inversions vs. heap-files/indexes
- Use of TID-lists common in modern DBMS so be doctrinaire? What about Data Independence?
  - One answer: you have to get transactions right for each “access method”

Some important points of discussion

- System R was often CPU bound (though that’s a coarse-grained assertion -- really means NOT disk-bound)
  - This is common today in well-provisioned DBMSs as well. Why?
- DBMSs are not monolithic designs, really
  - The RSS stuff does intertwine locking and logging into disk access, indexing and buffer management. But RDS/RSS boundary is clean, and RDS is decomposable.

Some important points of discussion

- Access control via views: a deep application of data independence?!
- Transactional contribution of System R (both conceptual and implementation) as important as relational model, and in fact should be decoupled from relational model.

The "Convoy Problem"

- A classic cross-level scheduling interaction
  - We will see this again!
- Poorly explained in the paper, three big issues
- Two interactions between OS and DB scheduling:
  - #1: OS can preempt a database "process" even when that process is holding a high-traffic DB lock
  - #2: DB processes sitting in DB lock queues use up their OS scheduling quanta while waiting (poorly explained in text), then are removed from the "multiprogramming set" and go to "sleep" – and an expensive OS dispatch is required to run them again

Convoy Problem (Con't)

- Last issue is DBMS uses a FCFS wait queue for lock
  - For a high-traffic DB lock, DB processes will request it on average every T timesteps
  - If the OS preempts a DB process holding that high-traffic DB lock, the queue behind the lock grows to include almost all DB processes
  - Moreover, the queue is too long to be drained in T timesteps, so it's "stable" -- every DB process queues back up before the queue drains, and they burn up their quanta pointlessly waiting in line, after which they are sent to sleep
  - Hence each DB process is awake for only one grant of the lock and the subsequent T timesteps of useful work, after which they queue for the lock again, waste their quanta in the queue, and are put back to sleep
- The result is that the useful work per OS waking period is about T timesteps, which is shorter than the overhead of scheduling – hence the system is thrashing

"Solution"

- Attacks the only issue that can be handled without interacting with the OS: #3 the FCFS DB lock queue
  - Paper's explanation is confusing
- Point is to always allow any one of the DB processes currently in the "multiprogramming set" to immediately get the lock without burning a quantum waiting on the lock
  - Hence no quanta are wasted on waiting, so each process spends almost all of its allotted quanta on "real work"
  - Technically, this is not "fair", however, it is "efficient"!
- Note that the proposed policy achieves this without needing to know which processes are in the OS' multiprogramming set

System R and INGRES

- The prototypes that all current systems are based on
- Basic architecture is the same, and many of the ideas remain in today's systems:
  - Optimizer remains, largely unchanged
  - RSS/RDS divide remains in many systems
  - SQL, cursors, duplicates, NULLs, etc.
    - » the pros and cons of duplicates. Alternatives?
    - » pros and cons of NULLs. Alternatives?
    - » grouping and aggregation
  - updatable single-table views
  - begin/end xact at user level, savepoints and restore, catalogs as relations, flexible security (GRANT/REVOKE), integrity constraints
  - triggers (!!), clustering, compiled queries, B-trees
  - Nest-loop & sort-merge join, all joins 2-way
  - dual logs to support log failure

Stuff they got wrong:

- Shadow paging
- Predicate locking
- SQL language
  - Duplicate semantics
  - Subqueries vs. joins
  - Outer join
- Rejected hashing

OS and DBMS: Philosophical Similarities

- UNIX paper: "The most important job of UNIX is to provide a file system"
  - UNIX and System R are both "information management" systems!
  - both also provide programming APIs for code
- Both providing what they think are crucial APIs for applications

Difference in Focus

- Bottom-Up (elegance of system) vs. Top-Down (elegance of semantics)
  - main goal of UNIX: provide a small *elegant* set of mechanisms, and have programmers (i.e. C programmers) build on top of it.
    - » They are proud that "No large 'access method' routines are required to insulate the programmer from system calls". OS viewed its role as *presenting hardware to computer programmers.*
    - » No native locking mechanisms!
  - main goal of System R and Ingres: provide a complete system that insulated programmers (i.e. SQL + scripting) from the system
    - » Guarantee clearly defined *semantics* of data and queries. After all, DBMS views its role as *managing data for application programmers.*
- Affects where the complexity goes!
  - to the system, or the end-programmer?
  - question: which is better? in what environments?
  - follow-on question: are internet systems more like enterprise apps (traditionally built on DBMSs) or scientific/end-user apps (traditionally built over OSes and files)? Why?

Achilles' Heel

- Achilles' heel of RDBMSs: a closed box
  - Cannot leverage technology without going through the full SQL stack
  - One solution: make the system extensible, convince the world to download code into the DBMS
  - Another solution: componentize the system (hard? RSS is hard to bust up, due to transaction semantics)
- Achilles' heel of OSes: hard to decide on the "right" level of abstraction
  - As we'll read, many UNIX abstractions (e.g. virtual memory) *hide too much detail*, messing up semantics. On the other hand, too low a level can cause too much programmer burden, and messes up the elegance of the system
  - One solution: make the system extensible, convince the fancy apps to download code into the OS
  - Another solution: componentize the system (hard, due to protection issues)

Communities

- Traditionally separate communities, despite subsequently clear need to integrate
  - UNIX paper: "We take the view that locks are neither necessary nor sufficient, in our environment, to prevent interference between users of the same file. They are unnecessary because we are not faced with large, single-file data bases maintained by independent processes."
  - System R: "has illustrated the feasibility of compiling a very high-level data sublanguage, SQL, into machine-level code".

Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

Summary

- Main goal of this class is to work from both of these directions, cull the lessons from each, and ask how to use these lessons today both within and OUTSIDE the context of these historically separate systems.