System design issues

• Systems often have many goals:
  - Performance, reliability, availability, consistency, scalability, security, versatility, modularity/simplicity

• Designers face trade-offs:
  - Availability vs. consistency
  - Scalability vs. reliability
  - Reliability vs. performance
  - Performance vs. modularity
  - Modularity vs. versatility
Engineering vs. research

• **Engineering:**
  - Find the right design point in the trade-off
  - Minimize cost/benefit, etc.

• **Research:**
  - Fundamentally alter the trade-offs
  - Ideally get “best of both worlds”
Example: Scheduler activations

- **Problem: Kernel-level threads suck**
  - Many expensive context switches
  - Kernel doesn’t know about application-specific priorities

- **Problem: User-level threads suck**
  - Scheduler doesn’t know which system calls block

- **Solution: New kernel interface**
  - Expose information needed by user-level scheduler: preemption, blocking system calls, I/O completion, …
  - Provides the best of both worlds
  - Facilitates other abstractions, too! (async I/O)
The end-to-end principle

- Place functionality closer to the endpoints
Example applications of principle

• **Link-by-link reliable message delivery**
  - Often ensured by application (higher-level reply)
  - Can’t trust every component of network
  - Inappropriate for many applications (e.g., voice over IP)

• **FIFO message delivery, duplicate suppression**
  - Redundant, just slows down two-phase commit, etc.

• **Security and data integrity checks**
  - Only make sense end-to-end
Applying the end-to-end argument

• Keep lower-level functionality for performance
  - E.g., Ethernet tries several times after a collision
  - Avoids unnecessarily triggering TCP retransmits

• Provide “least common denominator” abstractions
  - Can implement threads on async I/O, but not vice versa
  - Can implement threads or async I/O on sched. activations
  - Can implement POSIX on top of NFS, not vice versa
  - Can implement file system on Petal, not vice versa
Hints for low-level abstraction design

• Expose information
  - Lets applications/libraries make intelligent decisions (Is thread runnable? How much memory is available?)

• Expose hardware and other low-level functionality
  - Appel & Li: Exposing VM helps applications
  - Frangipani: Exploits low-level block protocol, locks

• Avoid “outsmarting” higher-level software
  - We still see papers on buffer cache management (UBM)
  - Maybe OS shouldn’t dictate the policy
Example: Security and key management

• Traditional approach
  - Application takes server name, provides secure abstraction
  - SSL: server name $\rightarrow$ encrypted socket
  - SSH: server name $\rightarrow$ encrypted remote login
  - Echo, AFS: server name $\rightarrow$ secure file system

• Problem: Many trade-offs in key management

• SFS approach: Key management in higher layer
  - Expose public keys in pathnames
  - Applications can use any key management
  - Use file system itself to implement key management
Other lessons in system design

• Determine an application’s exact reliability needs
  - RDBMS vs. Porcupine or DDS

• Determine application’s exact consistency needs
  - Bayou: general-purpose library, application-specific reconciliation

• Find useful abstractions that are not overkill
  - Petal (definitely), DDS (probably)

• Use feedback in allocating resources
  - Porcupine server selection, queue length in Mogul paper, eliminating hot spots in web caching
  - Shed work early in overload conditions (livelock)
Mechanisms

- **Concurrency:**
  - Threads
  - Asynchronous I/O
  - RPC & Network objects

- **Crash-recovery**
  - Write-ahead logging
  - Snapshot/checkpoint functionality

- **Failure recovery:** Two-phase commit (+ BFS)

- **Server selection:** consistent hashing
Conclusions

• System designers face many trade-offs
• When possible, gain the best of both trade-offs
  - Rethink layer interfaces and abstractions
  - Push functionality upwards (end-to-end principle)
• High-performance servers particularly demanding
  - Often uncomfortable fit on traditional OS abstractions
• Use “OS techniques” at application level