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

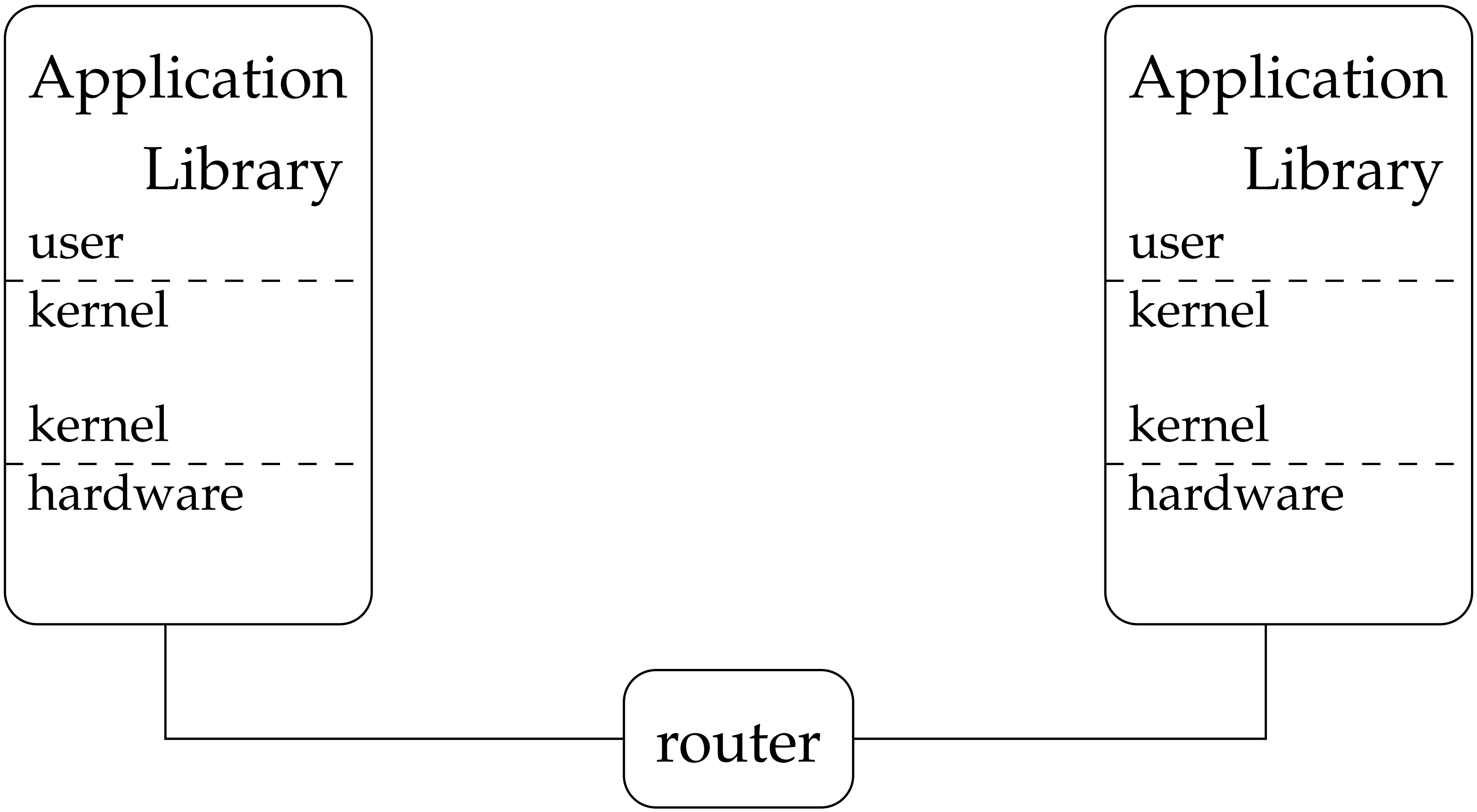
Engineering vs. research

- **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 abastractions, 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 mamagement (UBM)
  - Maybe OS shouldn't dictate the policy
  - Exokernel provides lower-level interface than buffer cache

Example: Security and key management

- **Traditional approach**
  - Application takes server name, provides secure abstraction
  - SSL: server name → encrypted socket
  - SSH: server name → encrypted remote login
  - TAOS: user/server name → secure connection
- **Problem: Many trade-offs in key management**
- **SFS (in lab 4): Key management in higher layer**
  - Expose public keys in pathnames:
    `/sfs/@class1.scs.cs.nyu.edu,wny5zs84js67egnhcq3aj2w5s8uymp4q`
  - Applications can use any key management
  - Use file system itself to implement key management

Current research at NYU

- **SUNDR secure file system**
  - End-to-end security requirement:
    Users should read data written other legitimate users
  - File system guarantees this without trusting server
- **Coral content-distribution network**
  - Most P2P data storage systems dicate data placement
    (E.g., store on closest node to ID in Chord or Pastry.)
  - Also attempt to provide reliability and consistency
  - Coral is optimized for placement of pointers
    End nodes determine placement of data
  - Gains efficiency by sacrificing consistency
    (perfect when want *some* copy of data, not *all*)

Other lessons in system design

- **Determine an application's exact reliability needs**
  - RDBMS vs. DDS / web caching
- **Determine application's exact consistency needs**
  - Ficus: application-specific resolvers
  - Bayou: general-purpose library, application-specific reconciliation
- **Find useful abstractions that are not overkill**
  - Petal (definitely), DDS (probably), Pastry/Scribe (maybe)
- **Use feedback in allocating resources**
  - Hot bucket handling in Cache Resolver, queue length in Mogul paper
  - Shed work early in overload conditions (livelock)

Conclusions

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

Brief Quiz Review

Transparent distributed systems

- Frangipani
- Amoeba
- Network Objects

Distributed system building blocks

- Ficus
- DDS
- Bayou
- Consistent hashing
- Scribe

Security

- TAOS
- BFS

Mechanisms

- **Concurrency:**
  - Threads
  - Asynchronous I/O
  - RPC & Network objects
- **Crash-recovery**
  - Write-ahead logging
  - Snapshot/checkpoint functionality
- **Distributed consistency: Two-phase commit, BFS**
- **Server selection: consistent hashing**