Distributed Systems

Operating System Support

Middleware layers

Applications

RMI, RPC and events

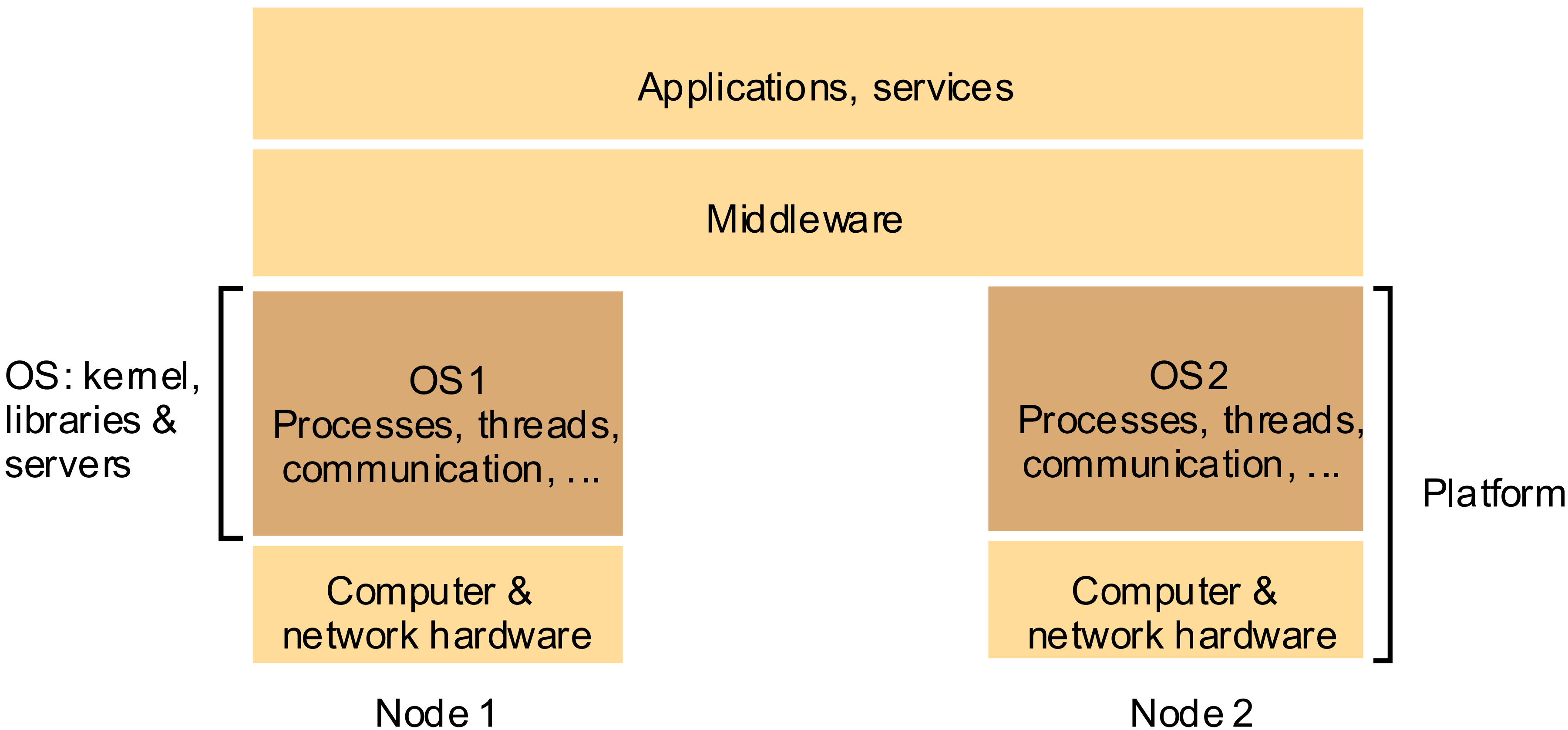
Request reply protocol

External data representation

Operating System

Middleware layers

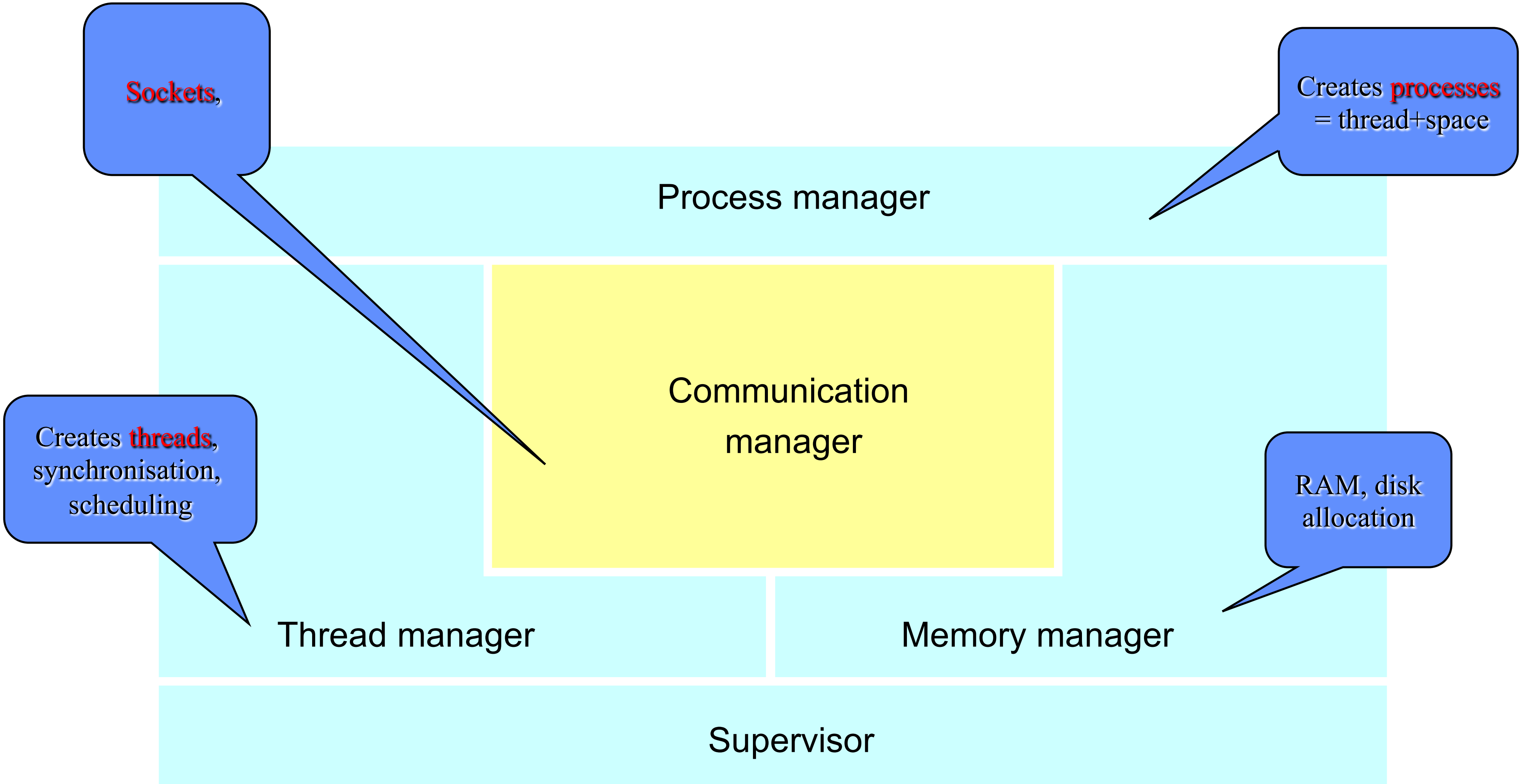
System layers



Why Operating Systems?

- I/O management, memory management, …
- Running concurrent processes
- Multiuser: more than one user can access the computer at the same time
- File ownership
- Security
- communication (sockets,…)
- protection of processes
  - Kernel

Core OS functionality



Core OS components

- Process manager
  - creation and operations on processes (= space+threads)
- Threads manager
  - threads creation, synchronisation, scheduling
- Communication manager
  - communication between threads (sockets, semaphores)
- Memory manager
  - physical (RAM) and virtual (disk) memory
- Supervisor
  - hardware abstraction (interrupts, exceptions, caches)

Program, Process, Thread

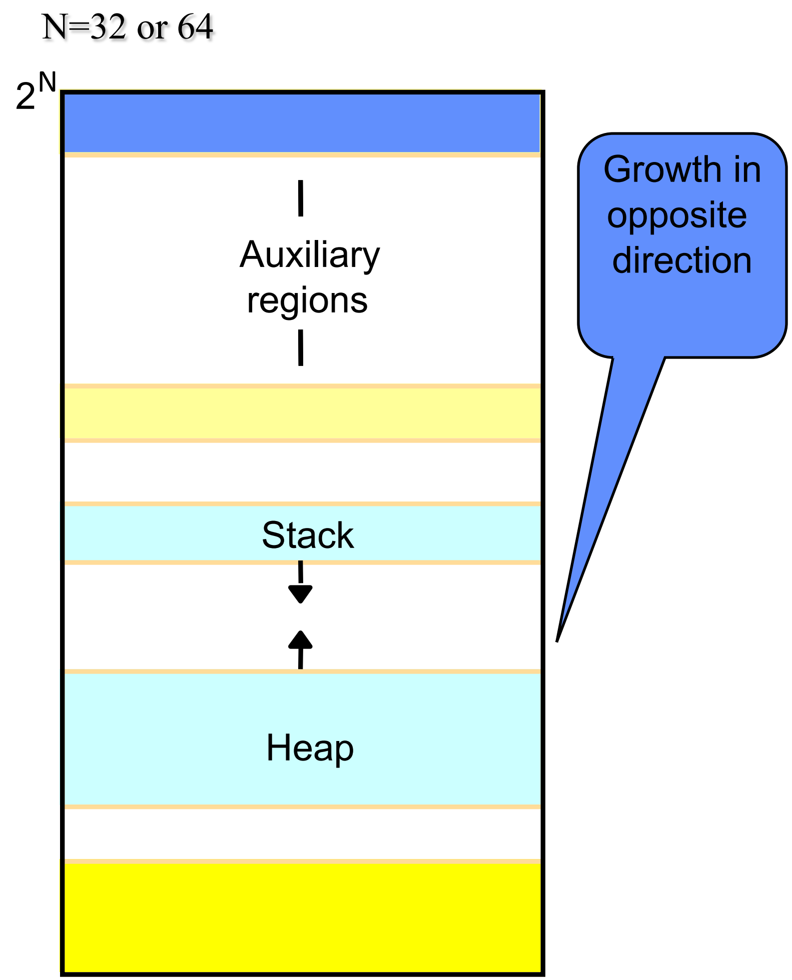
- process: a program that is currently executing
  - program ≠ process
- Thread (lightweight process): OS abstraction of an activity
  - : a path of code execution in a program
- Process = execution environment + one or more thread
- Example: when Java VM starts by an OS, then
  - a new process is created and
  - "a process spawns many threads"

Execution Environment

- Threads within the same process share the execution environment
- Execution Environment consists of :
  - an address space
  - thread synchronisation mechanism
  - communication interface (socket)
  - high level resources (file and window)

Address Space

- Unit of virtual memory
- One or more regions
- Text: where program stored
- Stack: local variables, such as program counters and return addresses are stored
- Heap: dynamically allocated memory is stored
- region vs contents



Processes vs Threads

- Processes
  - historically first abstraction of single thread of activity
  - can run concurrently, CPU sharing if single CPU need own execution environment
    - address space, registers, synchronisation resources (semaphores)
  - scheduling requires switching of environment (context switching)
- Threads (=lightweight processes)
  - can share execution environment
    - no need for expensive switching
  - can be created/destroyed dynamically
    - multi-threaded processes
    - increased parallelism of operations (=speed up)

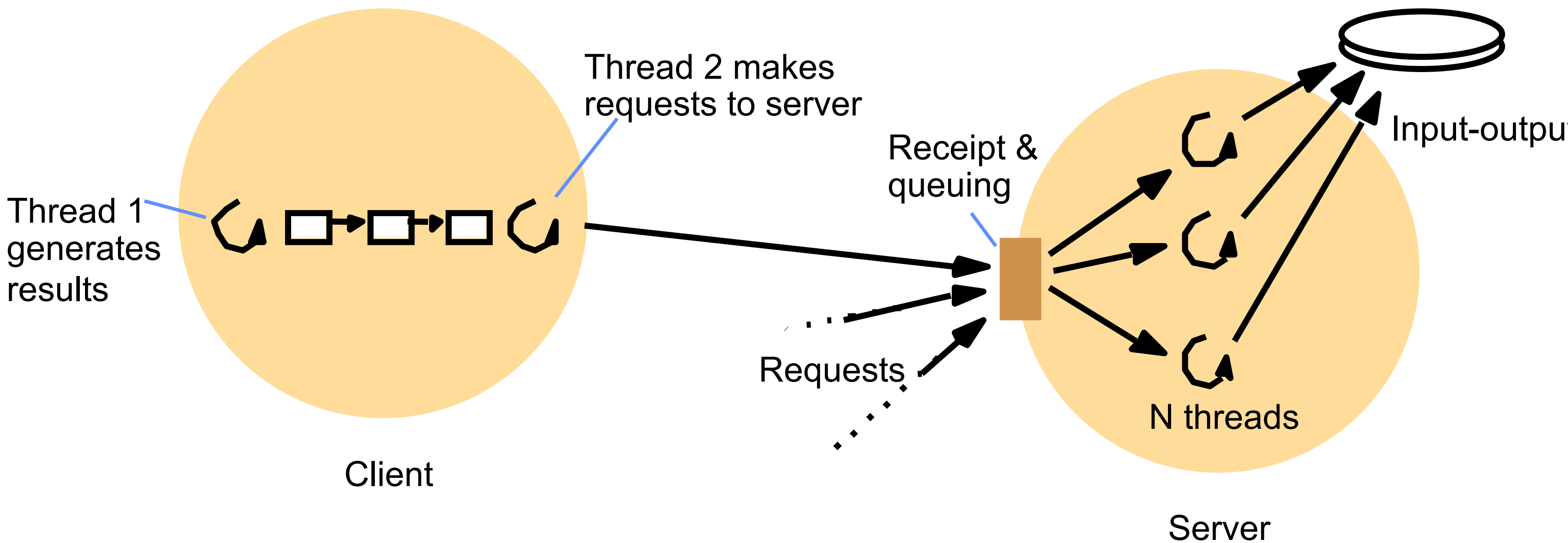
Why threads not processes?

- Process context switching
  - requires save/restore of execution environment
    - registers, program counters, etc
- Threads within a process
  - cheaper to create/manage
  - no need to save execution environments (shared between threads)
  - resource sharing more efficient and convenient
  - but less protection from interference by other threads

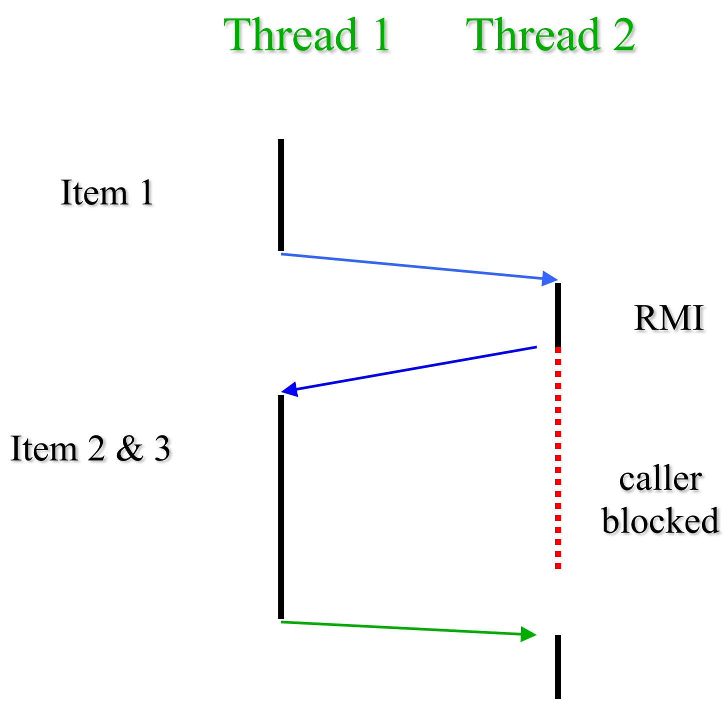
Role of threads in clients/servers

- On a single CPU system
  - threads help to logically decompose problem
  - not much speed-up from CPU-sharing
- In a distributed system, more waiting
  - for remote invocations (blocking of invoker)
  - for disk access (unless caching)
  - obtain better speed up with threads

Client and server with threads



Threads within clients



- Separate
  - data production
  - RMI calls to server
- Pass data via buffer
- Run concurrently
- Improved speed, throughput

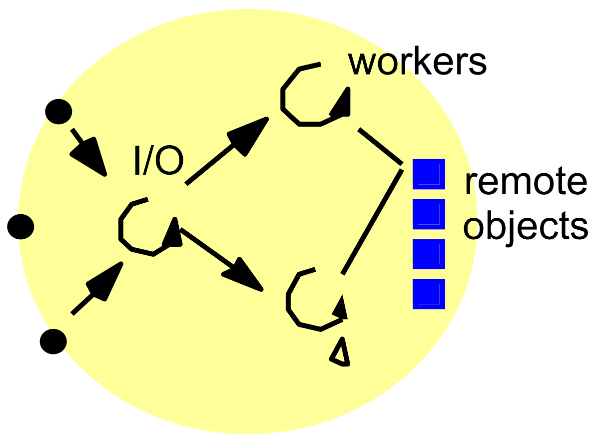
Server threads and throughput

- Assume stream of client requests, each 2ms processing + 8ms I/O.
- Single thread
  - max 100 client requests per second =1000/(2+8)
- Two threads, no disk caching
  - max 125 client requests per second =1000/8
- Two threads, with disk caching (75% hit rate)
  - theoretical mean max 500 client requests per second =1000/(0.75*0+0.25*8) = 1000/2
  - caching takes CPU time, so better estimate 1000/2.5 = 400

Multi-threaded server architectures

- Worker pool
  - fixed pool of worker threads, size does not change
  - can accommodate priorities but inflexible
- Other architectures
  - thread-per-request
  - thread-per-connection
  - thread-per-object
- Physical parallelism
  - multi-processor machines (cf casper, SoCS file server;)

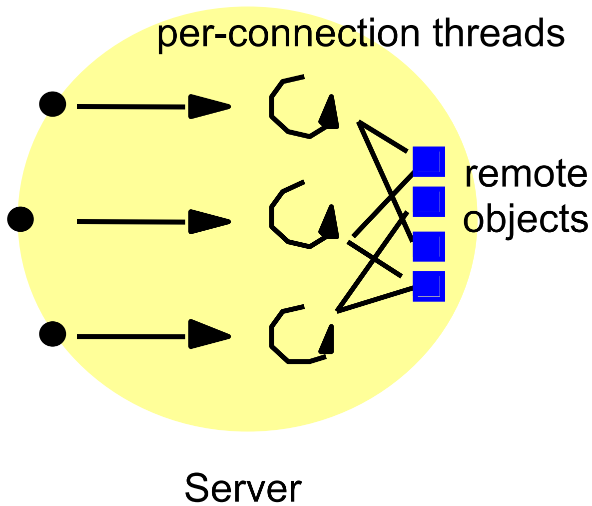
Thread-per-request



Server

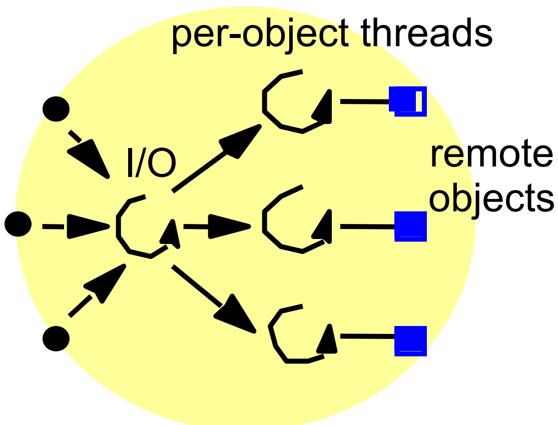
- Spawns
  - new worker for each request
  - worker destroys itself when finished
- Allows max throughput
  - no queuing
  - no I/O delays
- but overhead of creation & destruction high

Thread-per-connection



- •Create new thread for each connection
- •Multiple requests
- •Destroy thread on close
- •Lower o/heads
- •But unbalanced load

Thread-per-object



- As per-connection, but new thread created for each object.