COMP610: Topics in Engineering Enterprise Middleware Platforms

Charles Zhang

charlesz@cse.ust.hk

HongKong University of Science and Technology

Objectives

- Introduce basic concepts in middleware systems
- Familiarize with classic and recent research contributions
- Explore new areas for establishing research topics
- Obtain hands-on experience in engineering system code

Administration

- Research focus:
  - No tests, mid-terms, and final exams
  - Based on paper discussions and class projects
  - Don't believe anything that I said……
- Format:
  - 20% class participation
  - 30% class presentation
  - 50% class project
- Class intro:
  - … …

Class Presentation

- Cover a group of papers representing classic and recent research results
  - focusing on software engineering aspects of distributed systems in general
  - With emphasis on server-side computation
- Topics include:
  - Basic concepts (logical clock, middleware, AOP, architectural style, sync/async duality)
  - Conventional programming models (RPC, MOM, Pub/Sub, CORBA)
  - New models (MapReduce, REST, Stream, etc)
  - Naming and lookup (P2P, LDAP)
  - Heavy focus on server architecture
    - Locks, STM, Event-based programming, micro-kernel, SOA, etc
  - Debugging and fault detection
    - Predicate-based. Statistical methods. Capture and replay methods.
- You should send me your choices of papers by the end of first week
  - Each student can have one “easy” paper.
  - First come first serve
  - Grades divided by number of paper presentations

Presentation Specification

- Format
  - Each presentation is limited to 40 mins
  - 15 mins presentation/ 25 mins Q&A
  - Welcome to send me a week before for comments
- Presentation suggestion
  - Thesis or the goals of the paper, i.e., what is the paper trying to achieve.
  - Major contributions of the work, i.e., what is new about the work.
  - Briefly describe each contribution. Choose one (or two) contribution(s) that you think is most interesting or novel and explain it in some detail.
  - If there are experiments in the paper that highlight the benefits of the work, present some of these results. Ideally, the results you show will focus on the contributions that you explained in detail.
  - Next, present related work in the area, i.e., how is this work related to other projects or systems.
  - Present your conclusions about the work, i.e., does the paper achieve what it set out to achieve.

Q&A suggestion

- At least 5 questions aiming at understanding the following:
  - What were the main contributions of the work?
  - What were the advantages and disadvantages of the approach?
  - What are potential avenues for further work and improvements?
- Ask intelligent questions
  - Good question:
    - What aspects of the problem, which the related approaches failed to solve, are addressed by the authors?
    - How do the authors support the soundness claim?
  - Bad question:
    - What do you think the main contribution of the paper is?
  - You should prepare the answers to the questions

Class Project

- Goal: explore some aspect of the applications of software engineering techniques in building or evaluating distributed systems, middleware or operating systems
- Scope:
  - Individual projects, no team effort.
  - You can propose any project with related topics to this course.
  - You are welcomed to work on your research if a proper relationship can be defined.
  - You should talk to me before finalizing your project
- Options:
  - Implementation of a new system → Novel ideas
  - Evaluation of existing systems → Empirical study
  - Position paper → Research students

Project deliverables (option 1,2)

- Project Description: 1 page (Due Feb 26, 2009)
  - Title of project, names of project members
  - Purpose of the project
  - Expected outcome or result of the project
  - Three or more intermediate steps in the project
- Status Report: 3-4 pages (Due March 26, 2009)
  - Title of project, names of project members
  - Purpose of the project
  - Expected outcome or result of the project
  - Background research with bibliography of relevant research
  - Research methodology or approach taken in the project
  - Status of implementation
  - Experiments that will be performed
- Final Report: 8-10 pages (Due May 16, 2009)
  - Title of project, names of project members
  - Purpose of the project
  - Expected outcome or result of the project
  - Background research with bibliography of relevant research
  - Details of the research methodology or approach taken in the project
  - Status of implementation
  - Evaluation results
  - Conclusion: did your results meet expectations
  - Future work
  - Code

Project deliverables (option 3)

- Format:
  - Conduct detailed background research and cover as much literature as possible.
  - Compare the approaches and discuss the benefits or drawbacks of each.
  - Come up with your "position".
    - Novel statement based on solid background research and sound judgment
    - Not a survey of previous work
    - Not essential to implement or evaluate a system.
    - Grading will be stricter regarding the quality of the final report and the novelty of the idea.
- Specifics:
  - There may be no implementation and evaluation
  - Background research should be more thorough
  - Focus of the paper should be on the details of your approach which should clearly justify your position, i.e. your novel statement.

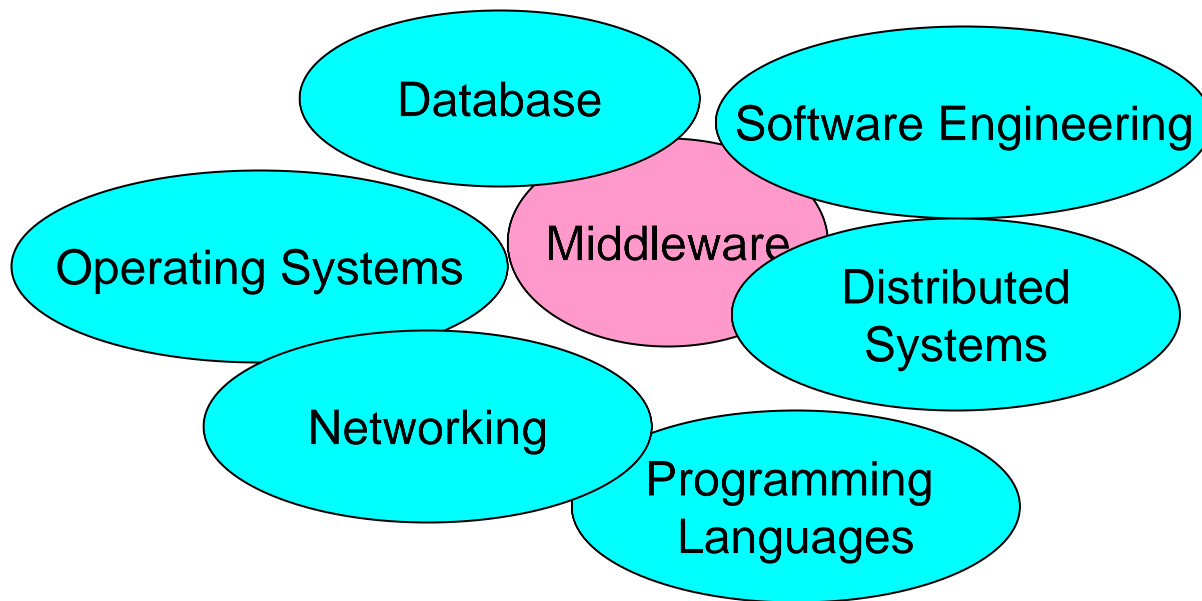
What is middleware?

- Definition by common references:
  - DCE, CORBA, DCOM, COM+, CCM, .NET, Java RMI, J2EE, JMS, Web Services
  - A mixture of standards, particular technologies, and conglomerations of technologies.
- Definition by technologies:
  - Remote procedure call
  - Event notification and messaging systems
  - Application container
  - Transaction monitors
  - Naming services
- Our definition:
  - Software component
    - supports a type of specialized distributed computation
    - gives programmers uniform programming abstractions

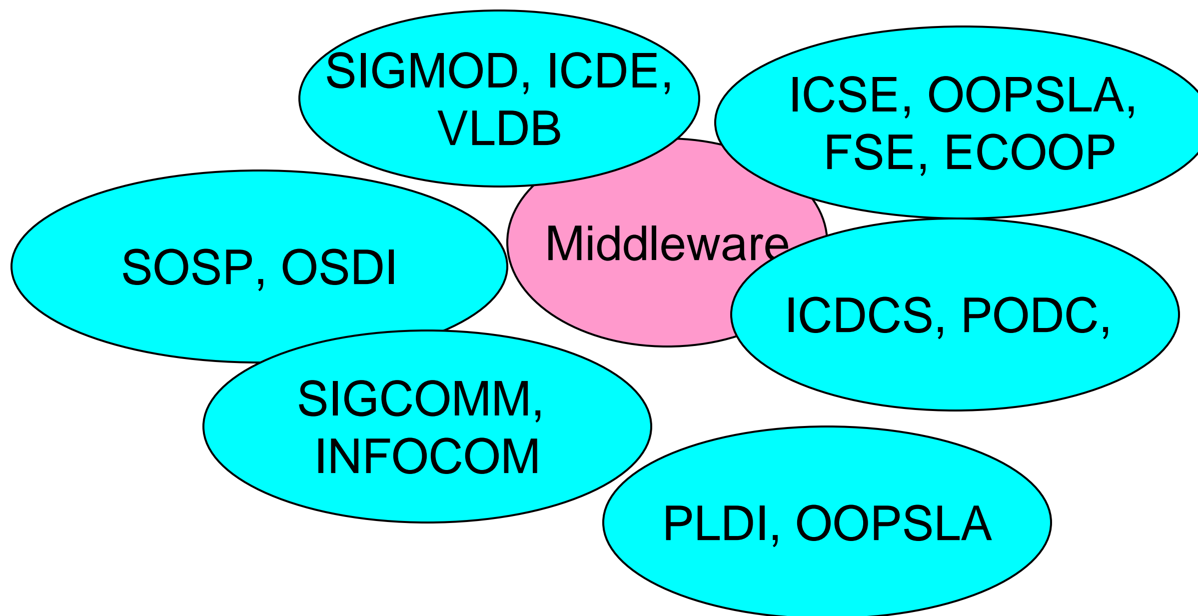
Why not just distributed systems?

- Commonalities:
  - Enable networked computations
  - Sensitive to QoS issues such as performance, reliability, and security
- Differences:
  - Issues in distributed systems more related to the dynamics of the interaction behaviors.
    - How should a distributed file system support a read-dominated traffic?
    - How does a composite event get delivered close to its source?
  - Issues in middleware more related to software engineering tasks in distributed systems.
    - Does RPC produce more error-prone applications?
    - How to build a distributed type system?
  - However, solutions to problems often need to address both

Where is middleware?



Where is middleware?



Categories of Middleware

- Remote invocation mechanisms:
  - DCOM, CORBA, DCE, Sun RPC, Java RMI, .Net Remoting, SOAP-RPC, Protocol Buffer
- Naming and directory service:
  - JNDI, LDAP, COS Naming, DNS, WS-*, DHT
- Message oriented middleware:
  - Event notification systems
  - Publish/Subscribe systems
- Tuple space
  - Linda, inspired JavaSpace, evolved to JINI
- Transaction monitors (TP Monitor)

Central Issues of Middleware

- Heterogeneity
- Transparency
- Openness
- Scalability

Heterogeneity

- Equipment (Networks/Hardware/VM)
  - IP/ATM/Proprietary networks
  - TCP/UDP/Overlay
  - Endianness/Connectedness
- Operating systems
  - System calls
  - Scheduling characteristics (distributed priority)
- Programming languages
  - Type systems/Primitive data/Memory model

Transparency (Tanaubaum et al.)

- Access
  - Data representation/Naming convention
- Location
  - Physical location (IP/port) of the resource
- Migration
  - Resource moves from one node to another node
- Relocation
  - Mobility of users of resources
- Replication
  - Addition of computation resources
- Concurrency
  - Maximization of the resource sharing
- Failure
  - Specific failures (resource / hardware)
  - Recoverable failures

Openness

- Implementation independence
  - A middleware implemented by company A vs. company B
- Language independence
  - A C++ based system vs. a Java-based system
- Interoperability
  - Can RPC package of company A talk to that of company B?
- Portability
  - Application for system A runs on system B without modification
- Middleware is heavily based on standards
  - Object management group (OMG): CORBA/UML/Meta Information
  - Internet engineering task force (IETF) RFCs: TCP, HTTP
  - WWW consortium. XML, WS-XXX

Scalability

- Definition for scalability:
  - Computational capacity w.r.t. users and resources
  - Geographical scalability w.r.t. physical distance between users and resources
  - “Scale to administration” w.r.t. to management tasks
- Computation capacity:
  - Centralized/Decentralized
  - Synchronous/Asynchronous
  - RPC(stateful) / Messages(stateless)
  - Caching/Consistency
  - … …

Remote Procedure Call

- RPC a version of client/server communication
  - attempts to make remote procedure calls look like ordinary procedure calls.
  - easy to write programs with model programmers are familiar with
  - good match for many distributed applications (client/server)
  - hides details (e.g., marshaling/unmarshaling)
- alternatives?
  - directly programming with sockets
  - distributed-shared memory
  - map/reduce
  - Dryad
  - MPI ...
- RPC seems to have won (or lost against sockets?)
  - XML RPC
  - Java RMI
  - Sun RPC
  - Protocol buffer
  - map/reduce + dryad implemented using RPC

Benefit of RPC

Socket based approach

```
socket_ = new Socket();
socket_.setSoTimeout(timeout);
socket_.connect(new InetSocketAddress(host_,port_));
in_ = socket_.getInputStream();
out_ = socket_.getOutputStream();
byte type[];
byte len[];
byte src[];
type = Util.intToBytes(JOBTYPE);
len = Util.intToBytes(numbers_.length);
cpudata_ = new byte[numbers_.length+len.
length+type.length];
src = numbers_;
System.arraycopy(type, 0, cpudata_, 0, type.length);
System.arraycopy(len, 0, cpudata_, 4, len.length);
System.arraycopy(src, 0, cpudata_, 8, src.length);
out_.write(data_);
out_.flush();
socket_close();
```

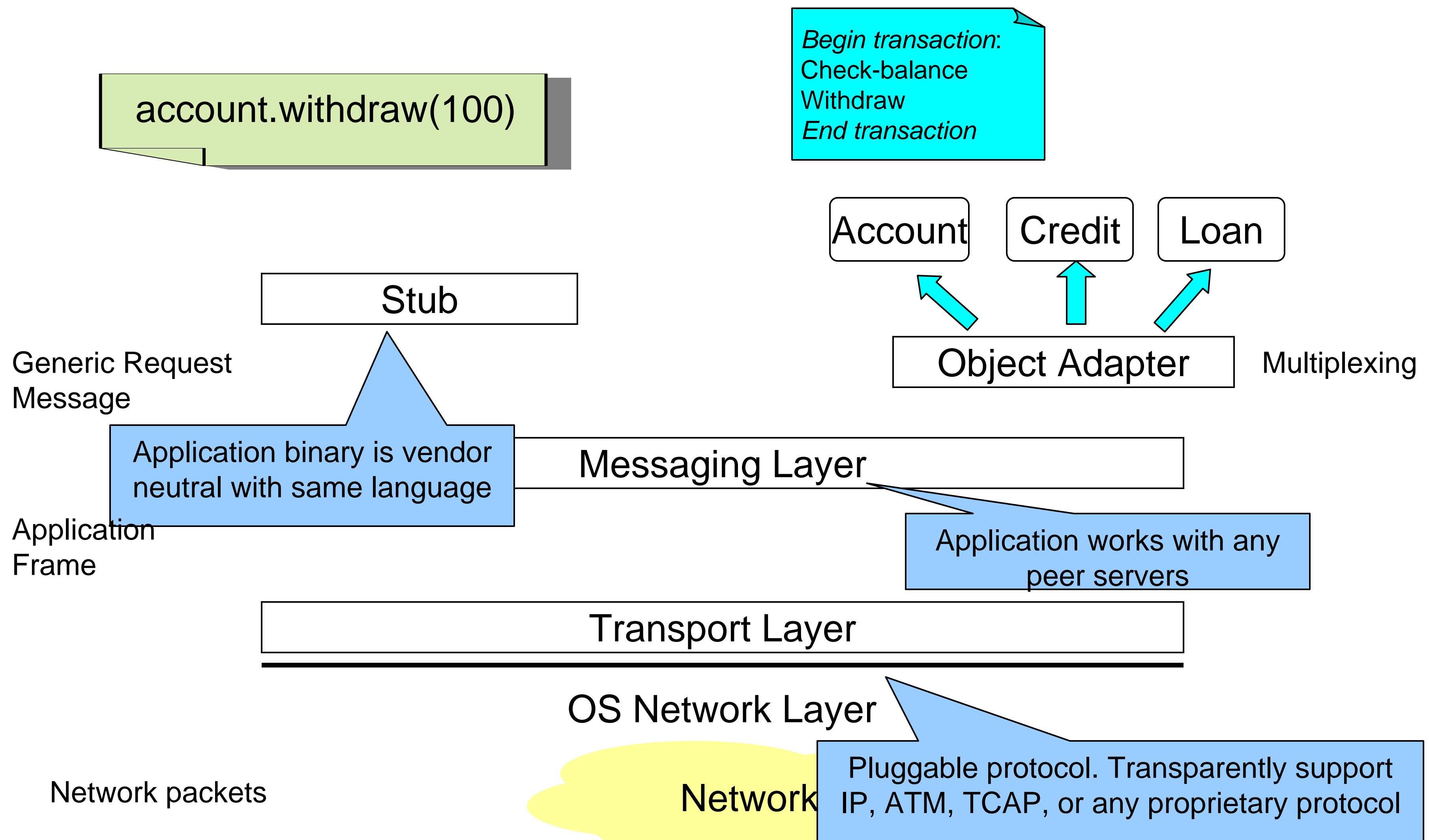
CORBA RPC

```
interface Data {
int type;
long number;
}
```

```
interface Server {
bool receive(in Data d);
}
```

```
ORB orb = org.omg.CORBA.ORB.init(…);
Data d = new Data(0,12345667);
String ref_= //get the object reference,
possibly through naming
org.omg.CORBA.Object obj =
        orb.string_to_object(ref_);
Server server = ServerHelper.narrow(obj);
server.receive(d);
```

RPC middleware: a layered construction



Challenges of RPC: Non-Idempotency

- Idempotent operations → Operations can be executed more than once
  - What the balance of my account?
  - Deduct 10 dollars from my balance…
- Semantics in the face of:
  - communication failures (messages may be delayed, variable round trip, never arrive)
  - machines failures (did server fail just before the processing the request or just after?)
  - sometimes impossible tell the difference between communication failures and machine failures
- Example: acquiring a lock using RPC
  - if client and server stay up, client receives lock
  - if client fails, it may have the lock or not (server needs a plan!)
  - if server fails, client may have lock or not
  - if server immediately recovers, client will receive an exception
    - what does a client do in the case of an exception? need to implement some application-specific protocol
    - ask server, do i have the lock? server needs to have a plan for remembering state across reboots e.g., store locks on disk.

Challenges of RPC: Scalability

- Hardwired knowledge of identities between communication parties (coupling)
- Complexity
  - Difficulty in changing declared operations.
  - Overhead of modifying a multi-party interaction
  - Caching consistency in face of binding changes
  - Distributed deadlock

Challenges of RPC: Transparency

- Method calls are expensive

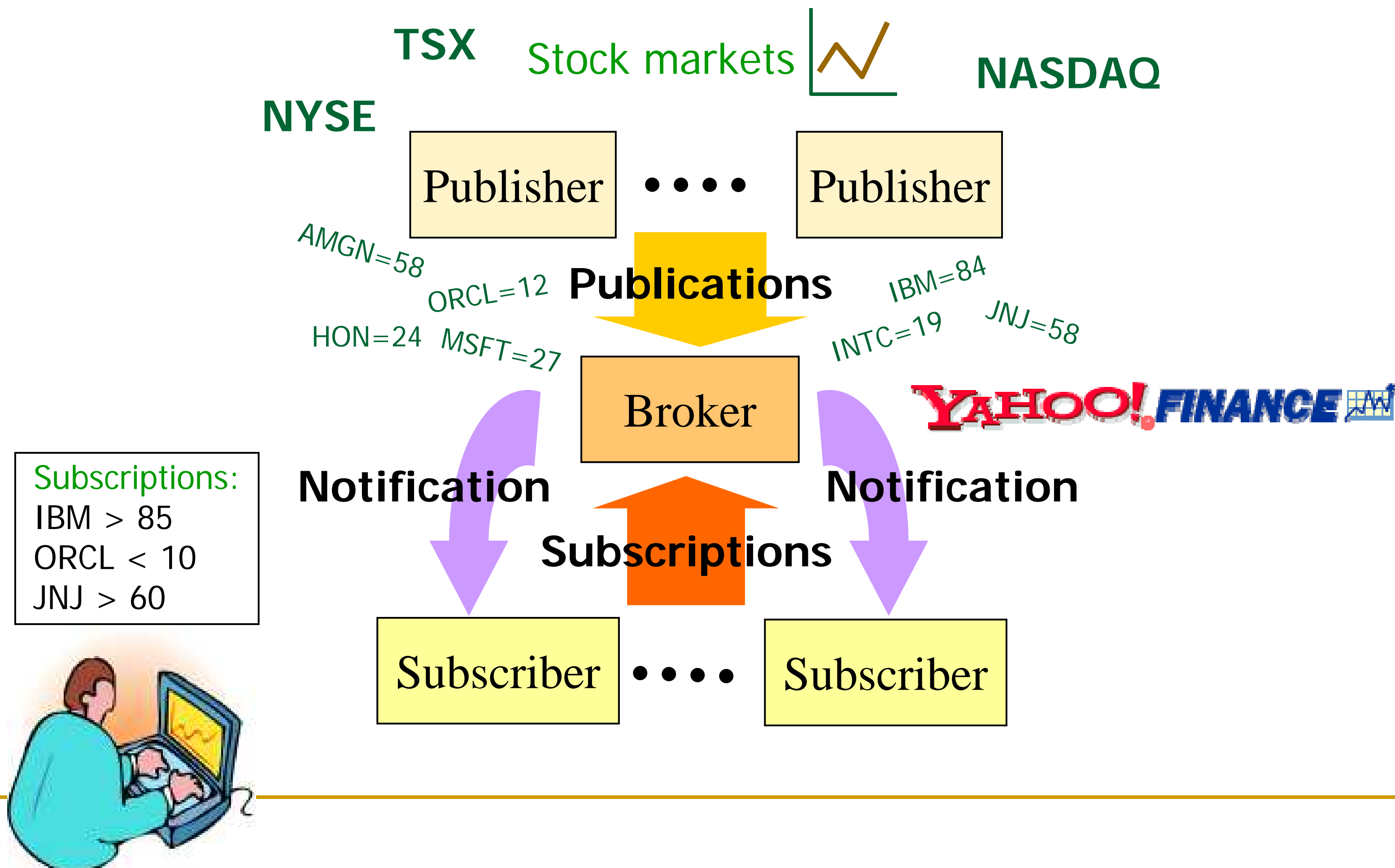
```
for(int id=0;id<1000;id++) {
    int balance = bank.getBalance(id);
    bank.setBalance(id, balance+10);
}
```

- Parameter passing is tricky

```
for (int id=0;id<1000;id++){
    blog.setImage(id, image);
or
    blog.setImage(id, imagedb.getImage(id));
}
```

- Often requires compiler or infrastructure support

Publish/subscribe systems



Application examples & domains

- Alerting services
- Enterprise application integration
- Monitoring, surveillance, and control
- Workload management
- Network and distributed system management
- Selective information dissemination applications
- Location-based services
- Financial applications (e.g., stock order processing)

Subscriptions

- Subscribe to a News-Channel (***channel-based***)
- /news/Canada/sports (***topic-based***)
- programming language type (***type-based***)
- trigger registered with a table (***DB trigger***)
- (category=soccer) AND (team=KSC) AND (score > 2) (***content-based***)

Publications

- A message to a news group
- A message classified according to a topic-hierarchy (type hierarchy)
- A (typed) programming language object
- A database operation modifying a table or a constraint on a table (e.g., insert, update, delete from table)
- {(category, soccer), (team, KSC), (score, 2)}

The matching problem(s)

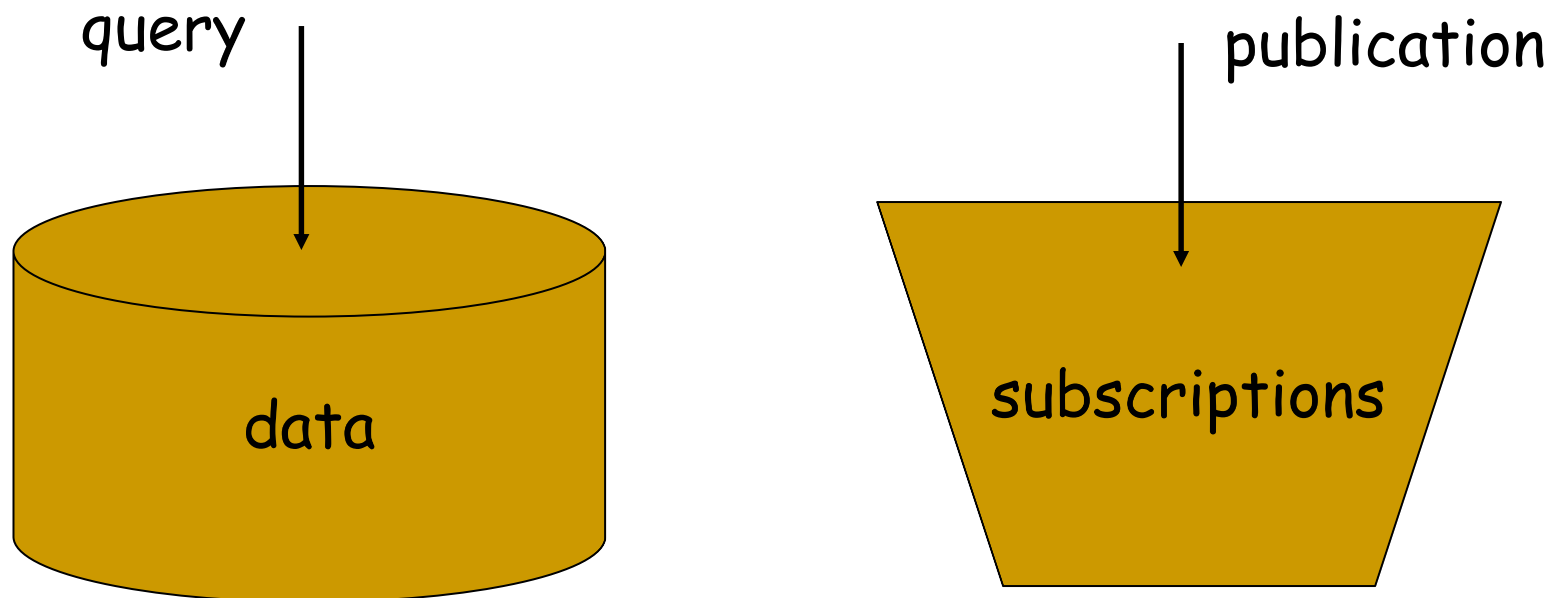
Given a set **S** of subscriptions and a publication **p**, find **all s in S** such that **p matches s.**

- S = (category=soccer) AND (team=KSC) AND (score > 2)
- P = {(category, soccer), (team, KSC), (score, 2) }
- No match

semi-structured	structured	text-based
XML/Xpath		text / search strings
	DB tables/SQL	
$\{(a_1,v_1), \ldots,(a_n,v_n)\}$ / Boolean functions		sentences / regular expressions

Interesting, …

but isn't this a database problem?



Query and ***subscription*** is essentially the same.
Data and ***publication*** is essentially the same.

However, the two <u>problems are inverse.</u>

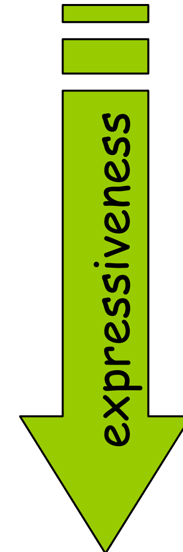
Benefits of the p/s paradigm

- **independence** of participants
- lends itself well to distributed system development
  - de-coupled development & processing
  - (dynamic) system evolution
- **interaction** with **large number** of entities facilitated
- naturally supports **non-continuous** operations
- potential for scalability & fault-tolerance
- open for (legacy) **system integration** on either end

*Of course it is not a **one size fits all** paradigm, but a good solution for certain kinds of problems.*

The evolution of publish/subscribe

- Channel-based model
  - USENET news, CORBA Event Service
- Topic-based model
  - TIBCO/RV, JMS, OMG DDS
- Type-based model
  - JavaSpaces, JMS, CORBA Notification Services,
  - OMG DDS
- Object-based model
- Content-based model
  - Various research prototypes, including: IBM Gryphon, ToPSS, LeSubscribe, …
- Subject Spaces
- Orthogonally: Content-based routing (a network of publish/subscribe systems that routes information)

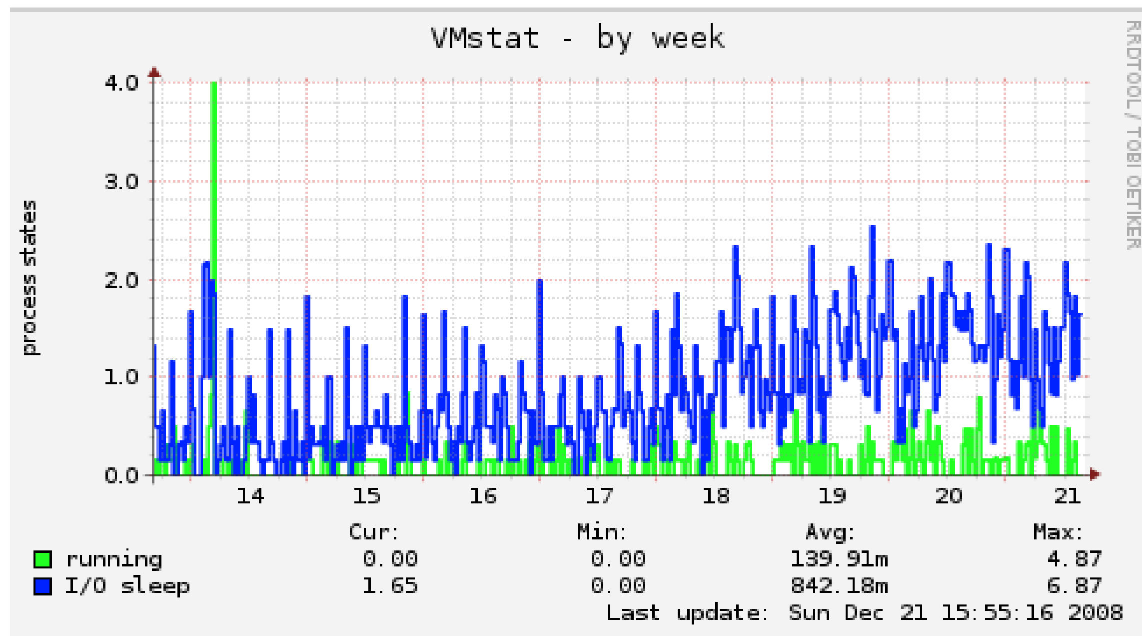


Server-side issues

- Concurrency
- Events
- Patterns oriented architecture

Concurrency

- Hardware is inherently parallel, but software is inherently sequential
- Can we make more efficient use of resources?



Ways to get concurrency

- What we want is *I/O concurrency*
  - Ability to overlap I/O wait with other useful work.
  - In web server case, I/O wait mostly for net transfer to client.
  - Could be disk I/O: compile 1st part of file while fetching 2nd part.
  - Could be user interaction: emacs GC while waiting for you to type.
- Performance benefits of I/O concurrency can be huge
  - Suppose we're waiting for disk for client one, 10 milliseconds We can probably serve 100 other clients from cache during that time!
- Typical ways to get concurrency.
  - Multiple processes
  - One process, many threads
  - Event-driven Depends on O/S facilities and type of application

Multi-process I/O concurrency

- Mechanism:
  - Start a new UNIX process for each client connection / request
  - Master process hands out connections.
  - Now plenty of work available to keep system busy : fork() after accept()
- Benefit:
  - Preserves original s/w structure.
  - Isolated: transient bug for one client does not crash the whole server
  - Most interaction hidden by O/S. e.g. lock the disk queue.
  - If > 1 CPU, CPU concurrency as a side effect
- Drawbacks:
  - Cost of starting a new process (fork()) may be high. (What are the costs?)
  - Processes are fairly isolated by default, e.g. they do not share memory
    - What if you want a web cache? Must be shared among processes. Or even just keep statistics?

Multi-thread I/O concurrency

- Mechanism
  - Looks a bit like multiple processes But thread_fork() leaves address space alone
  - So all threads share memory
  - One stack per thread, inside process
- Benefit
  - Seems simple -- still preserves single-process structure.
  - Potentially easier to have e.g. shared web cache
- Drawbacks
  - Programmer needs to know about some kind of locking.
  - Easier for one thread to corrupt another
- There are some low-level but very important details that are hard to get right.
  - What happens when a thread calls read()? Or some other blocking system call? Does the whole process block until disk I/O has finished? If you don't get this right, you don't get I/O concurrency.

Kernel-supported threads

- Mechanism:
  - O/S kernel knows about each thread
    - It knows a thread was just blocked, e.g. in disk read wait Can schedule another thread
  - What does kernel need for this?
    - Per-thread kernel stack.
    - Per-thread tables (e.g. saved registers).
  - Semantics:
    - per-process resources: addr space, file descriptors
    - per-thread resources: user stack, kernel stack, kernel state
  - Kernel can schedule one thread per CPU
- Drawbacks (kernel pays expenses)
  - Just like processes, kernel has to help create each thread
  - Kernel has to help with each context switch So it knows which thread took a fault.
  - lock/unlock must go through kernel, but bad for them to be slow
  - Many O/S do not provide kernel-supported threads, not portable

User-level Threads

- Mechanism
  - Implemented purely inside program, kernel does not know
  - User scheduler for threads inside the program
    - Know when a thread is making a blocking system call. Don't actually block, but switch to another thread.
    - Know when I/O has completed so it can wake up original thread.
    - Thread library has fake read(), write(), accept(). system calls library knows how to *start* syscall operations without waiting
    - Library marks threads as waiting, switches to a runnable
    - Thread kernel notifies library of I/O completion and other events library marks waiting thread runnable
- Drawbacks
  - User-level threads need significant kernel support
    - Non-blocking system calls are hard to support in general
    - Uniform event delivery mechanism.

Discussion

- I/O centric concurrency ←→ Multi-core
- Issues with programming models

Event-driven programming

- Mechanism
  - Organize code around arrival of events
  - Write software in state-machine style
    - When this event occurs, execute this function
  - Library support to register interest in events
- Benefit
  - Preserves the serial natures of the events
  - Programmer sees events/functions occurring one at a time
- Drawback
  - Unnatural for some continuous logic
  - Manual stack management

Event-driven Programming

- How to program in event style?
  - Identify events and appropriate responses:
  - Write a loop that handles incoming events (I/O events)
  - Translate low-level system events into application level events
  - Maintain individual application state
- Drawbacks
  - Writing this event loop for each program is tedious
  - What if your program does the one thing in parallel?
    - Have to partition up state for each client
    - Need to maintain sets of file descriptors
  - What if your program does many things? e.g. let's add DNS resolution
    - Hard to be modular if event loop knows about all activities.
    - And knows how to consult all state.
- We would prefer abstraction
  - Use a library to provide main loop (e.g. libasync)
  - Programmer provides "callbacks" to handle events
  - Break up code into functions with non-blocking ops let the library handle the boring async stuff

Event-driven Programming (cont.)

- It's unfortunately hard for async programs to maintain state
  - Ordinary programs and threads use variables
    - Persist across function calls, and blocking operations.
    - Stored on the stack.
    - Async programs can't keep state on the stack. Since each callback must return immediately.
- How to maintain state across calls?
  - Use global variables
  - Use the heap: In C or C++, programmers package up state in a generic data structure.
    - Hard to program
    - No type safety
    - Must declare structs for every set of state transfer
    - User has to manage memory in potentially tricky cases
  - Use closure
    - Library help
    - Language support : LISP/Java Inner classes/AspectJ around advices

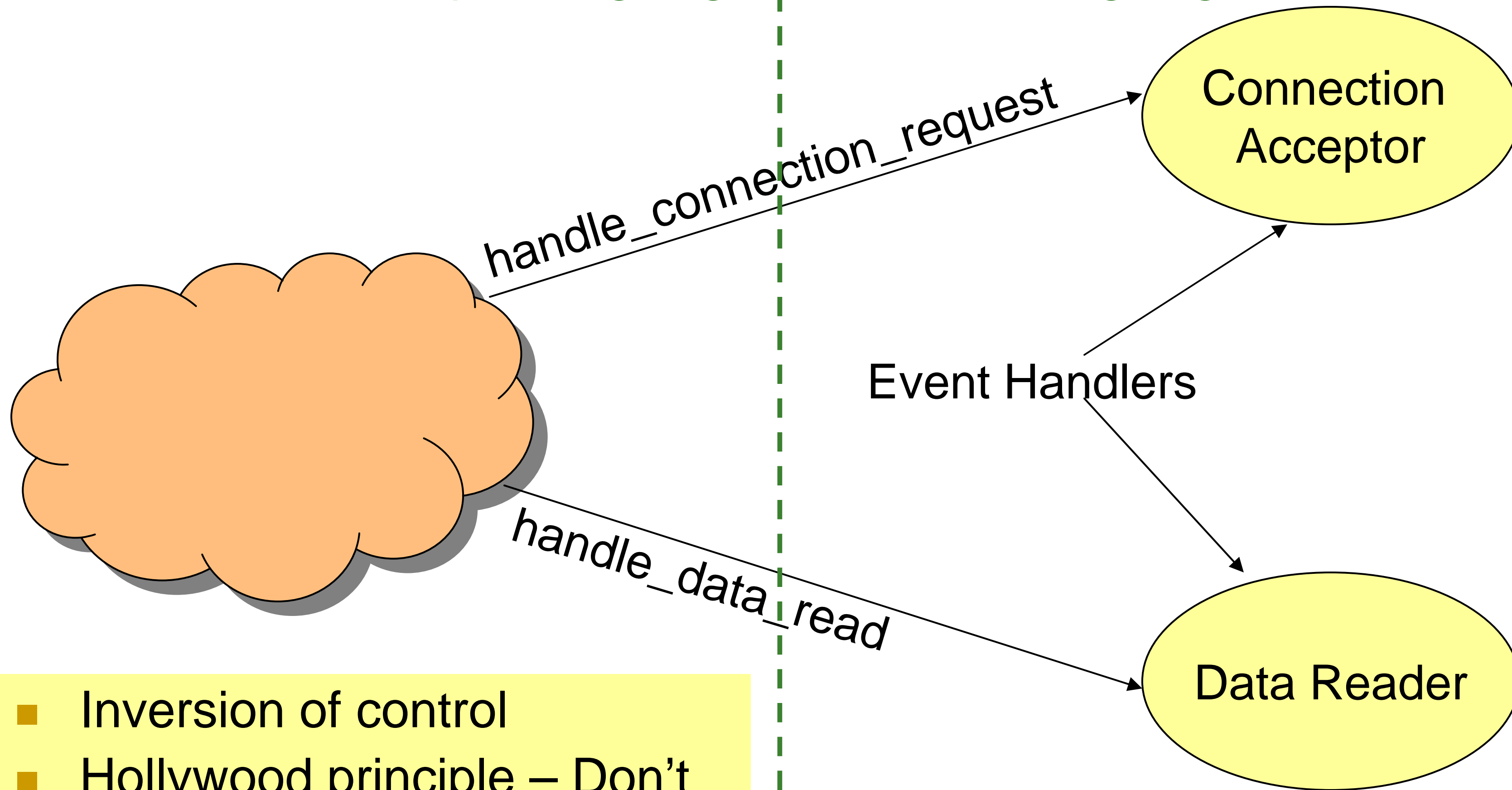
Pattern-centric Server Architecture

- Design server architecture is challenging
  - Loads
  - Platforms
- Conventionally dominated by scenario-specific designs
  - Specific to load profiles
  - Naturally described in design patterns
- Examples:
  - Reactor
  - Half Sync/Half Async

Problems with Threaded Approach

- Multi-threading may increase code complexity
- Multi-threading/processing adds overhead
  - Context switching (especially among processes)
  - Synchronization for shared data, other resources
- What if we could make 1 thread responsive?
  - Better resource utilization by aligning threading strategy to # of available resources (like CPUs)
  - Also, multi-threading may not be available in all OS platforms (e.g., embedded ones)

Alternative: Event Driven Server



- Inversion of control
- Hollywood principle – Don't call us, we'll call you ("*there is no main*")

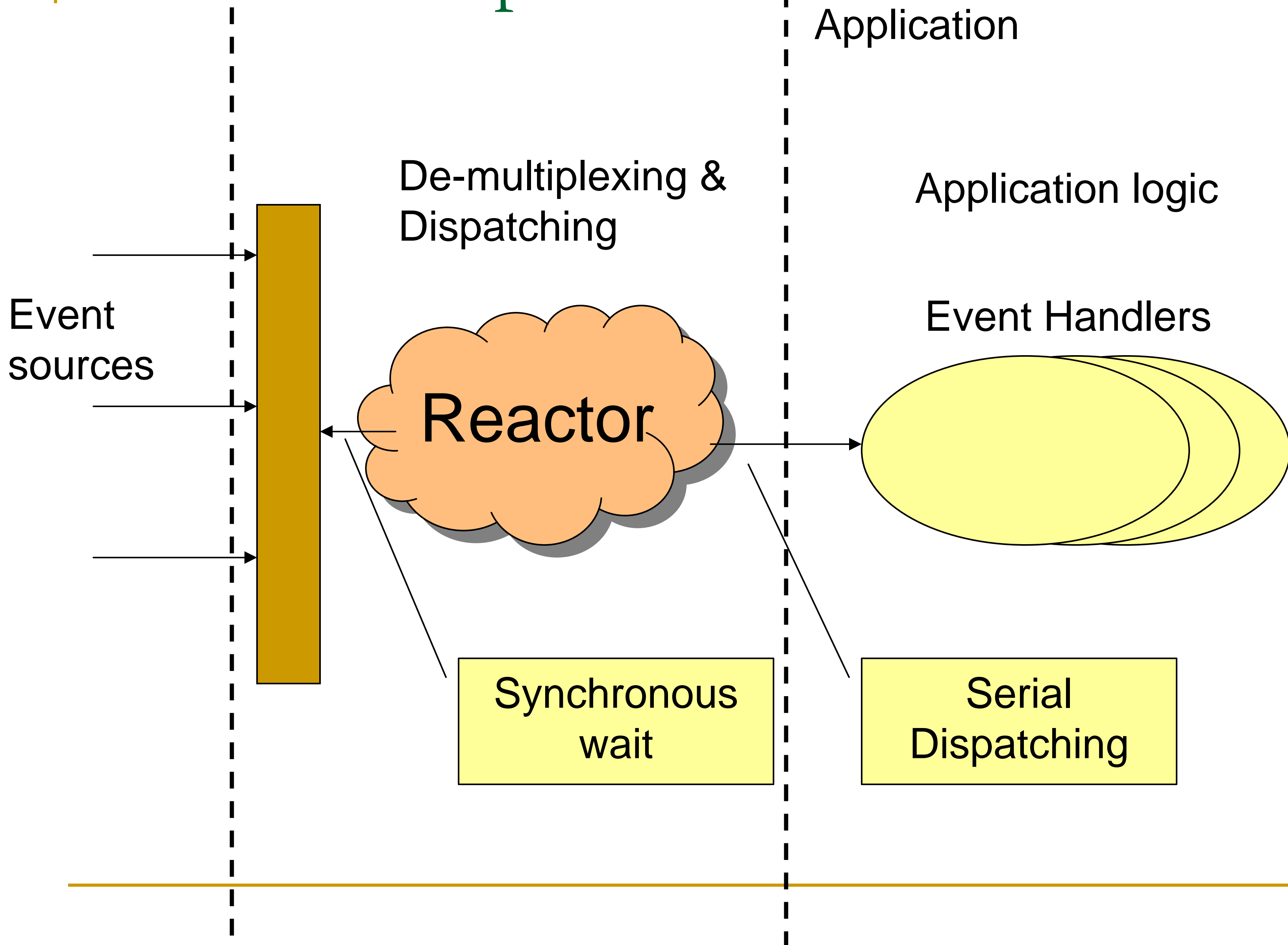
Reactor Pattern (Dispatching Logic)

- An *architectural pattern*
  - Context: event-driven application
  - Concurrent reception of multiple service requests, but serial processing of each one
- Dispatch service requests
  - *Calls* the appropriate event handler
- Also known as
  - Dispatcher, Notifier, Selector (see Java NIO)

Design Forces

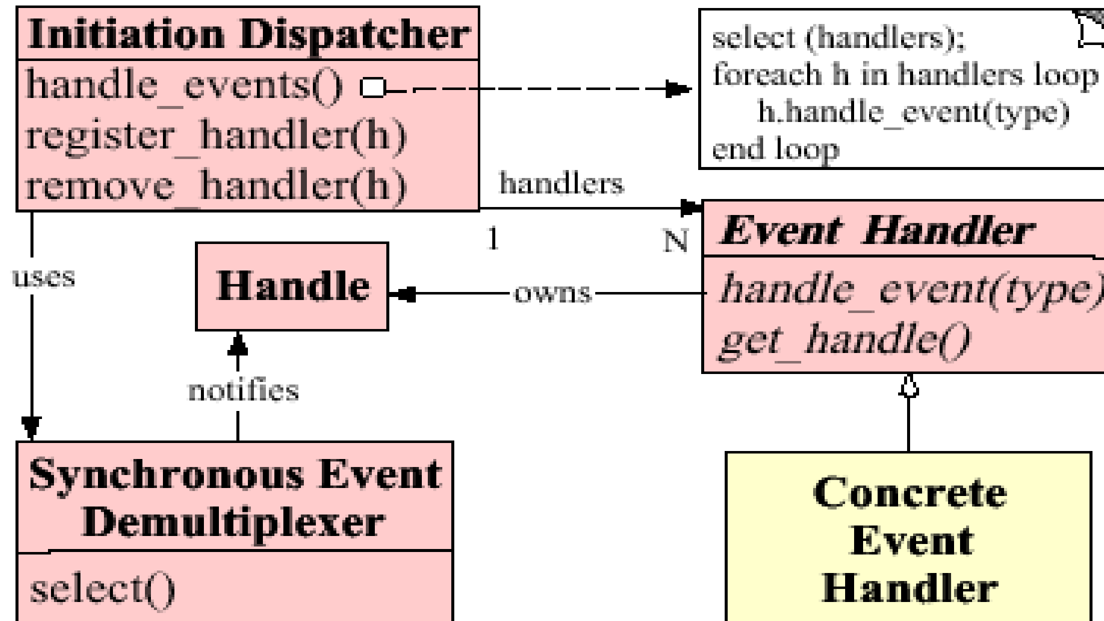
- Enhance scalability
- Maximize throughput
- Minimize latency
- Reduce effort that is needed to integrate new services into server

Solution – Separation of Concerns



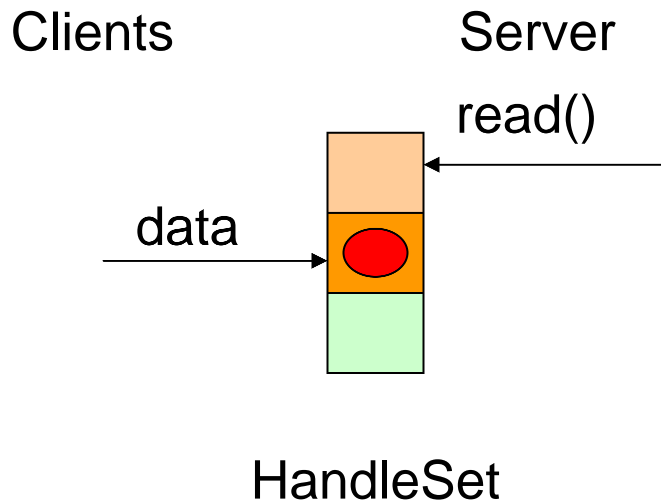
Reactor Pattern Structure

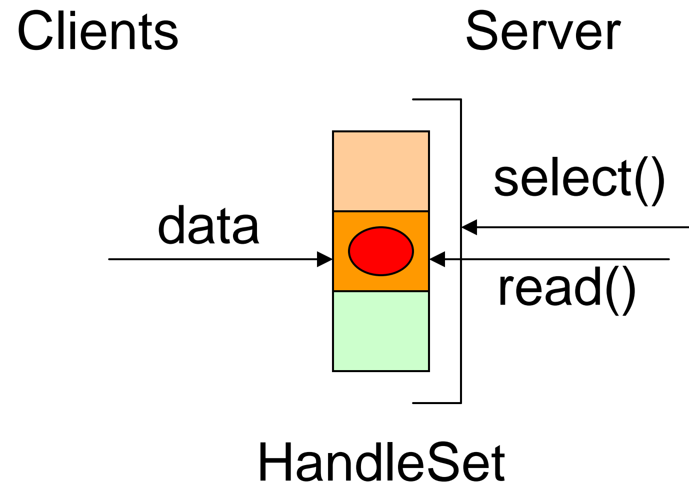
a.k.a
"the reactor"



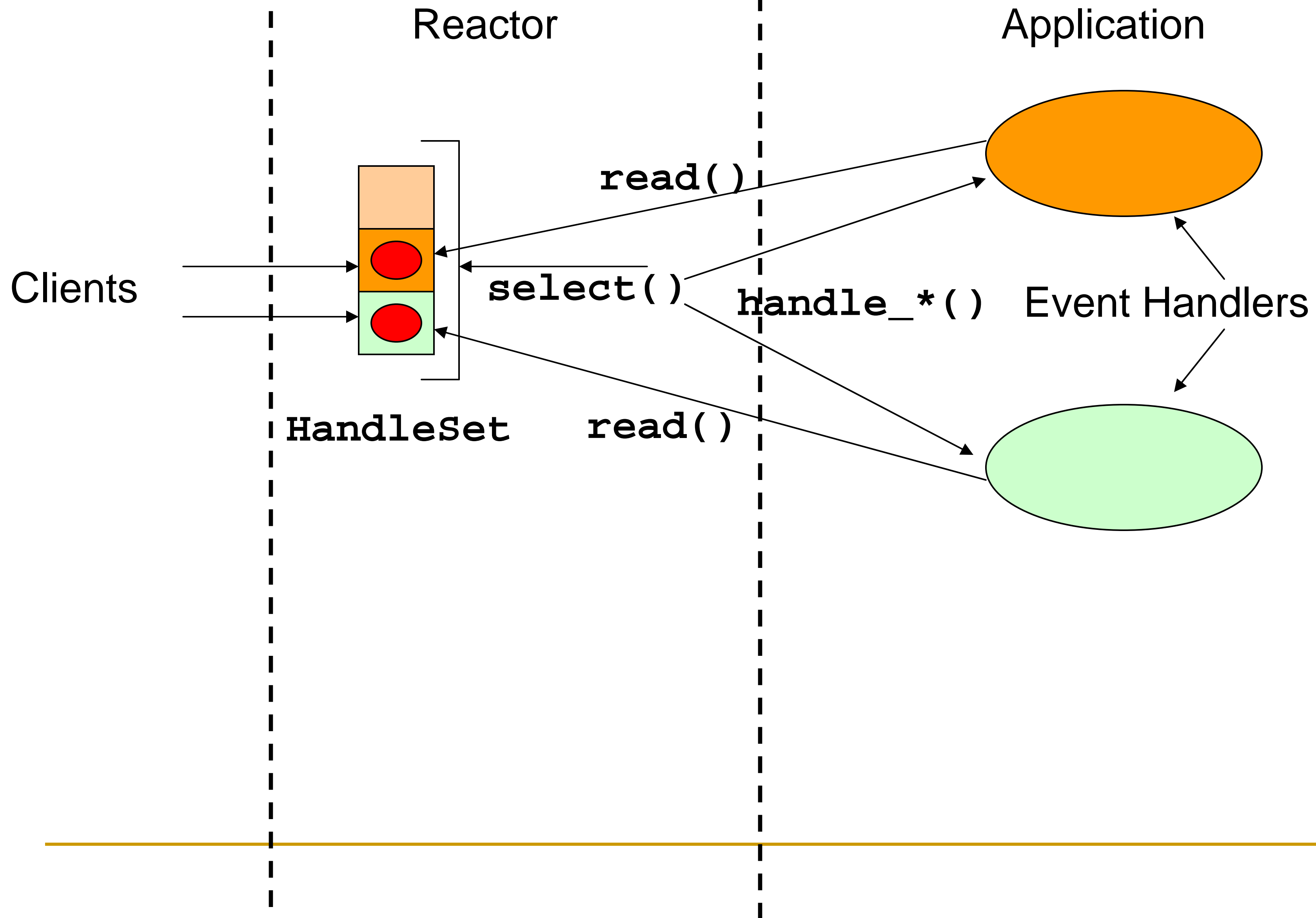
From http://www.cs.wustl.edu/~schmidt/patterns-ace.html

Synchronous vs. Reactive Read





Serial Event Dispatching



Multiple Reactors

- A single reactor instance will work in most cases
  - Sometimes desirable, *e.g.*, for handler serialization
  - Can use Singleton (e.g., **`ACE_Reactor::instance()`**)
- Limits on number of OS handles may restrict this
  - Total available (rarely an issue in general-purpose OS)
  - Max a single thread can wait for
    - *E.g.*, 64 in Win32
  - May need multiple reactors, each with its own thread
  - Note that handlers are not serialized across Reactor instances – treat remote/concurrent reactors similarly

Half-Sync/Half-Async

- Architectural pattern
- Decouples asynchronous and synchronous service processing
- Synchronous layer eases programming complexity
- Asynchronous layer improves overall performance

Context

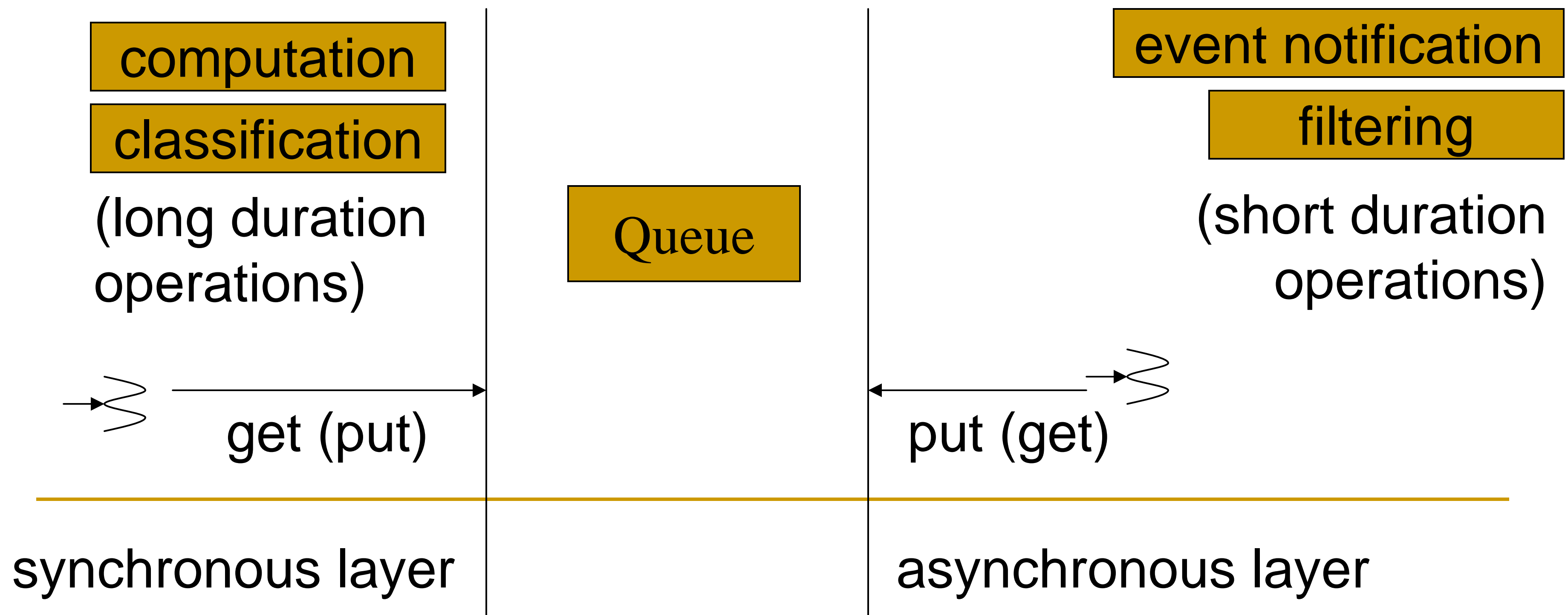
- A concurrent system
- Performs both synchronous and asynchronous services
  - Synchronous: copy data from one container to another
  - Asynchronous: socket reads and writes
- Both kinds of services must interact

Problem

- Asynchronous processing may be more efficient
  - E.g., reactive socket management vs. polling
  - Services may map directly to asynchronous mechanisms
    - E.g., hardware interrupts, signals, asynchronous I/O
- Synchronous processing simpler, easier to debug
- How to bring these paradigms together?

Solution

- Decompose architecture into two service layers
  - Synchronous
  - Asynchronous
- Add a queueing layer between them to facilitate communication



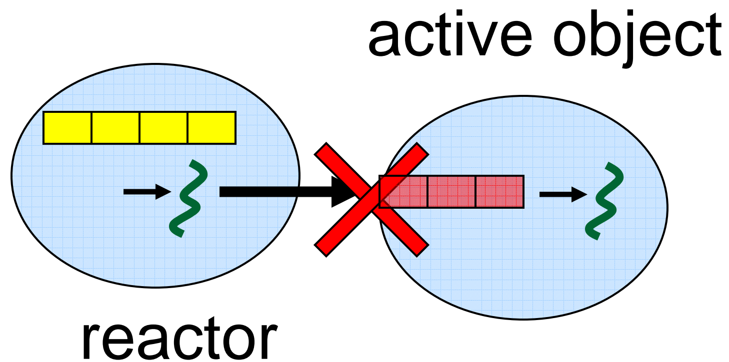
Summary: Separation of Concerns

- Long-running activities
  - Synchronous layer (simpler programming)
- Short-lived activities
  - Asynchronous layer (performance)
- How to bridge between these
  - Queueing layer

Implementation

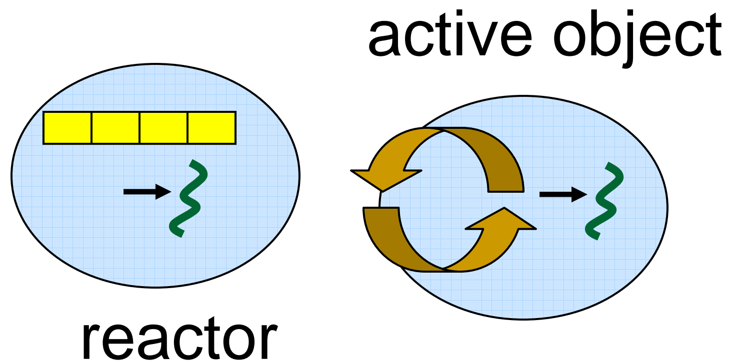
- Decompose system into three layers
  - Synchronous
    - Higher level/long duration operations
  - Asynchronous
    - Lower level/short duration operations
  - Queueing layer
    - Transfers results between the layers (may be bidirectional)
- Identify inter-layer communication strategies
  - queue, mailboxes, etc.
- Implement services in the synchronous layer
- Implement services in the asynchronous layer
- Integrate with the queueing layer (Monitor Object)

Problem: Race Conditions



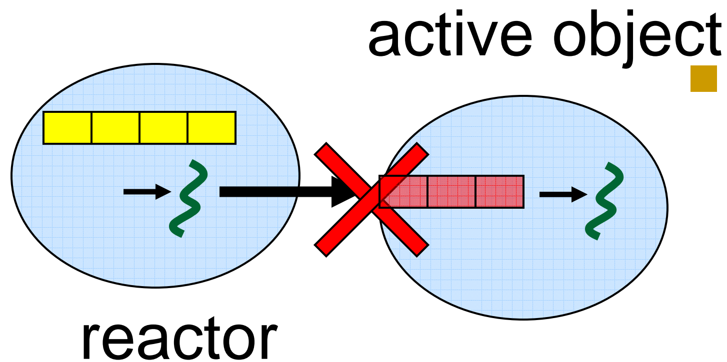
- Reactor thread wants to enqueue
- Active object thread wants to dequeue
- Queue could end up in an inconsistent state
- Solution
  - Mutex serializes access to the queue
    - Queue as a Monitor Object

Problem: Busy Waiting



- Empty queue
- Active object thread keeps locking/polling
  - Chews up CPU time
  - Slows down enqueue
- Need a notification mechanism
  - Is there work?
- Solution
  - Condition variable + notification

Problem: Overflow



- Full queue
  - Bounded memory, or other resource prevents transfer
- Could buffer message elsewhere for later transfer
  - *E.g.*, in the handler
  - Makes code more complex
- Really want flow control
  - Avoid "silly window" syndrome
- Need another notification mechanism
  - Another condition variable
  - Is there room for request?
  - High and low water marks