Pattern-Oriented Software Architecture

Applying Concurrent & Networked Objects to Develop & Use Distributed Object Computing Middleware

Dr. Douglas C. Schmidt

`schmidt@uci.edu`

`http://www.posa.uci.edu/`

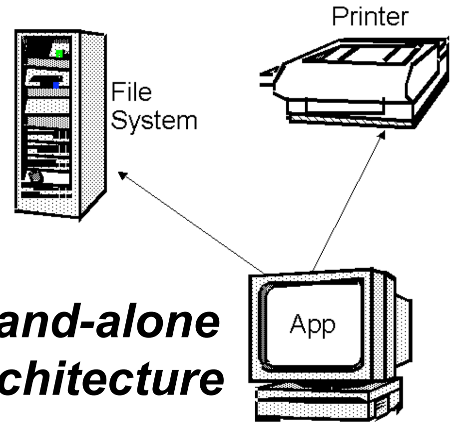
Electrical & Computing Engineering Department
The Henry Samueli School of Engineering
University of California, Irvine

Montag, 19. April 2004

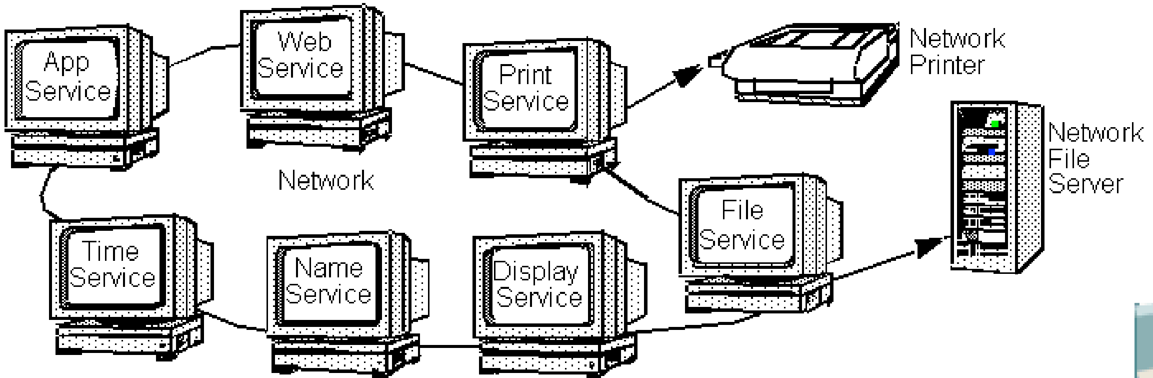
Middleware Patterns Tutorial Outline

Illustrate how/why it's hard to build robust, efficient, & extensible concurrent & networked applications

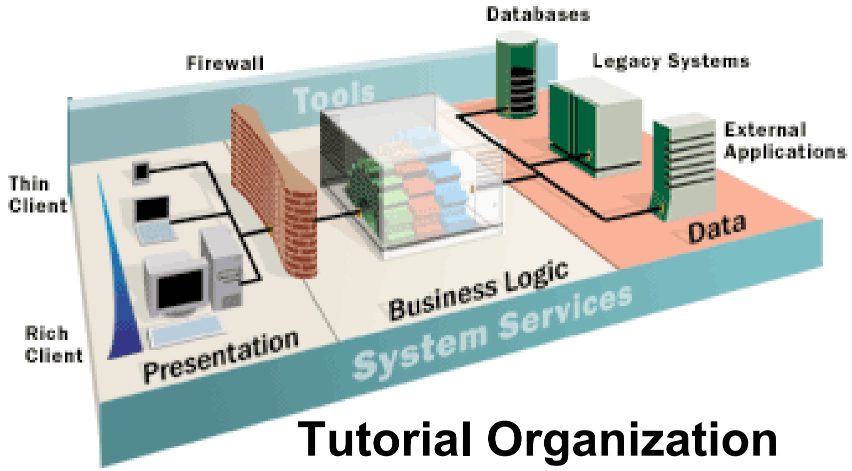
- *e.g.,* we must address many complex topics that are less problematic for non-concurrent, stand-alone applications



Stand-alone Architecture



Distributed Architecture



Describe OO techniques & language features to enhance software quality

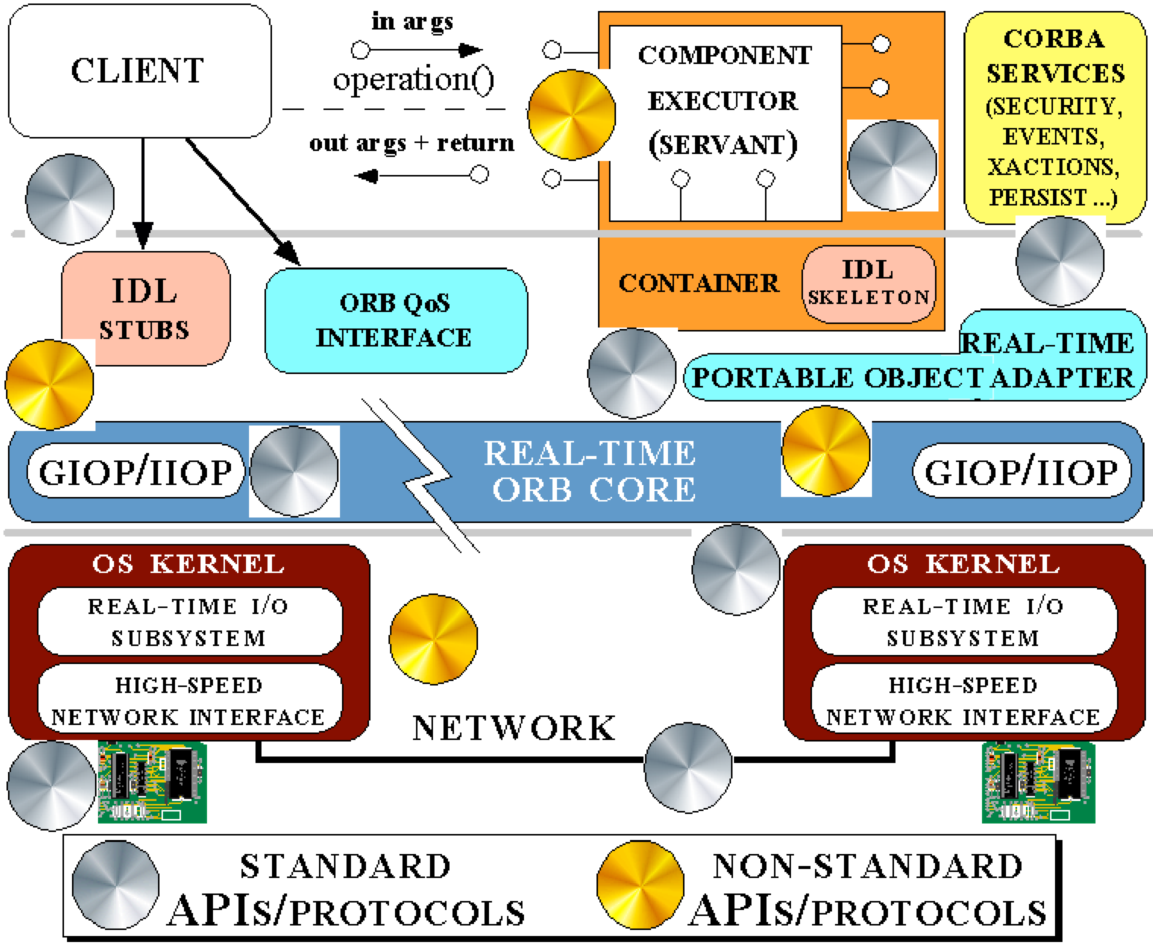
OO techniques & language features include:

- ***Patterns*** *(25+)*, which embody reusable software *architectures* & *designs*
- ***Frameworks & components***, which embody reusable software *implementations*
- ***OO language features****, e.g., classes, inheritance & dynamic binding, parameterized types & exceptions*

Tutorial Organization

1. *Background & motivation*
2. *Concurrent & network challenges & solution approaches*
3. *Multiple case studies*
4. *Wrap-up and Q&A*

The Road Ahead



Increasing software productivity and QoS depends heavily on COTS

CPUs and networks have increased by 3-7 orders of magnitude in the past decade

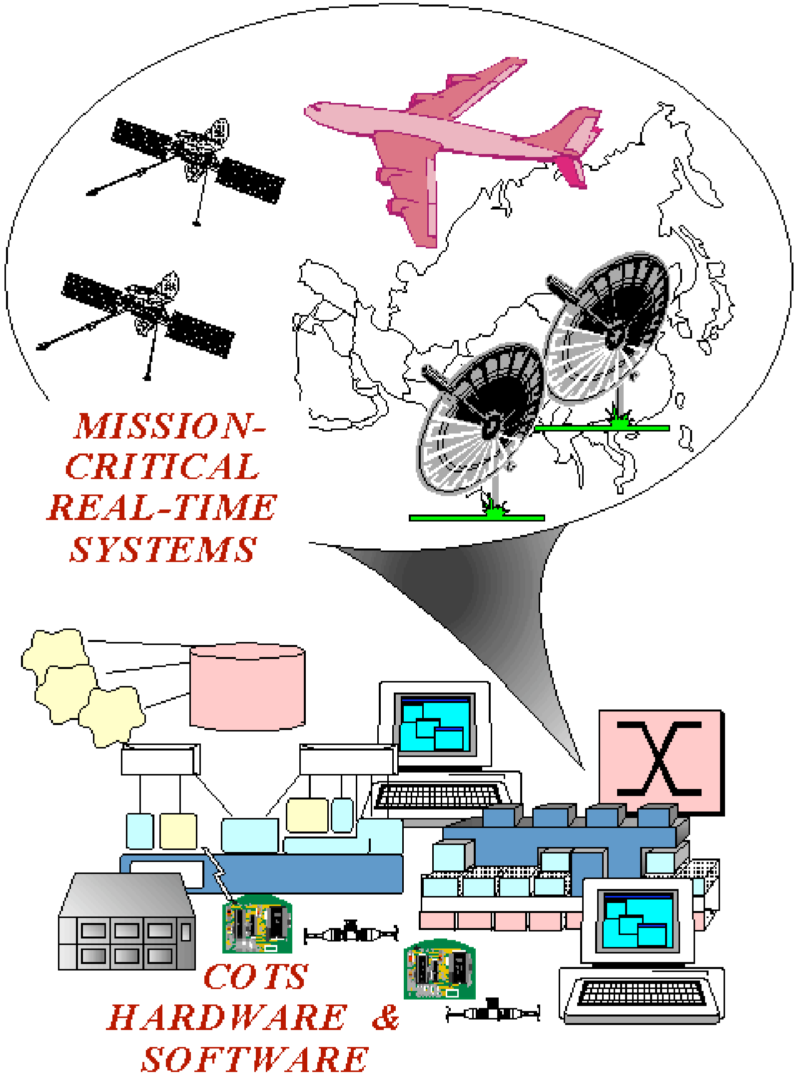
Extrapolating this trend to 2010 yields

- **~100 Gigahertz desktops**
- **~100 Gigabits/sec LANs**
- **~100 Megabits/sec wireless**
- **~10 Terabits/sec Internet backbone**

These advances stem largely from standardizing hardware & software APIs and protocols, *e.g.:*

- Intel x86 & Power PC chipsets
- TCP/IP, ATM
- POSIX & JVMs
- Middleware & components
- Ada, C, C++, RT Java

Addressing the COTS "Crisis"



Distributed systems must increasingly reuse commercial-off-the-shelf (COTS) hardware & software

- ***i.e.*, COTS is essential to R&D success**

However, this trend presents many vexing R&D challenges for mission-critical systems, *e.g.*,

- **Inflexibility and lack of QoS**
- **Security & global competition**

Why we should care:

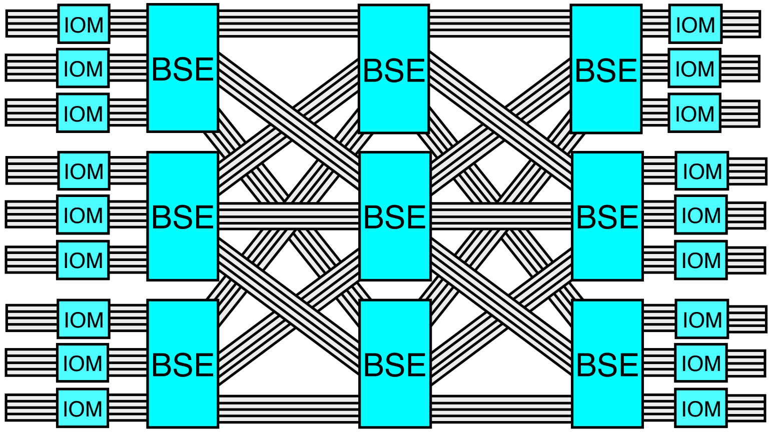
- **Despite IT commoditization, progress in COTS hardware & software is often *not* applicable for mission-critical distributed systems**
- **Recent advances in COTS software technology can help to fundamentally reshape distributed system R&D**

R&D Challenges & Opportunities

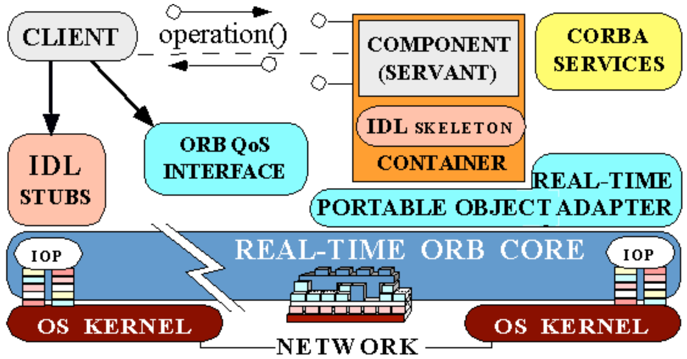
Challenges

Opportunities

High-performance, real-time, fault-tolerant, & secure systems

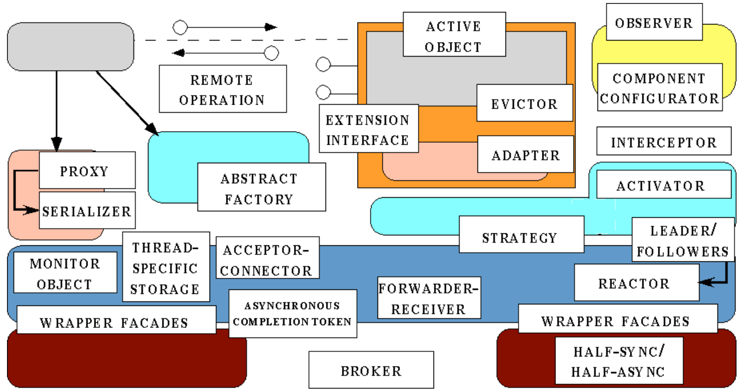


Middleware, Frameworks, & Components



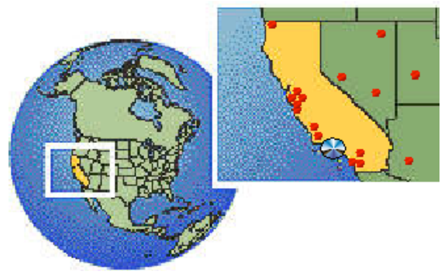
Autonomous systems

Patterns & Pattern Languages

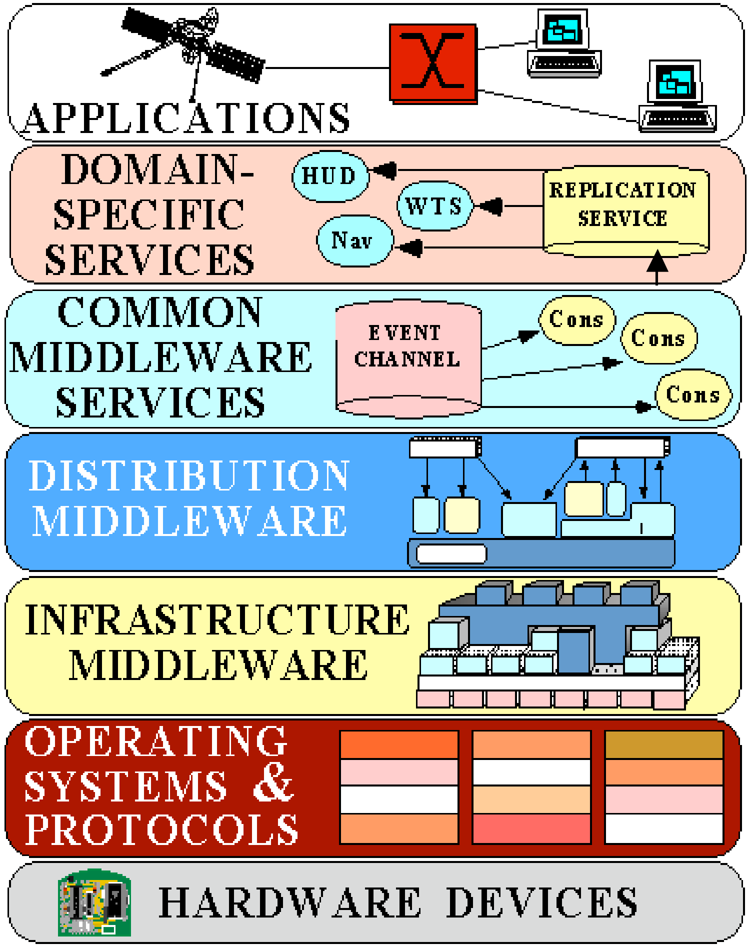


Power-aware *ad hoc*, mobile, distributed, & embedded systems

Standards & Open-source



The Evolution of COTS



There are multiple COTS layers & research/ business opportunities

Historically, mission-critical apps were built directly atop hardware & OS

- Tedious, error-prone, & costly over lifecycles

Standards-based COTS middleware helps:

- Manage end-to-end resources
- Leverage HW/SW technology advances
- Evolve to new environments & requirements

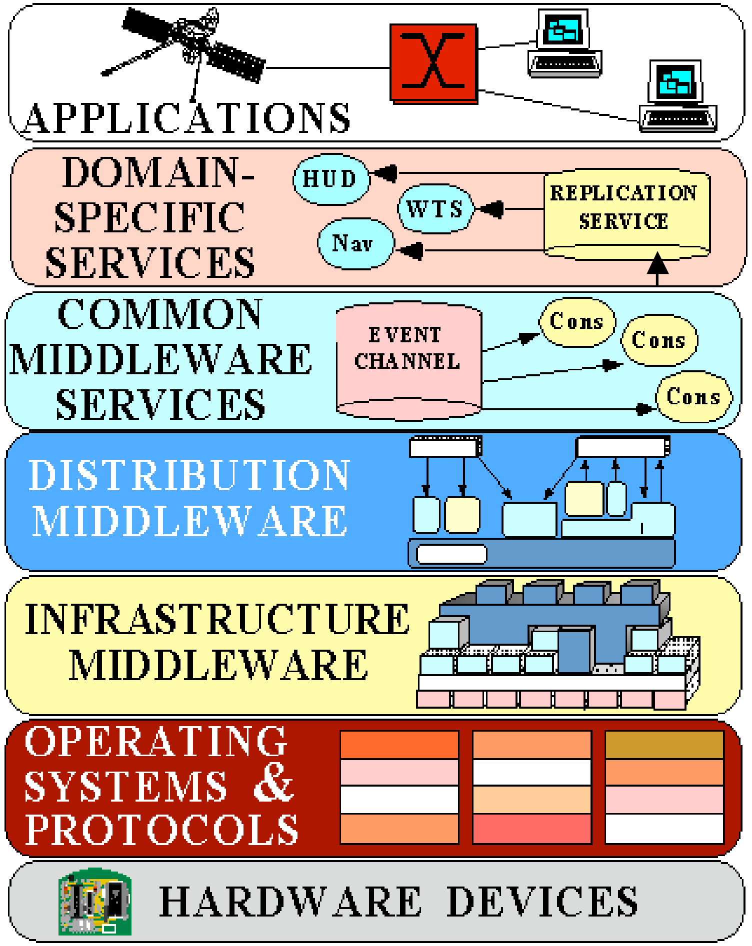
The domain-specific services layer is where system integrators can provide the most value & derive the most benefits

Key R&D challenges include:

- **Layered QoS specification & enforcement**
- **Separating policies & mechanisms across layers**
- **Time/space optimizations for middleware & apps**
- **Multi-level global resource mgmt. & optimization**
- **High confidence**
- **Stable & robust adaptive systems**

Prior R&D efforts have address some, *but by no means all*, of these issues

Consequences of COTS & IT Commoditization



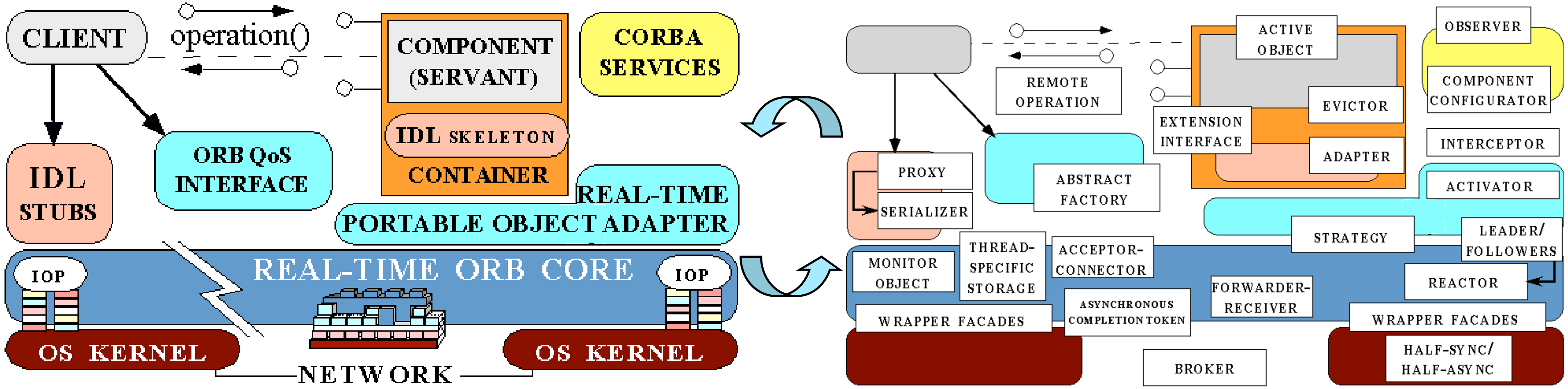
- More emphasis on integration rather than programming
- Increased technology convergence & standardization
- Mass market economies of scale for technology & personnel
- More disruptive technologies & global competition
- Lower priced--but often lower quality--hardware & software components
- The decline of internally funded R&D
- Potential for complexity cap in next-generation complex systems

Not all trends bode well for long-term competitiveness of traditional R&D leaders

Ultimately, competitiveness will depend on success of long-term R&D efforts on *complex* distributed & embedded systems

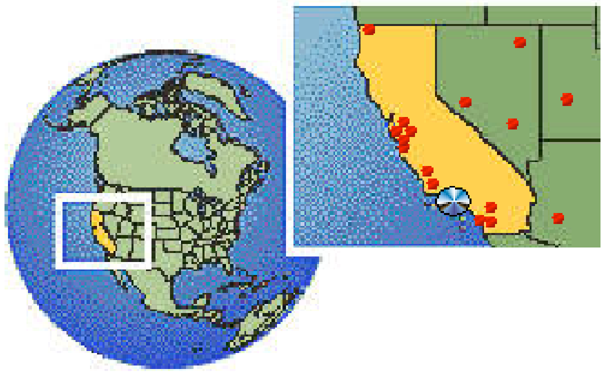
Why We are Succeeding Now

Recent synergistic advances in fundamentals:



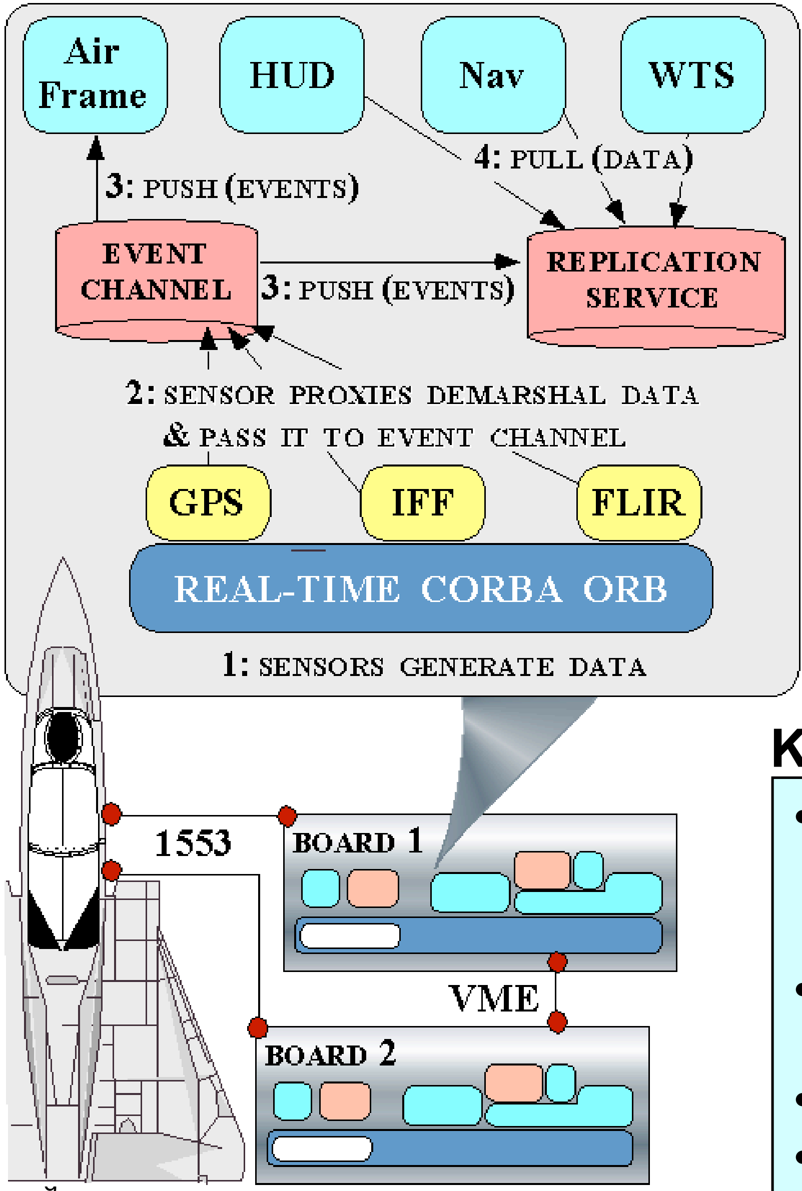
Standards-based QoS-enabled Middleware: Pluggable service & micro-protocol components & reusable "semi-complete" application frameworks

Patterns & Pattern Languages: Generate software architectures by capturing recurring structures & dynamics & by resolving design forces



Revolutionary changes in software process: Open-source, refactoring, extreme programming (XP), advanced V&V techniques

Example:
Applying COTS in Real-time Avionics



Goals
- Apply COTS & open systems to mission-critical real-time avionics

Key System Characteristics
- Deterministic & statistical deadlines
  - ~20 Hz
- Low latency & jitter
  - ~250 *u*secs
- Periodic & aperiodic processing
- Complex dependencies
- Continuous platform upgrades

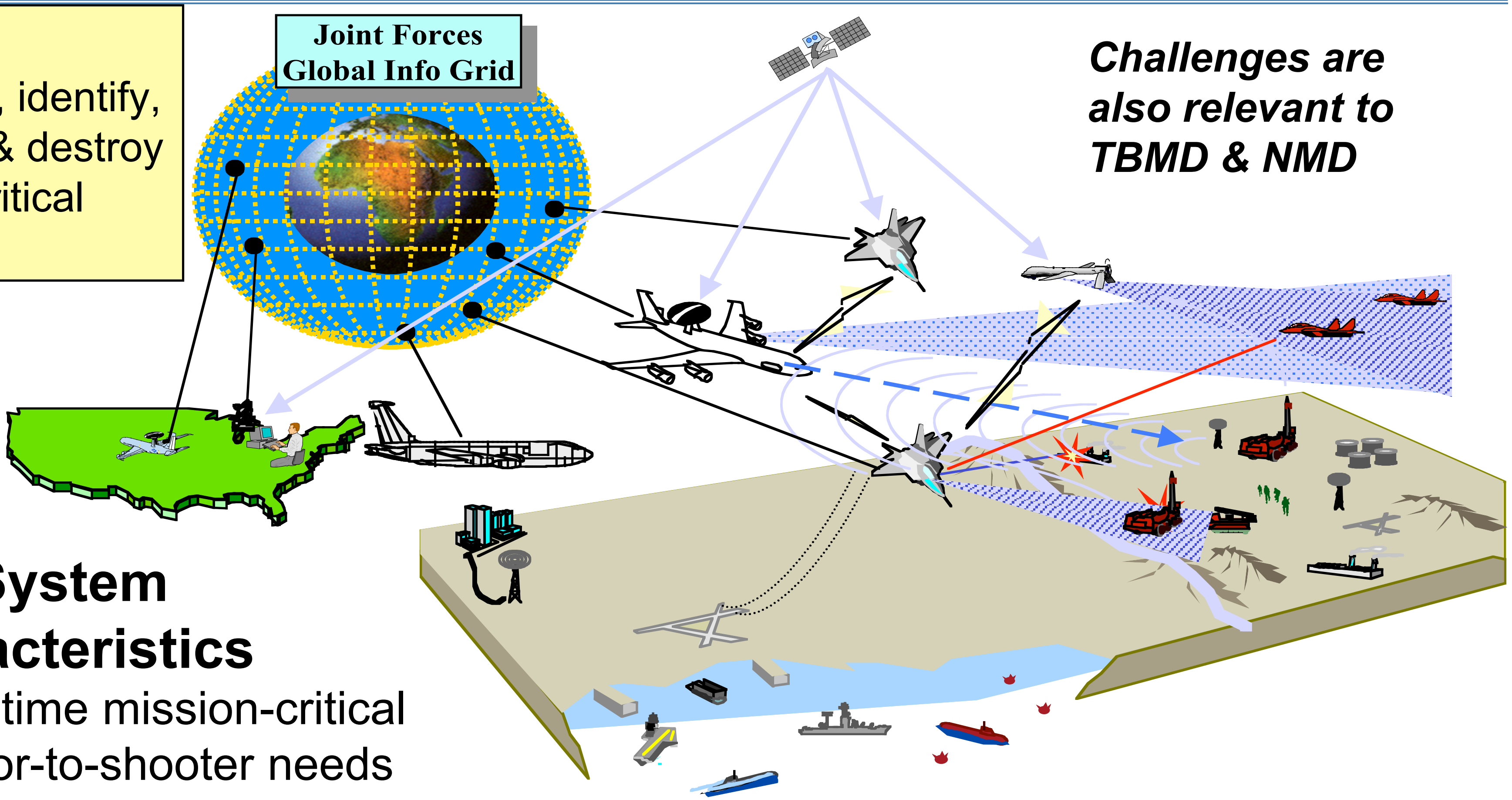
Key Results
- Test flown at China Lake NAWS by Boeing OSAT II '98, funded by OS-JTF
  - www.cs.wustl.edu/~schmidt/TAO-boeing.html
- Also used on SOFIA project by Raytheon
  - sofia.arc.nasa.gov
- First use of RT CORBA in mission computing
- Drove Real-time CORBA standardization

Example:

Applying COTS to Time-Critical Targets

Goals

- Detect, identify, track, & destroy time-critical targets



Challenges are also relevant to TBMD & NMD

Key System Characteristics

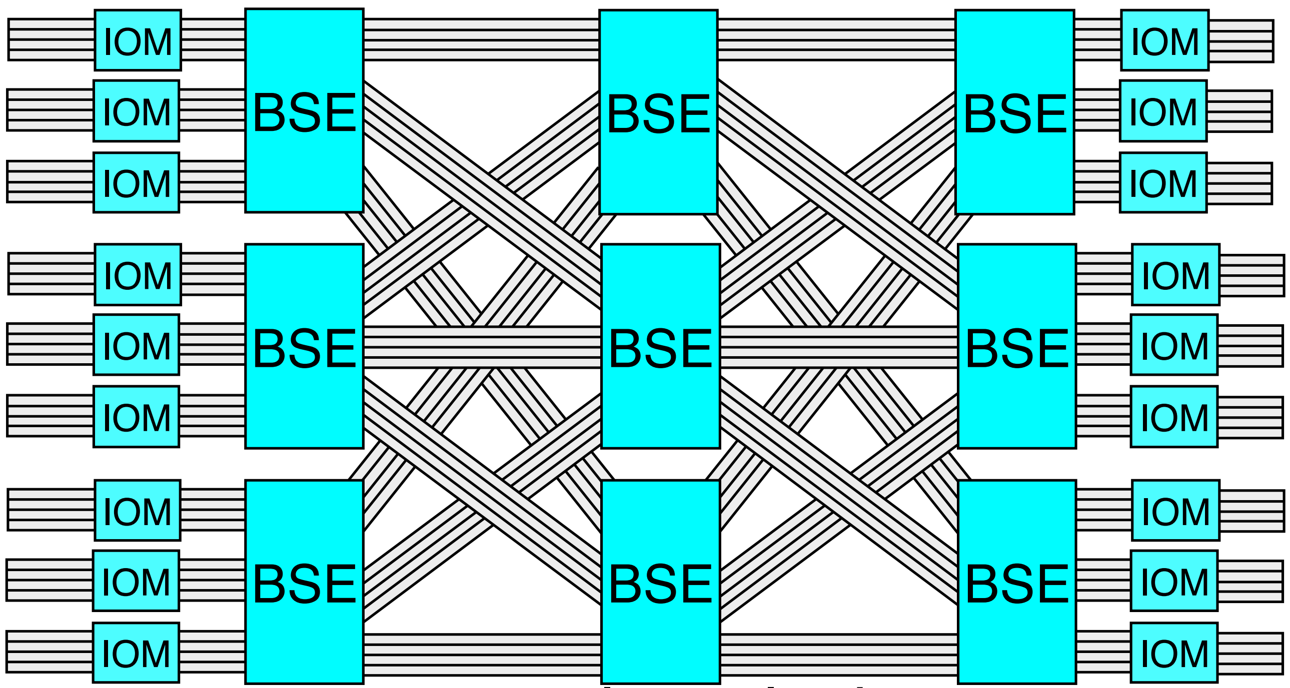
- Real-time mission-critical sensor-to-shooter needs
- Highly dynamic QoS requirements & environmental conditions
- Multi-service & asset coordination

Key Solution Characteristics

- Adaptive & reflective
- High confidence
- Safety critical

- Efficient & scalable
- Affordable & flexible
- COTS-based

Example:
Applying COTS to Large-scale Routers



www.arl.wustl.edu

Goal
- Switch ATM cells + IP packets at terabit rates

Key System Characteristics
- Very high-speed WDM links
- $10^2/10^3$ line cards
- Stringent requirements for availability
- Multi-layer load balancing, *e.g.:*
  - Layer 3+4
  - Layer 5

Key Software Solution Characteristics
- High confidence & scalable computing architecture
  - Networked embedded processors
  - Distribution middleware
  - FT & load sharing
  - Distributed & layered resource management
- Affordable, flexible, & COTS

Example:
Applying COTS to Hot Rolling Mills

Goals

- Control the processing of molten steel moving through a hot rolling mill in real-time

System Characteristics

- Hard real-time process automation requirements
  - *i.e.,* 250 ms real-time cycles
- System acquires values representing plant's current state, tracks material flow, calculates new settings for the rolls & devices, & submits new settings back to plant

Key Software Solution Characteristics

www.siroll.de

- Affordable, flexible, & COTS
  - Product-line architecture
  - Design guided by patterns & frameworks
  - Windows NT/2000
  - Real-time CORBA (ACE+TAO)

Example:
Applying COTS to Real-time Image Processing

Goals
- Examine glass bottles for defects in real-time

System Characteristics
- Process 20 bottles per sec
  - *i.e.,* ~50 msec per bottle
- Networked configuration
- ~10 cameras

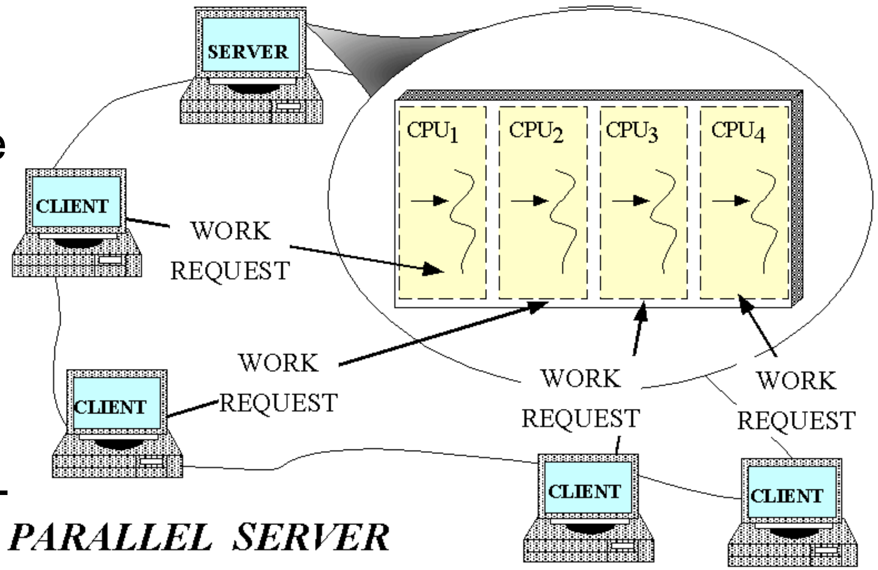
Key Software Solution Characteristics
- Affordable, flexible, & COTS
  - Embedded Linux (Lem)
  - Compact PCI bus + Celeron processors
  - Remote booted by DHCP/TFTP
  - Real-time CORBA (ACE+TAO)

Key Opportunities & Challenges in Concurrent & Networked Applications

Concurrency & Synchronization

Motivations

- Leverage hardware/software advances
- Simplify program structure
- Increase performance
- Improve response-time



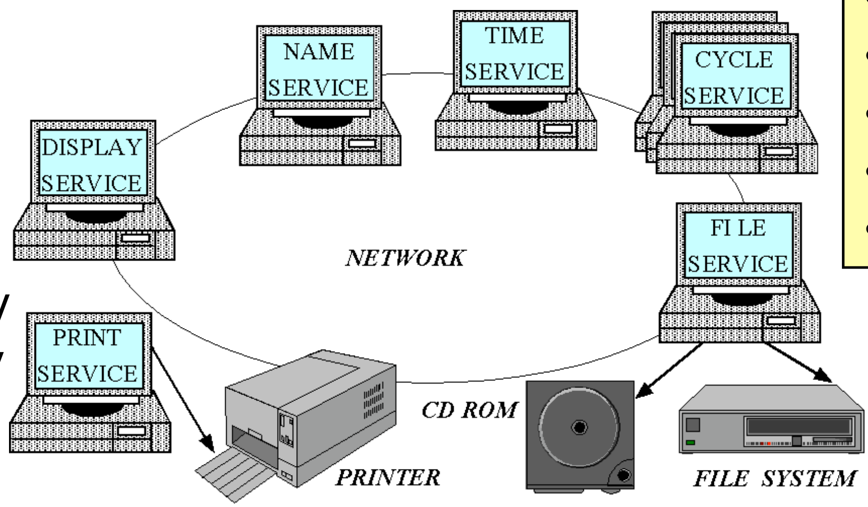
Accidental Complexities

- Low-level APIs
- Poor debugging tools
- Algorithmic decomposition
- Continuous re-invention & re-discover of core concepts & components

Networking & Distribution

Motivations

- Collaboration
- Performance
- Reliability & availability
- Scalability & portability
- Extensibility
- Cost effectiveness



Inherent Complexities

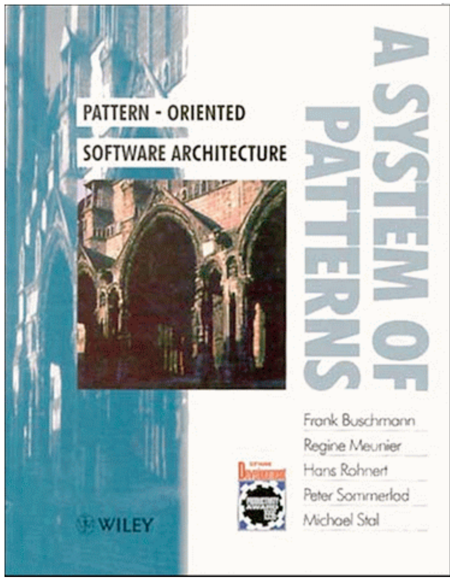
- Latency
- Reliability
- Load balancing
- Scheduling
- Causal ordering
- Synchronization
- Deadlocks

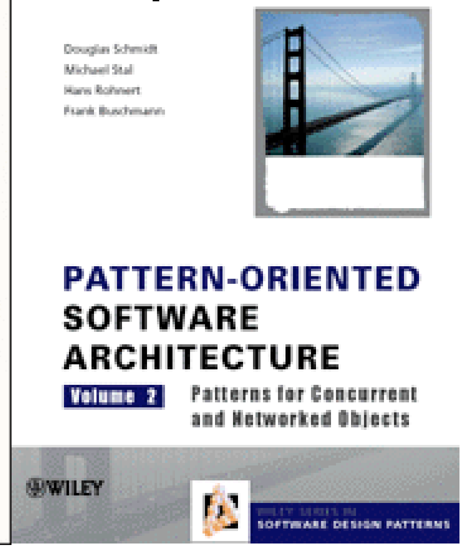
Overview of Patterns & Pattern Languages

Patterns

- Present *solutions* to common software *problems* arising within a certain *context*
- Help resolve key design forces
- Capture recurring structures & dynamics among software participants to facilitate reuse of successful designs
- Generally codify expert knowledge of design constraints & “best practices”

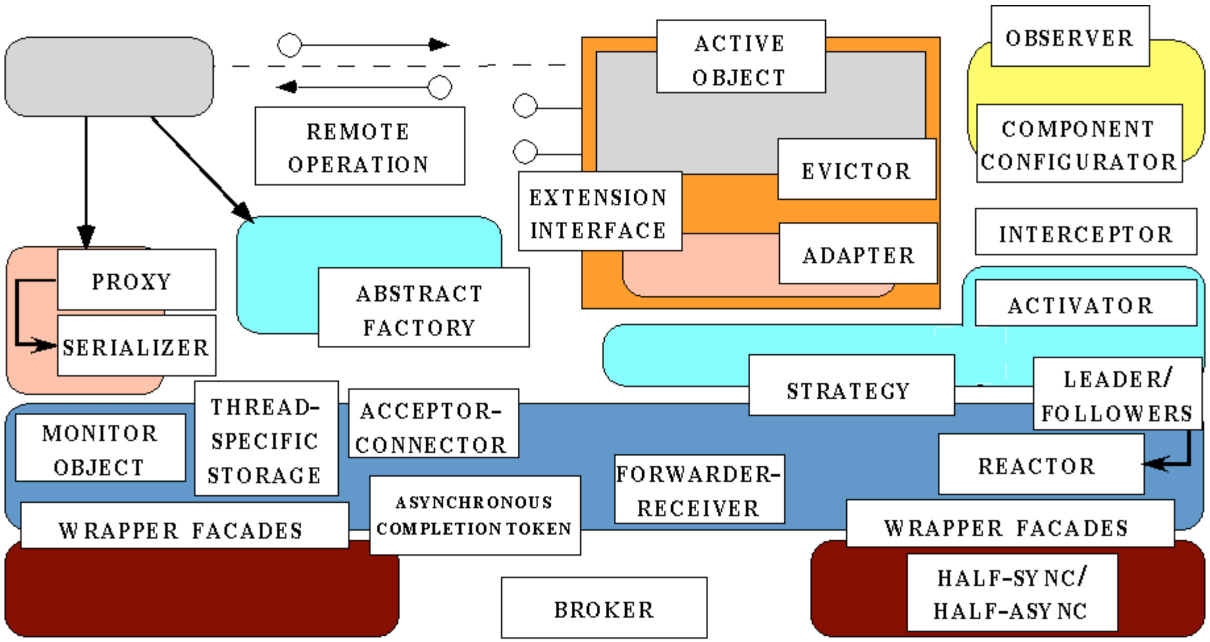
www.posa.uci.edu





Pattern Languages

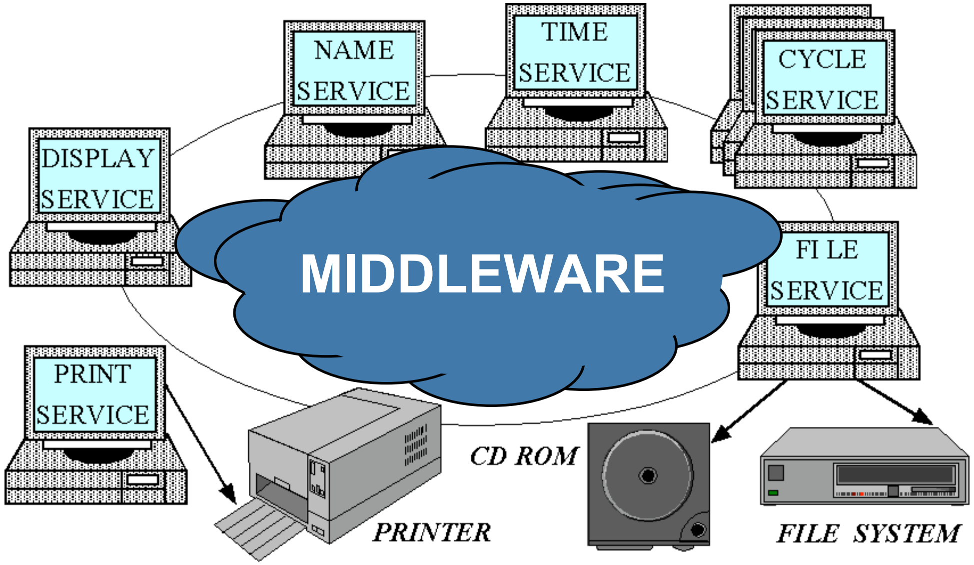
- Define a *vocabulary* for talking about software development problems
- Provide a *process* for the orderly resolution of these problems
- Help to generate & reuse software *architectures*



Software Design Abstractions for Concurrent & Networked Applications

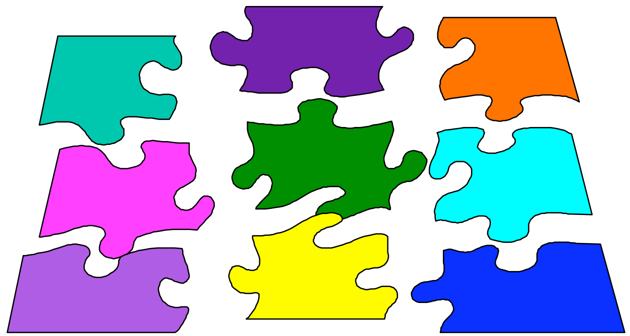
Problem

- Distributed application functionality is subject to change since it is often reused in unforeseen contexts, e.g.,
  - Accessed from different clients
  - Run on different platforms
  - Configured into different run-time contexts



Solution

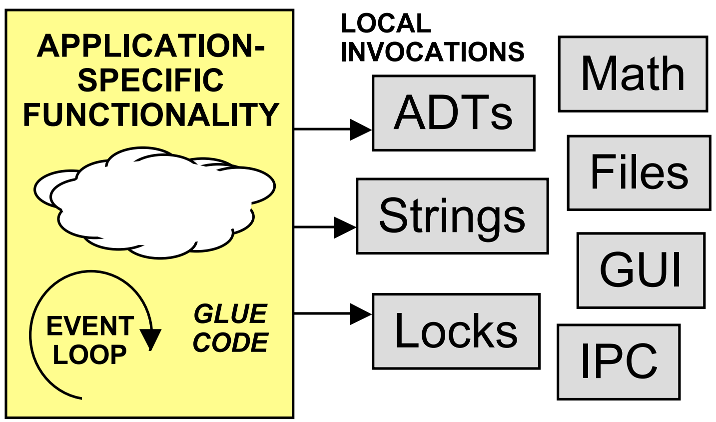
- Don‘t structure distributed applications as a monoliths, but instead decompose them into *classes*, *frameworks*, & *components*



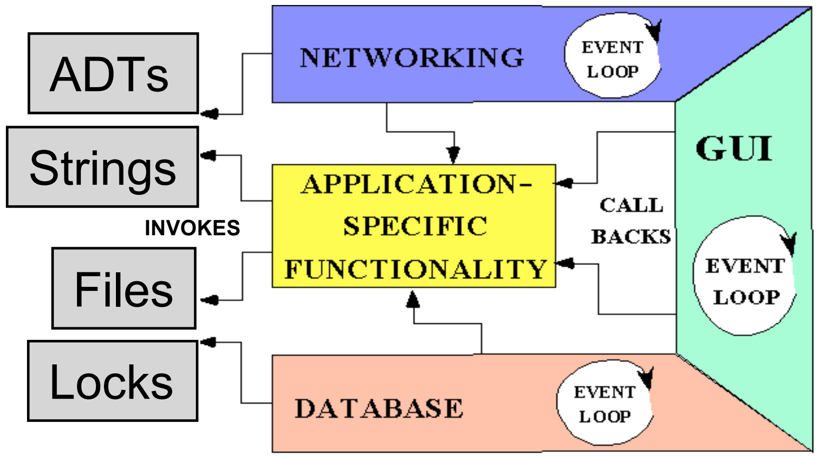
A ***class*** is a unit of abstraction & implementation in an OO programming	A ***framework*** is an integrated collection of classes that collaborate to produce a reusable architecture for a family of related applications	A ***component*** is an encapsulation unit with one or more interfaces that provide clients with access to its services

A Comparison of Class Libraries, Frameworks, & Components

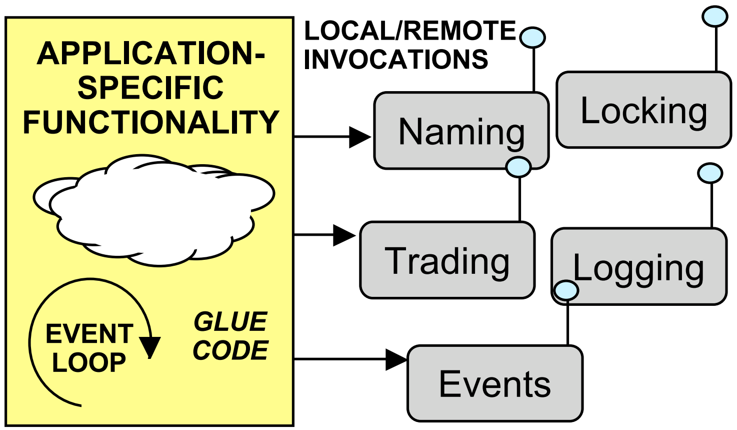
Class Library Architecture



Framework Architecture

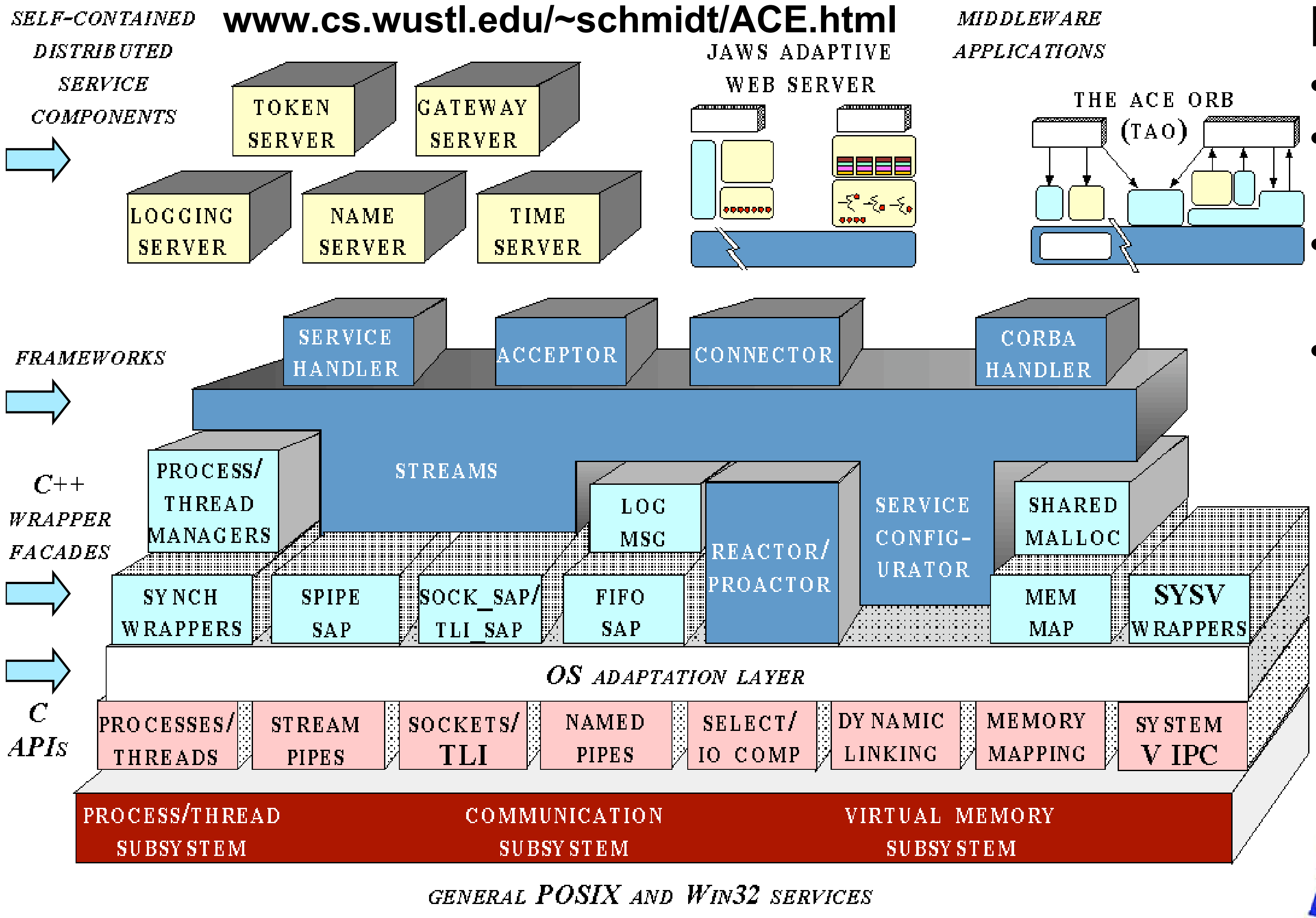


Component Architecture



Class Libraries	Frameworks	Components
Micro-level	Meso-level	Macro-level
Stand-alone language entities	"Semi-complete" applications	Stand-alone composition entities
Domain-independent	Domain-specific	Domain-specific or Domain-independent
Borrow caller's thread	Inversion of control	Borrow caller's thread

Overview of the ACE Framework



Features

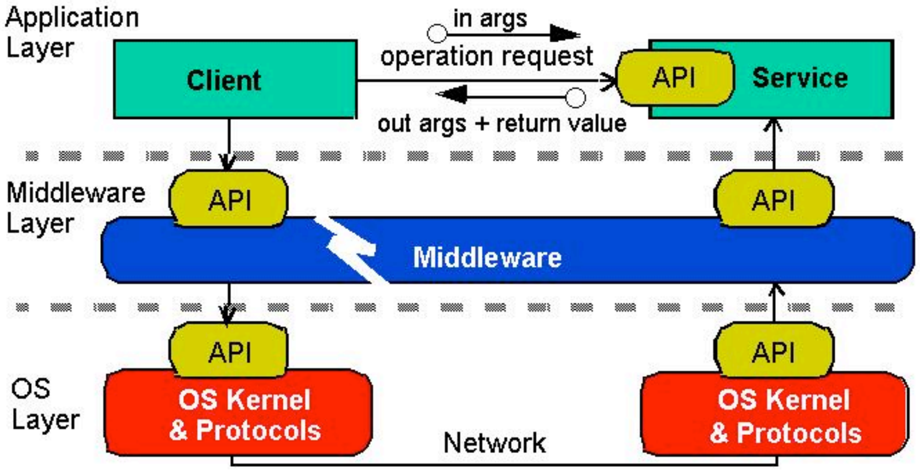
- Open-source
- 200,000+ lines of C++
- 30+ person-years of effort
- Ported to Win32, UNIX, & RTOSs
  - *e.g.*, VxWorks, pSoS, LynxOS, Chorus, QNX

- **Large open-source user community**
  - www.cs.wustl.edu/~schmidt/ACE-users.html

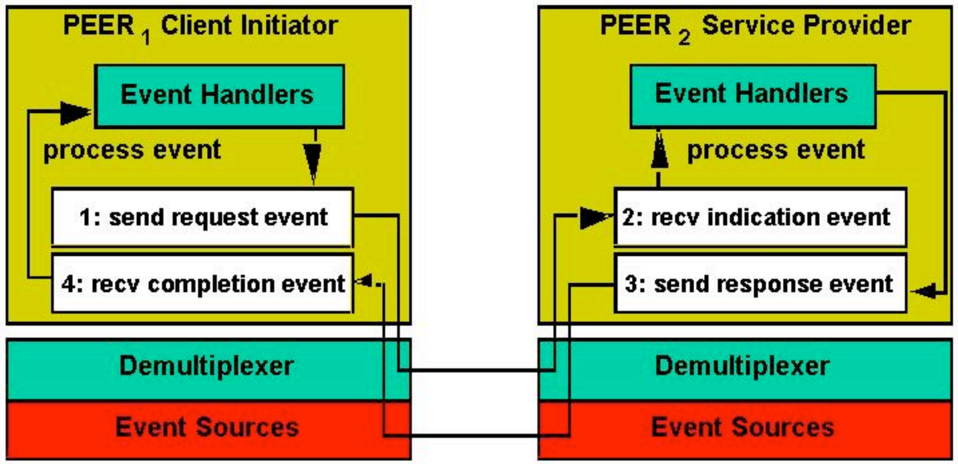
- **Commercial support by Riverace**
  - www.riverace.com/

Key Capabilities Provided by ACE

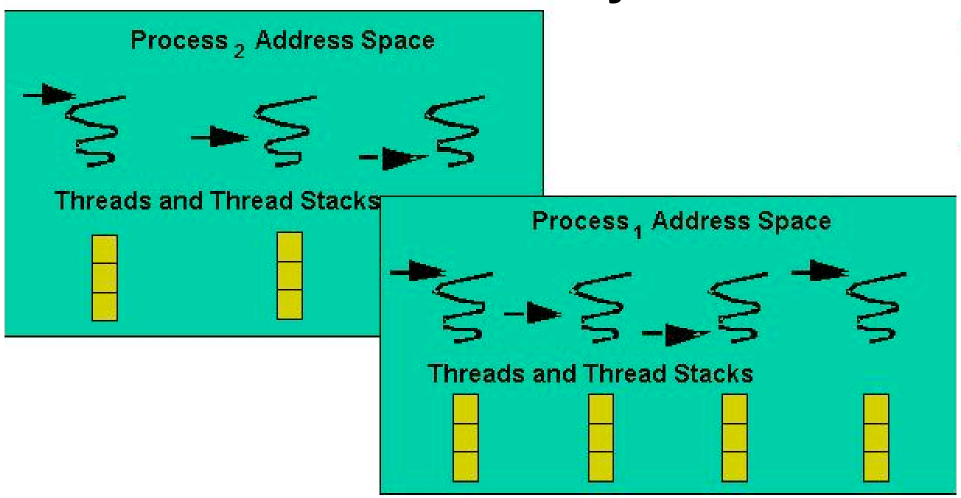
Service Access & Control



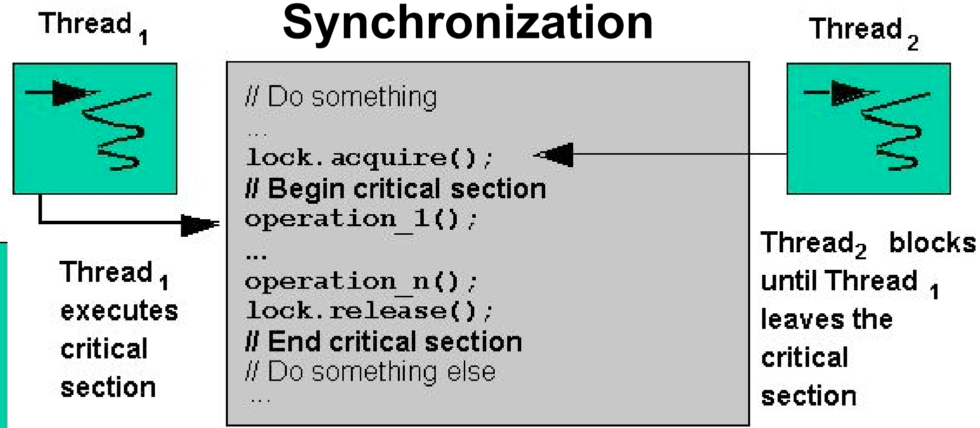
Event Handling



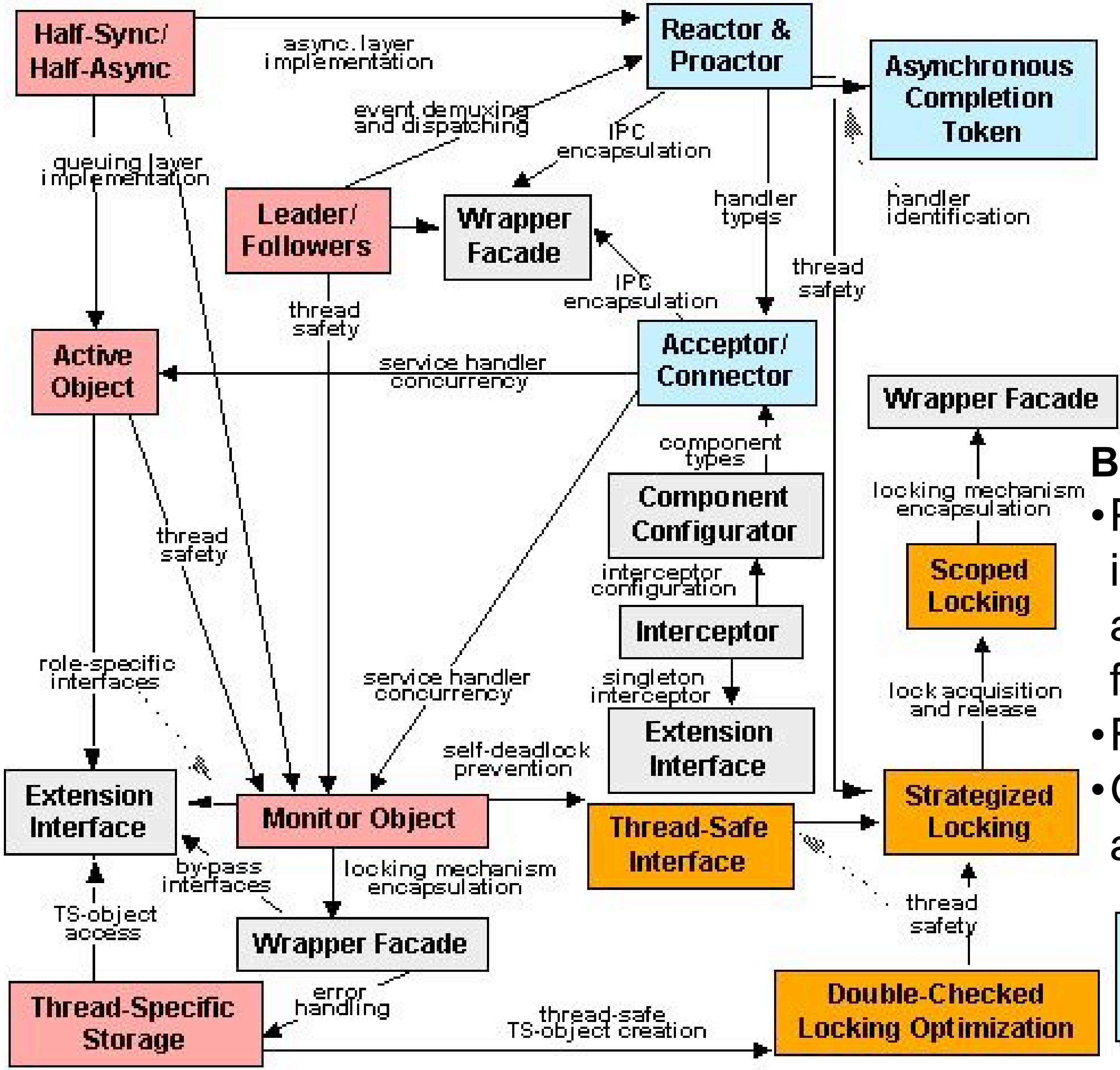
Concurrency



Synchronization



The POSA2 Pattern Language



Observation

- *Failure rarely results from unknown scientific principles, but from failing to apply proven engineering practices & patterns*

Benefits of POSA2 Patterns

- Preserve crucial design information used by applications & underlying frameworks/components
- Facilitate design reuse
- Guide design choices for application developers

URL for POSA Books
`www.posa.uci.edu`

POSA2 Pattern Abstracts

Service Access & Configuration Patterns

The *Wrapper Facade* design pattern encapsulates the functions and data provided by existing non-object-oriented APIs within more concise, robust, portable, maintainable, and cohesive object-oriented class interfaces.

The *Component Configurator* design pattern allows an application to link and unlink its component implementations at run-time without having to modify, recompile, or statically relink the application. Component Configurator further supports the reconfiguration of components into different application processes without having to shut down and re-start running processes.

The *Interceptor* architectural pattern allows services to be added transparently to a framework and triggered automatically when certain events occur.

The *Extension Interface* design pattern allows multiple interfaces to be exported by a component, to prevent bloating of interfaces and breaking of client code when developers extend or modify the functionality of the component.

Event Handling Patterns

The *Reactor* architectural pattern allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients.

The *Proactor* architectural pattern allows event-driven applications to efficiently demultiplex and dispatch service requests triggered by the completion of asynchronous operations, to achieve the performance benefits of concurrency without incurring certain of its liabilities.

The *Asynchronous Completion Token* design pattern allows an application to demultiplex and process efficiently the responses of asynchronous operations it invokes on services.

The *Acceptor-Connector* design pattern decouples the connection and initialization of cooperating peer services in a networked system from the processing performed by the peer services after they are connected and initialized.

POSA2 Pattern Abstracts (cont'd)

Synchronization Patterns

The *Scoped Locking* C++ idiom ensures that a lock is acquired when control enters a scope and released automatically when control leaves the scope, regardless of the return path from the scope.

The *Strategized Locking* design pattern parameterizes synchronization mechanisms that protect a component's critical sections from concurrent access.

The *Thread-Safe Interface* design pattern minimizes locking overhead and ensures that intra-component method calls do not incur 'self-deadlock' by trying to reacquire a lock that is held by the component already.

The *Double-Checked Locking Optimization* design pattern reduces contention and synchronization overhead whenever critical sections of code must acquire locks in a thread-safe manner just once during program execution.

Concurrency Patterns

The *Active Object* design pattern decouples method execution from method invocation to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control.

The *Monitor Object* design pattern synchronizes concurrent method execution to ensure that only one method at a time runs within an object. It also allows an object's methods to cooperatively schedule their execution sequences.

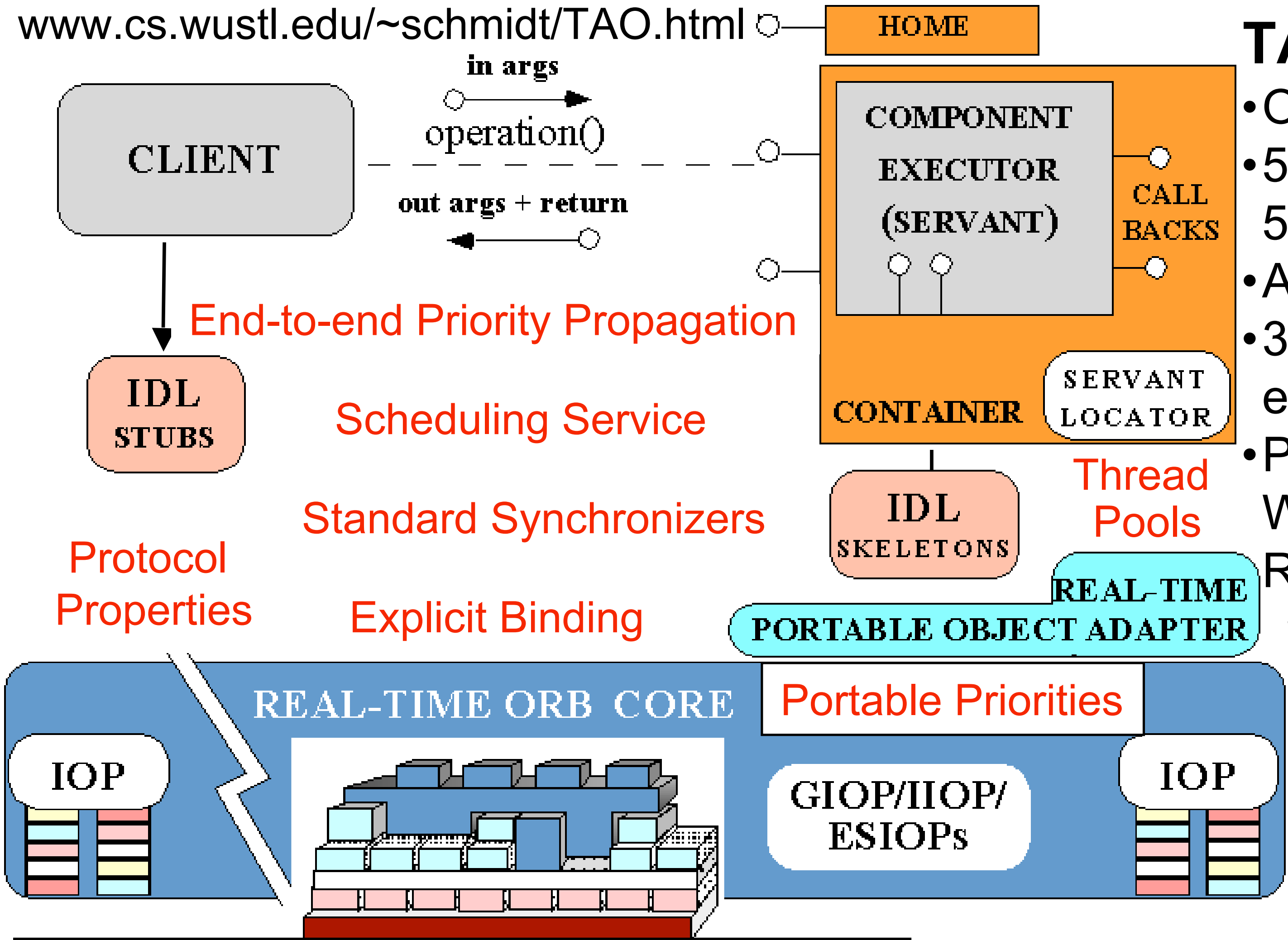
The *Half-Sync/Half-Async* architectural pattern decouples asynchronous and synchronous service processing in concurrent systems, to simplify programming without unduly reducing performance. The pattern introduces two intercommunicating layers, one for asynchronous and one for synchronous service processing.

The *Leader/Followers* architectural pattern provides an efficient concurrency model where multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process service requests that occur on the event sources.

The *Thread-Specific Storage* design pattern allows multiple threads to use one 'logically global' access point to retrieve an object that is local to a thread, without incurring locking overhead on each object access.

Example of Applying Patterns & Frameworks:

Real-time CORBA & The ACE ORB (TAO)



TAO Features

- Open-source
- 500+ classes & 500,000+ lines of C++
- ACE/patterns-based
- 30+ person-years of effort
- Ported to UNIX, Win32, MVS, & many RT & embedded OSs
  - *e.g.*, VxWorks, LynxOS, Chorus, QNX

- **Large open-source user community**
  - www.cs.wustl.edu/~schmidt/TAO-users.html

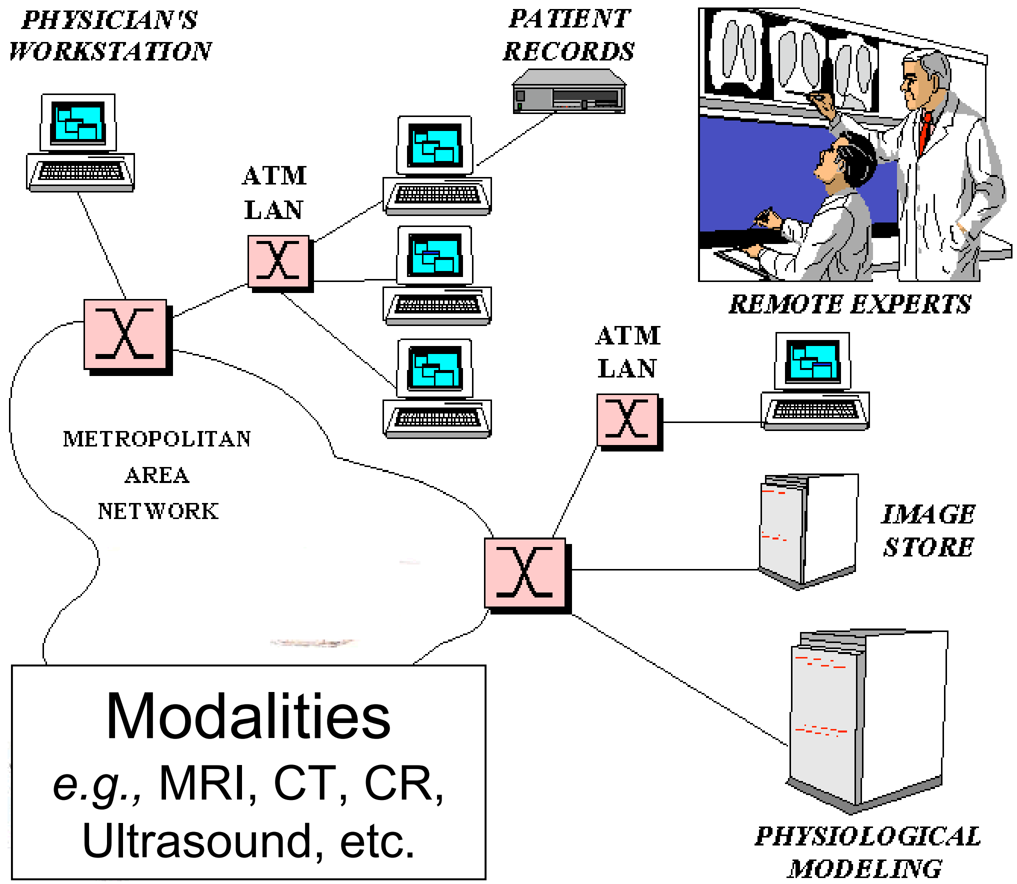
- **Commercial support by OCI**
  - www.theaceorb.com/

Tutorial Example 1:

Electronic Medical Imaging Systems

Goal

- Route, manage, & manipulate electronic medical images robustly, efficiently, & securely thoughout a distributed environment



System Characteristics

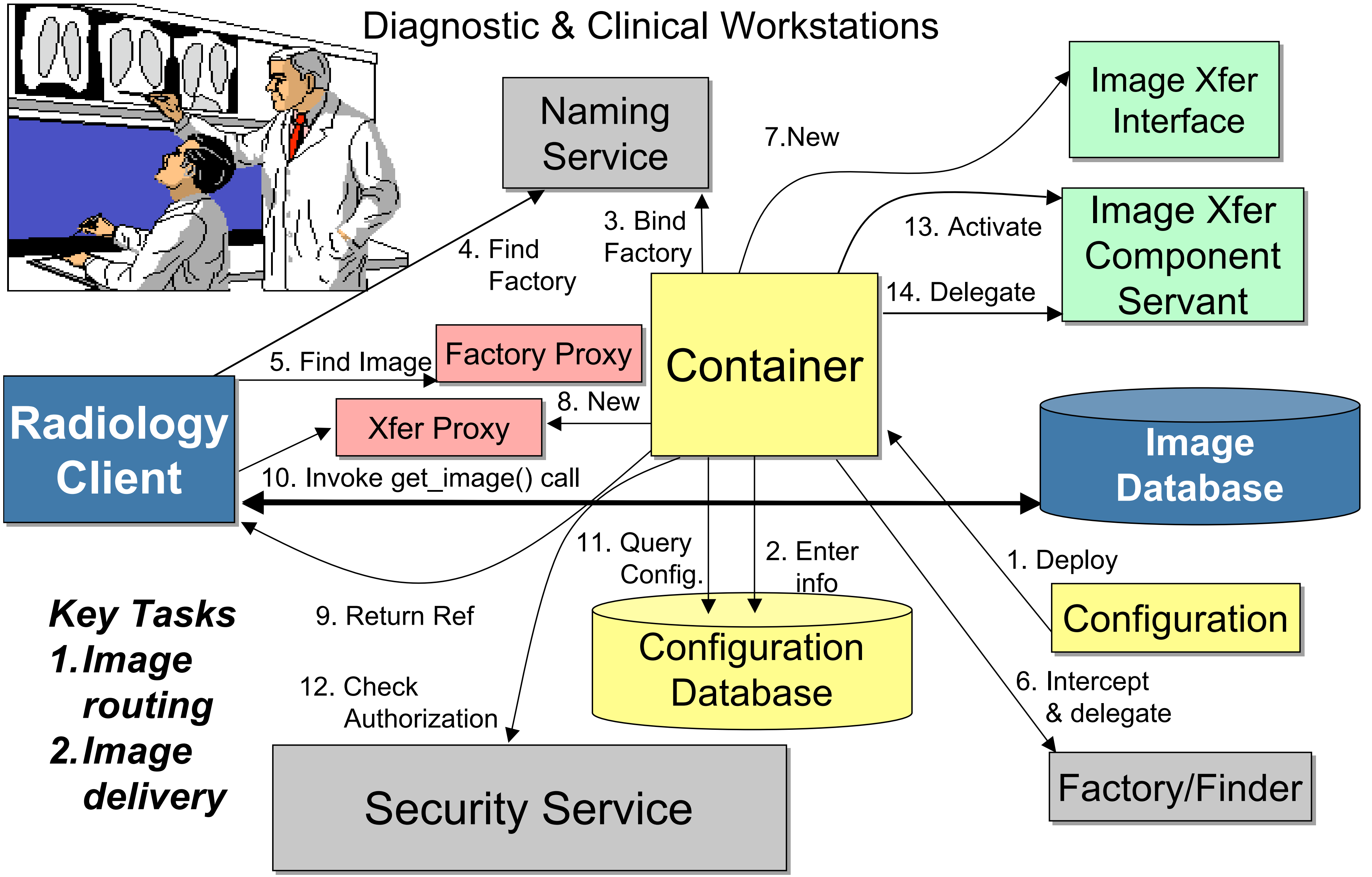
- Large volume of “blob” data
  - *e.g.,*10 to 40 Mps
  - “Lossy” compression isn’t viable due to liability concerns
- Diverse QoS requirements, *e.g.,*
  - Synchronous & asynchronous communication
  - Streaming communication
  - Prioritization of requests & streams
  - Distributed resource management

www.syngo.com

Key Software Solution Characteristics

- Affordable, flexible, & COTS
  - Product-line architecture
  - Design guided by patterns & frameworks
- General-purpose & embedded OS platforms
- Middleware technology agnostic

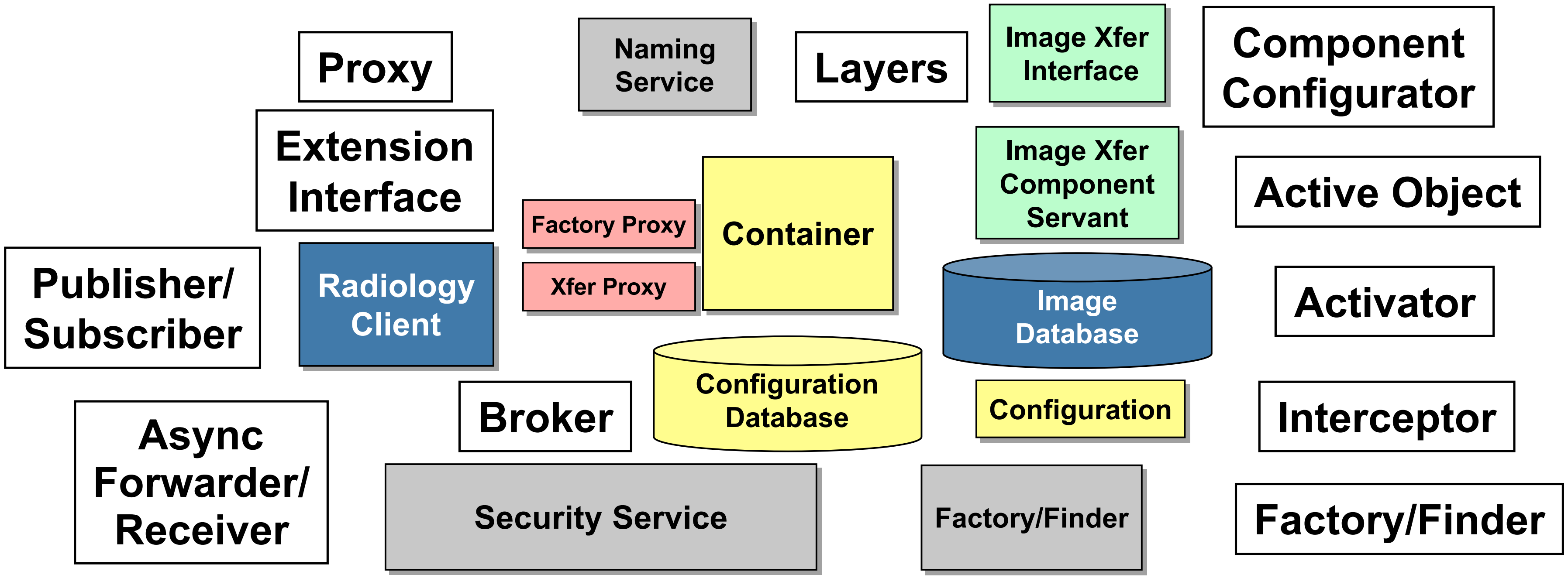
Image Acquisition Scenario



Key Tasks

1. ***Image routing***
2. ***Image delivery***

Applying Patterns to Resolve Key Design Challenges



Patterns help resolve the following common design challenges:

- Separating concerns between tiers
- Improving type-safety & performance
- Enabling client extensibility
- Ensuring platform-neutral & network-transparent OO comm.
- Supporting async comm.
- Supporting OO async comm.

- Decoupling suppliers & consumers
- Providing mechanisms to find & create remote components
- Locating & creating components effectively
- Extending components transparently
- Minimizing resource utilization
- Enhancing server (re)configurability

Separating Concerns Between Tiers

Context

- Distributed systems are now common due to the advent of
  - The global Internet
  - Ubiquitous mobile & embedded devices

Problem

- One reason it's hard to build COTS-based distributed systems is because a large number of capabilities must be provided to meet end-to-end application requirements

Solution

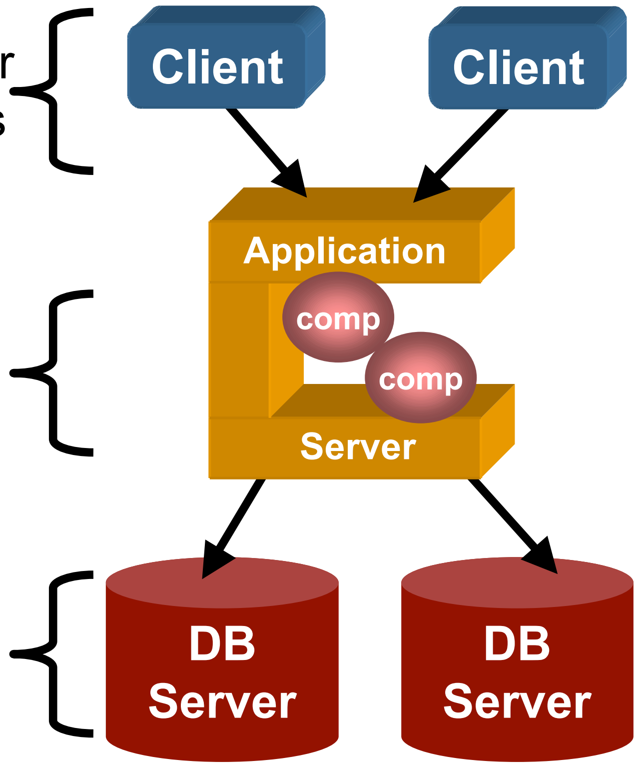
- Apply the *Layers* architectural pattern to create a multi-tier architecture that separates concerns between groups of subtasks occurring at distinct layers in the distributed system

- Services in the *middle-tier* participate in various types of tasks, *e.g.*,
  - Workflow of integrated "business" processes
  - Connect to databases & other backend systems for data storage & access

Presentation Tier
- *e.g.*, thin clients

Middle Tier
- *e.g.*, common business logic

Database Tier
- *e.g.*, persistent data

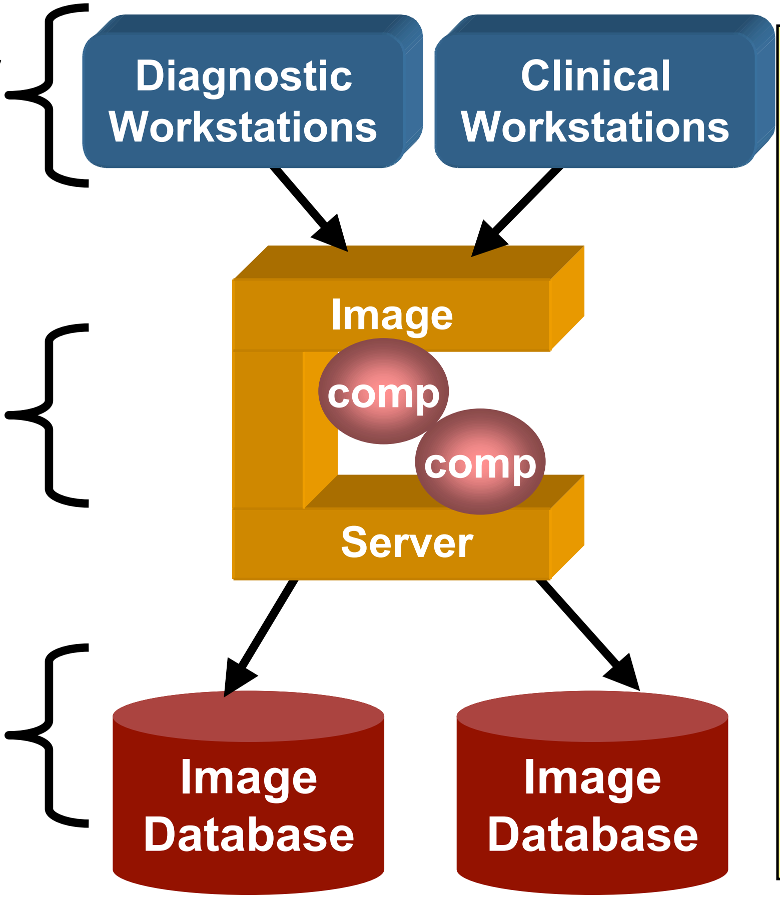


Applying the Layers Pattern to Image Acquisition

Presentation Tier
- *e.g.*, radiology clients

Middle Tier
- *e.g.*, image routing & image transfer logic

Database Tier
- *e.g.*, persistent image data



Diagnostic & clinical workstations are presentation tier components that:
- Typically represent sophisticated GUI elements
- Share the same address space with their clients
  - Their clients are containers that provide all the resources
- Exchange messages with the middle tier components

Image servers are middle tier components that:
- Provide server-side functionality
  - *e.g.*, they are responsible for scalable concurrency & networking
- Can run in their own address space
- Are integrated into containers that hide low-level system details

Pros & Cons of the Layers Pattern

This pattern has four **benefits**:

- ***Reuse of layers***
  - If an individual layer embodies a well-defined abstraction & has a well-defined & documented interface, the layer can be reused in multiple contexts
- ***Support for standardization***
  - Clearly-defined and commonly-accepted levels of abstraction enable the development of standardized tasks & interfaces
- ***Dependencies are localized***
  - Standardized interfaces between layers usually confine the effect of code changes to the layer that is changed
- ***Exchangeability***
  - Individual layer implementations can be replaced by semantically-equivalent implementations without undue effort

This pattern also has **liabilities**:

- ***Cascades of changing behavior***
  - If layer interfaces & semantics aren't abstracted properly then changes can ripple when behavior of a layer is modified
- ***Lower efficiency***
  - A layered architecture can be less efficient than a monolithic architecture
- ***Unnecessary work***
  - If some services performed by lower layers perform excessive or duplicate work not actually required by the higher layer, performance can suffer
- ***Difficulty of establishing the correct granularity of layers***
  - It's important to avoid too many & too few layers

Overview of Distributed Object Computing Communication Mechanisms

Context

In multi-tier systems both the *tiers* & the *components* within the tiers must be connected via *communication mechanisms*

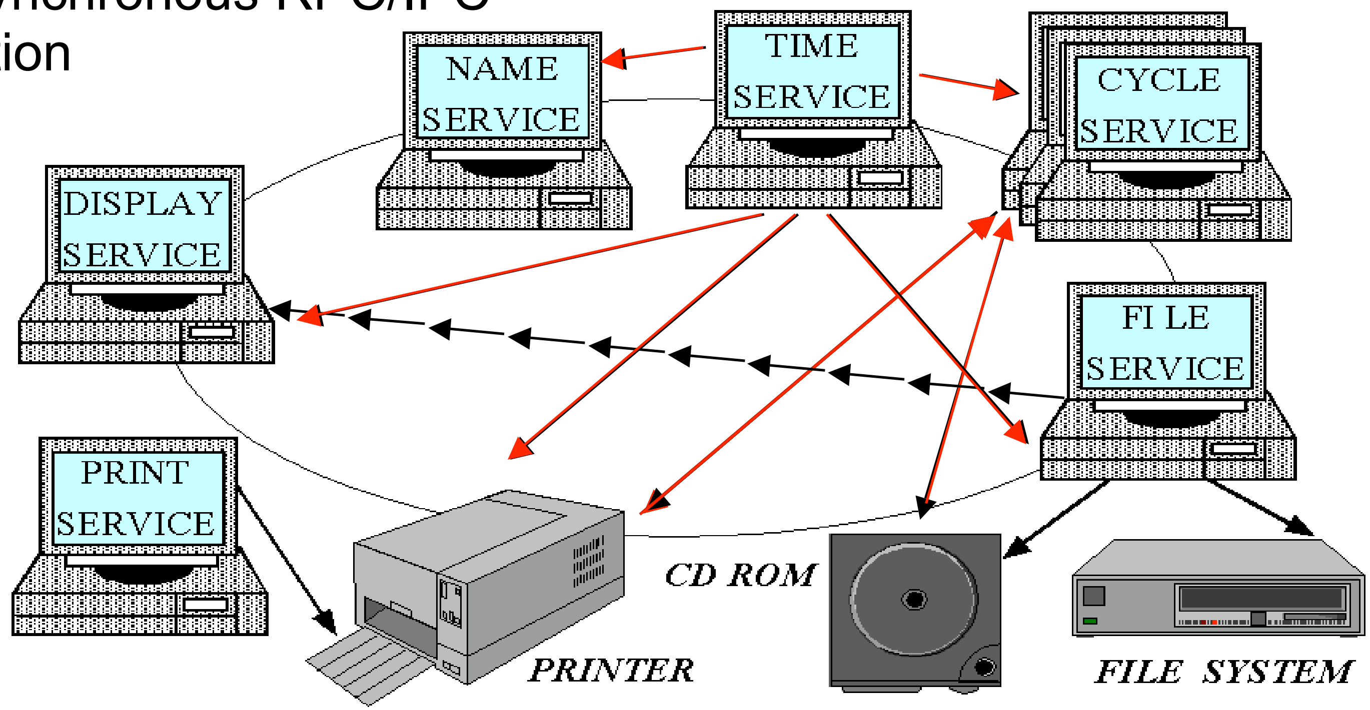
Problem

- A single communication mechanism does not fit all uses!

Solution

- DOC middleware provides multiple types of communication mechanisms
  - Collocated client/server (*i.e.,* native function call)
  - Synchronous & asynchronous RPC/IPC
  - Group communication
  - Data streaming

Next, we'll explore various patterns that applications can apply to leverage these communication mechanisms



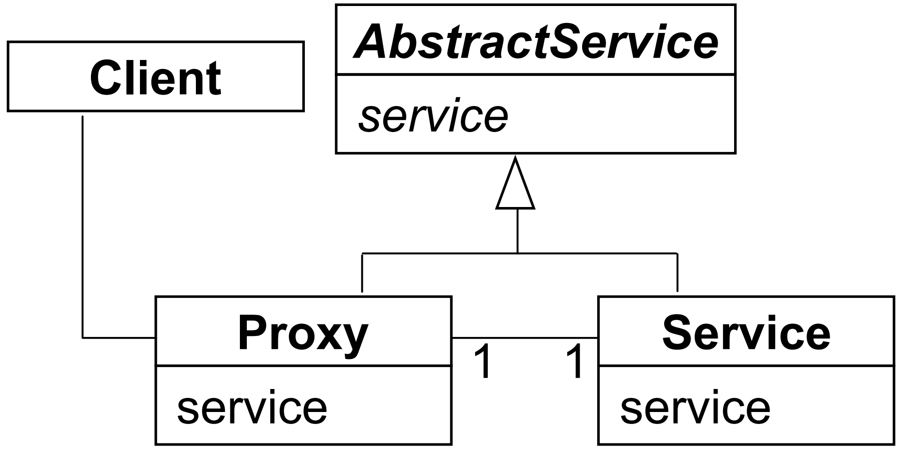
Improving Type-safety & Performance

Context

- The configuration of components in distributed systems is often subject to change as requirements evolve

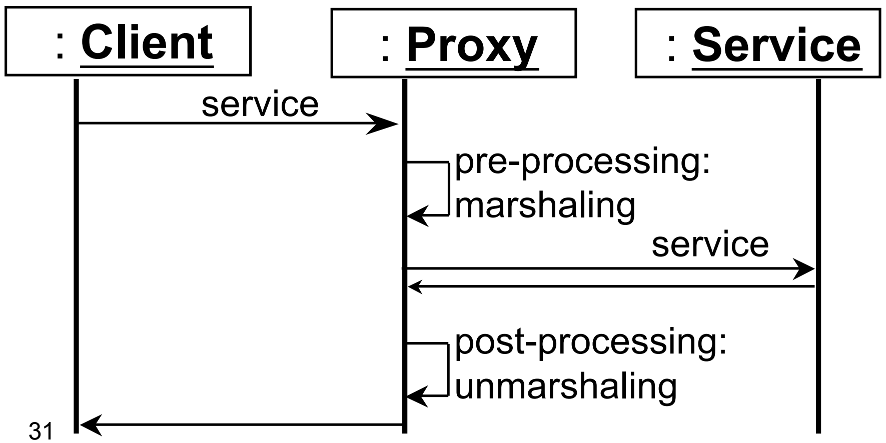
Problems

- Low-level message passing is fraught with accidental complexity
- Remote components should look like local components from an application perspective
  - *i.e.,* clients & servers should be oblivious to communication issues



Solution

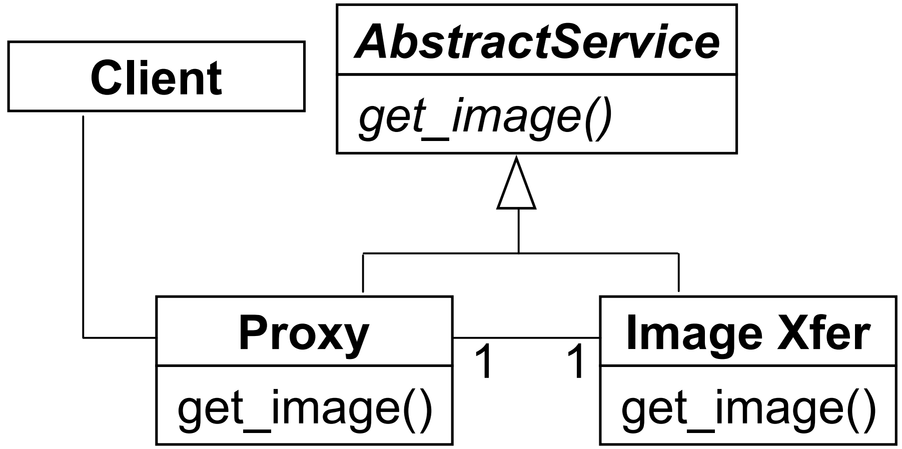
Apply the *Proxy* design pattern to provide an OO surrogate through which clients can access remote objects

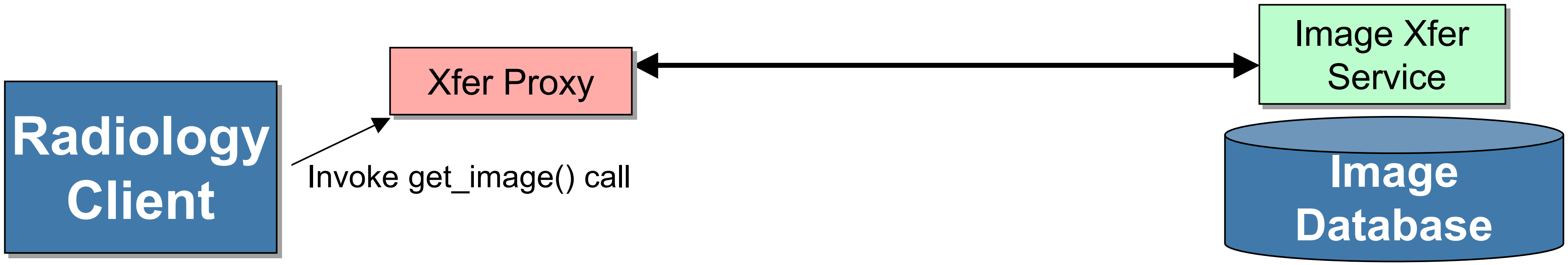


- A *Service* implements the object, which is not accessible directly
- A *Proxy* represents the Service and ensures the correct access to it
  - Proxy offers same interface as Service
- *Clients* use the Proxy to access the Service

Applying the Proxy Pattern to Image Acquisition

We can apply the Proxy pattern to provide a strongly-typed interface to initiate & coordinate the downloading of images from an image database





When proxies are generated automatically by middleware they can be optimized to be much more efficient than manual message passing

- *e.g.,* improved memory management, data copying, & compiled marshaling/demarshaling

Pros & Cons of the Proxy Pattern

This pattern provides three **benefits**:

- ***Decoupling clients from the location of server components***
  - By putting all location information & addressing functionality into a proxy clients are not affected by migration of servers or changes in the networking infrastructure
- ***Potential for time & space optimizations***
  - Proxy implementations can be loaded "on-demand" and can also be used to cache values to avoid remote calls
  - Proxies can also be optimized to improve both type-safety & performance
- ***Separation of housekeeping & functionality***
  - A proxy relieves clients from burdens that do not inherently belong to the task the client performs

This pattern has two **liabilities**:

- ***Potential overkill via sophisticated strategies***
  - If proxies include overly sophisticated functionality they many introduce overhead that defeats their intended purpose
- ***Less efficiency due to indirection***
  - Proxies introduce an additional layer of indirection that can be excessive if the proxy implementation is inefficient

Enabling Client Extensibility

Context

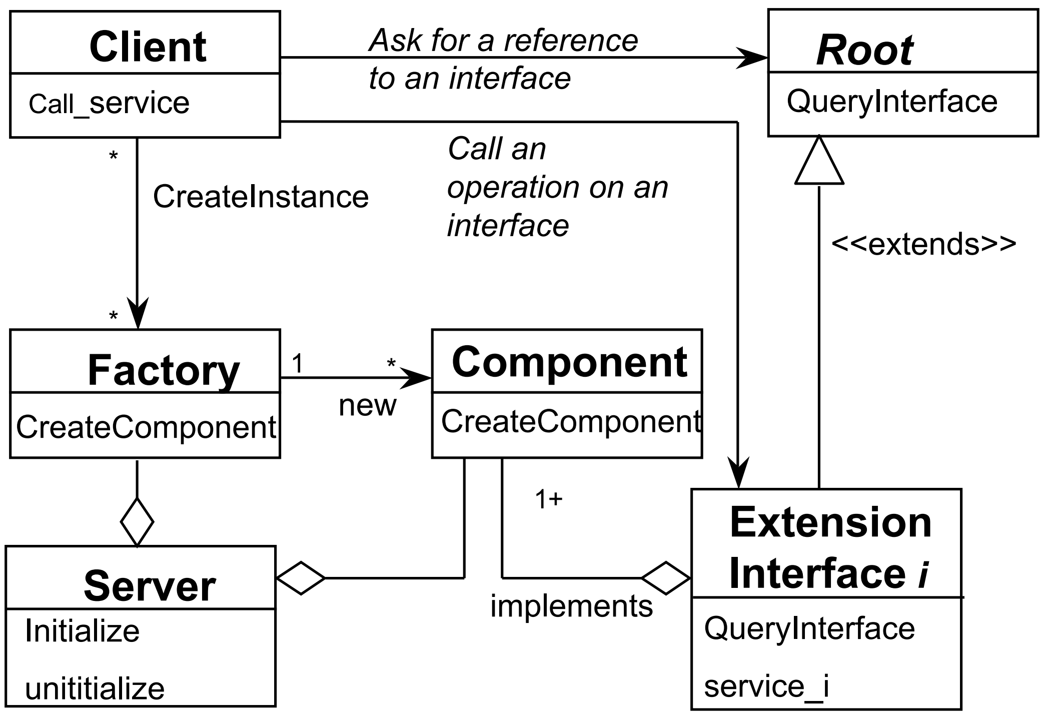
- Object models define how components import & export functionality
  - *e.g.*, UML class diagrams specify well-defined OO interfaces

Problem

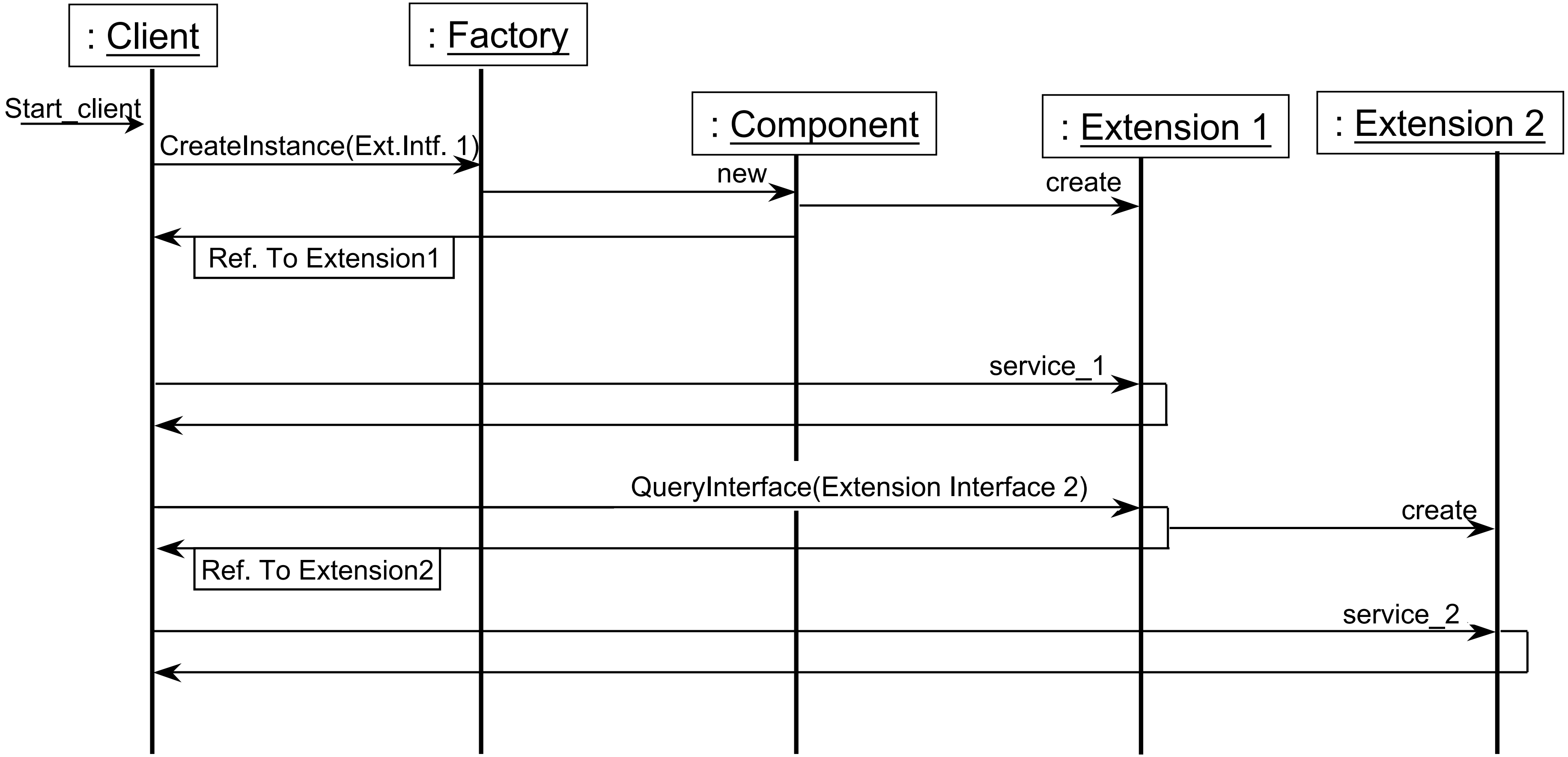
- Many object models assign a single interface to each component
- This design makes it hard to evolve components *without*
  - Breaking existing client interfaces
  - Bloating client interfaces

Solution

- Apply the *Extension Interface* design pattern to allow multiple interfaces to be exported by a component, to prevent bloating of interfaces & breaking of client code when developers extend or modify component functionality



Extension Interface Pattern Dynamics



Note how each extension interface can serve as a “factory” to return object reference to other extension interfaces

Pros & Cons of the Extension Interface Pattern

This pattern has five **benefits**:

- ***Separation of concerns***
  - Interfaces are strictly decoupled from implementations
- ***Exchangeability of components***
  - Component implementations can evolve independently from clients that access them
- ***Extensibility through interfaces***
  - Clients only access components via their interfaces, which reduces coupling to representation & implementation details
- ***Prevention of interface bloating***
  - Interfaces need not contain all possible methods, just the ones associated with a particular capability
- ***No subclassing required***
  - Delegation—rather than inheritance—is used to customize components

This pattern also has **liabilities**:

- ***Overhead due to indirection***
  - Clients must incur the overhead of several round-trips to obtain the appropriate object reference from a server component
- ***Complexity & cost for development & deployment***
  - This pattern off-loads the responsibility for determining the appropriate interface from the component designer to the client application

Ensuring Platform-neutral & Network-transparent OO Communication

Context

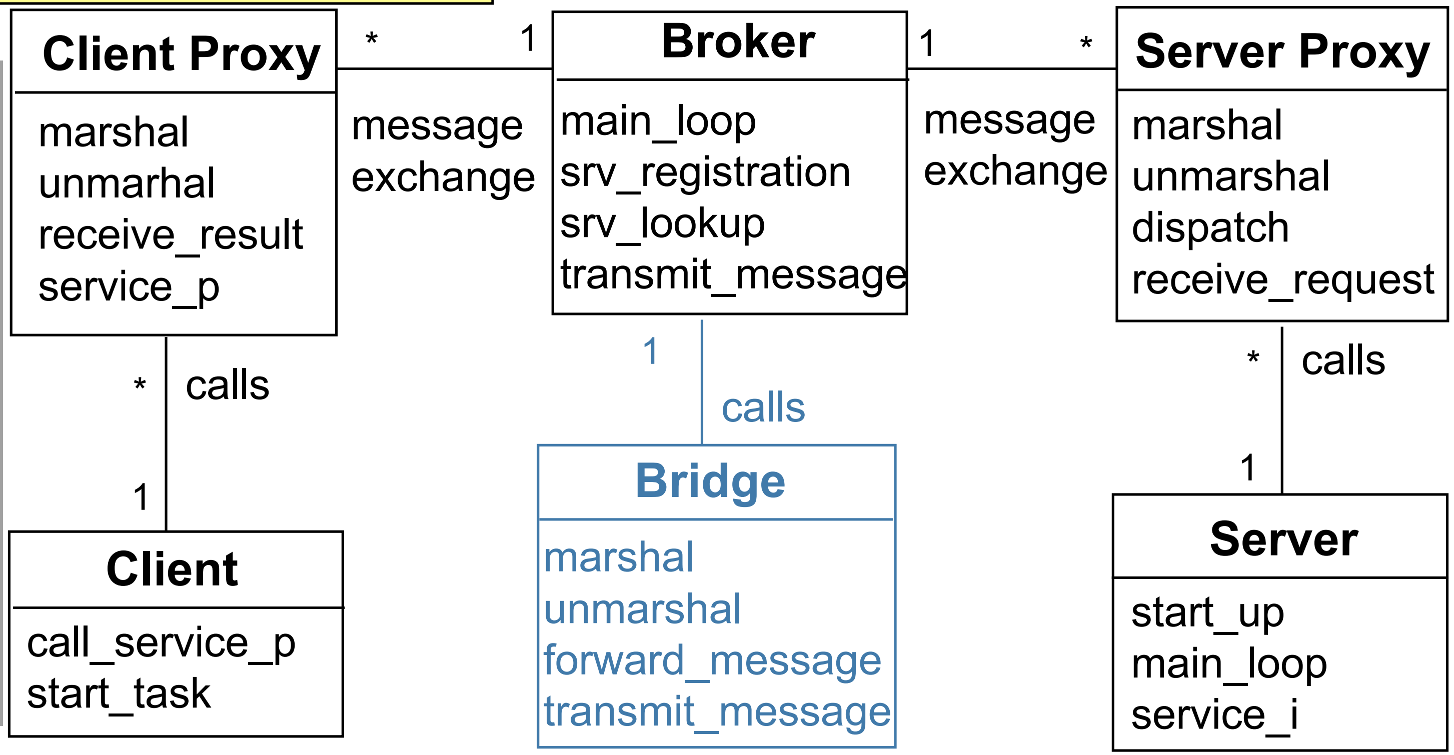
- Using the Proxy pattern is insufficient since it doesn‘t address how
  - Remote components are located
  - Connections are established
  - Messages are exchanged across a network
  - *etc.*

Problem

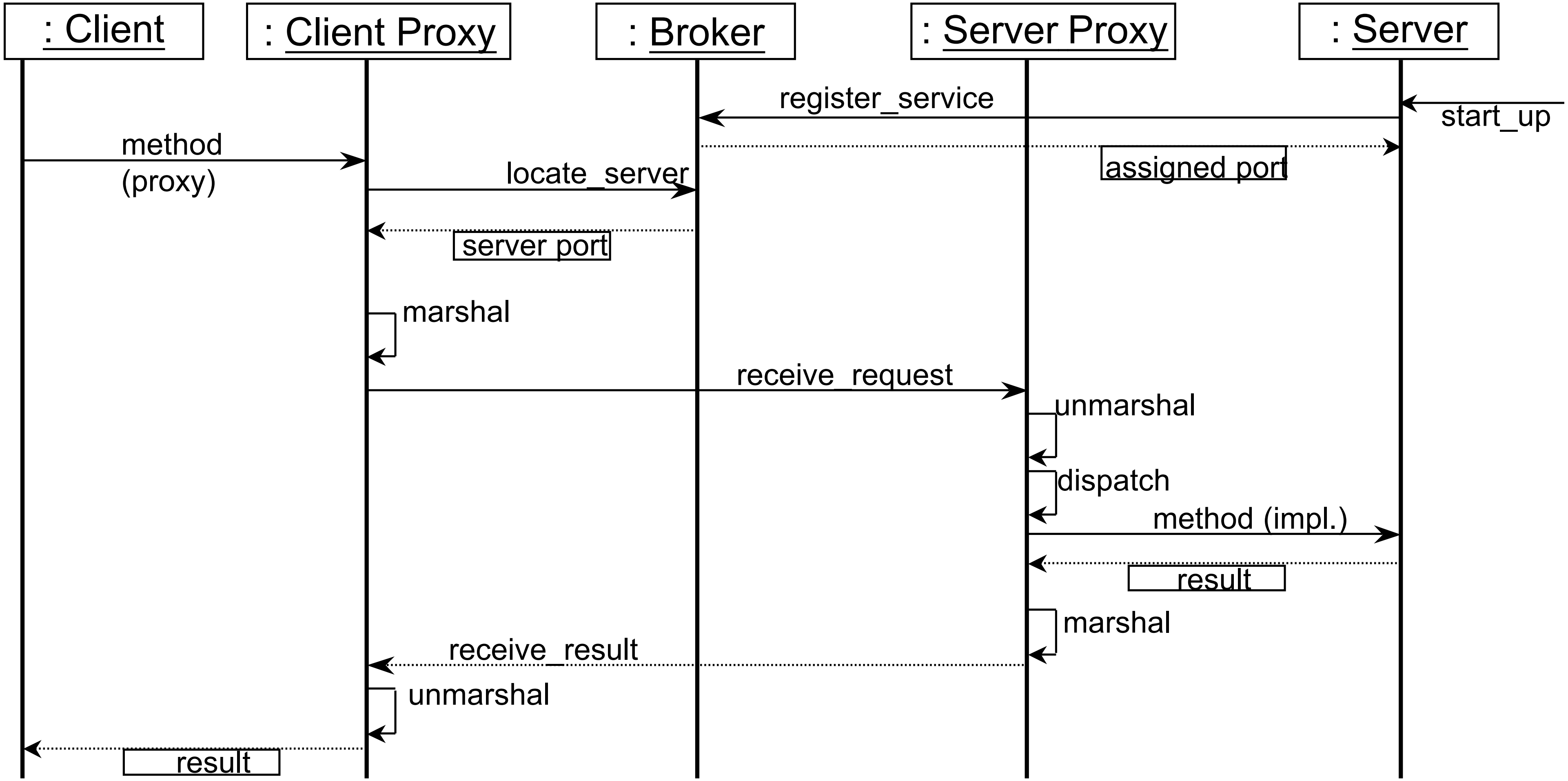
- We need an *architecture* that:
  - Supports remote method invocation
  - Provides location transparency
  - Allows the addition, exchange, or remove of services dynamically
  - Hides system details from the developer

Solution

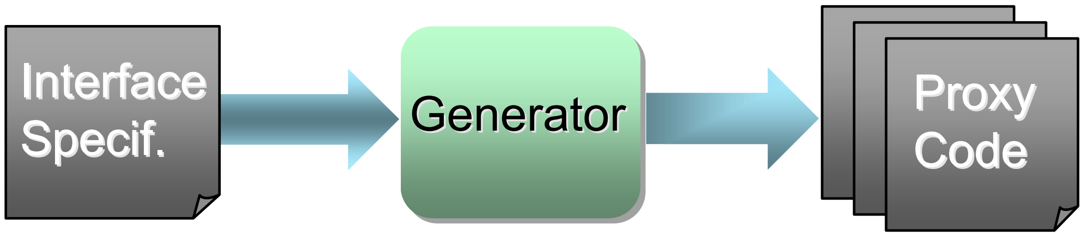
- Apply the *Broker* pattern to provide OO platform-neutral communication between networked client & server components



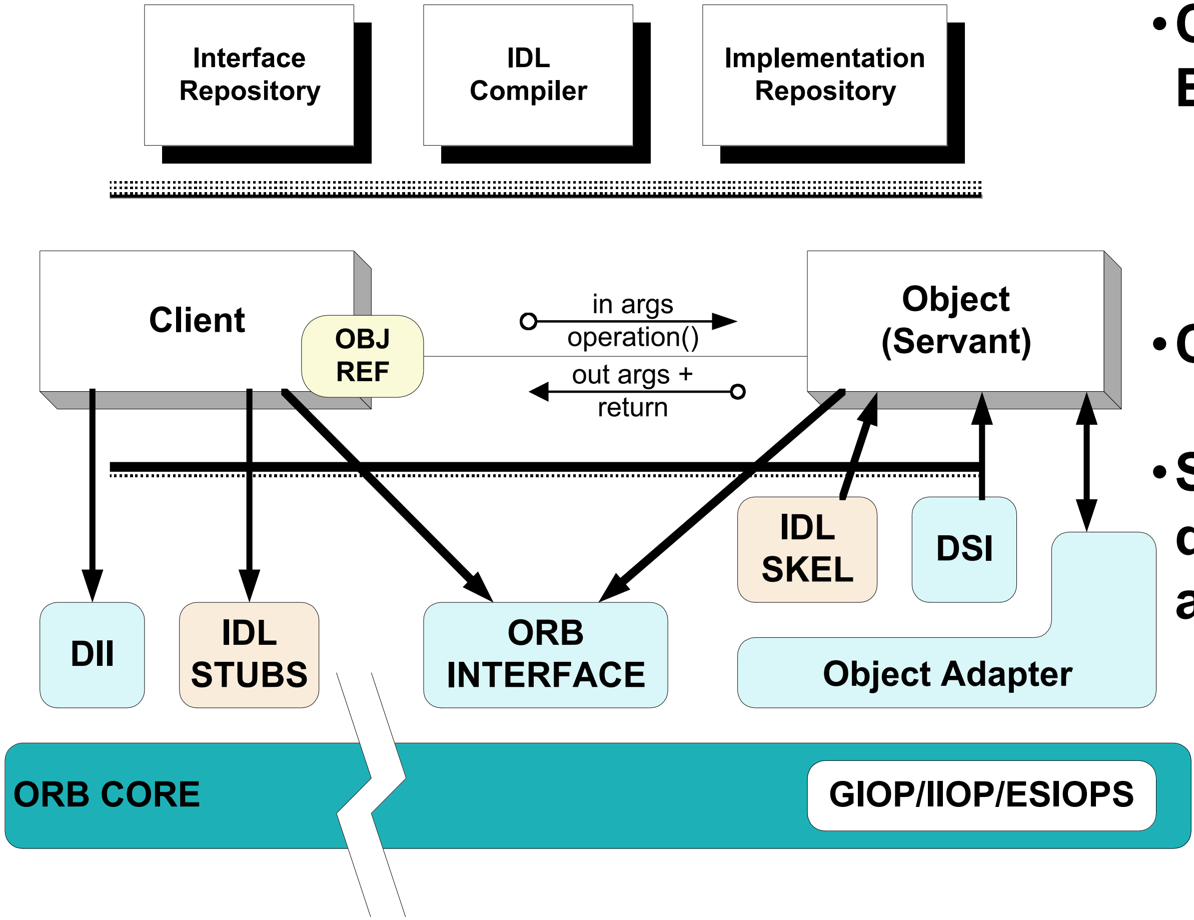
Broker Pattern Dynamics



Broker tools provide the generation of necessary client & server proxies from higher level interface definitions



Applying the Broker Pattern to Image Acquisition



- CORBA shields applications from environment heterogeneity
  - *e.g.*, programming languages, operating systems, networking protocols, hardware

- **Common Object Request Broker Architecture (CORBA)**
  - A family of specifications
  - OMG is the standards body
  - Over 800 companies
- **CORBA defines interfaces**
  - Rather than implementations
- **Simplifies development of distributed applications by automating**
  - Object location
  - Connection management
  - Memory management
  - Parameter (de)marshaling
  - Event & request demuxing
  - Error handling
  - Object/server activation
  - Concurrency

Pros & Cons of the Broker Pattern

This pattern has five **benefits**:

- ***Portability enhancements***
  - A broker hides OS & network system details from clients and servers by using indirection & abstraction layers, such as APIs, proxies, adapters, & bridges
- ***Interoperability with other brokers***
  - Different brokers may interoperate if they understand a common protocol for exchanging messages
- ***Reusability of services***
  - When building new applications, brokers enable application functionality to reuse existing services
- ***Location transparency***
  - A broker is responsible for locating servers, so clients need not know where servers are located
- ***Changeability & extensibility of components***
  - If server implementations change without affecting interfaces clients should not be affected

This pattern also has **liabilities**:

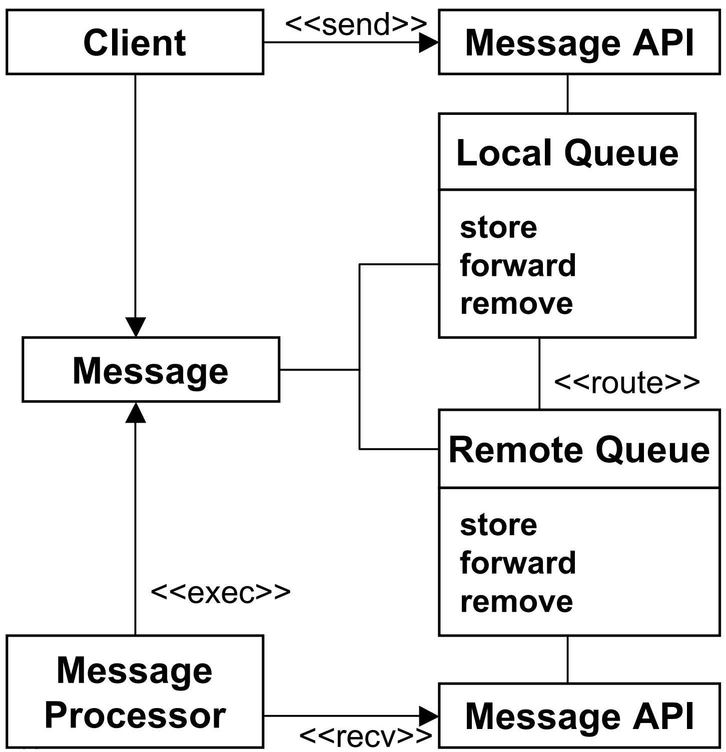
- ***Restricted efficiency***
  - Applications using brokers may be slower than applications written manually
- ***Lower fault tolerance***
  - Compared with non-distributed software applications, distributed broker systems may incur lower fault tolerance
- ***Testing & debugging may be harder***
  - Testing & debugging of distributed systems is tedious because of all the components involved

Supporting Async Communication

Context
- Some clients want to send requests, continue their work, & receive the results at some later point in time

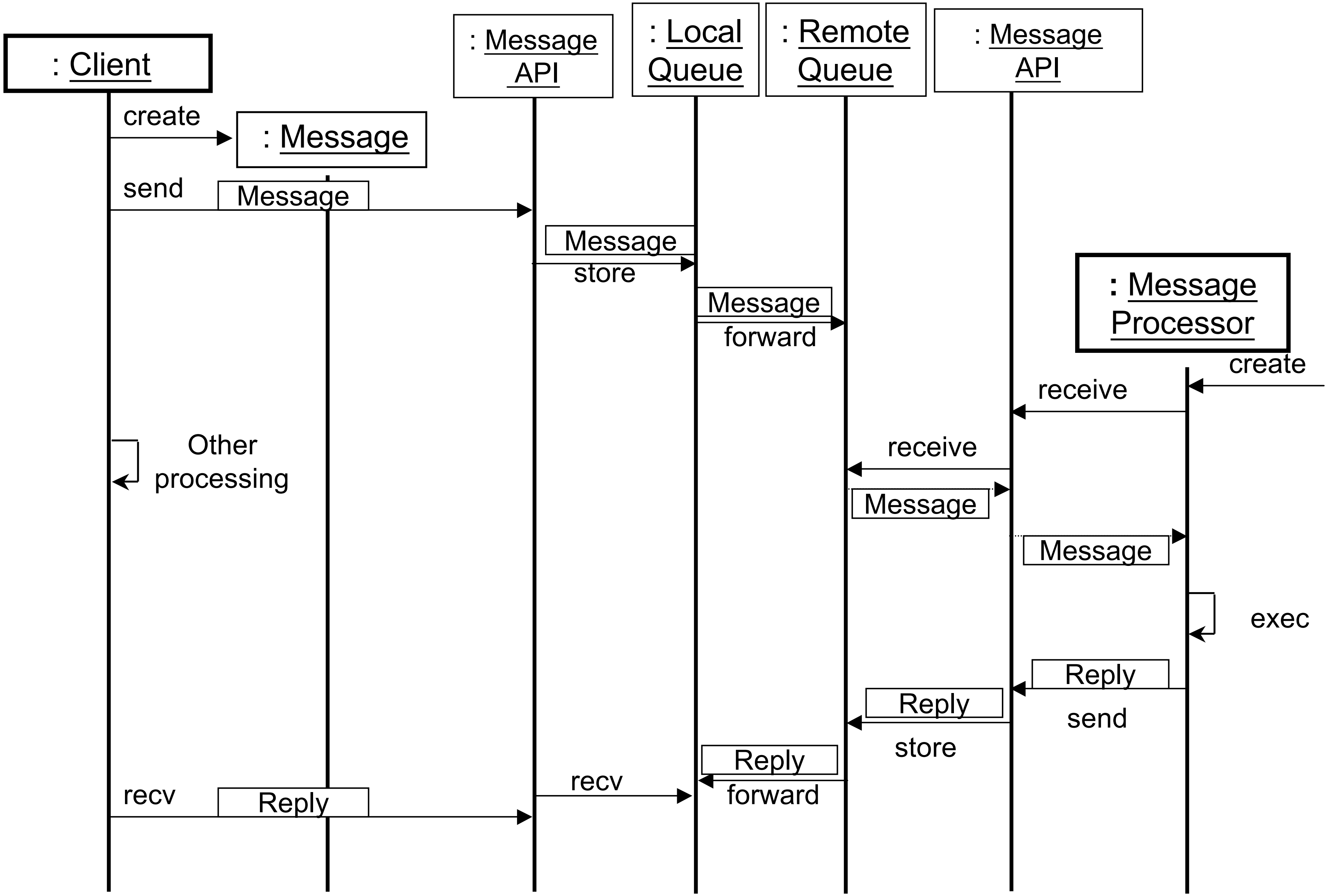
Problem
- Broker implementations based on conventional RPC semantics often just support *blocking* operations
  - *i.e.,* clients must wait until two-way invocations return
- Unfortunately, this design can reduce scalability & complicate certain use-cases

Solution
- Apply the *Async Forwarder/Receiver* design pattern to allow asynchronous communication between clients & servers



Introduce intermediary queue(s) between clients & servers:
- A *queue* is used to store messages
  - A queue can cooperate with other queues to route messages
- *Messages* are sent from sender to receiver
- A *client* sends a message, which is queued & then forwarded to a *message processor* on a server that receives & executes them
- A *Message API* is provided for clients & servers to send/receive messages

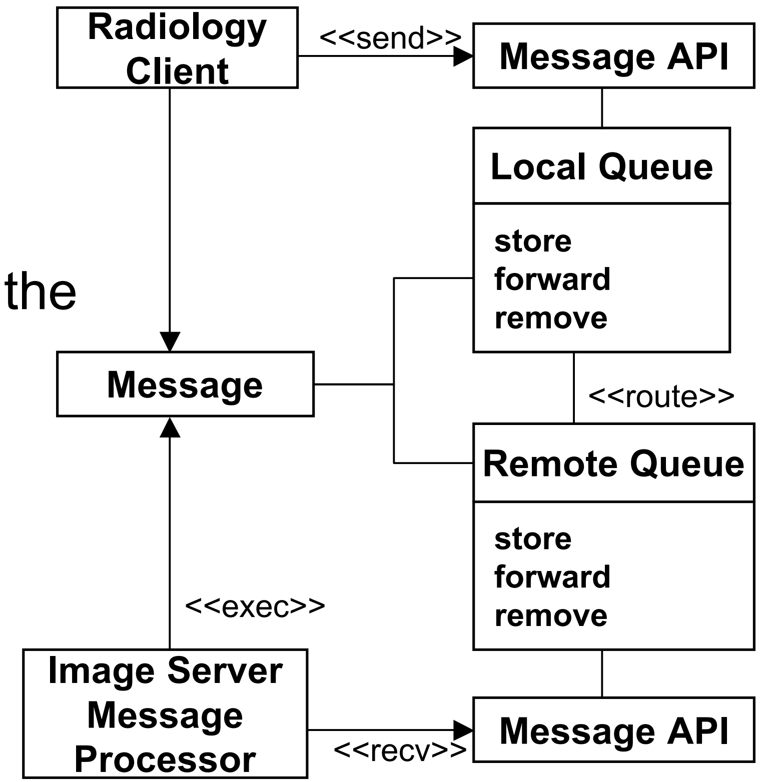
Async Forwarder/Receiver Pattern Dynamics

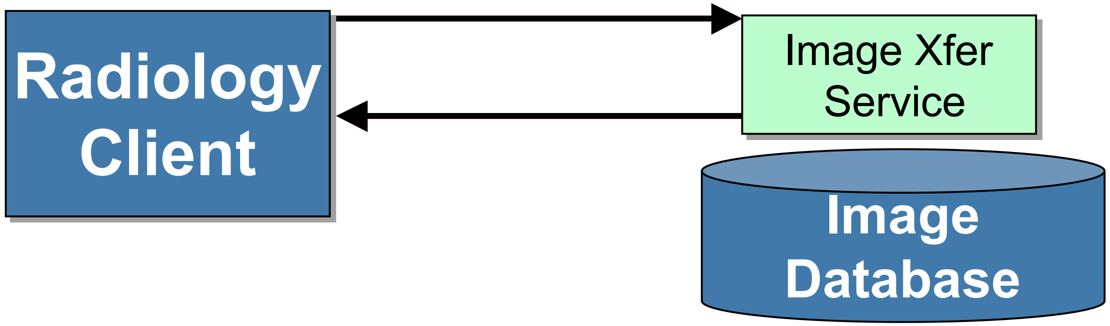


Applying the Async Forwarder/Receiver Pattern to Image Acquisition

We can apply the Async Forwarder/Receiver pattern to

- Queue up image request messages remotely without blocking the diagnostic/clinical workstation clients
- Execute the requests at a later point & return the results to the client





This design also enables other, more advanced capabilities, *e.g.*,

- Multi-hop store & forward persistence
- QoS-driven routing, where requests can be delivered to the "best" image database

Pros & Cons of the Async Forwarder/Receiver Pattern

This pattern provides three **benefits**:

- ***Enhances concurrency by transparently leveraging available parallelism***
  - Messages can be executed remotely on servers while clients perform other processing
- ***Simplifies synchronized access to a shared object that resides in its own thread of control***
  - Since messages are processed serially by a message processor target objects often need not be concerned with synchronization mechanisms
- ***Message execution order can differ from message invocation order***
  - This allows reprioritizing of messages to enhance quality of service

This pattern also has some **liabilities**:

- ***Message execution order can differ from message invocation order***
  - As a result, clients must be careful not to rely on ordering dependencies
- ***Lack of type-safety***
  - Clients & servers are responsible for formatting & passing messages
- ***Complicated debugging***
  - As with all distributed systems, debugging & testing is complex

Supporting OO Async Communication

Context

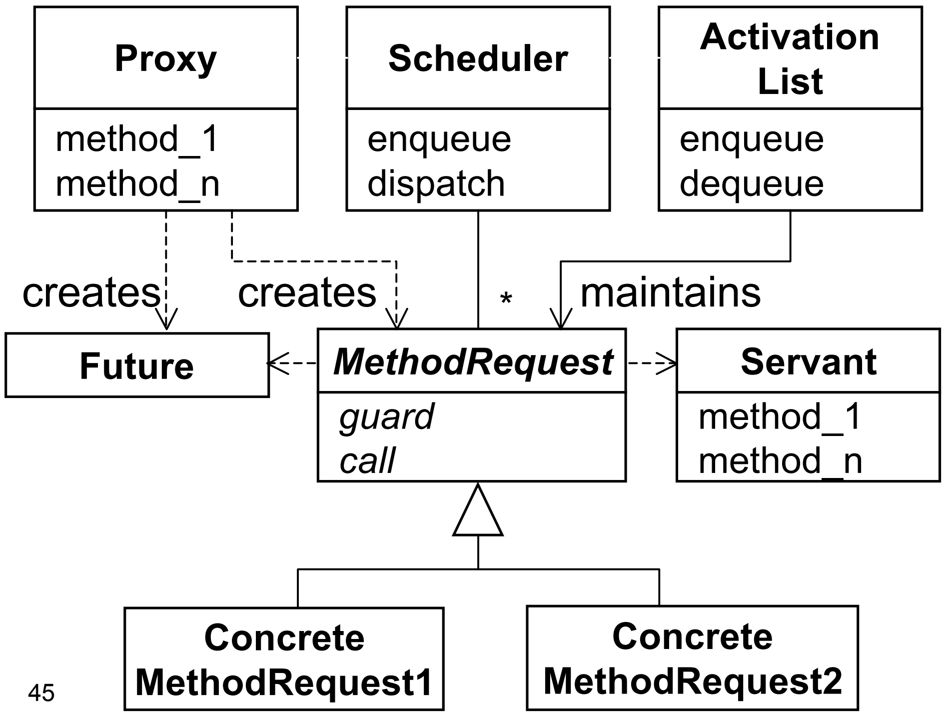
- Some clients want to invoke remote operations, continue their work, & retrieve the results at a later point in time

Problem

- Using the explicit message-passing API of the Async Forwarder/Receiver pattern can reduce type-safety & performance
  - Similar to motivation for Proxy pattern...

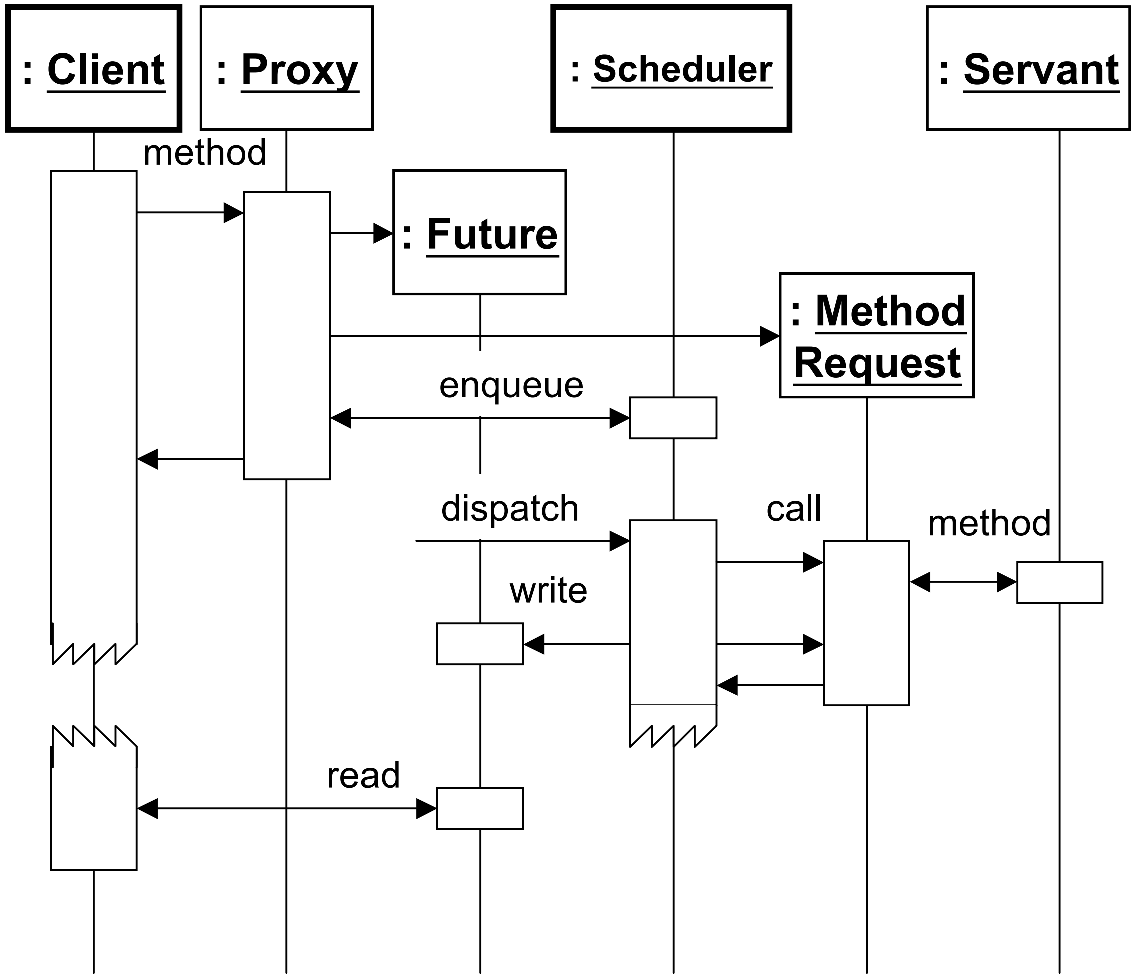
Solution

- Apply the *Active Object* design pattern to decouple method invocation from method execution using an object-oriented programming model



- A *proxy* provides an interface that allows clients to access methods of an object
- A *concrete method request* is created for every method invoked on the proxy
- A *scheduler* receives the method requests & dispatches them on the servant when they become runnable
- An *activation list* maintains pending method requests
- A *servant* implements the methods
- A *future* allows clients to access the results of a method call on the proxy

Active Object Pattern Dynamics

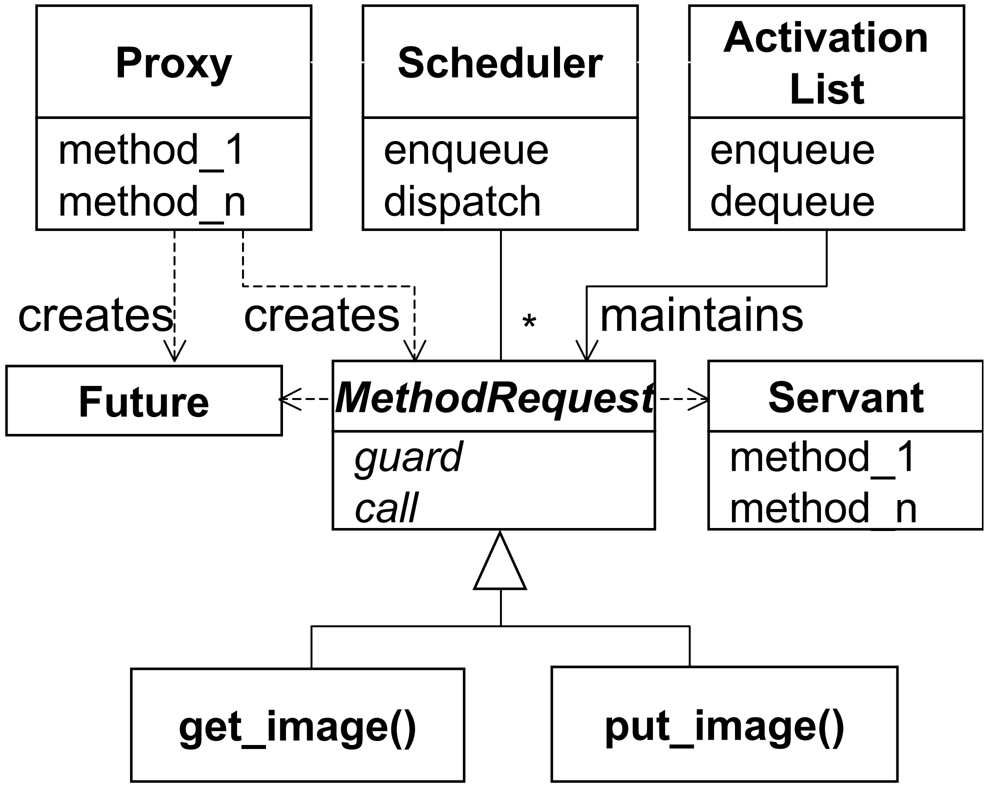


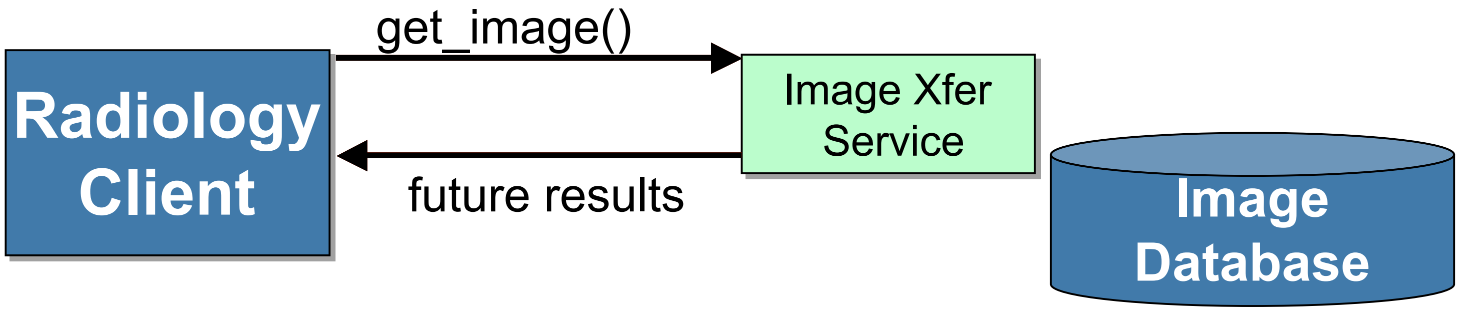
Clients can obtain result from futures via blocking, polling, or callbacks

- A *client* invokes a method on the *proxy*
- The *proxy* returns a future to the client, & creates a *method request,* which it passes to the *scheduler*
- The *scheduler* enqueues the *method request* into the *activation list* (not shown here)
- When the *method request* becomes runnable, the *scheduler* dequeues it from the *activation list* (not shown here) & executes it in a different thread than the client
- The *method request* executes the method on the *servant* & writes results, if any, to the *future*
- *Clients* obtain the method's results via the *future*

Applying the Active Object Pattern to Image Acquisition

- OO developers generally prefer *method-oriented* request/response semantics to *message-oriented* semantics
- The Active Object pattern supports this preference via strongly-typed async method APIs:
  - Several types of parameters can be passed:
    - Requests contain in/inout arguments
    - Results carry out/inout arguments & results
  - Callback object or poller object can be used to retrieve results





Pros & Cons of the Active Object Pattern

This pattern provides four **benefits**:

- ***Enhanced type-safety***
  - Compared with async message passing
- ***Enhances concurrency & simplifies synchronized complexity***
  - Concurrency is enhanced by allowing client threads & asynchronous method executions to run simultaneously
  - Synchronization complexity is simplified by using a scheduler that evaluates synchronization constraints to guarantee serialized access to servants
- ***Transparent leveraging of available parallelism***
  - Multiple active object methods can execute in parallel if supported by the OS/hardware
- ***Method execution order can differ from method invocation order***
  - Methods invoked asynchronous are executed according to the synchronization constraints defined by their guards & by scheduling policies

This pattern also has some **liabilities**:

- ***Performance overhead***
  - Depending on how an active object's scheduler is implemented, context switching, synchronization, & data movement overhead may occur when scheduling & executing active object invocations
- ***Complicated debugging***
  - It is hard to debug programs that use the Active Object pattern due to the concurrency & non-determinism of the various active object schedulers & the underlying OS thread scheduler

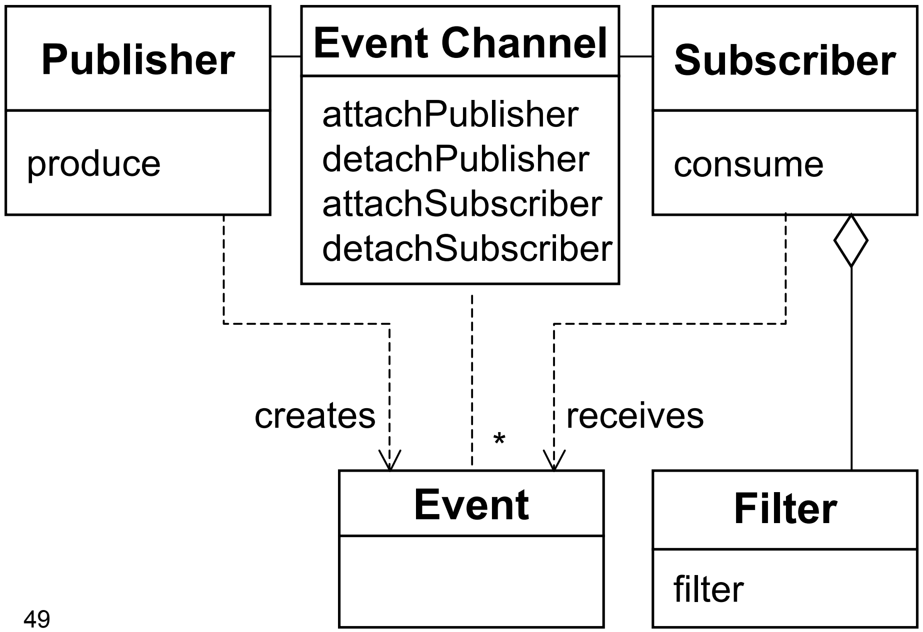
Decoupling Suppliers & Consumers

Context

- In large-scale electronic medical imaging systems, radiologists may share “work lists” of patient images to balance workloads effectively

Solution

- Apply the Publisher/Subscriber pattern to decouple image suppliers from image consumers



Problem

- Having each client call a specific server is inefficient & non-scalable
  - A polling strategy leads to performance bottlenecks
  - Work lists could be spread across different servers
  - More than one client may be interested in work list content

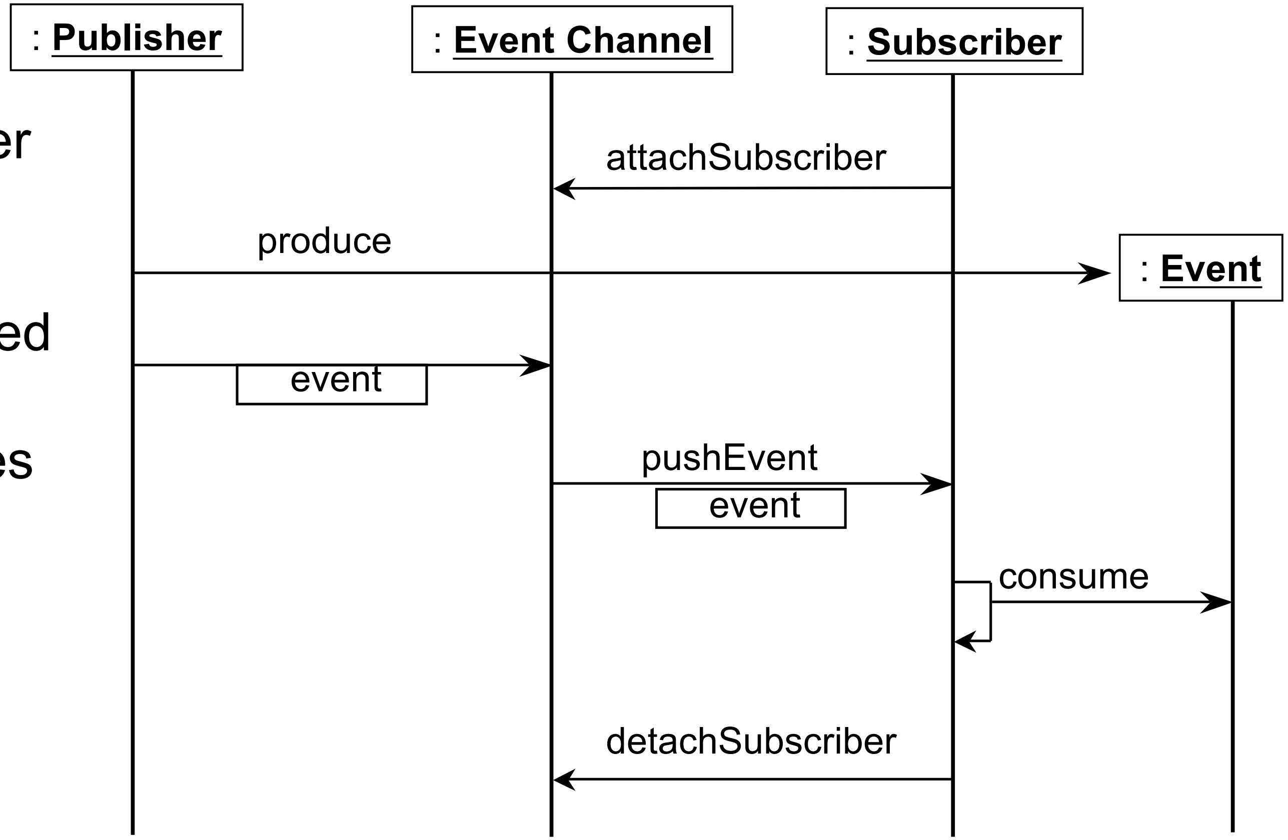
Decouple suppliers (publishers) & consumers (subscribers) of events:

- An *Event Channel* stores/forwards events
- *Publishers* create events & store them in a queue maintained by the Event Channel
- *Consumers* register with event queues, from which they retrieve events
- *Events* are used to transmit state change info from publishers to consumers
- For event transmission *push-models* & *pull-models* are possible
- *Filters* can filter events for consumers

Publisher/Subscriber Pattern Dynamics

- The Publisher/Subscriber pattern helps keep the state of cooperating components synchronized

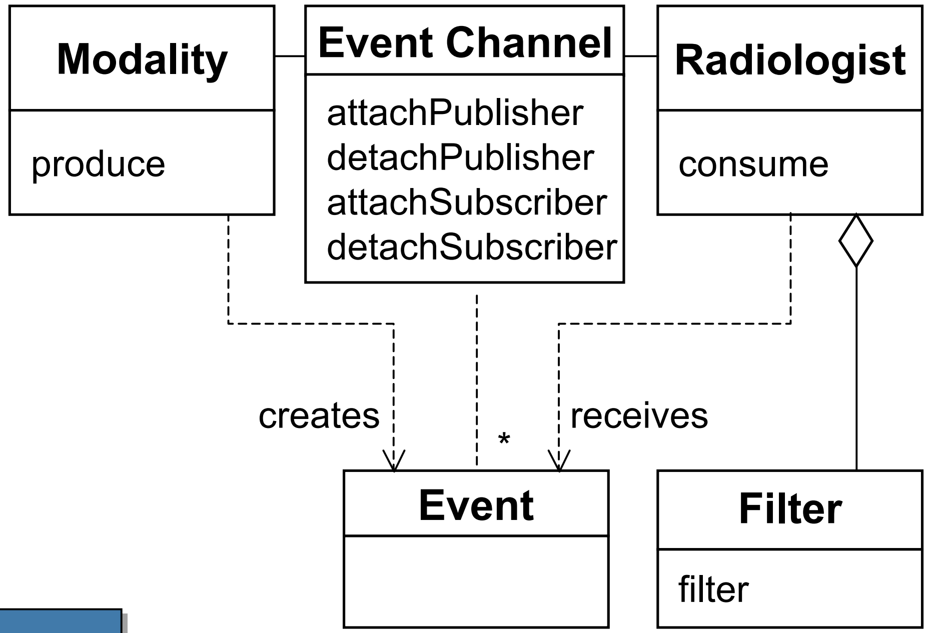
- To achieve this it enables one-way propagation of changes: one publisher notifies any number of subscribers about changes to its state

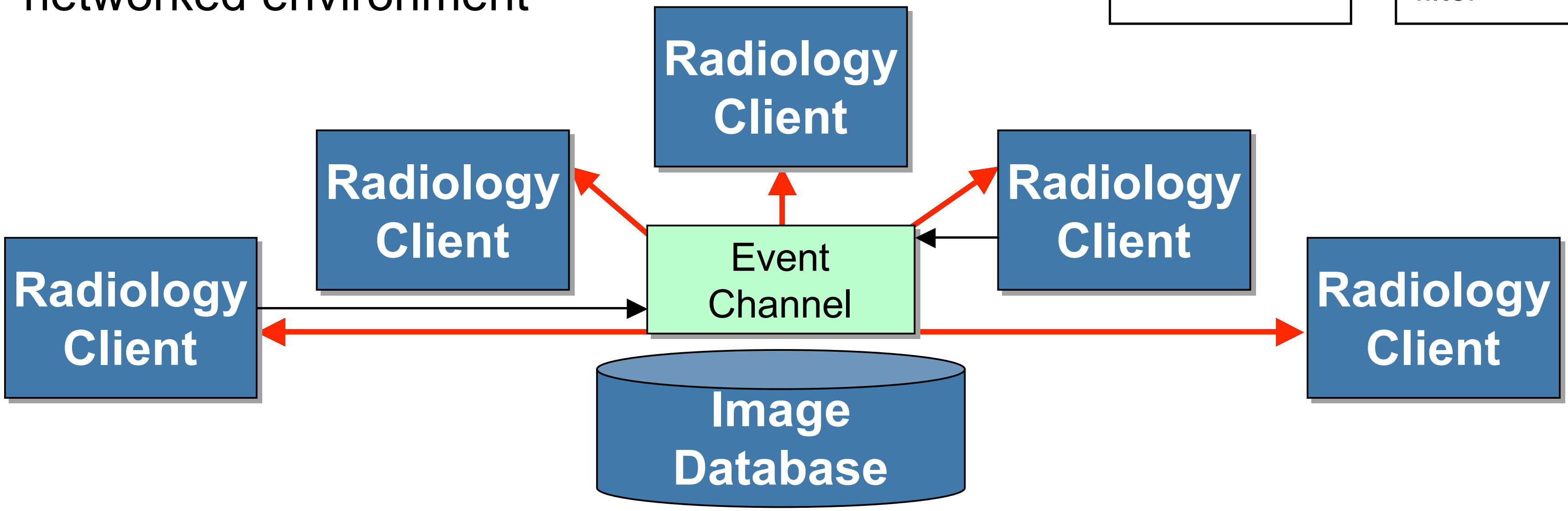


Key design considerations for the Publisher/Subscriber pattern include:
- Push vs. pull interaction models
- Control vs. data event notification models
- Multicast vs. unicast communication models
- Persistence vs. transient queueing models

Applying the Publisher/Subscriber Pattern to Image Acquisition

- Radiologists can subscribe to an event channel in order to receive notifications produced when modalities publish events indicating the arrival of a new image
- This design enables a group of distributed radiologists to collaborate effectively in a networked environment





Pros & Cons of the Publisher/Subscriber Pattern

This pattern has two **benefits**:

- ***Decouples consumers & producers of events***
  - All an event channel knows is that it has a list of consumers, each conforming to the simple interface of the **`Subscriber`** class
  - Thus, the coupling between the publishers and subscribers is abstract & minimal
- ***n:m communication models are supported***
  - Unlike an ordinary request/response interaction, the notification that a publisher sends needn't designate its receiver, which enables a broader range of communication topologies, including multicast & broadcast

There is also a **liability**:

- ***Must be careful with potential update cascades***
  - Since subscribers have no knowledge of each other's presence, applications can be blind to the ultimate cost of publishing events through an event channel
  - Thus, a seemingly innocuous operation on the subject may cause a cascade of updates to observers & their dependent objects

Locating & Creating Components Effectively

Context

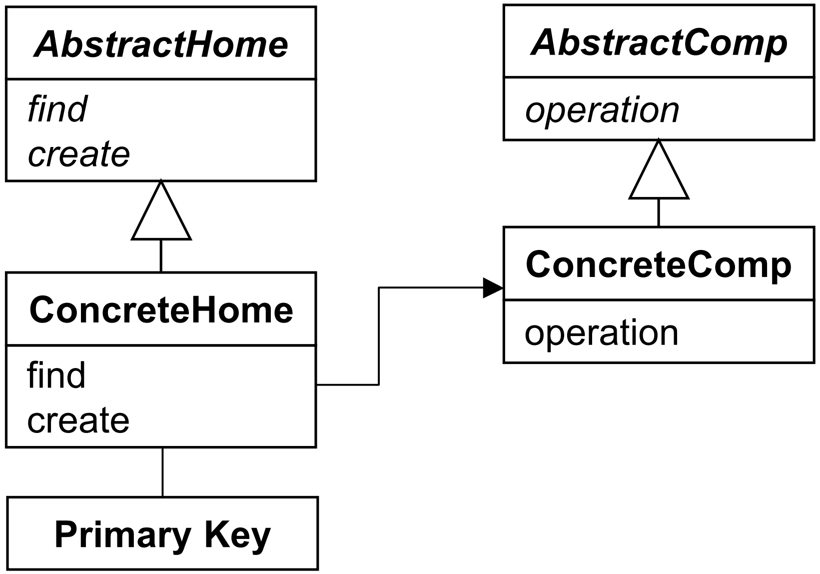
- Our electronic medical imaging system contains many components distributed in a network

Problem

- How to create new components and/or find existing ones
  - Simple solutions appropriate for stand-alone applications don’t scale
  - “Obvious” solutions for distribution also don’t scale

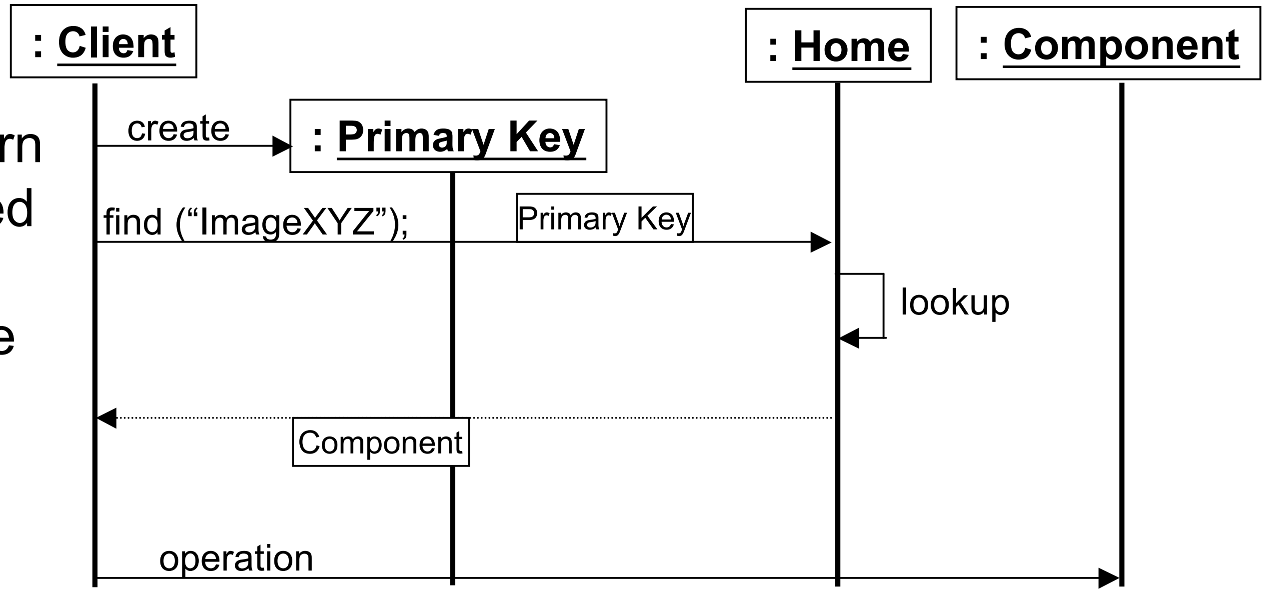
Solution

- Apply the *Factory/Finder* pattern to separate the management of component lifecycles from their use by client applications

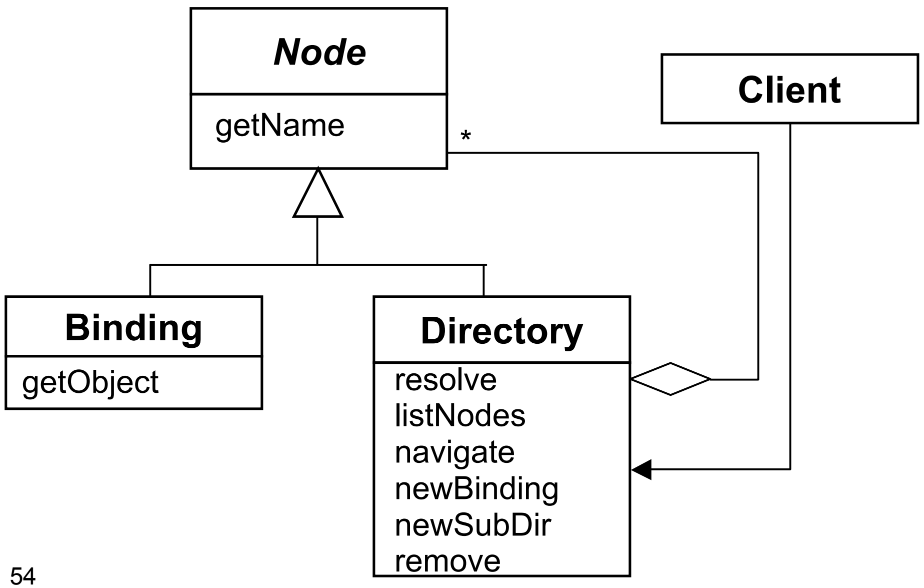


- An *Abstract Home* declares an interface for operations that find and/or create abstract instances of components
- *Concrete Homes* implements the abstract Home interface to find specific instances and/or create new ones
- *Abstract Comp* declares interface for a specific type of component class
- *Concrete Comp* define instances
- A *Primary Key* is associated with a component

Factory/Finder Pattern Dynamics



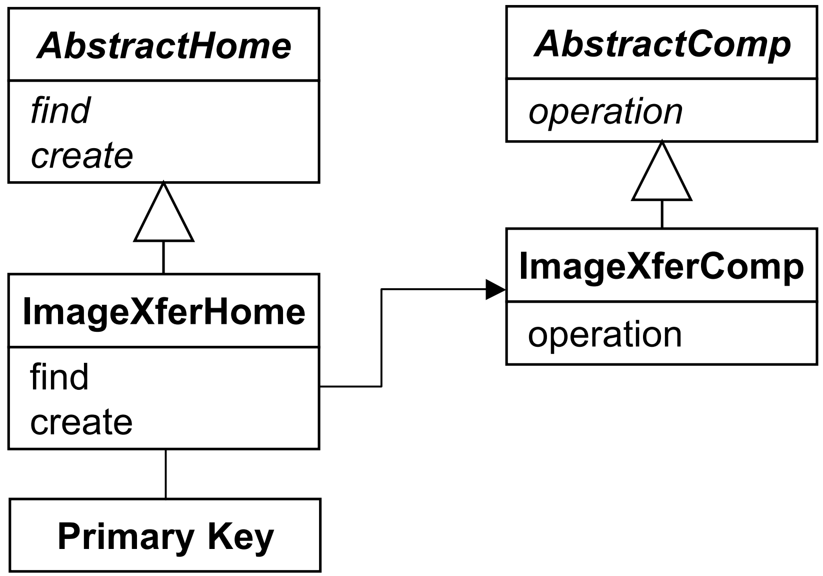
- The Factory/Finder pattern is supported by distributed component models
  - *e.g.*, EJB, COM+, & the CCM

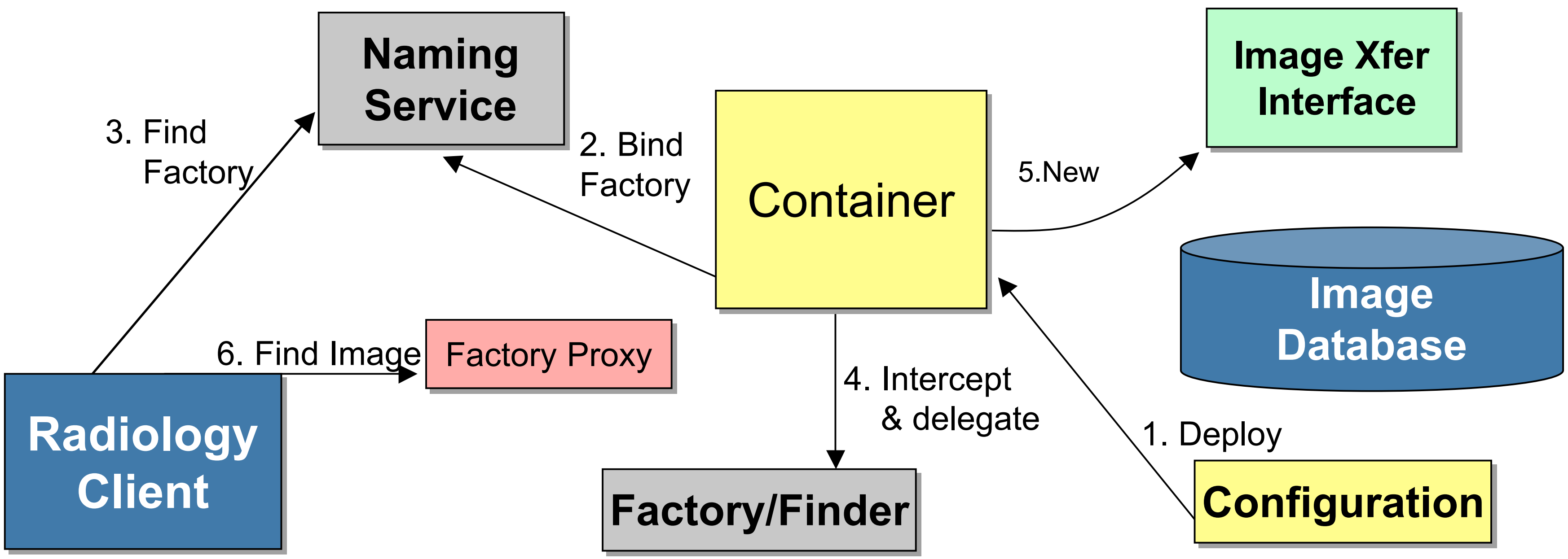


- Homes enable the creation & location of components, but we still need a global naming service to locate the homes

Applying the Factory/Finder Pattern to Image Acquisition

- We can apply the Factory/Finder pattern to create/locate image transfer components for images needed by radiologists
- If a suitable component already exists the component home will use it, otherwise, it will create a new component





Pros & Cons of the Factory/Finder Pattern

This pattern has three **benefits**:

- ***Separation of concerns***
  - Finding/creating individual components is decoupled from locating the factories that find/create these components
- ***Improved scalability***
  - *e.g.,* general-purpose directory mechanisms need not manage the creation & location of large amounts of finer-grained components whose lifetimes may be short
- ***Customized capabilities***
  - The location/creation mechanism can be specialized to support key capabilities that are unique for various types of components

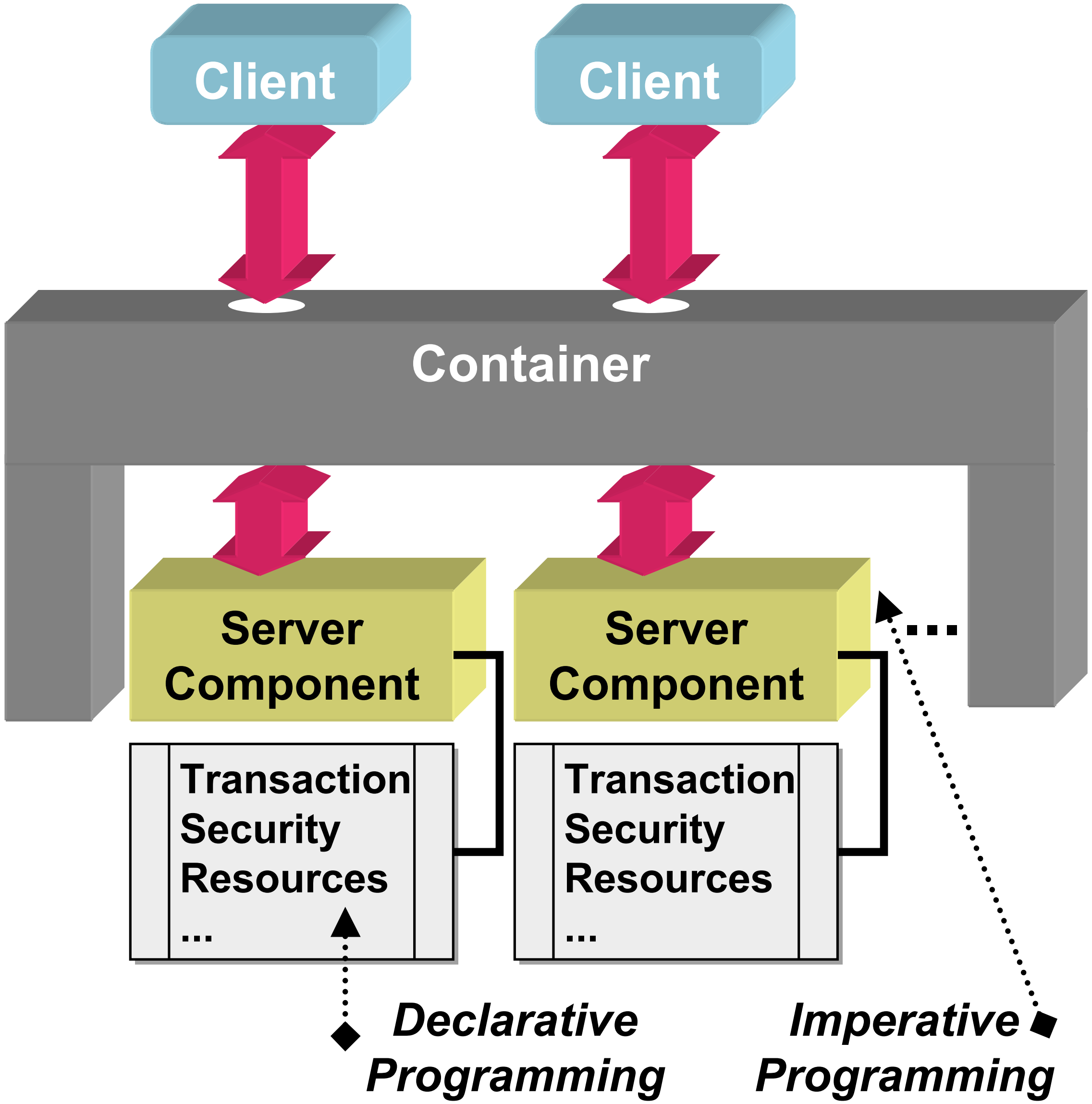
This pattern also has some **liabilities**:

- ***Overhead due to indirection***
  - Clients must incur the overhead of several round-trips to obtain the appropriate object reference
- ***Complexity & cost for development & deployment***
  - There are more steps involved in obtaining object references, which can complicate client programming

Extending Components Transparently

Context

- Component developers may not know *a priori* where their components will execute
- Thus, *containers* are introduced to:
  - Shield clients & components from the details of the underlying middleware, services, network & OS
  - Manage the lifecycle of components & notify components about lifecycle events
    - *e.g.*, activation, passivation, & transaction progress
  - Provide components uniform access to infrastructure services
    - *e.g.*, transactions, security, & persistence
  - Register & deploy components



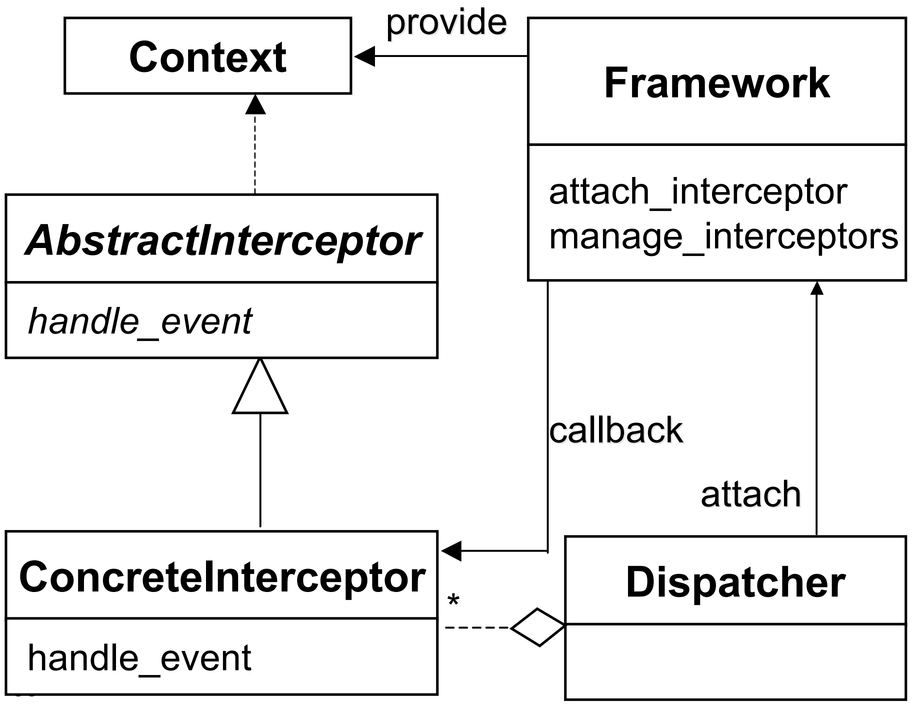
Extending Components Transparently (cont'd)

Problem

- Components should be able to specify *declaratively* in configuration files which execution environment they require
- Containers then should provide the right execution environment
  - *e.g.,* by creating a new transaction or new servant when required

Solution

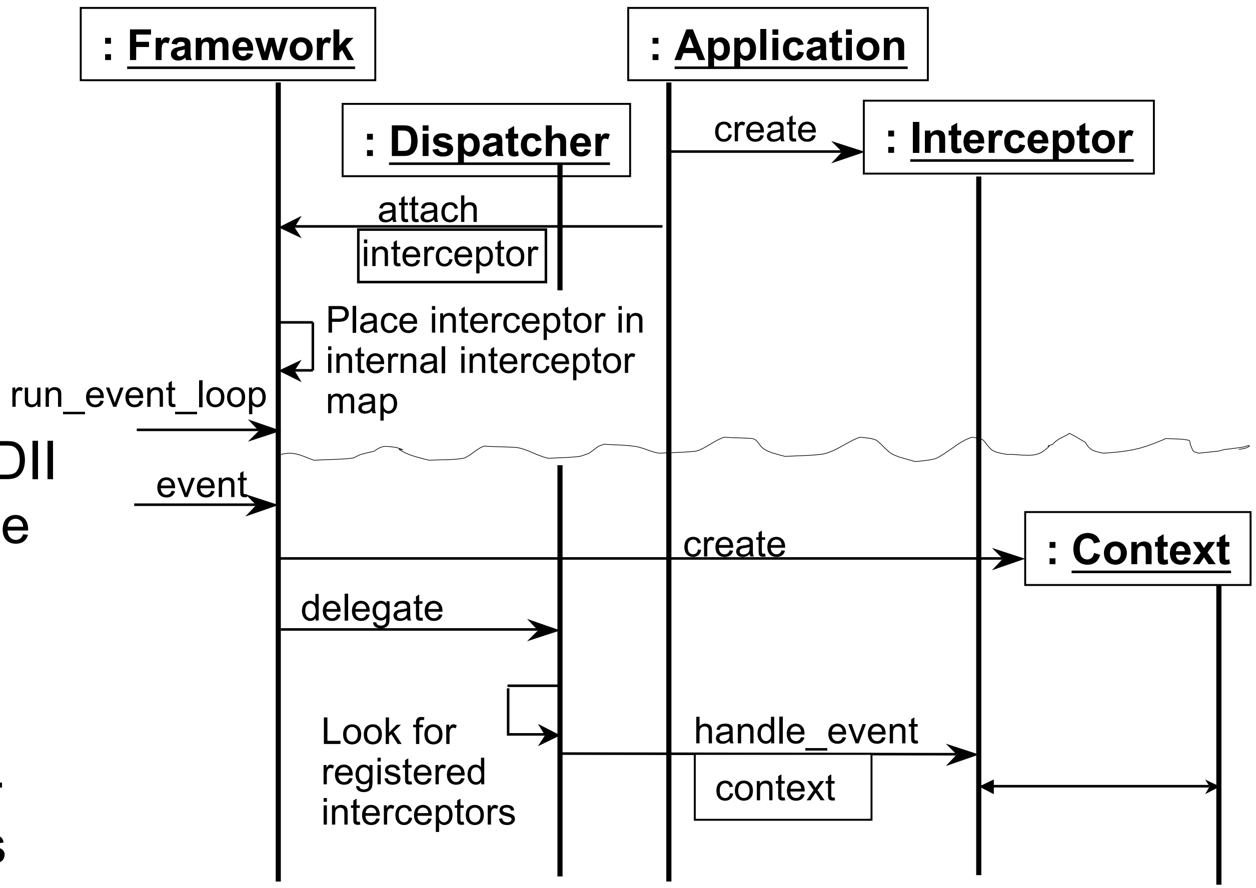
- Apply the *Interceptor* architectural pattern to attach interceptors to a framework that can handle particular events by invoking associated interceptors automatically



- *Framework* represents the concrete framework to which we attach interceptors
- *Concrete Interceptors* implement the event handler for the system-specific events they have subscribed for
- *Context* contains information about the event & allows modification of system behavior after interceptor completion
- The *Dispatcher* allows applications to register & remove interceptors with the framework & to delegate events to interceptors

Interceptor Pattern Dynamics

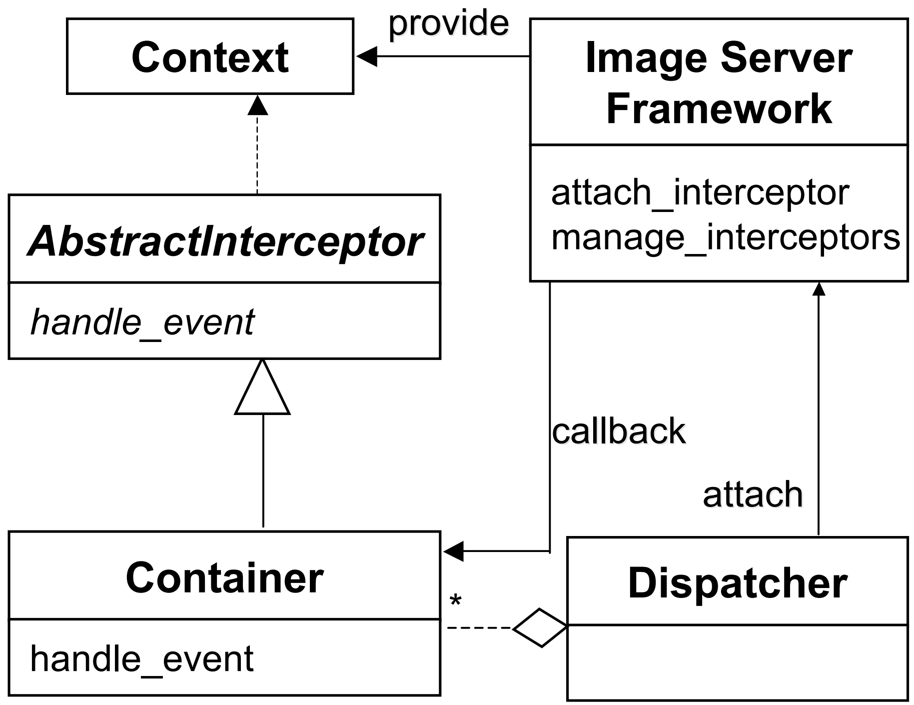
- Interceptor are a "*meta-programming mechanism*"
- Other meta-programming mechanisms include
  - Smart proxies
  - Pluggable protocols
  - Gateways/bridges
  - Interface repositories & DII
- These mechanisms provide building-blocks to handle variation *translucently* & *reflectively*
- More information on meta-programming mechanisms can be found at **`www.cs.wustl.edu/~schmidt/PDF/IEEE.pdf`**

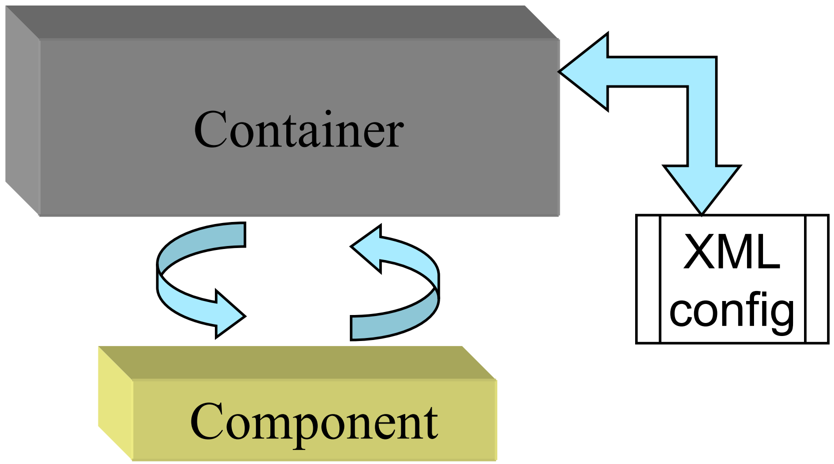


- Interception can also enable performance enhancement strategies
  - *e.g.*, just-in-time activation, object pooling, & caching

Applying the Interceptor Pattern to Image Acquisition

- A container provides generic interfaces to a component that it can use to access container functionality
  - *e.g.*, transaction control, persistence, security,etc.
- A container intercepts all incoming requests from clients
  - It reads the component's requirements from a XML configuration file & does some pre-processing before actually delegating the request to the component
- A component provides event interfaces the container invokes automatically when particular events occur
  - *e.g.*, activation or passivation





Pros & Cons of the Interceptor Pattern

This pattern has five **benefits**:

- ***Extensibility & flexibility***
  - Interceptors allow an application to evolve without breaking existing APIs & components
- ***Separation of concerns***
  - Interceptors decouple the "functional" path from the "meta" path
- ***Support for monitoring & control of frameworks***
  - *e.g.,* generic logging mechanisms can be used to unobtrusively track application behavior
- ***Layer symmetry***
  - Interceptors can perform transformations on a client-side whose inverse are performed on the server-side
- ***Reusability***
  - Interceptors can be reused for various general-purpose behaviors

This pattern also has **liabilities**:

- ***Complex design issues***
  - Determining interceptor APIs & semantics is non-trivial
- ***Malicious or erroneous interceptors***
  - Mis-behaving interceptors can wreak havoc on application stability
- ***Potential interception cascades***
  - Interceptors can result in infinite recursion

Minimizing Resource Utilization

Context

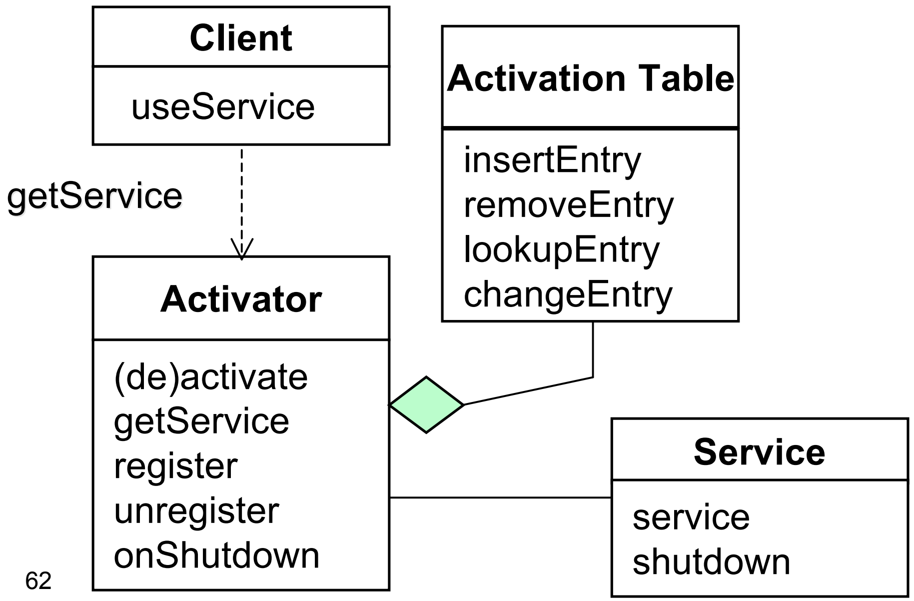
- Image servers are simply one of many services running throughout an distributed electronic medical image system

Problem

- It may not feasible to have all image server implementations running all the time since this ties up end-system resources unnecessarily

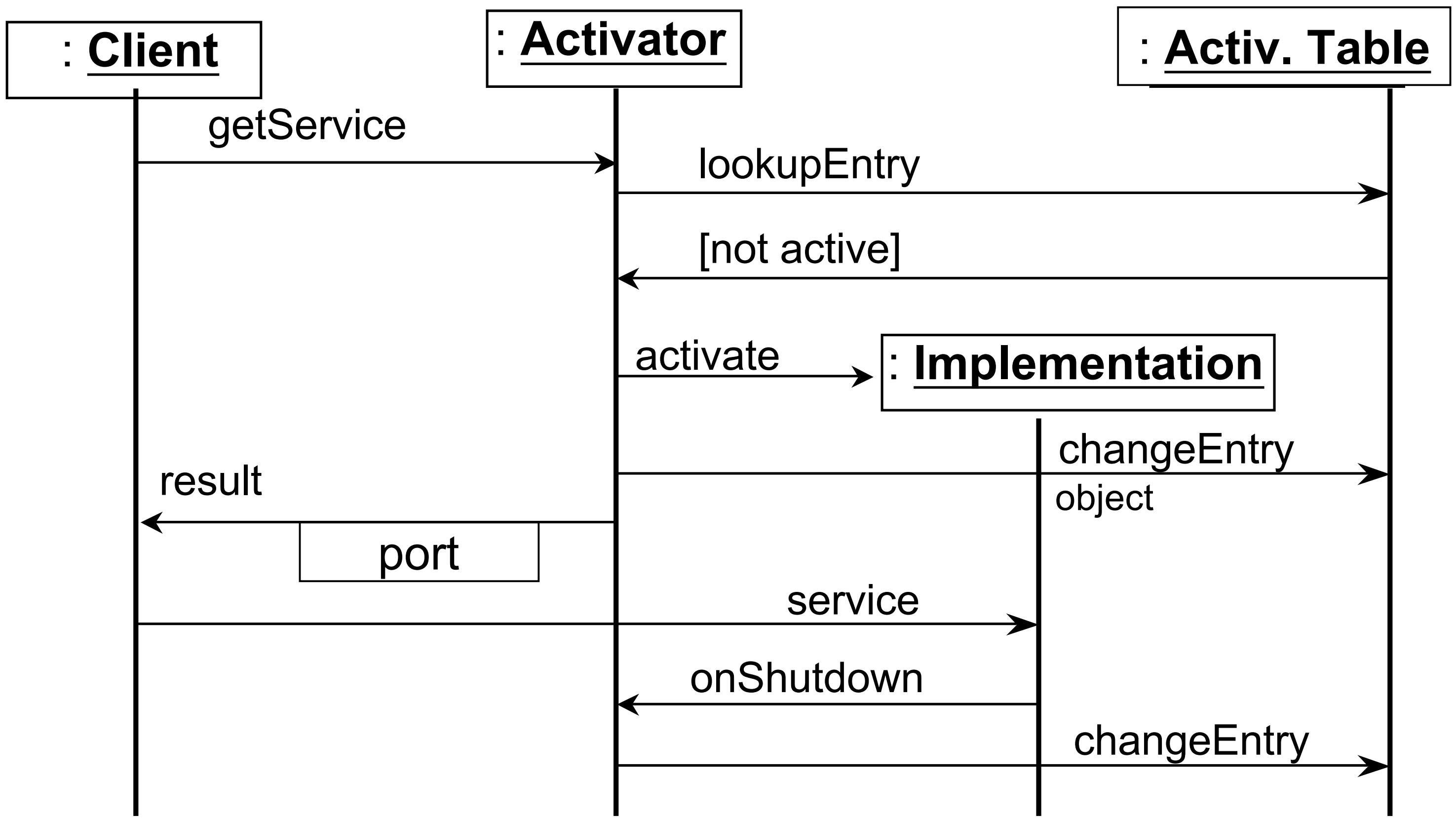
Solution

- Apply the *Activator* pattern to spawn servers on-demand in order to minimize end-system resource utilization



- When incoming requests arrive, the *Activator* looks up whether a target object is already active & if the object is not running it activates the implementation
- The *Activation Table* stores associations between services & their physical location
- The *Client* uses the Activator to get service access
- A *Service* implements a specific type of functionality that it provides to clients

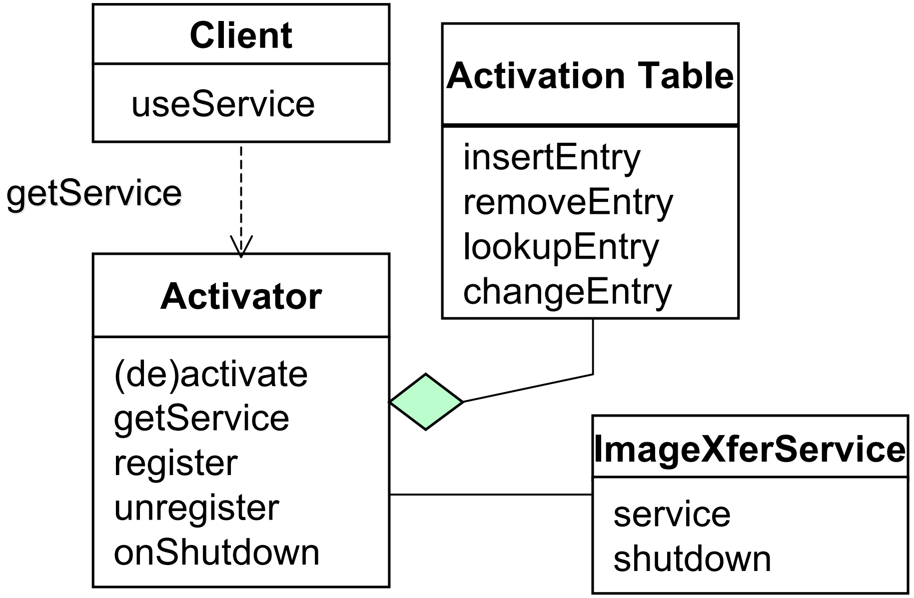
Activator Pattern Dynamics

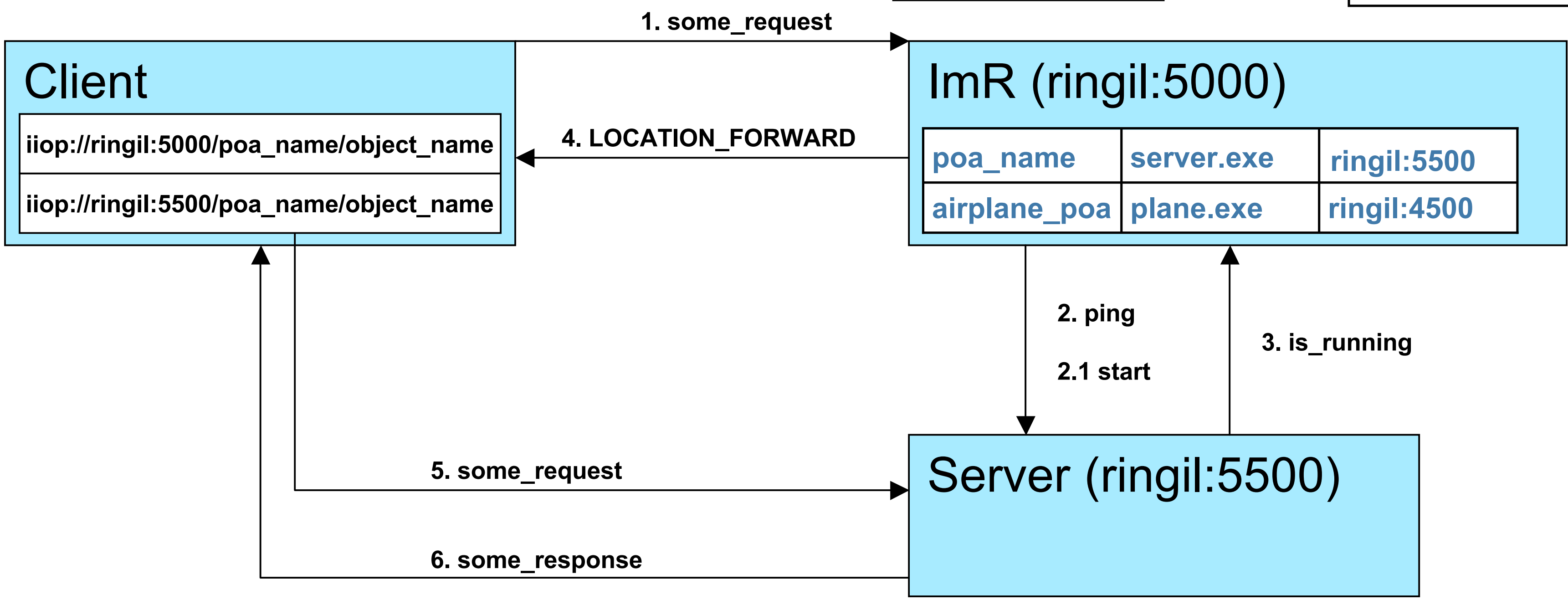


- A container can be used to activate & passivate a component
  - A component can be activated/passivated by itself, the container, after each method call, after each transaction, etc.

Applying the Activator Pattern to Image Acquisition

- We can use the Activator pattern to launch image transfer servers on-demand
- The Activator pattern is available in various COTS technologies:
  - UNIX Inetd "super server"
  - CORBA Implementation Repository





Pros & Cons of the Activator Pattern

This pattern has three **benefits**:

- ***Uniformity***
  - By imposing a uniform activation interface to spawn & control servers
- ***Modularity, testability, & reusability***
  - Application modularity & reusability is improved by decoupling server implementations from the manner in which the servers are activated
- ***More effective resource utilization***
  - Servers can be spawned "on-demand," thereby minimizing resource utilization until clients actually require them

This pattern also has **liabilities**:

- ***Lack of determinism & ordering dependencies***
  - This pattern makes it hard to determine or analyze the behavior of an application until its components are activated at run-time
- ***Reduced security or reliability***
  - An application that uses the Activator pattern may be less secure or reliable than an equivalent statically-configured application
- ***Increased run-time overhead & infrastructure complexity***
  - By adding levels of abstraction & indirection when activating & executing components

Enhancing Server (Re)Configurability

Context

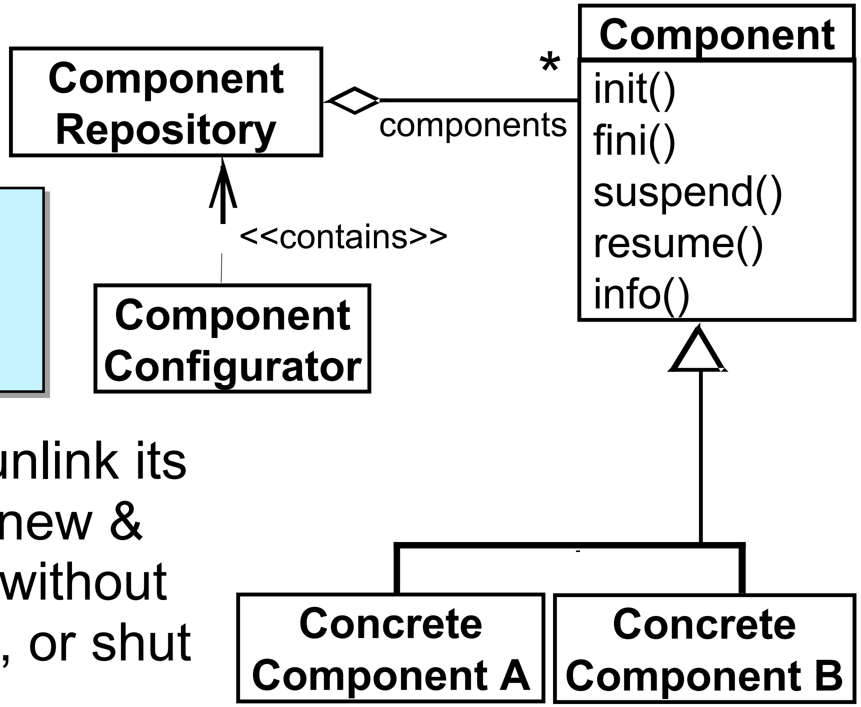
The implementation of certain image server components depends on a variety of factors:

- Certain factors are *static*, such as the number of available CPUs & operating system support for asynchronous I/O
- Other factors are *dynamic*, such as system workload

Problem

Prematurely committing to a particular image server component configuration is inflexible and inefficient:

- No single image server configuration is optimal for all use cases
- Certain design decisions cannot be made efficiently until run-time



Solution

- Apply the *Component Configurator* design pattern to enhance server configurability

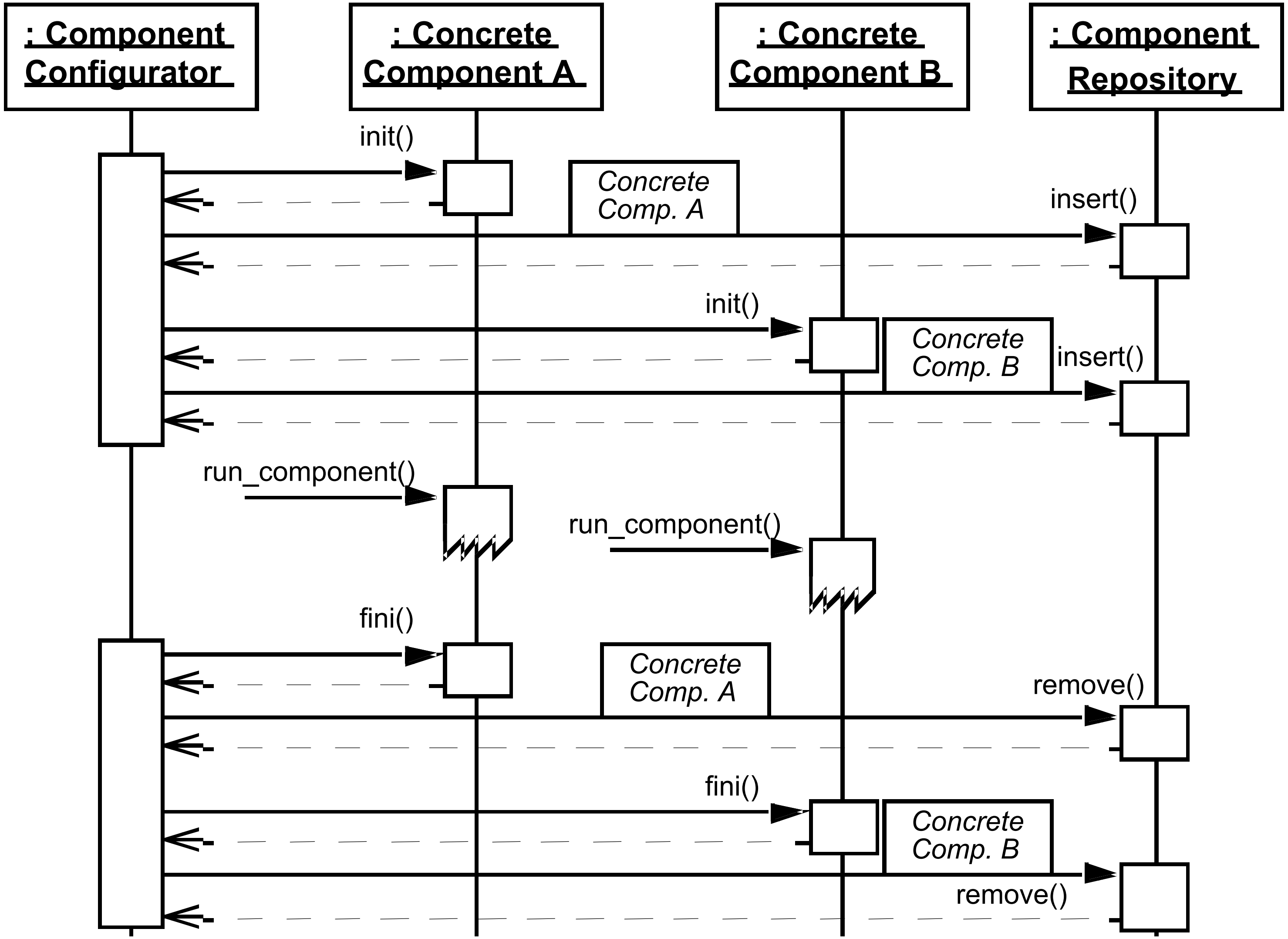
This pattern allows an application to link & unlink its component implementations at run-time so new & enhanced services can therefore be added without having to modify, recompile, statically relink, or shut down & restart a running application

Component Configurator Pattern Dynamics

1.Component initialization

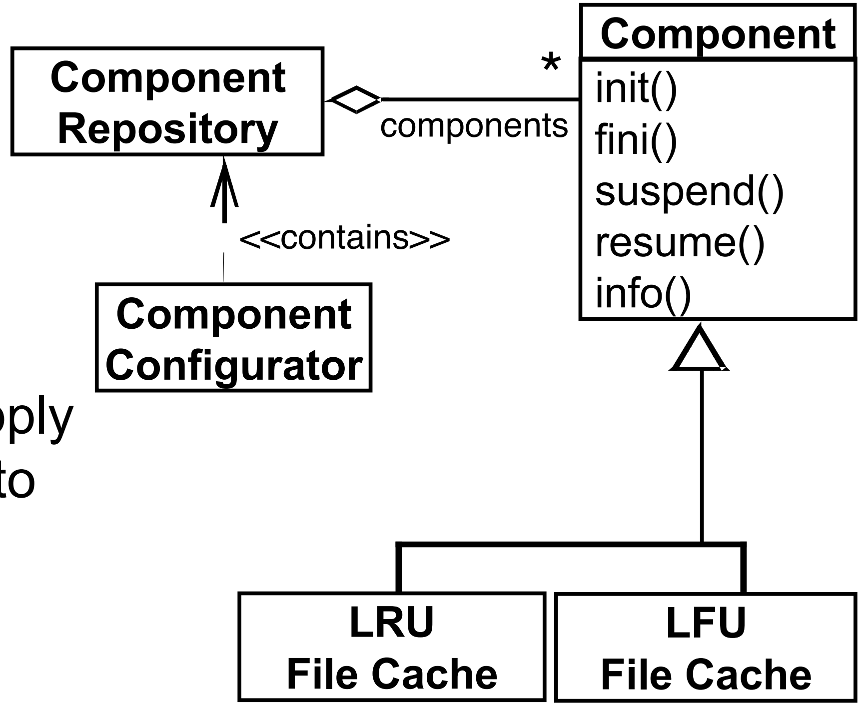
2.Component processing

3.Component termination



Applying the Component Configurator Pattern to Image Acquisition

Image servers can use the Component Configurator pattern to dynamically optimize, control, & reconfigure the behavior of its components at installation-time or during run-time



- For example, an image server can apply the Component Configurator pattern to configure various *Cached Virtual Filesystem* strategies
  - *e.g.*, least-recently used (LRU) or least-frequently used (LFU)

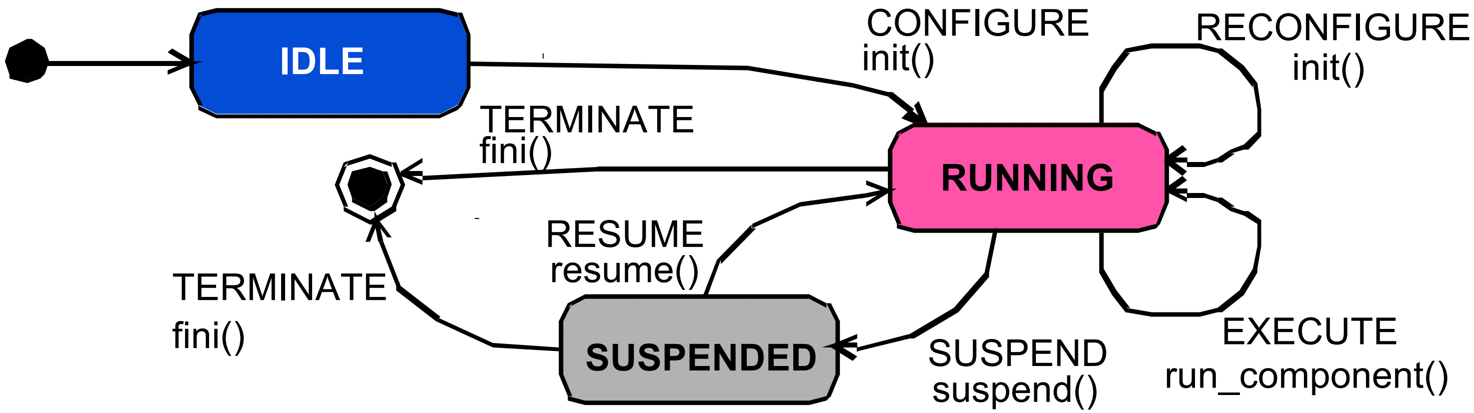
Concrete components can be packaged into a suitable unit of configuration, such as a dynamically linked library (DLL)

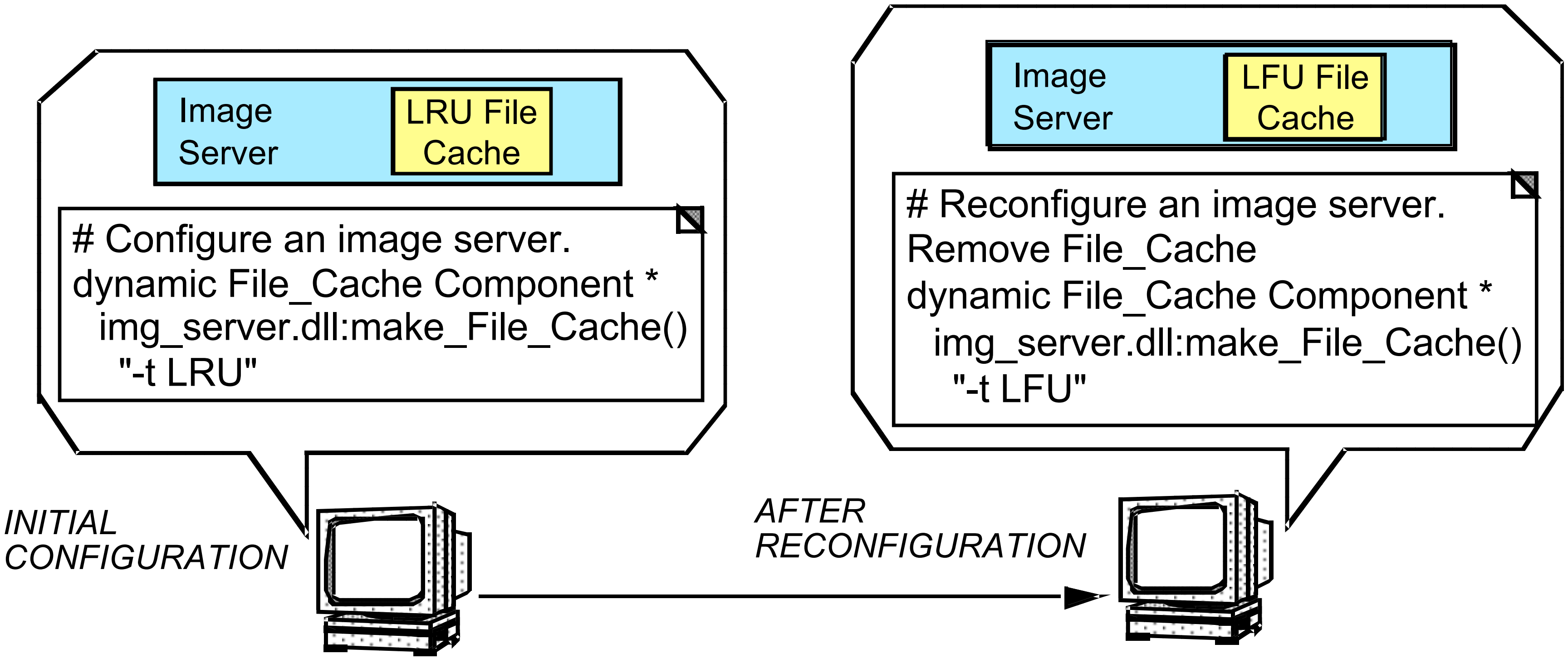
Only the components that are currently in use need to be configured into an image server

Reconfiguring an Image Server

Image servers can also be reconfigured dynamically to support new components & new component implementations

Reconfiguration State Chart





Pros and Cons of the Component Configurator Pattern

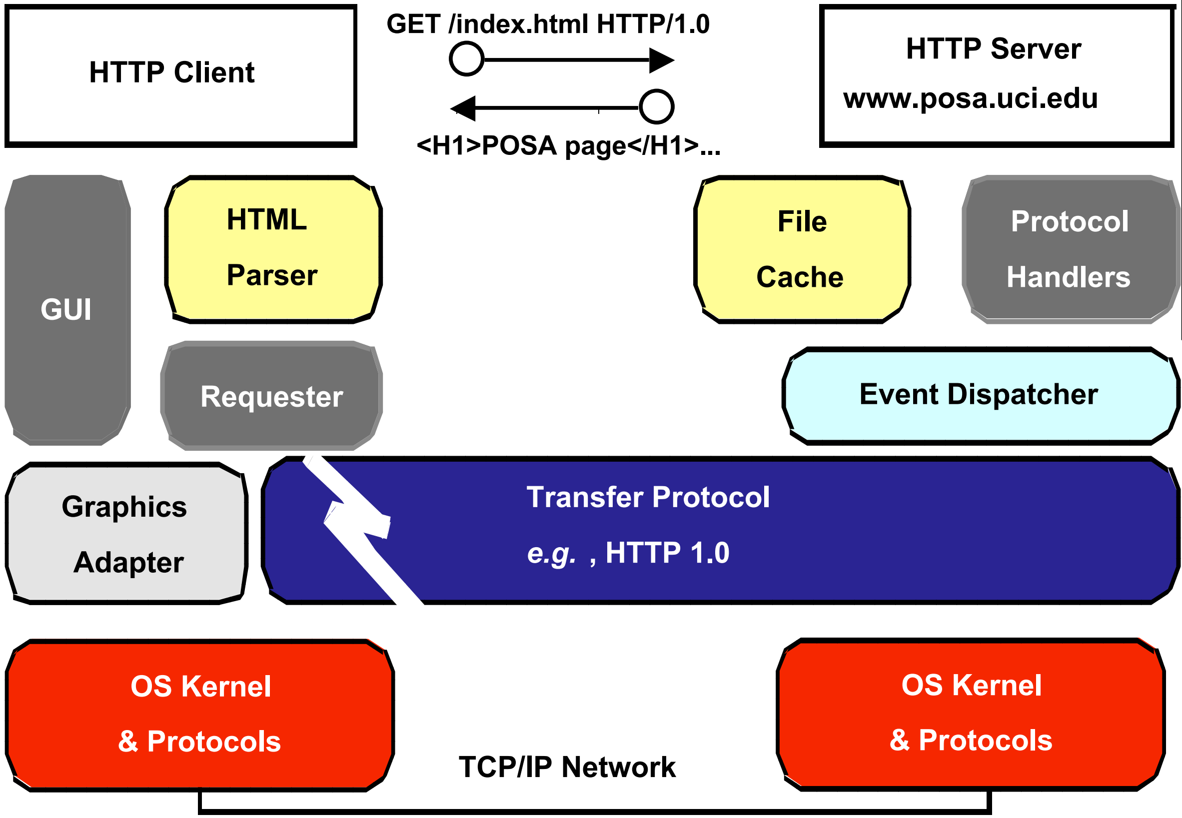
This pattern offers four **benefits**:

- ***Uniformity***
  - By imposing a uniform configuration & control interface to manage components
- ***Centralized administration***
  - By grouping one or more components into a single administrative unit that simplifies development by centralizing common component initialization & termination activities
- ***Modularity, testability, & reusability***
  - Application modularity & reusability is improved by decoupling component implementations from the manner in which the components are configured into processes
- ***Configuration dynamism & control***
  - By enabling a component to be dynamically reconfigured without modifying, recompiling, statically relinking existing code & without restarting the component or other active components with which it is collocated

This pattern also incurs **liabilities**:

- ***Lack of determinism & ordering dependencies***
  - This pattern makes it hard to determine or analyze the behavior of an application until its components are configured at run-time
- ***Reduced security or reliability***
  - An application that uses the Component Configurator pattern may be less secure or reliable than an equivalent statically-configured application
- ***Increased run-time overhead & infrastructure complexity***
  - By adding levels of abstraction & indirection when executing components
- ***Overly narrow common interfaces***
  - The initialization or termination of a component may be too complicated or too tightly coupled with its context to be performed in a uniform manner

Tutorial Example 2:
High-performance Content Delivery Servers

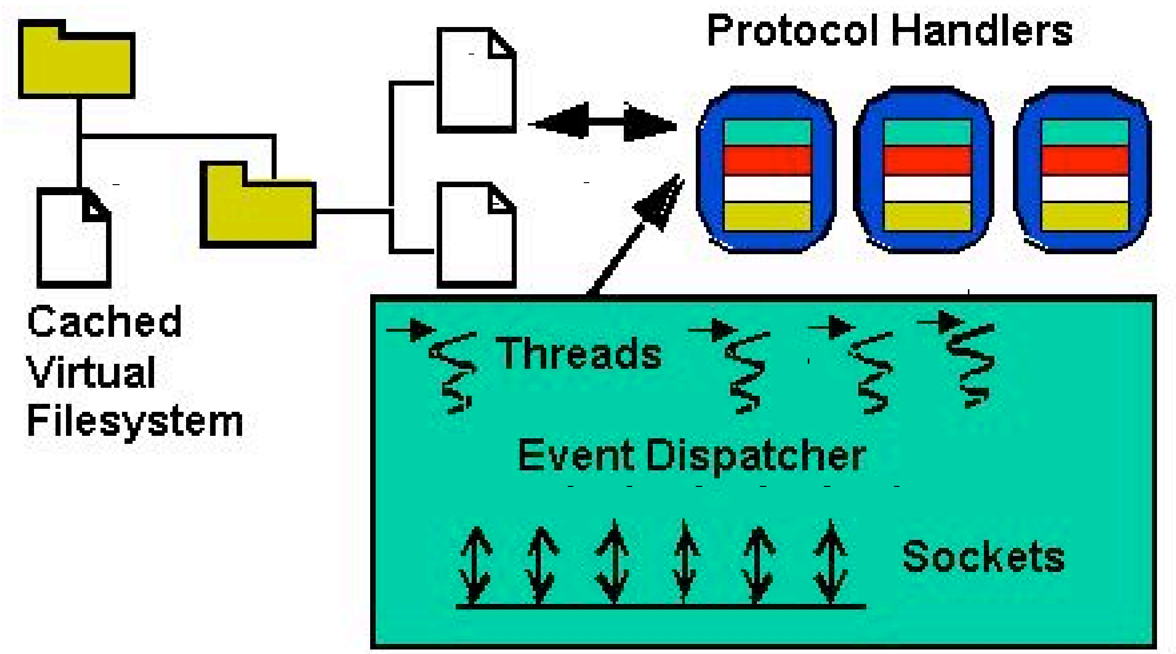


Goal
- Download content scalably & efficiently
  - *e.g.,* images & other multi-media content types

Key System Characteristics
- Robust implementation
  - *e.g.,* stop malicious clients
- Extensible to other protocols
  - *e.g.,* HTTP 1.1, IIOP, DICOM
- Leverage advanced multi-processor hardware & software

Key Solution Characteristics
- Support many content delivery server design alternatives seamlessly
  - *e.g.,* different concurrency & event models
- Design is guided by patterns to leverage time-proven solutions
- Implementation is based on ACE framework components to reduce effort & amortize prior effort
- Open-source to control costs & to leverage technology advances

JAWS Content Server Framework



Key Sources of Variation

- Concurrency models
  - *e.g.,*thread pool vs. thread-per request
- Event demultiplexing models
  - *e.g.,*sync vs. async
- File caching models
  - *e.g.,*LRU vs. LFU
- Content delivery protocols
  - *e.g.,*HTTP 1.0+1.1, HTTP-NG, IIOP, DICOM

Event Dispatcher

- Accepts client connection request events, receives HTTP GET requests, & coordinates JAWS's event demultiplexing strategy with its concurrency strategy.
  - As events are processed they are dispatched to the appropriate Protocol Handler.

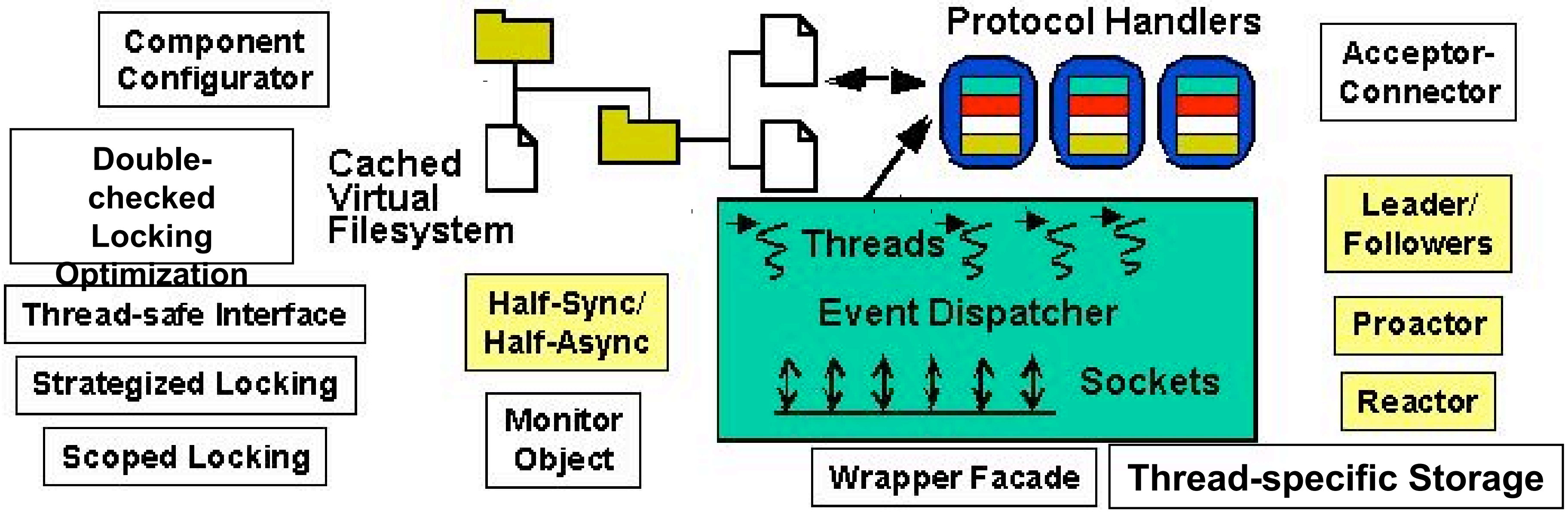
Protocol Handler

- Performs parsing & protocol processing of HTTP request events.
  - JAWS Protocol Handler design allows multiple Web protocols, such as HTTP/1.0, HTTP/1.1, & HTTP-NG, to be incorporated into a Web server.
  - To add a new protocol, developers just write a new Protocol Handler component & configure it into the JAWS framework.

Cached Virtual Filesystem

- Improves Web server performance by reducing the overhead of file system accesses when processing HTTP GET requests.
  - Various caching strategies, such as least-recently used (LRU) or least-frequently used (LFU), can be selected according to the actual or anticipated workload & configured statically or dynamically.

Applying Patterns to Resolve Key JAWS Design Challenges



Patterns help resolve the following common design challenges:

- Encapsulating low-level OS APIs
- Decoupling event demultiplexing & connection management from protocol processing
- Scaling up performance via threading
- Implementing a synchronized request queue
- Minimizing server threading overhead
- Using asynchronous I/O effectively
- Efficiently demuxing asynchronous operations & completions
- Transparently parameterizing synchronization into components
- Ensuring locks are released properly
- Minimizing unnecessary locking
- Synchronizing singletons correctly
- Logging access statistics efficiently

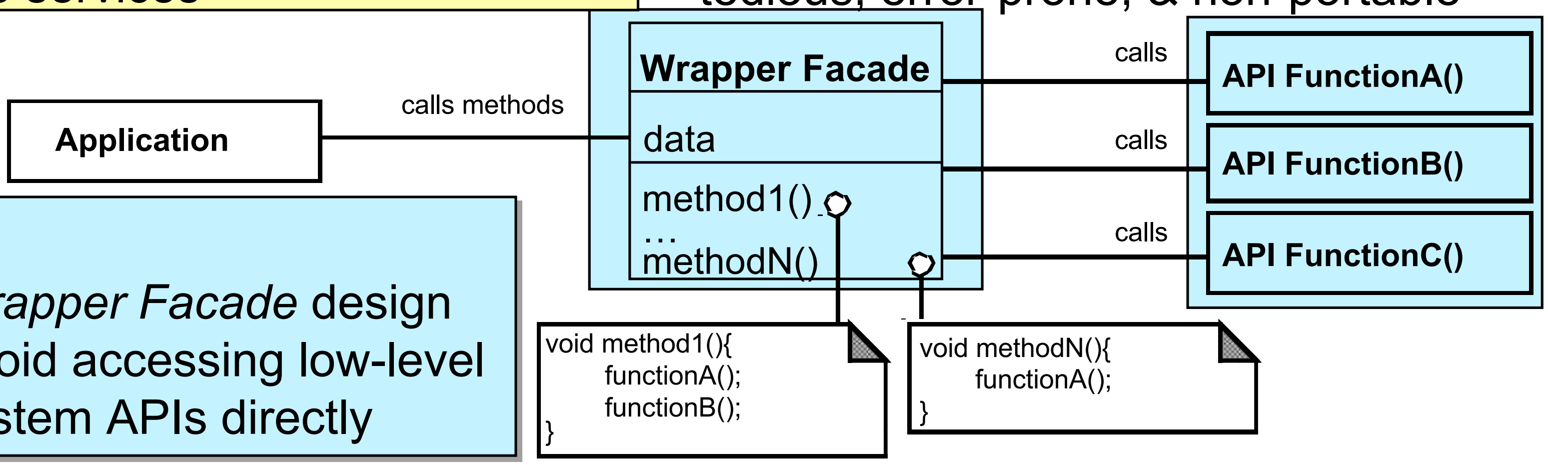
Encapsulating Low-level OS APIs

Context

- A Web server must manage a variety of OS services, including processes, threads, Socket connections, virtual memory, & files. Most operating systems provide low-level APIs written in C to access these services

Problem

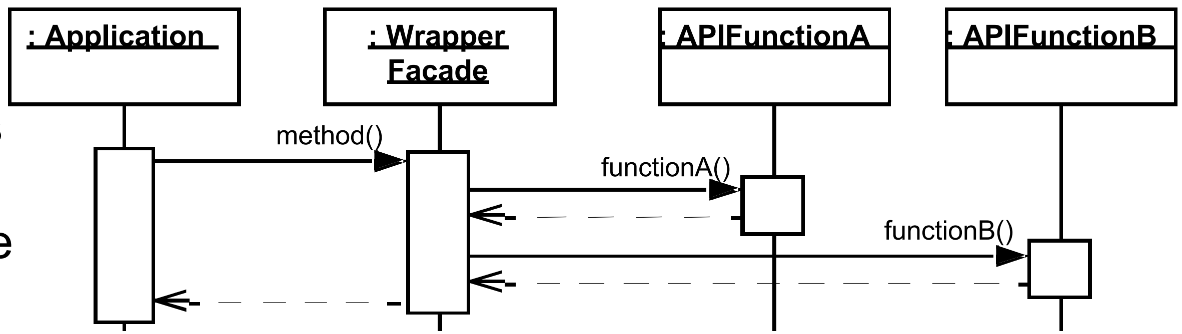
- The diversity of hardware & operating systems makes it hard to build portable & robust Web server software by programming directly to low-level operating system APIs, which are tedious, error-prone, & non-portable



Solution

- Apply the *Wrapper Facade* design pattern to avoid accessing low-level operating system APIs directly

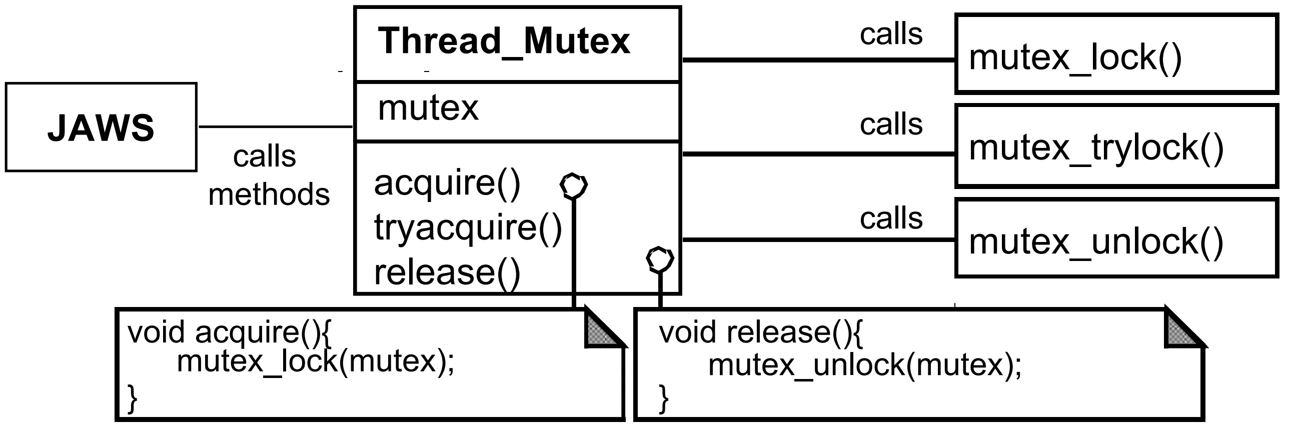
The Wrapper Facade pattern encapsulates data & functions provided by existing non-OO APIs within more concise, robust, portable, maintainable, & cohesive OO class interfaces



Applying the Wrapper Façade Pattern in JAWS

JAWS uses the wrapper facades defined by ACE to ensure its framework components can run on many operating systems, including Windows, UNIX, & many real-time operating systems

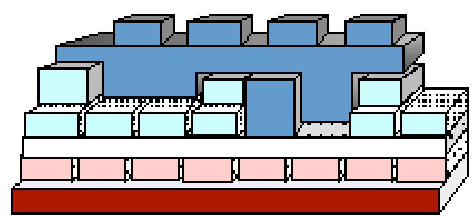
For example, JAWS uses the **`Thread_Mutex`** wrapper facade in ACE to provide a portable interface to operating system mutual exclusion mechanisms



The `Thread_Mutex` wrapper in the diagram is implemented using the Solaris thread API

The ACE `Thread_Mutex` wrapper facade is also available for other threading APIs, *e.g.*, pSoS, VxWorks, Win32 threads or POSIX Pthreads

Other ACE wrapper facades used in JAWS encapsulate Sockets, process & thread management, memory-mapped files, explicit dynamic linking, & time operations



www.cs.wustl.edu/~schmidt/ACE/

Pros and Cons of the Wrapper Façade Pattern

This pattern provides three **benefits*:***

- ***Concise, cohesive, & robust higher-level object-oriented programming interfaces***
  - These interfaces reduce the tedium & increase the type-safety of developing applications, which descreases certain types of programming errors
- ***Portability & maintainability***
  - Wrapper facades can shield application developers from non-portable aspects of lower-level APIs
- ***Modularity, reusability & configurability***
  - This pattern creates cohesive & reusable class components that can be 'plugged' into other components in a wholesale fashion, using object-oriented language features like inheritance & parameterized types

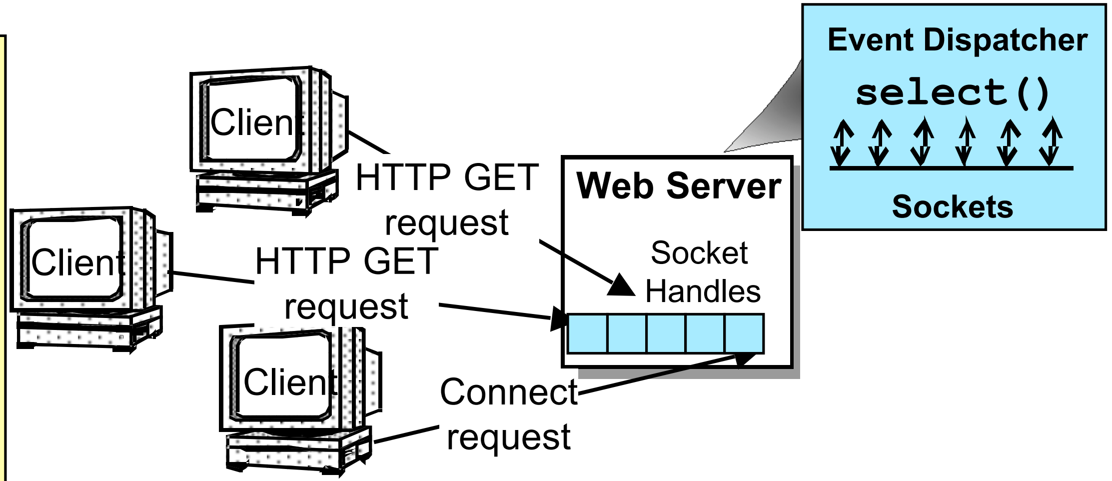
This pattern can incur **liabilities*:***

- ***Loss of functionality***
  - Whenever an abstraction is layered on top of an existing abstraction it is possible to lose functionality
- ***Performance degradation***
  - This pattern can degrade performance if several forwarding function calls are made per method
- ***Programming language & compiler limitations***
  - It may be hard to define wrapper facades for certain languages due to a lack of language support or limitations with compilers

Decoupling Event Demuxing & Connection Management from Protocol Processing

Context

- Web servers can be accessed simultaneously by multiple clients
- They must demux & process multiple types of indication events arriving from clients concurrently
- A common way to demux events in a server is to use `select()`



- Thus, changes to event-demuxing & connection code affects the server protocol code directly & may yield subtle bugs
  - *e.g.*, porting it to use TLI or `WaitForMultipleObjects()`

Problem

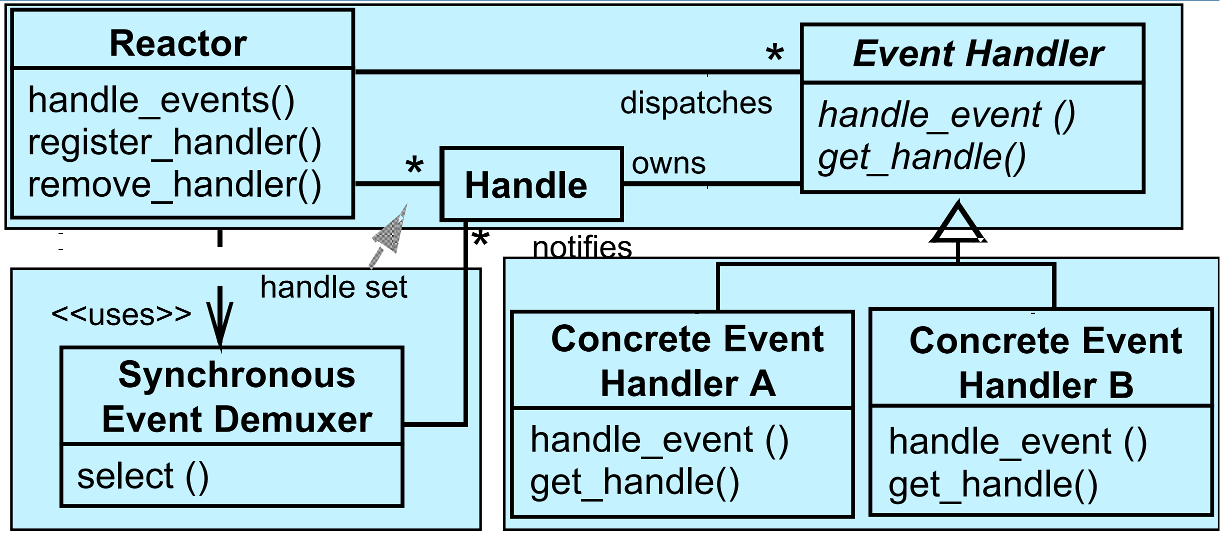
- Developers often couple event-demuxing & connection code with protocol-handling code
- This code cannot then be reused directly by other protocols or by other middleware & applications

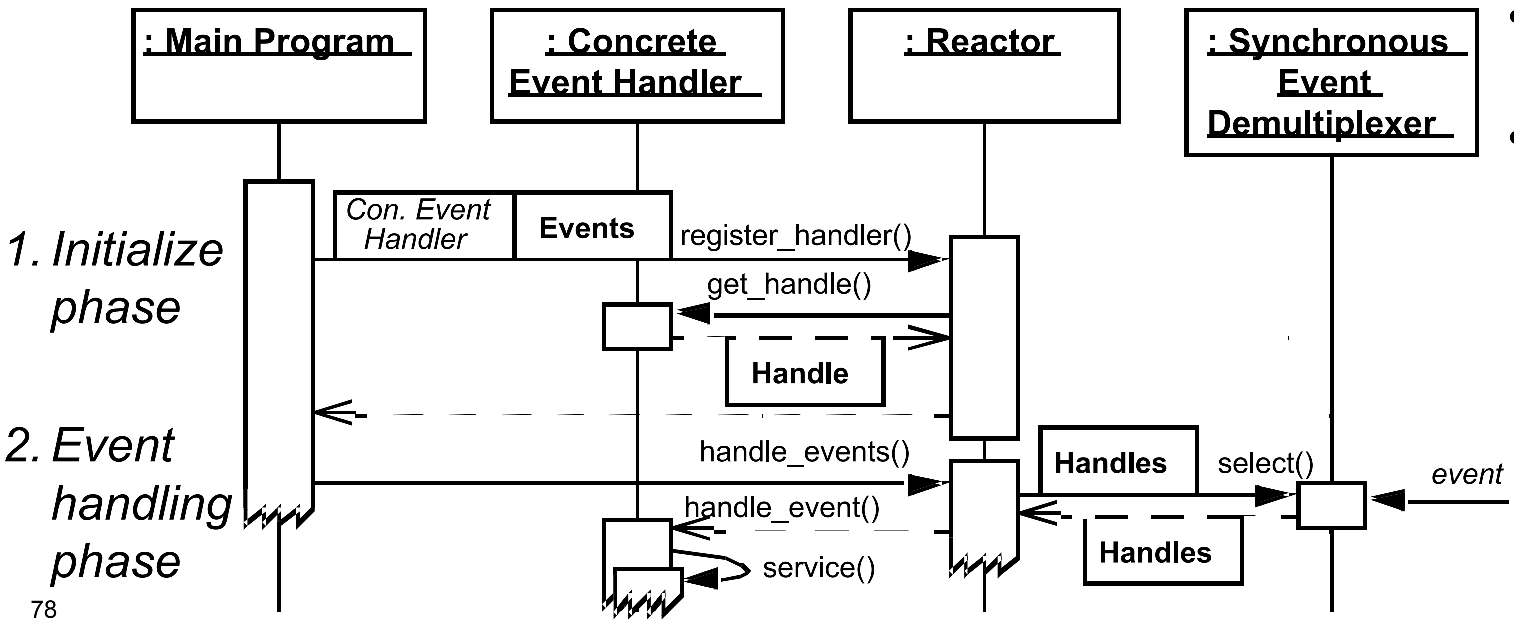
Solution

Apply the *Reactor* architectural pattern & the *Acceptor-Connector* design pattern to separate the generic event-demultiplexing & connection-management code from the web server's protocol code

The Reactor Pattern

The *Reactor* architectural pattern allows event-driven applications to demultiplex & dispatch service requests that are delivered to an application from one or more clients.



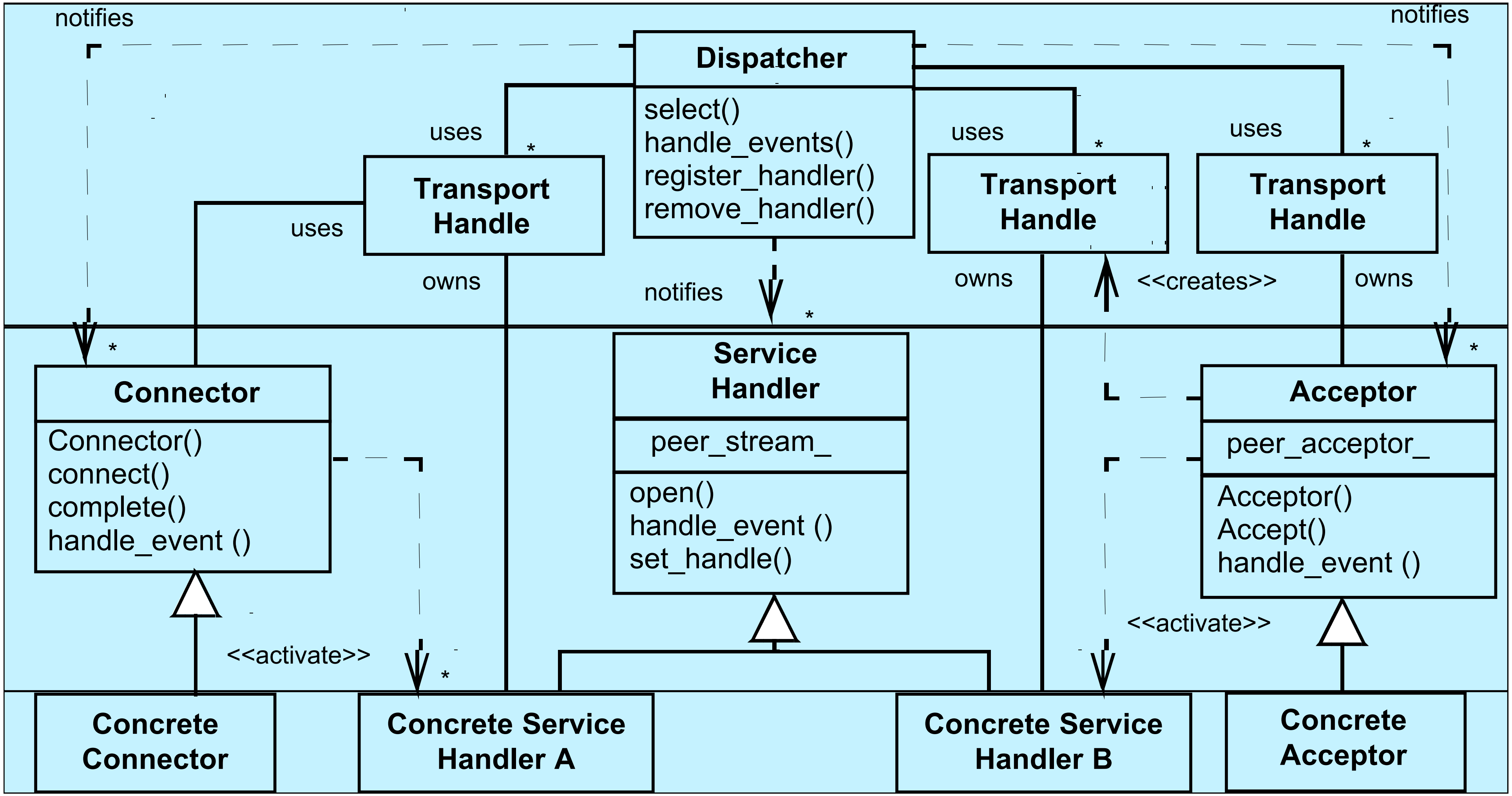


Observations

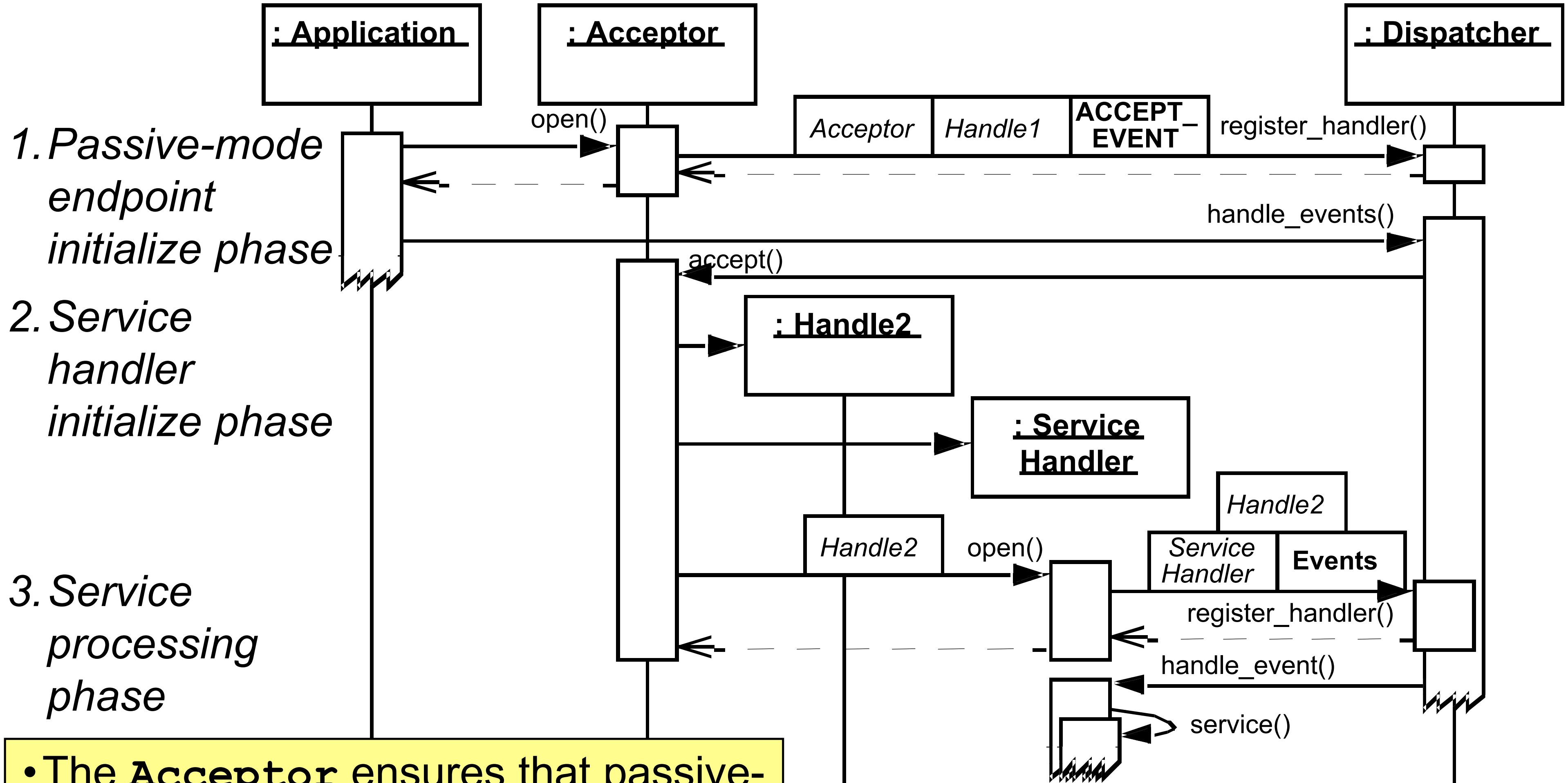
- Note inversion of control
- Also note how long-running event handlers can degrade the QoS since callbacks steal the reactor's thread!

The Acceptor-Connector Pattern

The *Acceptor-Connector* design pattern decouples the connection & initialization of cooperating peer services in a networked system from the processing performed by the peer services after being connected & initialized.



Acceptor Dynamics



- The **`Acceptor`** ensures that passive-mode transport endpoints aren't used to read/write data accidentally
  - And vice versa for data transport endpoints…

- There is typically one **`Acceptor`** factory per-service/per-port
  - Additional demuxing can be done at higher layers, *a la* CORBA

Synchronous Connector Dynamics

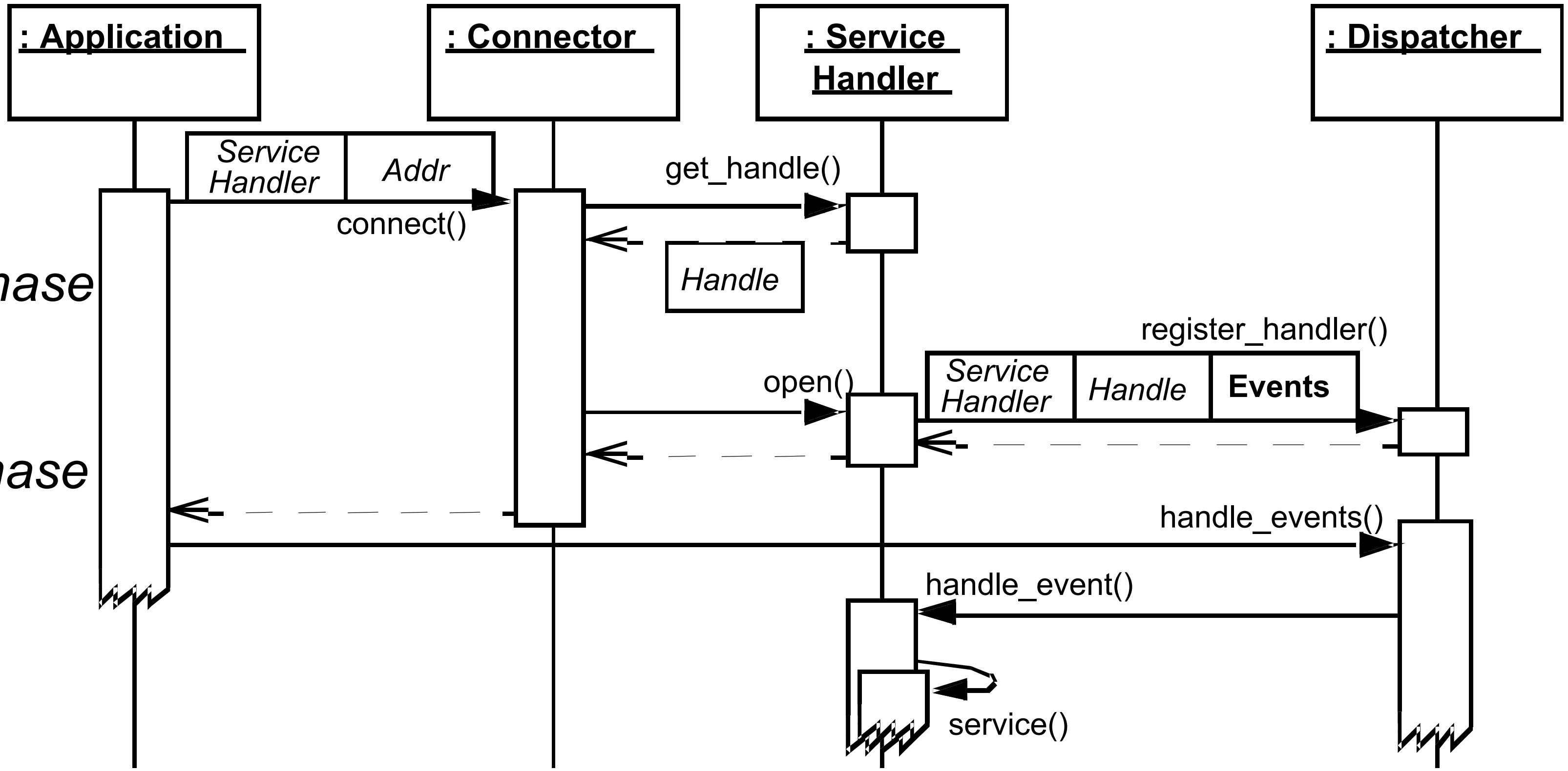
Motivation for Synchrony

- If connection latency is negligible
  - *e.g.*, connecting with a server on the same host via a 'loopback' device

- If multiple threads of control are available & it is efficient to use a thread-per-connection to connect each service handler synchronously

- If the services must be initialized in a fixed order & the client can't perform useful work until all connections are established

1. *Sync connection initiation phase*
2. *Service handler initialize phase*
3. *Service processing phase*



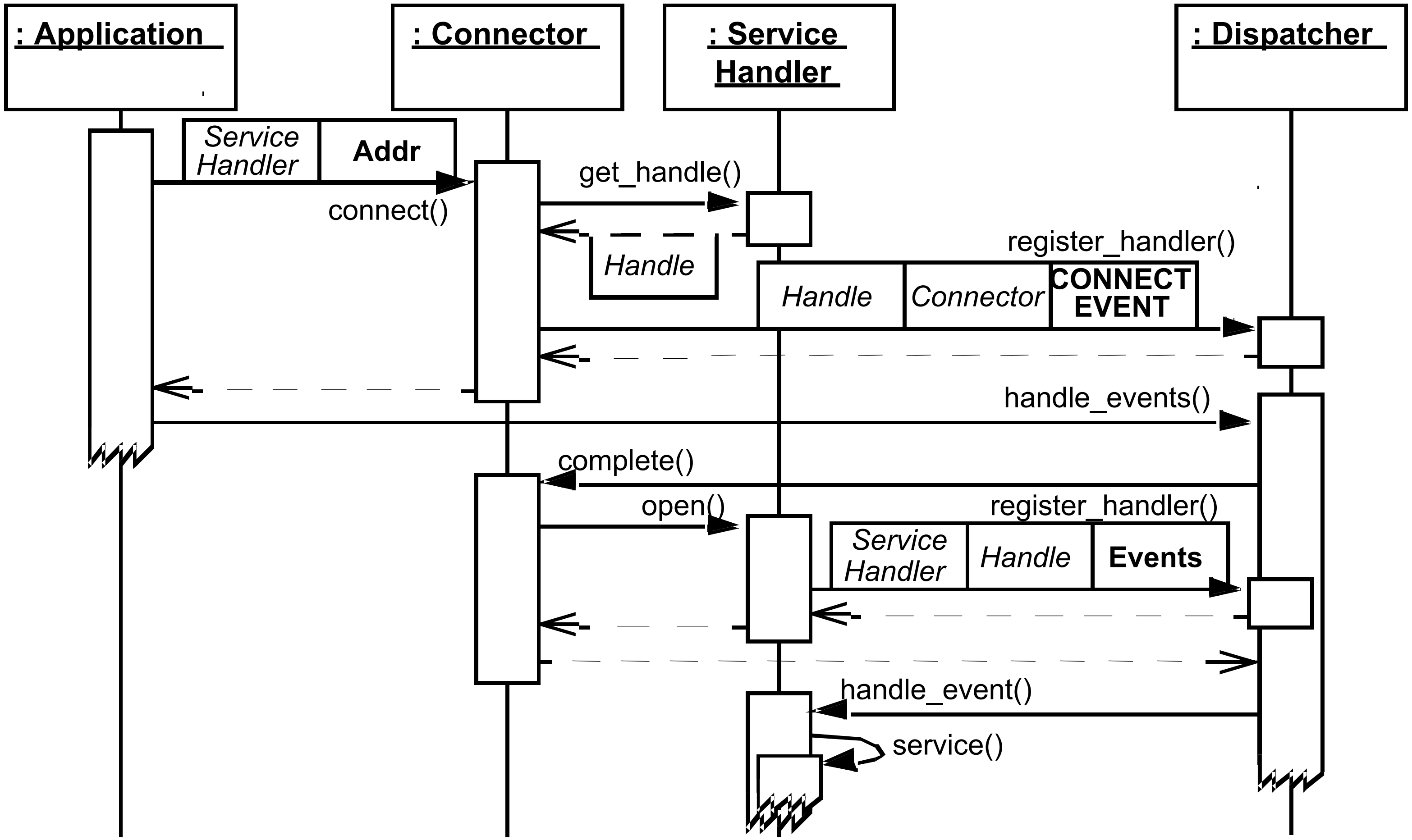
Asynchronous Connector Dynamics

Motivation for Asynchrony

- If client is establishing connections over high latency links

- If client is a single-threaded applications

- If client is initializing many peers that can be connected in an arbitrary order

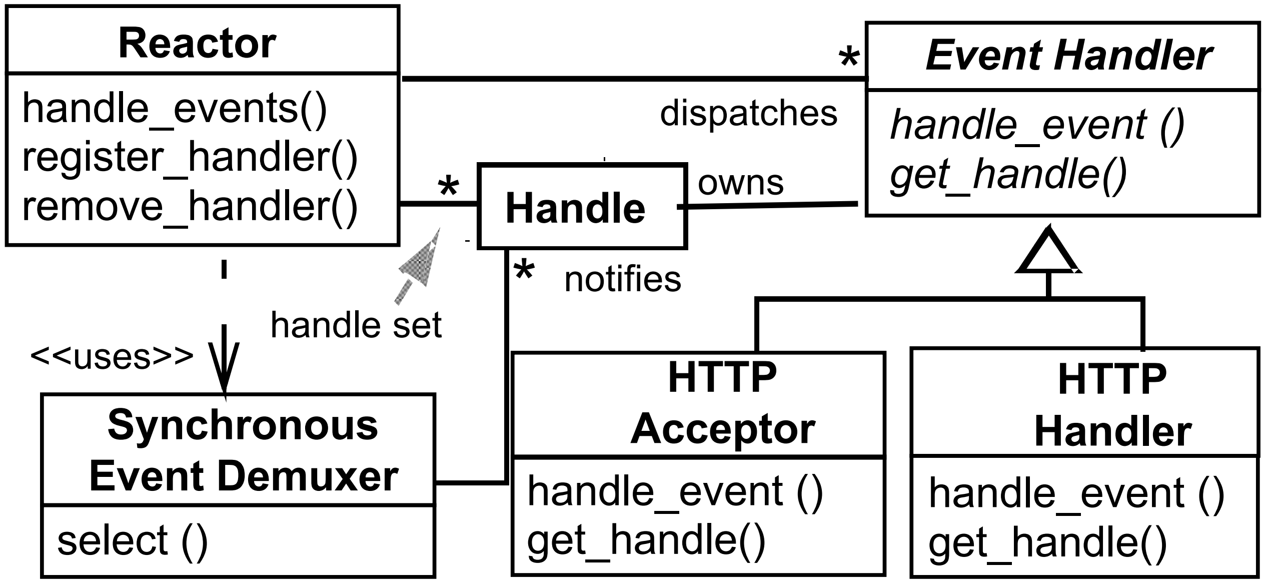


1. *Async connection initiation phase*
2. *Service handler initialize phase*
3. *Service processing phase*

Applying the Reactor and Acceptor-Connector Patterns in JAWS

The Reactor architectural pattern decouples:

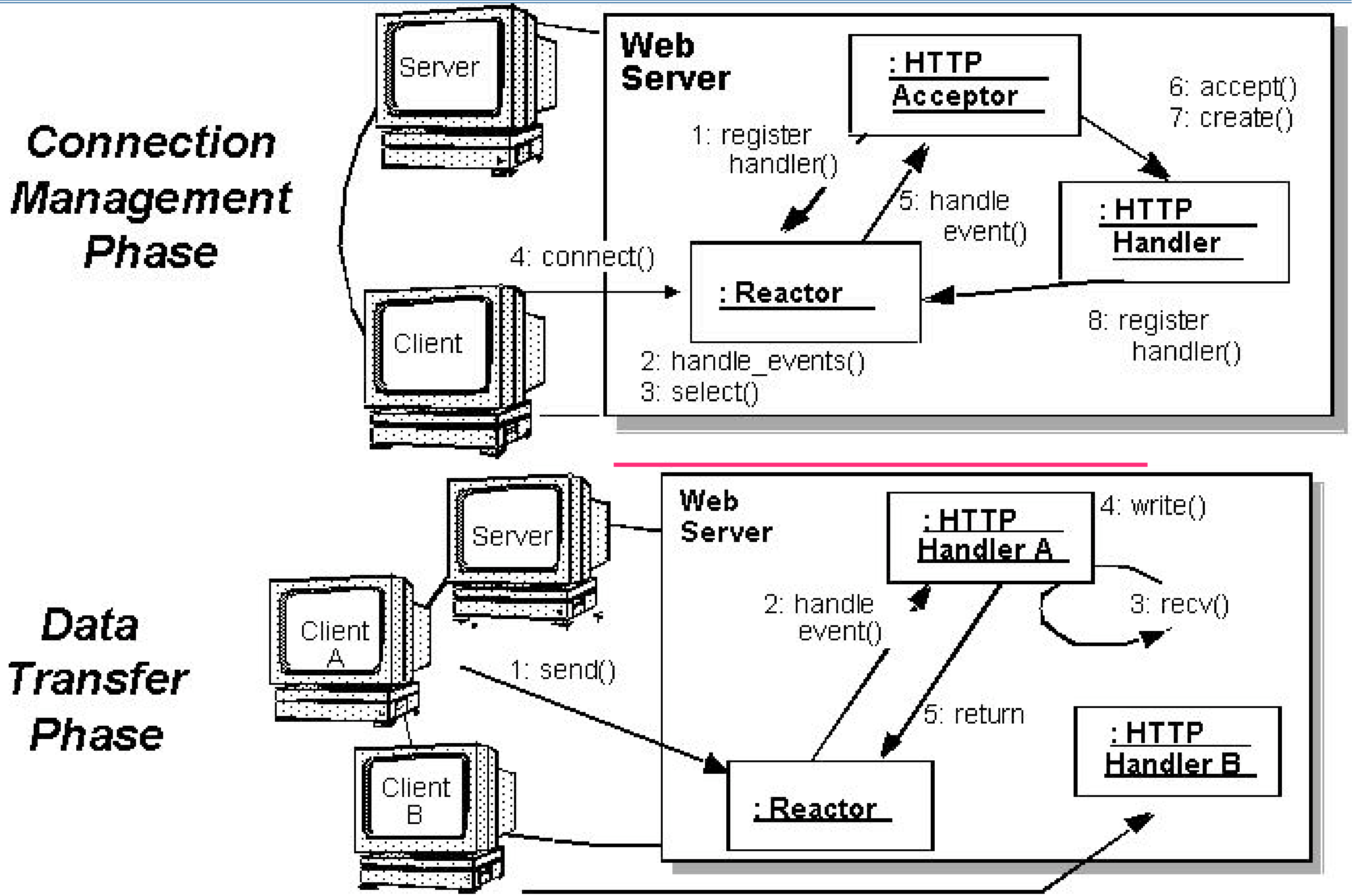
1. JAWS generic synchronous event demultiplexing & dispatching logic from
2. The HTTP protocol processing it performs in response to events



The Acceptor-Connector design pattern can use a Reactor as its *Dispatcher* in order to help decouple:

1. The connection & initialization of peer client & server HTTP services from
2. The processing activities performed by these peer services once they are connected & initialized

Reactive Connection Management & Data Transfer in JAWS



Pros and Cons of the Reactor Pattern

This pattern offers four **benefits**:

- ***Separation of concerns***
  - This pattern decouples application-independent demuxing & dispatching mechanisms from application-specific hook method functionality
- ***Modularity, reusability, & configurability***
  - This pattern separates event-driven application functionality into several components, which enables the configuration of event handler components that are loosely integrated via a reactor
- ***Portability***
  - By decoupling the reactor's interface from the lower-level OS synchronous event demuxing functions used in its implementation, the Reactor pattern improves portability
- ***Coarse-grained concurrency control***
  - This pattern serializes the invocation of event handlers at the level of event demuxing & dispatching within an application process or thread

This pattern can incur **liabilities*:***

- ***Restricted applicability***
  - This pattern can be applied efficiently only if the OS supports synchronous event demuxing on handle sets
- ***Non-pre-emptive***
  - In a single-threaded application, concrete event handlers that borrow the thread of their reactor can run to completion & prevent the reactor from dispatching other event handlers
- ***Complexity of debugging & testing***
  - It is hard to debug applications structured using this pattern due to its inverted flow of control, which oscillates between the framework infrastructure & the method call-backs on application-specific event handlers

Pros and Cons of the Acceptor-Connector Pattern

This pattern provides three **benefits**:

- ***Reusability, portability, & extensibility***
  - This pattern decouples mechanisms for connecting & initializing service handlers from the service processing performed after service handlers are connected & initialized
- ***Robustness***
  - This pattern strongly decouples the service handler from the acceptor, which ensures that a passive-mode transport endpoint can't be used to read or write data accidentally
- ***Efficiency***
  - This pattern can establish connections actively with many hosts asynchronously & efficiently over long-latency wide area networks
  - Asynchrony is important in this situation because a large networked system may have hundreds or thousands of host that must be connected

This pattern also has **liabilities:**

- ***Additional indirection***
  - The Acceptor-Connector pattern can incur additional indirection compared to using the underlying network programming interfaces directly
- ***Additional complexity***
  - The Acceptor-Connector pattern may add unnecessary complexity for simple client applications that connect with only one server & perform one service using a single network programming interface

Scaling Up Performance via Threading

Context

- HTTP runs over TCP, which uses flow control to ensure that senders do not produce data more rapidly than slow receivers or congested networks can buffer and process
- Since achieving efficient end-to-end *quality of service* (QoS) is important to handle heavy Web traffic loads, a Web server must scale up efficiently as its number of clients increases

Problem

- Processing all HTTP GET requests reactively within a single-threaded process does not scale up, because each server CPU time-slice spends much of its time blocked waiting for I/O operations to complete
- Similarly, to improve QoS for all its connected clients, an entire Web server process must not block while waiting for connection flow control to abate so it can finish sending a file to a client

Solution

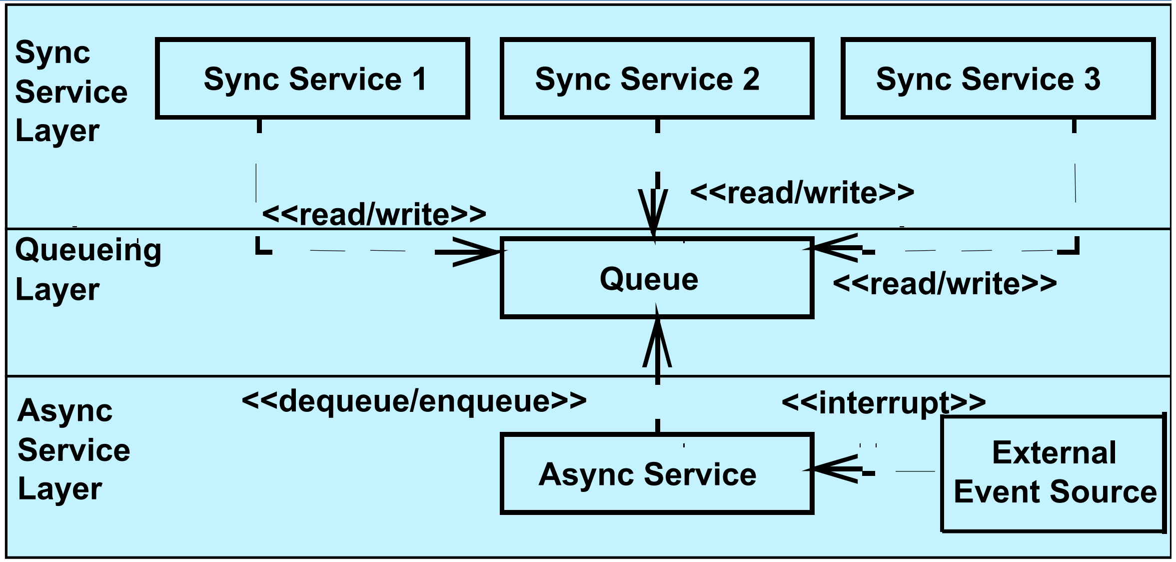
- Apply the *Half-Sync/Half-Async* architectural pattern to scale up server performance by processing different HTTP requests concurrently in multiple threads

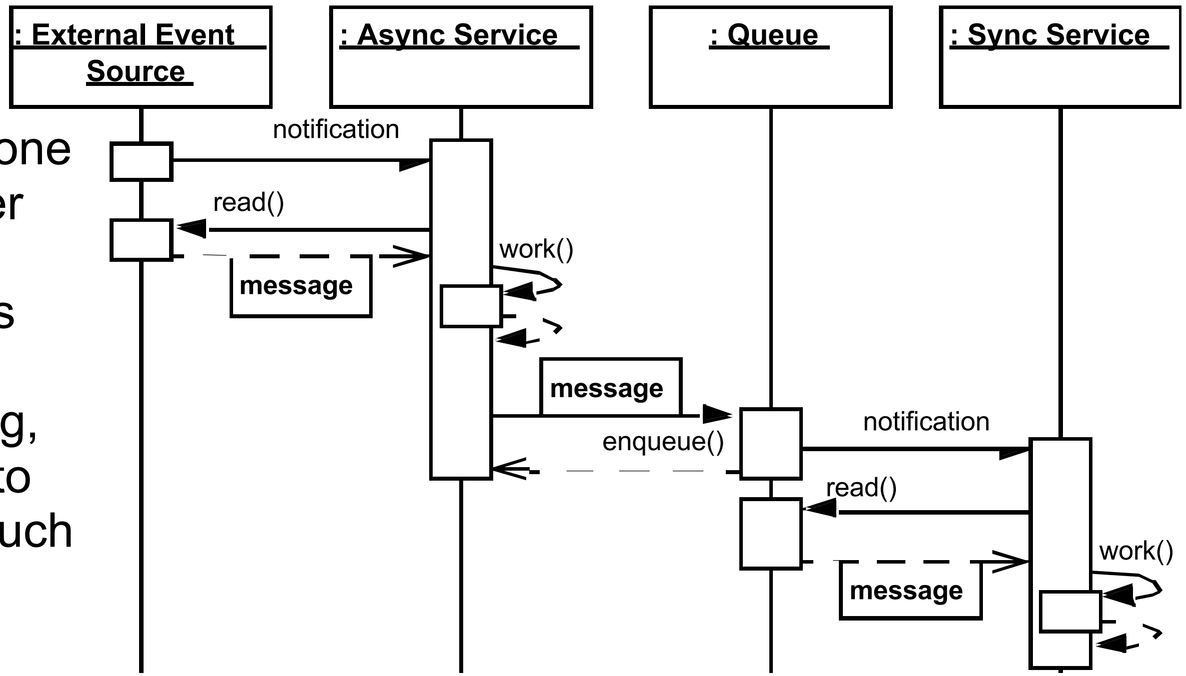
This solution yields two benefits:

1. Threads can be mapped to separate CPUs to scale up server performance via multi-processing
2. Each thread blocks independently, which prevents a flow-controlled connection from degrading the QoS other clients receive

The Half-Sync/Half-Async Pattern

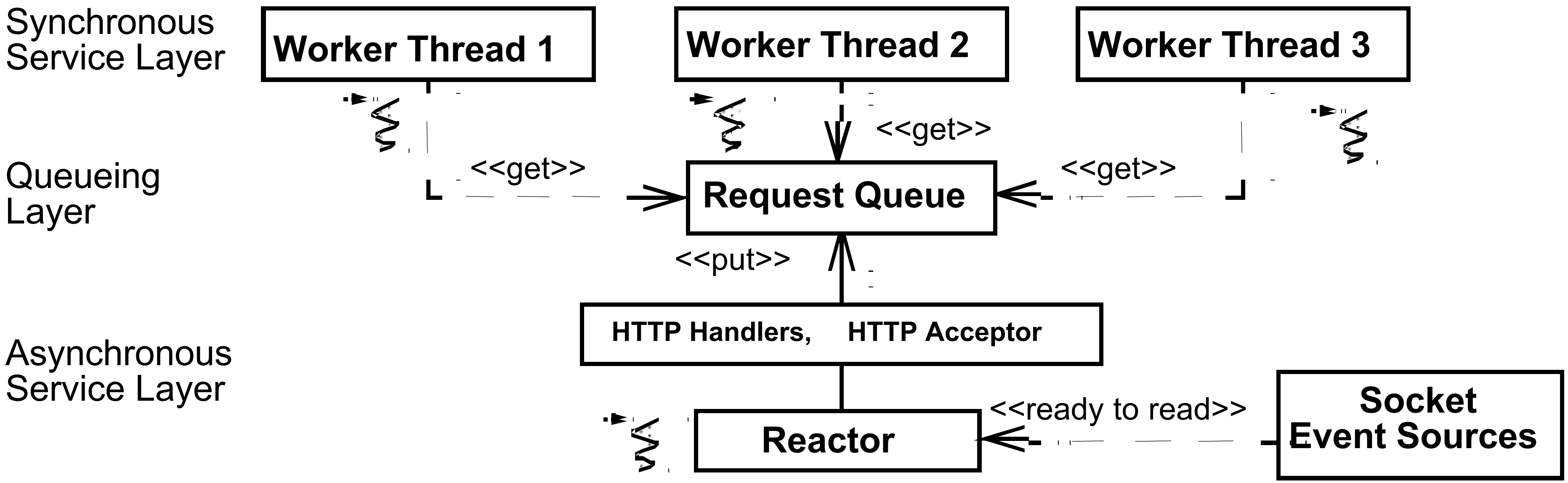
The *Half-Sync/Half-Async* architectural pattern decouples async & sync service processing in concurrent systems, to simplify programming without unduly reducing performance





- This pattern defines two service processing layers—one async & one sync—along with a queueing layer that allows services to exchange messages between the two layers
- The pattern allows sync services, such as HTTP protocol processing, to run concurrently, relative both to each other & to async services, such as event demultiplexing

Applying the Half-Sync/Half-Async Pattern in JAWS



- JAWS uses the Half-Sync/Half-Async pattern to process HTTP GET requests synchronously from multiple clients, but concurrently in separate threads

- The worker thread that removes the request synchronously performs HTTP protocol processing & then transfers the file back to the client

- If flow control occurs on its client connection this thread can block without degrading the QoS experienced by clients serviced by other worker threads in the pool

Pros & Cons of the Half-Sync/Half-Async Pattern

This pattern has three **benefits**:

- ***Simplification & performance***
  - The programming of higher-level synchronous processing services are simplified without degrading the performance of lower-level system services
- ***Separation of concerns***
  - Synchronization policies in each layer are decoupled so that each layer need not use the same concurrency control strategies
- ***Centralization of inter-layer communication***
  - Inter-layer communication is centralized at a single access point, because all interaction is mediated by the queueing layer

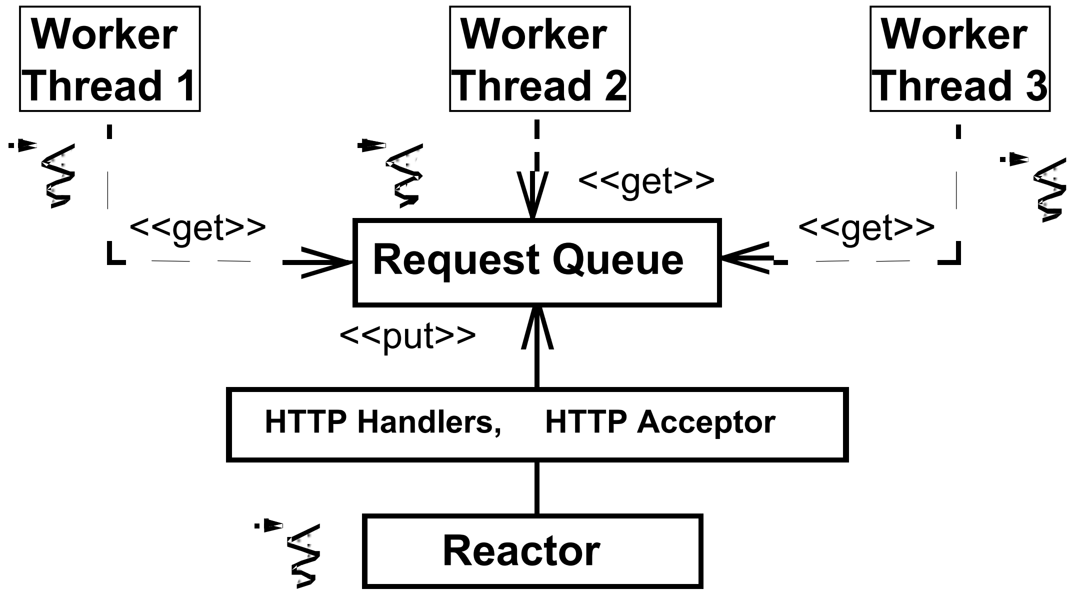
This pattern also incurs **liabilities**:

- ***A boundary-crossing penalty may be incurred***
  - This overhead arises from context switching, synchronization, & data copying overhead when data is transferred between the sync & async service layers via the queueing layer
- ***Higher-level application services may not benefit from the efficiency of async I/O***
  - Depending on the design of operating system or application framework interfaces, it may not be possible for higher-level services to use low-level async I/O devices effectively
- ***Complexity of debugging & testing***
  - Applications written with this pattern can be hard to debug due its concurrent execution

Implementing a Synchronized Request Queue

Context

- The Half-Sync/Half-Async pattern contains a queue
- The JAWS Reactor thread is a 'producer' that inserts HTTP GET requests into the queue
- Worker pool threads are 'consumers' that remove & process queued requests



Problem

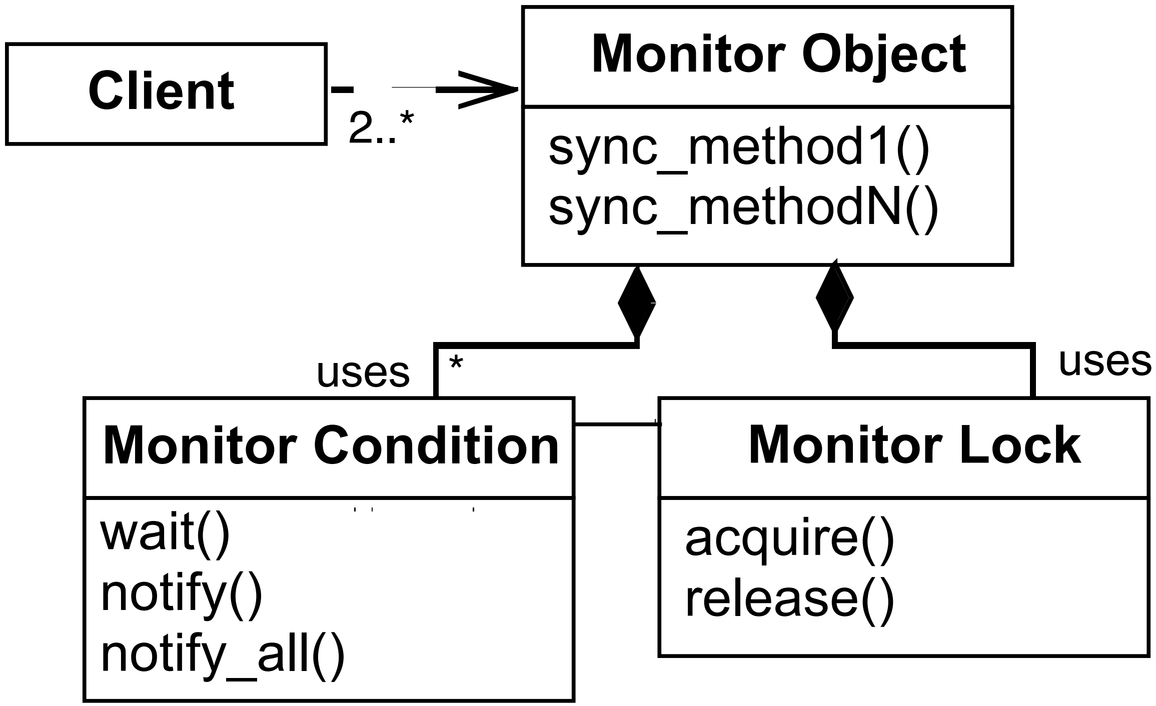
- A naive implementation of a request queue will incur race conditions or 'busy waiting' when multiple threads insert & remove requests
  - *e.g.,* multiple concurrent producer & consumer threads can corrupt the queue's internal state if it is not synchronized properly
  - Similarly, these threads will 'busy wait' when the queue is empty or full, which wastes CPU cycles unnecessarily

The Monitor Object Pattern

Solution

- Apply the *Monitor Object* design pattern to synchronize the queue efficiently & conveniently

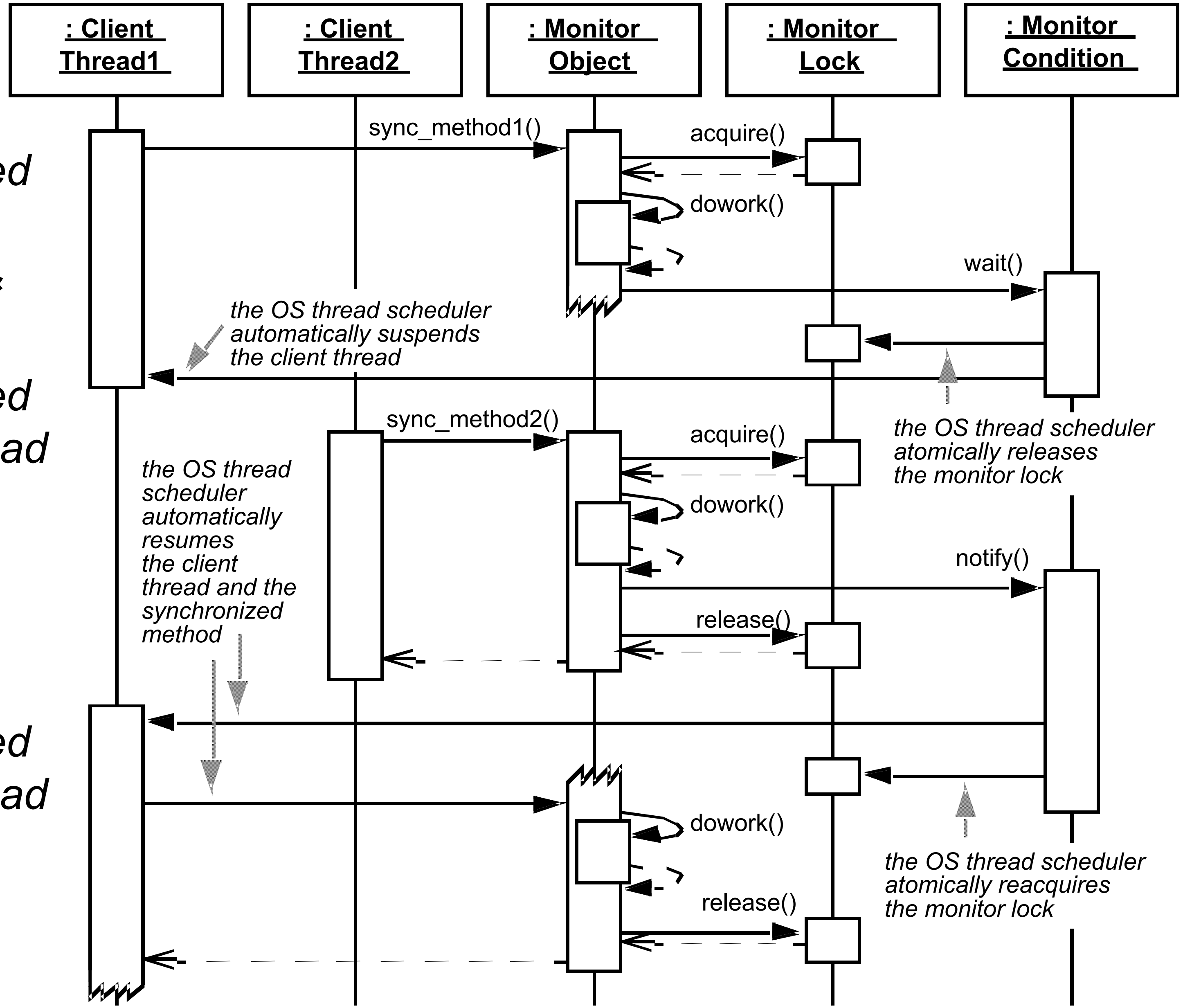
- This pattern synchronizes concurrent method execution to ensure that only one method at a time runs within an object
- It also allows an object's methods to cooperatively schedule their execution sequences



- It's instructive to compare Monitor Object pattern solutions with Active Object pattern solutions
  - The key tradeoff is efficiency vs. flexibility

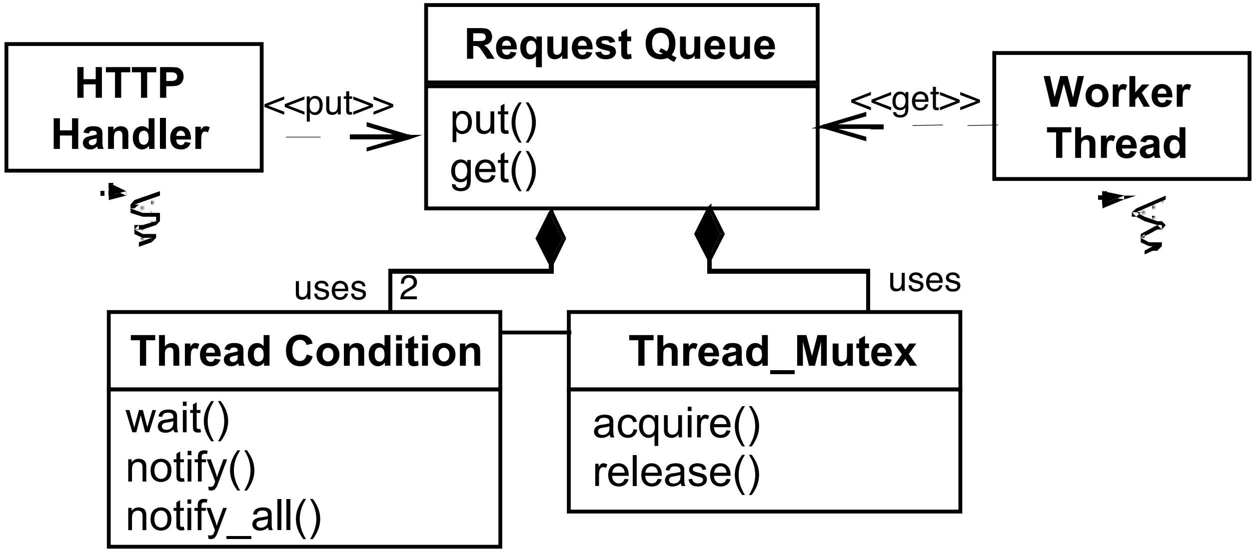
Monitor Object Pattern Dynamics

1. *Synchronized method invocation & serialization*
2. *Synchronized method thread suspension*
3. *Monitor condition notification*
4. *Synchronized method thread resumption*



Applying the Monitor Object Pattern in JAWS

The JAWS synchronized request queue implements the queue's *not-empty* and *not-full* monitor conditions via a pair of ACE wrapper facades for POSIX-style condition variables



- When a worker thread attempts to dequeue an HTTP GET request from an empty queue, the request queue's **`get()`** method atomically releases the monitor lock & the worker thread suspends itself on the *not-empty* monitor condition
- The thread remains suspended until the queue is no longer empty, which happens when an **`HTTP_Handler`** running in the Reactor thread inserts a request into the queue

Pros & Cons of the Monitor Object Pattern

This pattern provides two **benefits**:

- ***Simplification of concurrency control***
  - The Monitor Object pattern presents a concise programming model for sharing an object among cooperating threads where object synchronization corresponds to method invocations
- ***Simplification of scheduling method execution***
  - Synchronized methods use their monitor conditions to determine the circumstances under which they should suspend or resume their execution & that of collaborating monitor objects

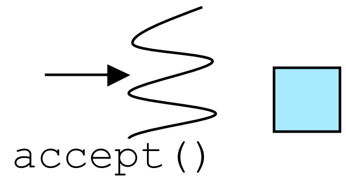
This pattern can also incur **liabilities**:

- The use of a single monitor lock can ***limit scalability*** due to increased contention when multiple threads serialize on a monitor object
- ***Complicated extensibility semantics***
  - These result from the coupling between a monitor object's functionality & its synchronization mechanisms
- It is also hard to inherit from a monitor object transparently, due to the ***inheritance anomaly*** problem
- ***Nested monitor lockout***
  - This problem is similar to the preceding liability & can occur when a monitor object is nested within another monitor object

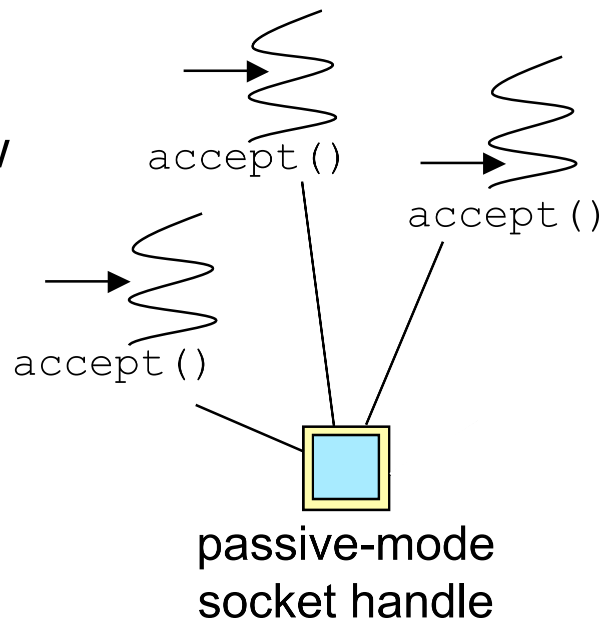
Minimizing Server Threading Overhead

Context

- Socket implementations in certain multi-threaded operating systems provide a concurrent **`accept()`** optimization to accept client connection requests & improve the performance of Web servers that implement the HTTP 1.0 protocol as follows:



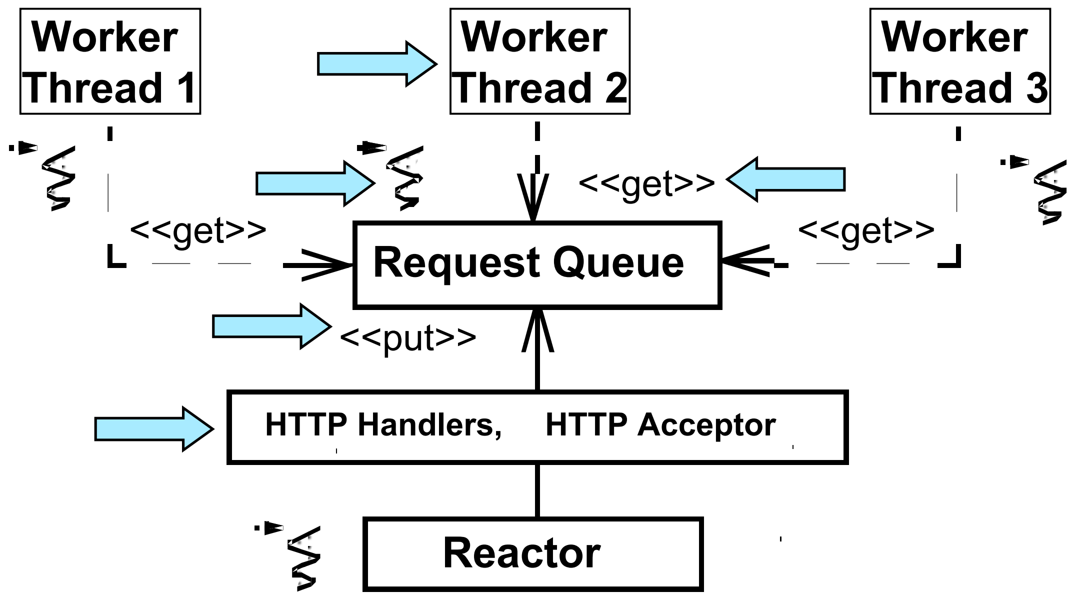
- The OS allows a pool of threads in a Web server to call **`accept()`** on the same passive-mode socket handle
- When a connection request arrives, the operating system's transport layer creates a new connected transport endpoint, encapsulates this new endpoint with a data-mode socket handle & passes the handle as the return value from **`accept()`**
- The OS then schedules one of the threads in the pool to receive this data-mode handle, which it uses to communicate with its connected client



Drawbacks with the Half-Sync/ Half-Async Architecture

Problem

- Although Half-Sync/Half-Async threading model is more scalable than the purely reactive model, it is not necessarily the most efficient design
  - *e.g.,* passing a request between the Reactor thread & a worker thread incurs:
    - *Dynamic memory (de)allocation*,
    - *Synchronization operations*,
    - *A context switch*, &
    - *CPU cache updates*
- This overhead makes JAWS' latency unnecessarily high, particularly on operating systems that support the concurrent **`accept()`** optimization



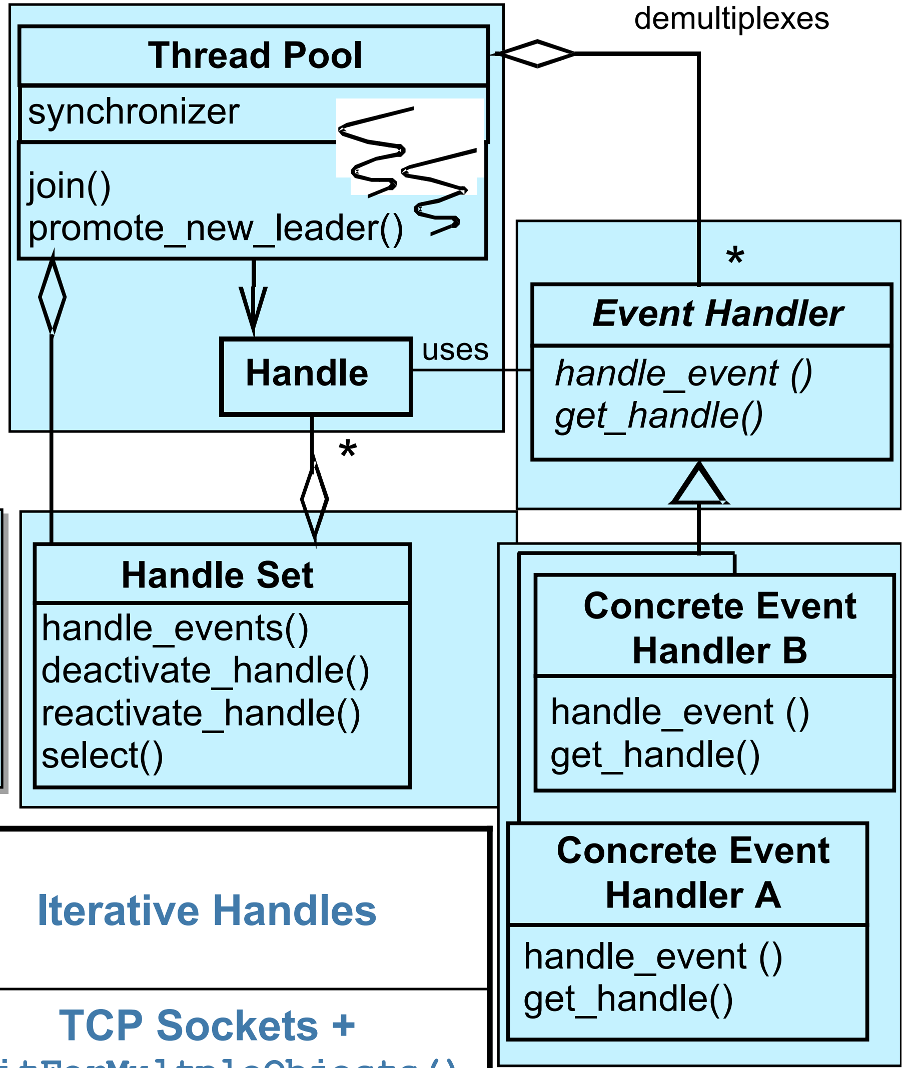
Solution

- Apply the *Leader/Followers* architectural pattern to minimize server threading overhead

The Leader/Followers Pattern

The Leader/Followers architectural pattern provides an efficient concurrency model where multiple threads take turns sharing event sources to detect, demux, dispatch, & process service requests that occur on the event sources

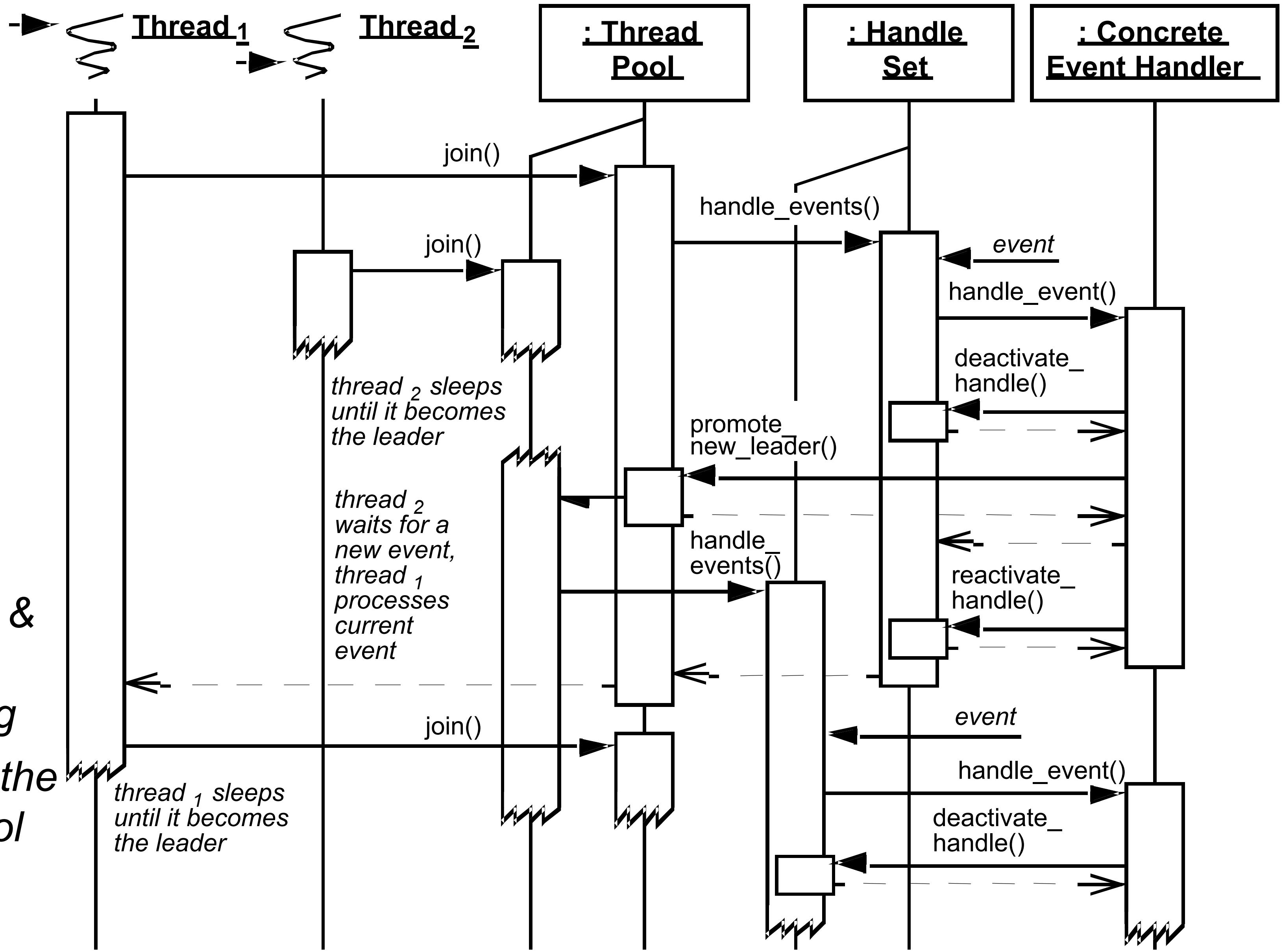
This pattern eliminates the need for—& the overhead of—a separate Reactor thread & synchronized request queue used in the Half-Sync/Half-Async pattern



Handles / Handle Sets	Concurrent Handles	Iterative Handles
Concurrent Handle Sets	UDP Sockets + `WaitForMultipleObjects()`	TCP Sockets + `WaitForMultpleObjects()`
Iterative Handle Sets	UDP Sockets + `select()/poll()`	TCP Sockets + `select()/poll()`

Leader/Followers Pattern Dynamics

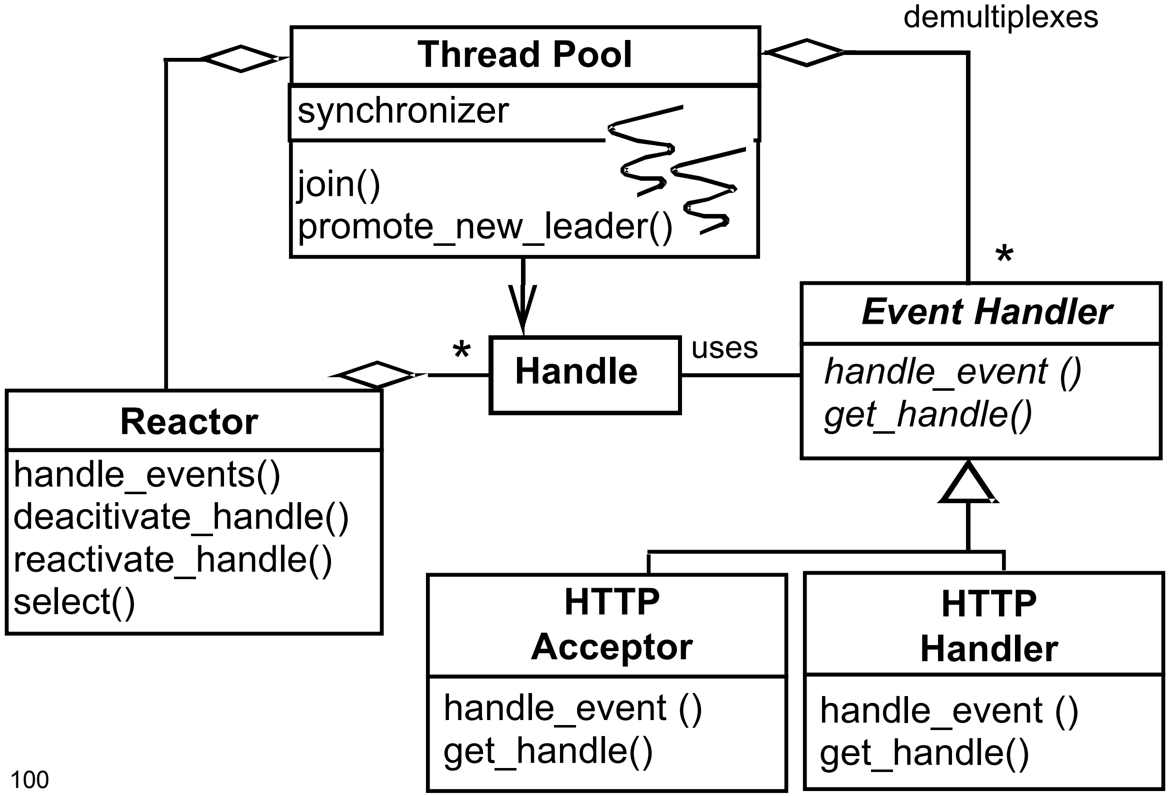
1. *Leader thread demuxing*
2. *Follower thread promotion*
3. *Event handler demuxing & event processing*
4. *Rejoining the thread pool*



Applying the Leader/Followers Pattern in JAWS

Two options:

1. If platform supports **`accept()`** optimization then the Leader/Followers pattern can be implemented by the OS
2. Otherwise, this pattern can be implemented as a reusable framework



Although Leader/Followers thread pool design is highly efficient the Half-Sync/Half-Async design may be more appropriate for certain types of servers, *e.g.:*

- The Half-Sync/Half-Async design can reorder & prioritize client requests more flexibly, because it has a synchronized request queue implemented using the Monitor Object pattern
- It may be more scalable, because it queues requests in Web server virtual memory, rather than the OS kernel

Pros and Cons of the Leader/Followers Pattern

This pattern provides two **benefits**:

- ***Performance enhancements***
  - This can improve performance as follows:
    - It enhances CPU cache affinity and eliminates the need for dynamic memory allocation & data buffer sharing between threads
    - It minimizes locking overhead by not exchanging data between threads, thereby reducing thread synchronization
    - It can minimize priority inversion because no extra queueing is introduced in the server
    - It doesn't require a context switch to handle each event, reducing dispatching latency
- ***Programming simplicity***
  - The Leader/Follower pattern simplifies the programming of concurrency models where multiple threads can receive requests, process responses, & demultiplex connections using a shared handle set

This pattern also incur **liabilities**:

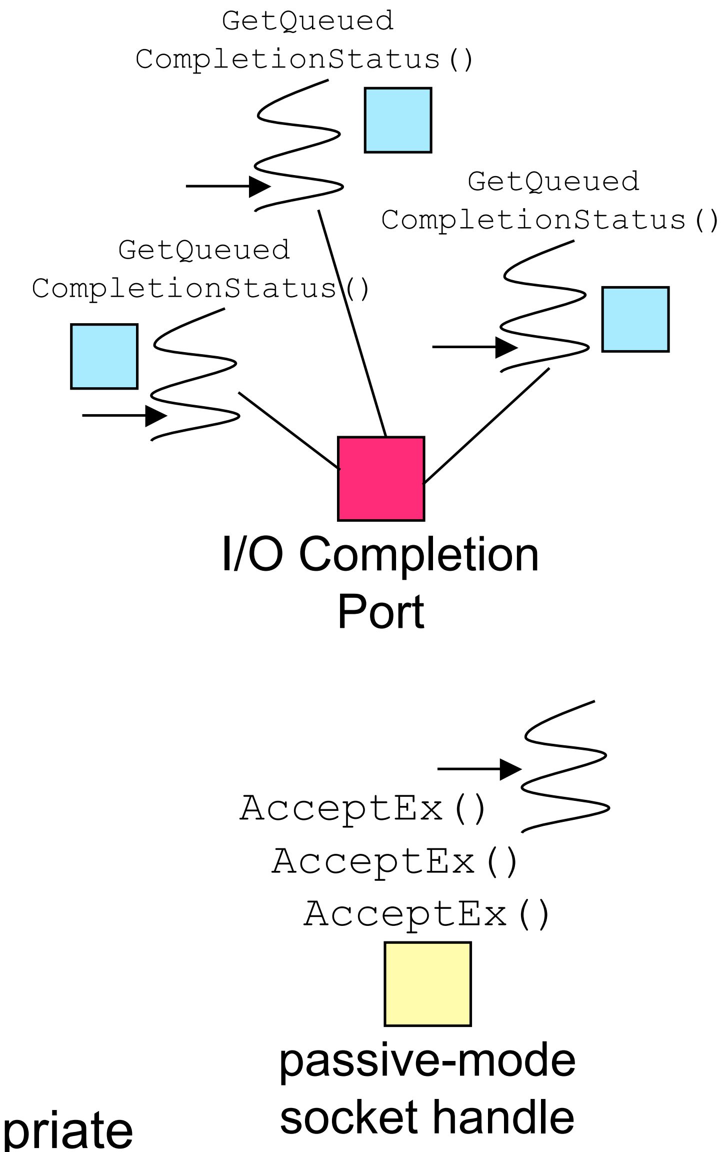
- ***Implementation complexity***
  - The advanced variants of the Leader/ Followers pattern are hard to implement
- ***Lack of flexibility***
  - In the Leader/ Followers model it is hard to discard or reorder events because there is no explicit queue
- ***Network I/O bottlenecks***
  - The Leader/Followers pattern serializes processing by allowing only a single thread at a time to wait on the handle set, which could become a bottleneck because only one thread at a time can demultiplex I/O events

Using Asynchronous I/O Effectively

Context

- Synchronous multi-threading may not be the most scalable way to implement a Web server on OS platforms that support async I/O more efficiently than synchronous multi-threading

- For example, highly-efficient Web servers can be implemented on Windows NT by invoking async Win32 operations that perform the following activities:
  - Processing indication events, such as TCP CONNECT and HTTP GET requests, via **`AcceptEx()`** & **`ReadFile()`**, respectively
  - Transmitting requested files to clients asynchronously via **`WriteFile()`** or **`TransmitFile()`**
- When these async operations complete, WinNT
  1. Delivers the associated completion events containing their results to the Web server
  2. Processes these events & performs the appropriate actions before returning to its event loop



The Proactor Pattern

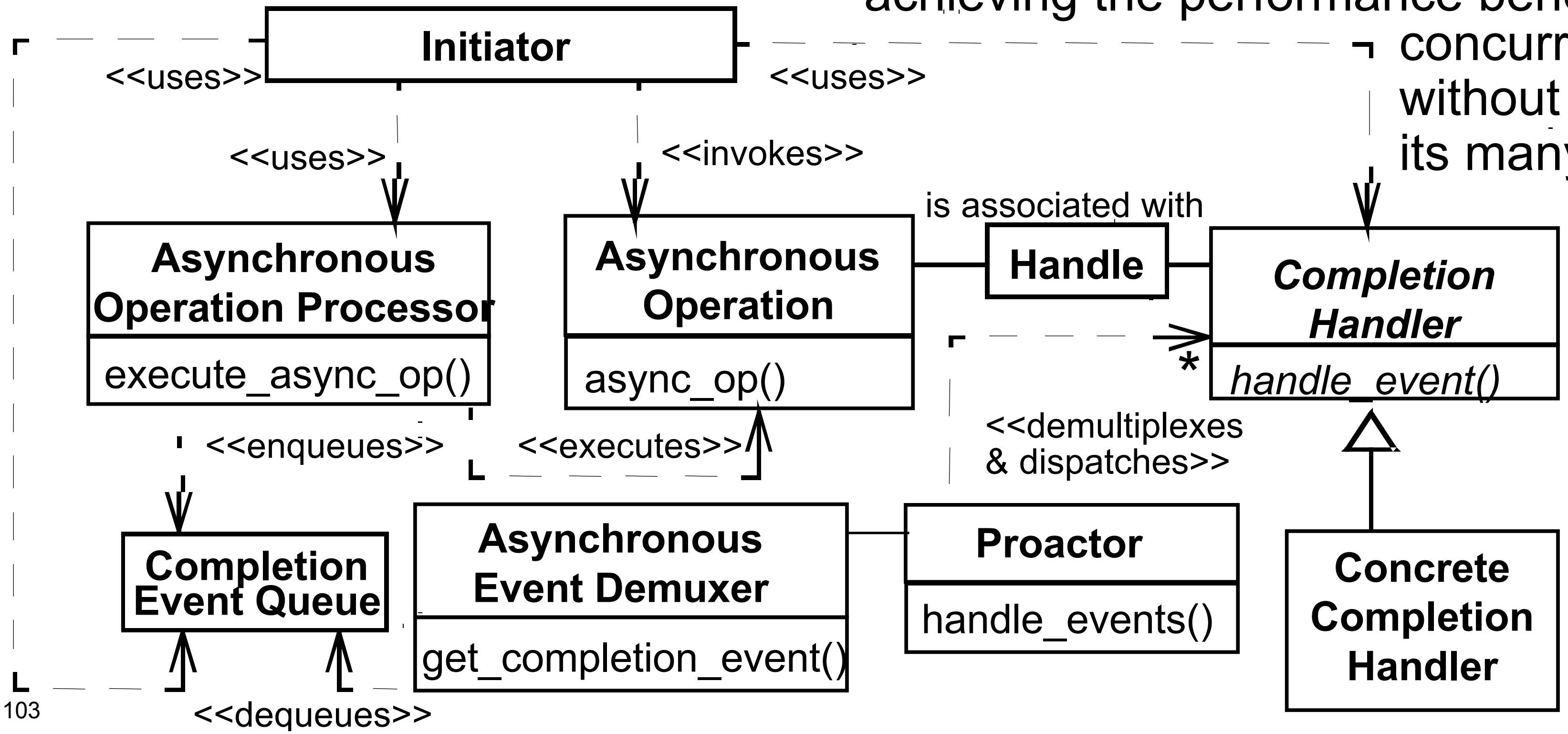
Problem

- Developing software that achieves the potential efficiency & scalability of async I/O is hard due to the separation in time & space of async operation invocations & their subsequent completion events

Solution

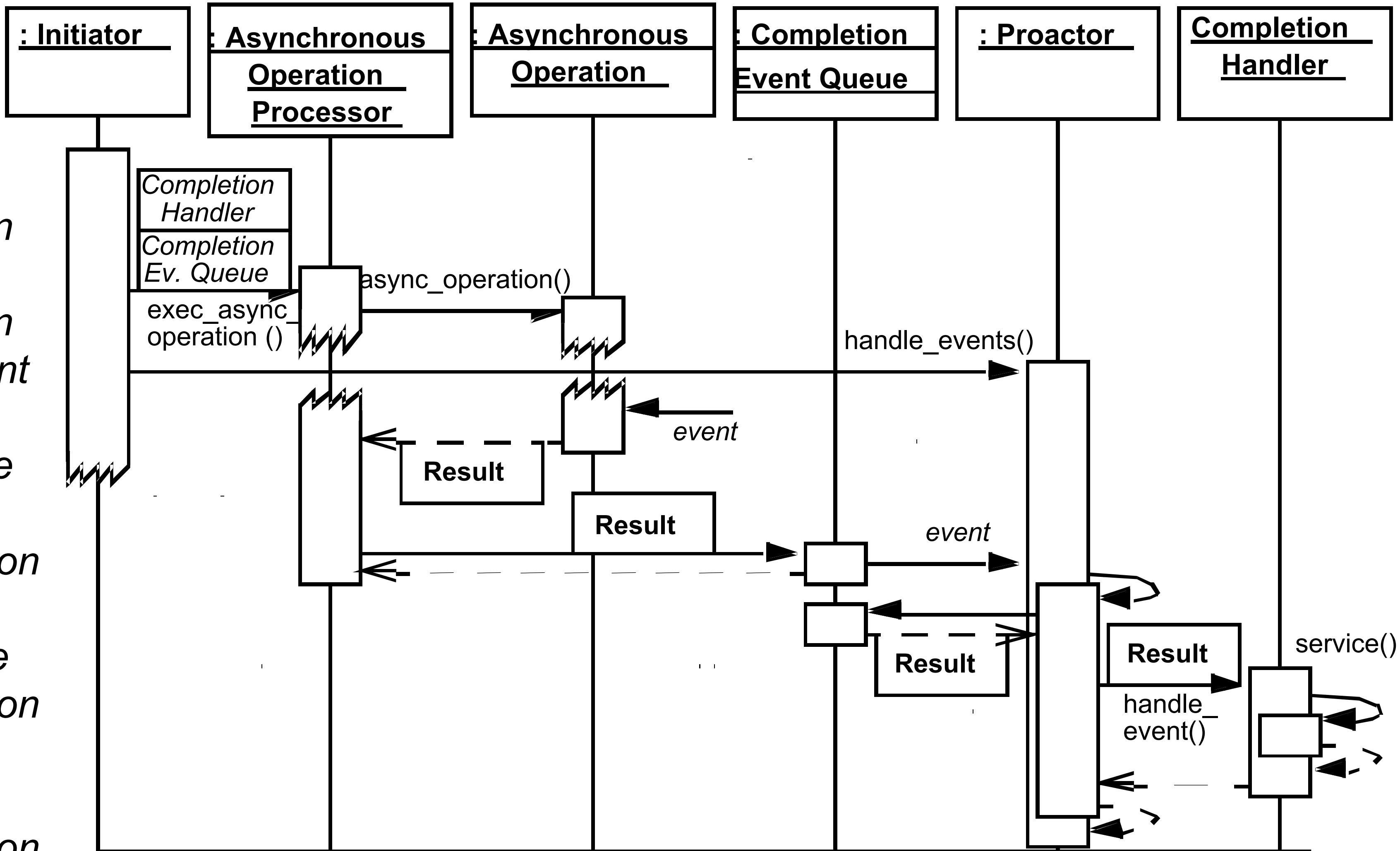
- Apply the *Proactor* architectural pattern to make efficient use of async I/O

This pattern allows event-driven applications to efficiently demultiplex & dispatch service requests triggered by the completion of async operations, thereby achieving the performance benefits of concurrency without incurring its many liabilities



Dynamics in the Proactor Pattern

1. *Initiate operation*
2. *Process operation*
3. *Run event loop*
4. *Generate & queue completion event*
5. *Dequeue completion event & perform completion processing*



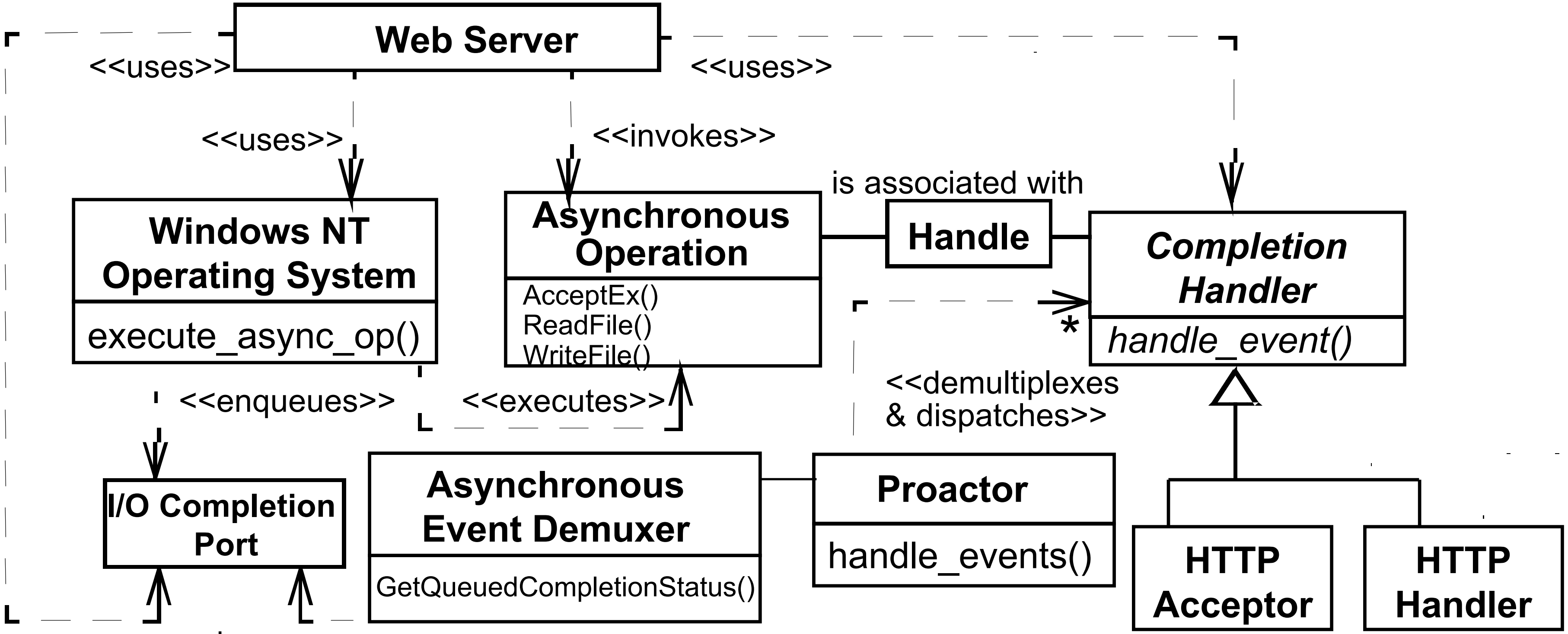
Note *similarities* & *differences* with the Reactor pattern, *e.g.:*
•Both process events via callbacks
•However, it's generally easier to multi-thread a proactor

Applying the Proactor Pattern in JAWS

The Proactor pattern structures the JAWS concurrent server to receive & process requests from multiple clients asynchronously

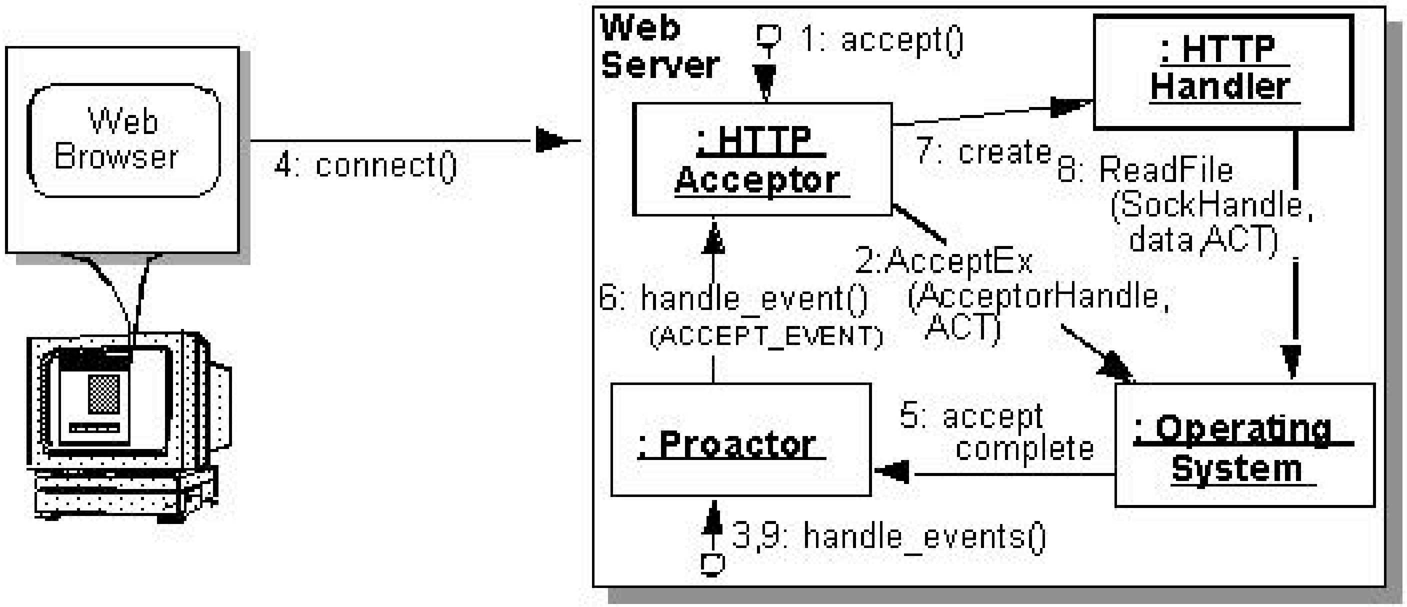
JAWS HTTP components are split into two parts:

1. Operations that execute asynchronously
   - *e.g.*, to accept connections & receive client HTTP GET requests
2. The corresponding completion handlers that process the async operation results
   - *e.g.*, to transmit a file back to a client after an async connection operation completes

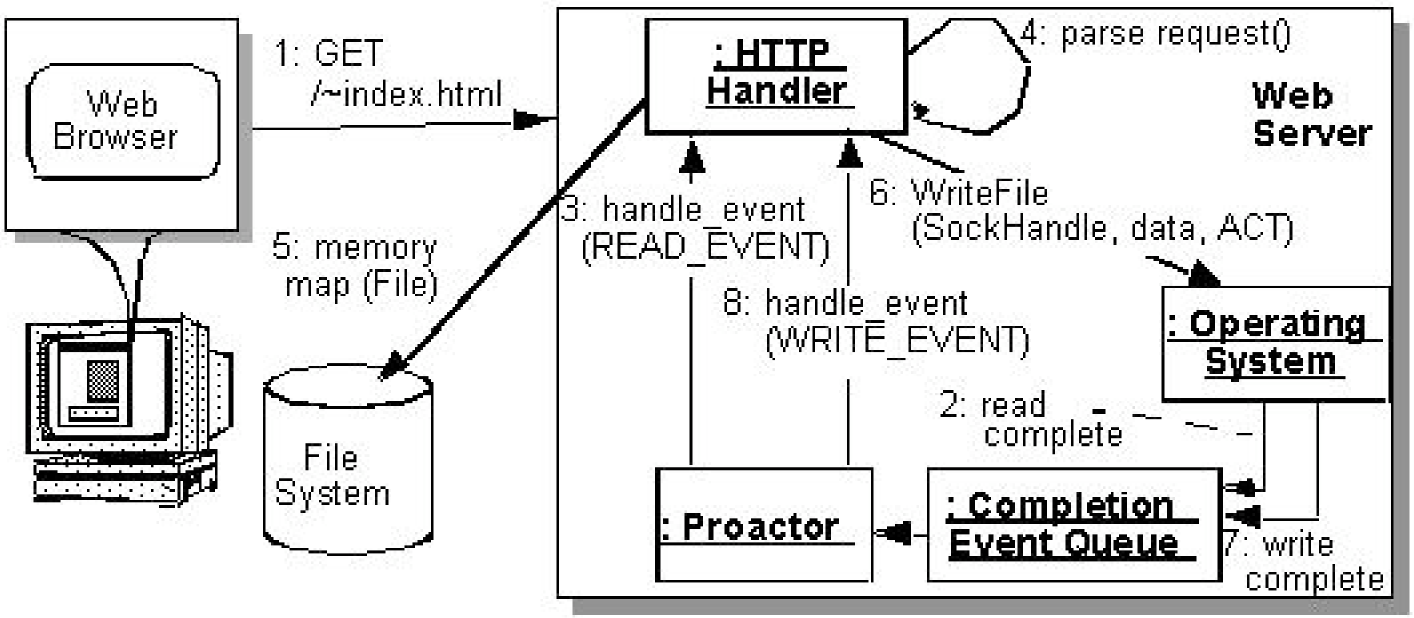


Proactive Connection Management & Data Transfer in JAWS

Connection Management Phase



Data Transfer Phase



Pros and Cons of the Proactor Pattern

This pattern offers five **benefits**:

- ***Separation of concerns***
  - Decouples application-independent async mechanisms from application-specific functionality
- ***Portability***
  - Improves application portability by allowing its interfaces to be reused independently of the OS event demuxing calls
- ***Decoupling of threading from concurrency***
  - The async operation processor executes long-duration operations on behalf of initiators so applications can spawn fewer threads
- ***Performance***
  - Avoids context switching costs by activating only those logical threads of control that have events to process
- ***Simplification of application synchronization***
  - If concrete completion handlers spawn no threads, application logic can be written with little or no concern for synchronization issues

This pattern incurs some **liabilities**:

- ***Restricted applicability***
  - This pattern can be applied most efficiently if the OS supports asynchronous operations natively
- ***Complexity of programming, debugging, & testing***
  - It is hard to program applications & higher-level system services using asynchrony mechanisms, due to the separation in time & space between operation invocation and completion
- ***Scheduling, controlling, & canceling asynchronously running operations***
  - Initiators may be unable to control the scheduling order in which asynchronous operations are executed by an asynchronous operation processor

Efficiently Demuxing Asynchronous Operations & Completions

Context

- In a proactive Web server async I/O operations will yield I/O completion event responses that must be processed efficiently

Problem

- As little overhead as possible should be incurred to determine how the completion handler will demux & process completion events after async operations finish executing
- When a response arrives, the application should spend as little time as possible demultiplexing the completion event to the handler that will process the async operation's response

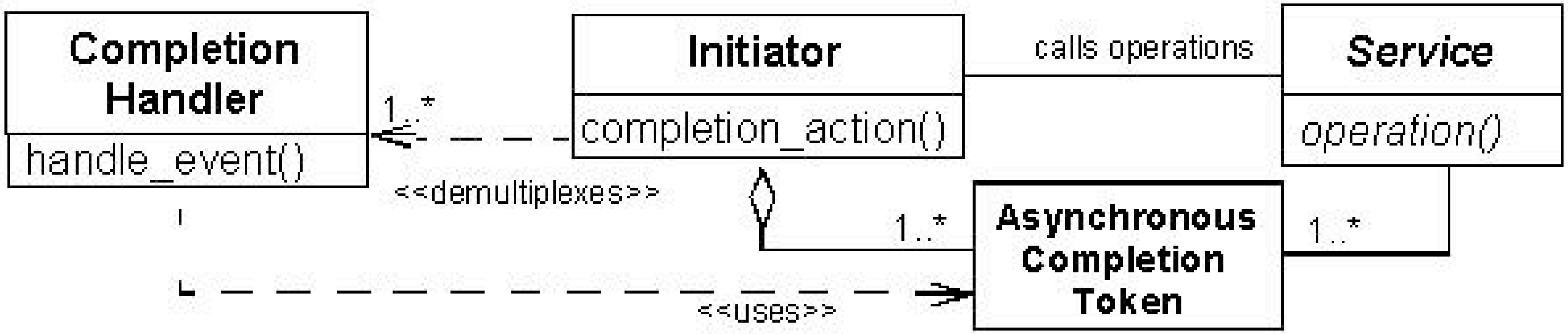
Solution

- Apply the *Asynchronous Completion Token* design pattern to demux & process the responses of asynchronous operations efficiently

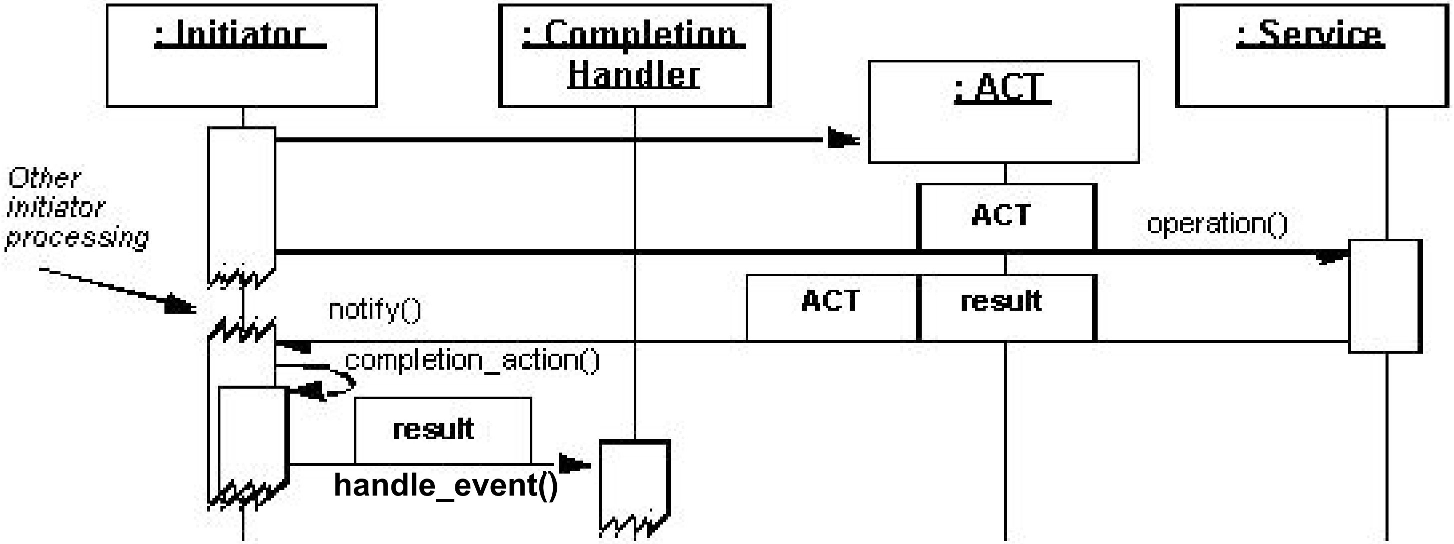
- Together with each async operation that a client *initiator* invokes on a *service*, transmit information that identifies how the initiator should process the service's response
- Return this information to the initiator when the operation finishes, so that it can be used to demux the response efficiently, allowing the initiator to process it accordingly

The Asynchronous Completion Token Pattern

Structure and Participants

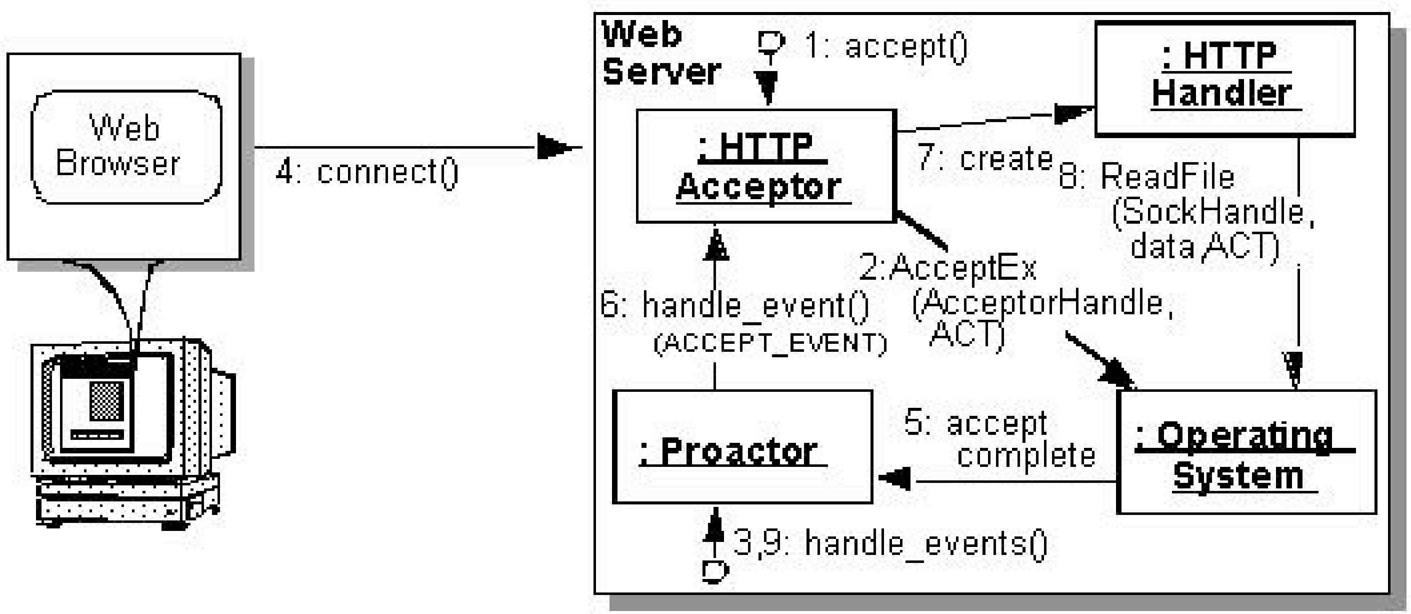


Dynamic Interactions

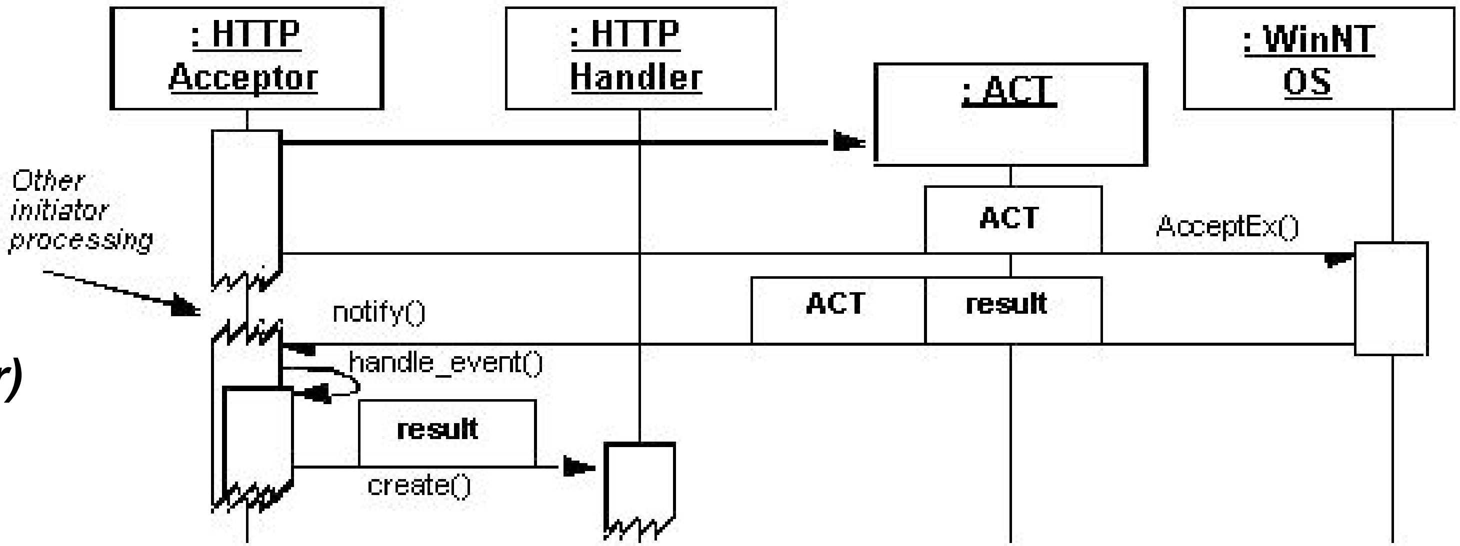


Applying the Asynchronous Completion Token Pattern in JAWS

Connection Management Phase



Detailed Interactions
(`HTTP_Acceptor` is both initiator & completion handler)



Pros and Cons of the Asynchronous Completion Token Pattern

This pattern has four **benefits**:

- ***Simplified initiator data structures***
  - Initiators need not maintain complex data structures to associate service responses with completion handlers
- ***Efficient state acquisition***
  - ACTs are time efficient because they need not require complex parsing of data returned with the service response
- ***Space efficiency***
  - ACTs can consume minimal data space yet can still provide applications with sufficient information to associate large amounts of state to process asynchronous operation completion actions
- ***Flexibility***
  - User-defined ACTs are not forced to inherit from an interface to use the service's ACTs

This pattern has some **liabilities**:

- ***Memory leaks***
  - Memory leaks can result if initiators use ACTs as pointers to dynamically allocated memory & services fail to return the ACTs, for example if the service crashes
- ***Authentication***
  - When an ACT is returned to an initiator on completion of an asynchronous event, the initiator may need to authenticate the ACT before using it
- ***Application re-mapping***
  - If ACTs are used as direct pointers to memory, errors can occur if part of the application is re-mapped in virtual memory

Transparently Parameterizing Synchronization into Components

Context

- The various concurrency patterns described earlier impact component synchronization strategies in various ways
  - *e.g.,*ranging from no locks to readers/writer locks
- In general, components must run efficiently in a variety of concurrency models

Problem

- It should be possible to customize JAWS component synchronization mechanisms according to the requirements of particular application use cases & configurations
- Hard-coding synchronization strategies into component implementations is *inflexible*
- Maintaining multiple versions of components manually is *not scalable*

Solution

- Apply the *Strategized Locking* design pattern to parameterize JAWS component synchronization strategies by making them ‘pluggable’ types

- Each type objectifies a particular synchronization strategy, such as a mutex, readers/writer lock, semaphore, or ‘null’ lock
- Instances of these pluggable types can be defined as objects contained within a component, which then uses these objects to synchronize its method implementations efficiently

Applying Polymorphic Strategized Locking in JAWS

Polymorphic Strategized Locking

```
class Lock {
public:
  // Acquire and release the lock.
  virtual void acquire () = 0;
  virtual void release () = 0;

  // ...
};
class Thread_Mutex : public Lock {
  // ...
};
```

```
class File_Cache {
public:
  // Constructor.
  File_Cache (Lock &l): lock_ (&l) { }

  // A method.
  const void *lookup (const string &path) const {
    lock_->acquire();
    // Implement the <lookup> method.
    lock_->release ();
  }

  // ...
private:
  // The polymorphic strategized locking object.
  mutable Lock *lock_;
  // Other data members and methods go here...
};
```

Applying Parameterized Strategized Locking in JAWS

Parameterized Strategized Locking

- Single-threaded file cache.

```
typedef File_Cache<Null_Mutex>
        Content_Cache;
```

- Multi-threaded file cache using a thread mutex.

```
typedef File_Cache<Thread_Mutex>
        Content_Cache;
```

- Multi-threaded file cache using a readers/writer lock.

```
typedef File_Cache<RW_Lock>
        Content_Cache;
```

```
template <class LOCK>
class File_Cache {
public:
  // A method.
  const void *lookup
    (const string &path) const {
    lock_.acquire ();
    // Implement the <lookup> method.
    lock_.release ();
  }

  // ...
private:
  // The polymorphic strategized locking object.
  mutable LOCK lock_;
  // Other data members and methods go here...
};
```

Note that the various locks need not inherit from a common base class or use virtual methods!

Pros and Cons of the Strategized Locking Pattern

This pattern provides three **benefits**:

- ***Enhanced flexibility & customization***
  - It is straightforward to configure & customize a component for certain concurrency models because the synchronization aspects of components are strategized
- ***Decreased maintenance effort for components***
  - It is straightforward to add enhancements & bug fixes to a component because there is only one implementation, rather than a separate implementation for each concurrency model
- ***Improved reuse***
  - Components implemented using this pattern are more reusable, because their locking strategies can be configured orthogonally to their behavior

This pattern also incurs **liabilities**:

- ***Obtrusive locking***
  - If templates are used to parameterize locking aspects this will expose the locking strategies to application code
- ***Over-engineering***
  - Externalizing a locking mechanism by placing it in a component's interface may actually provide *too much* flexibility in certain situations
    - *e.g.,* inexperienced developers may try to parameterize a component with the wrong type of lock, resulting in improper compile- or run-time behavior

Ensuring Locks are Released Properly

Context

- Concurrent applications, such as JAWS, contain shared resources that are manipulated by multiple threads concurrently

Problem

- Code that shouldn't execute concurrently must be protected by some type of lock that is acquired & released when control enters & leaves a critical section, respectively
- If programmers must acquire & release locks explicitly, it is hard to ensure that the locks are released in all paths through the code
  - *e.g.*, in C++ control can leave a scope due to a return, break, continue, or goto statement, as well as from an unhandled exception being propagated out of the scope

Solution

- In C++, apply the *Scoped Locking* idiom to define a guard class whose constructor automatically acquires a lock when control enters a scope & whose destructor automatically releases the lock when control leaves the scope

```
// A method.
const void *lookup
   (const string &path) const {
   lock_.acquire ();
   // The <lookup> method
   // implementation may return
   // prematurely…
   lock_.release ();
}
```

Applying the Scoped Locking Idiom in JAWS

Generic Guard Wrapper Facade

```
template <class LOCK>
class Guard {
public:
  // Store a pointer to the lock and acquire the lock.
  Guard (LOCK &lock)
    : lock_ (&lock)
  { lock_->acquire (); }

  // Release the lock when the guard goes out of scope,
  ~Guard () { lock_->release (); }
private:
  // Pointer to the lock we're managing.
  LOCK *lock_;
};
```

Applying the Guard

```
template <class LOCK>
class File_Cache {
public:
  // A method.
  const void *lookup
    (const string &path) const {
    // Use Scoped Locking idiom to acquire
    // & release the <lock_>
automatically.
    Guard<LOCK> guard (*lock);
    // Implement the <lookup> method.
    // lock_ released automatically…
  }
```

Instances of the guard class can be allocated on the run-time stack to acquire & release locks in method or block scopes that define critical sections

Pros and Cons of the Scoped Locking Idiom

This idiom has one **benefit**:

- ***Increased robustness***
  - This idiom increases the robustness of concurrent applications by eliminating common programming errors related to synchronization & multi-threading
  - By applying the Scoped Locking idiom, locks are acquired & released automatically when control enters and leaves critical sections defined by C++ method & block scopes

This idiom also has **liabilities**:

- ***Potential for deadlock when used recursively***
  - If a method that uses the Scoped Locking idiom calls itself recursively, 'self-deadlock' will occur if the lock is not a 'recursive' mutex
- ***Limitations with language-specific semantics***
  - The Scoped Locking idiom is based on a C++ language feature & therefore will not be integrated with operating system-specific system calls
    - Thus, locks may not be released automatically when threads or processes abort or exit inside a guarded critical section
    - Likewise, they will not be released properly if the standard C `longjmp()` function is called because this function does not call the destructors of C++ objects as the run-time stack unwinds

Minimizing Unnecessary Locking

Context

- Components in multi-threaded applications that contain intra-component method calls
- Components that have applied the Strategized Locking pattern

Problem

- Thread-safe components should be designed to avoid unnecessary locking
- Thread-safe components should be designed to avoid “self-deadlock”

Solution

- Apply the *Thread-safe Interface* design pattern to minimize locking overhead & ensure that intra-component method calls do not incur ‘self-deadlock’ by trying to reacquire a lock that is held by the component already

This pattern structures all components that process intra-component method invocations according two design conventions:

- *Interface methods check*
  - All interface methods, such as C++ public methods, should only acquire/release component lock(s), thereby performing synchronization checks at the ‘border’ of the component.
- *Implementation methods trust*
  - Implementation methods, such as C++ private and protected methods, should only perform work when called by interface methods.

Motivating the Need for the Thread-safe Interface Pattern

```
template <class LOCK>
class File_Cache {
public:
  // Return a pointer to the memory-mapped file associated with
  // <path> name, adding it to the cache if it doesn't exist.
  const void *lookup (const string &path) const {
    // Use the Scoped Locking idiom to acquire
    // & release the <lock_> automatically.
    Guard<LOCK> guard (lock_);
    const void *file_pointer = check_cache (path);
    if (file_pointer == 0) {
      // Insert the <path> name into the cache.
      // Note the intra-class <insert> call.
      insert (path);
      file_pointer = check_cache (path);
    }
    return file_pointer;
  }
  // Add <path> name to the cache.
  void insert (const string &path) {
    // Use the Scoped Locking idiom to acquire
    // and release the <lock_> automatically.
    Guard<LOCK> guard (lock_);
    // ... insert <path> into the cache...
  }
private:
  mutable LOCK lock_;
  const void *check_cache (const string &) const;
  // ... other private methods and data omitted...
};
```

Since `File_Cache` is a template we don't know the type of lock used to parameterize it.

Applying the Thread-safe Interface Pattern in JAWS

```
template <class LOCK>
class File_Cache {
public:
  // Return a pointer to the memory-mapped file associated with
  // <path> name, adding it to the cache if it doesn't exist.
  const void *lookup (const string &path) const {
    // Use the Scoped Locking idiom to acquire
    // and release the <lock_> automatically.
    Guard<LOCK> guard (lock_);
    return lookup_i (path);
  }
private:
  mutable LOCK lock_; // The strategized locking object.

  // This implementation method doesn't acquire or release
  // <lock_> and does its work without calling interface methods.
  const void *lookup_i (const string &path) const {
    const void *file_pointer = check_cache_i (path);
    if (file_pointer == 0) {
      // If <path> name isn't in the cache then
      // insert it and look it up again.
      insert_i (path);
      file_pointer = check_cache_i (path);
      // The calls to implementation methods <insert_i> and
      // <check_cache_i> assume that the lock is held & do work.
    }
    return file_pointer;
```

Note fewer constraints on the type of LOCK…

Pros and Cons of the Thread-safe Interface Pattern

This pattern has three **benefits**:

- ***Increased robustness***
  - This pattern ensures that self-deadlock does not occur due to intra-component method calls
- ***Enhanced performance***
  - This pattern ensures that locks are not acquired or released unnecessarily
- ***Simplification of software***
  - Separating the locking and functionality concerns can help to simplify both aspects

This pattern has some **liabilities**:

- ***Additional indirection and extra methods***
  - Each interface method requires at least one implementation method, which increases the footprint of the component & may also add an extra level of method-call indirection for each invocation
- ***Potential for misuse***
  - OO languages, such as C++ and Java, support class-level rather than object-level access control
  - As a result, an object can bypass the public interface to call a private method on another object of the same class, thus bypassing that object’s lock
- ***Potential overhead***
  - This pattern prevents multiple components from sharing the same lock & prevents locking at a finer granularity than the component, which can increase lock contention

Synchronizing Singletons Correctly

Context

- JAWS uses various singletons to implement components where only one instance is required
  - *e.g.,* the ACE Reactor, the request queue, etc.

Problem

- Singletons can be problematic in multi-threaded programs

Either too little locking…

```
class Singleton {
public:
  static Singleton *instance ()
  {
    if (instance_ == 0) {
      // Enter critical section.
      instance_ = new Singleton;
      // Leave critical section.
    }
    return instance_;
  }
  void method_1 ();
  // Other methods omitted.
private:
  static Singleton *instance_;
  // Initialized to 0 by linker.
};
```

… or too much

```
class Singleton {
public:
  static Singleton *instance ()
  {
    Guard<Thread_Mutex> g (lock_);
    if (instance_ == 0) {
      // Enter critical section.
      instance_ = new Singleton;
      // Leave critical section.
    }
    return instance_;
  }
private:
  static Singleton *instance_;
  // Initialized to 0 by linker.
  static Thread_Mutex lock_;
};
```

The Double-checked Locking Optimization Pattern

Solution

- Apply the *Double-Checked Locking Optimization* design pattern to reduce contention & synchronization overhead whenever critical sections of code must acquire locks in a thread-safe manner just once during program execution

```
// Perform first-check to
// evaluate 'hint'.
if (first_time_in is FALSE)
{
  acquire the mutex
  // Perform double-check to
  // avoid race condition.
  if (first_time_in is FALSE)
  {
    execute the critical section
    set first_time_in to TRUE
  }
  release the mutex
}
```

```
class Singleton {
public:
  static Singleton *instance ()
  {
  // First check
  if (instance_ == 0) {
    Guard<Thread_Mutex> g(lock_);
    // Double check.
    if (instance_ == 0)
      instance_ = new Singleton;
    }
    return instance_;
  }
private:
  static Singleton *instance_;
  static Thread_Mutex lock_;
};
```

Applying the Double-Checked Locking Optimization Pattern in ACE

```
template <class TYPE>
class ACE_Singleton {
public:
  static TYPE *instance () {
    // First check
    if (instance_ == 0) {
      // Scoped Locking acquires and release lock.
      Guard<Thread_Mutex> guard (lock_);
      // Double check instance_.
      if (instance_ == 0)
        instance_ = new TYPE;
    }
    return instance_;
  }
private:
  static TYPE *instance_;
  static Thread_Mutex lock_;
};
```

ACE defines a "singleton adapter" template to automate the double-checked locking optimization

Thus, creating a "thread-safe" singleton is easy

```
typedef ACE_Singleton
        <Request_Queue>
        Request_Queue_Singleton;
```

Pros and Cons of the Double-Checked Locking Optimization Pattern

This pattern has two **benefits**:

- ***Minimized locking overhead***
  - By performing two first-time-in flag checks, this pattern minimizes overhead for the common case
  - After the flag is set the first check ensures that subsequent accesses require no further locking
- ***Prevents race conditions***
  - The second check of the first-time-in flag ensures that the critical section is executed just once

This pattern has some **liabilities***:*

- ***Non-atomic pointer or integral assignment semantics***
  - If an `instance_` pointer is used as the flag in a singleton implementation, all bits of the singleton `instance_` pointer must be read & written atomically in a single operation
  - If the write to memory after the call to new is not atomic, other threads may try to read an invalid pointer
- ***Multi-processor cache coherency***
  - Certain multi-processor platforms, such as the COMPAQ Alpha & Intel Itanium, perform aggressive memory caching optimizations in which read & write operations can execute 'out of order' across multiple CPU caches, such that the CPU cache lines will not be flushed properly if shared data is accessed without locks held

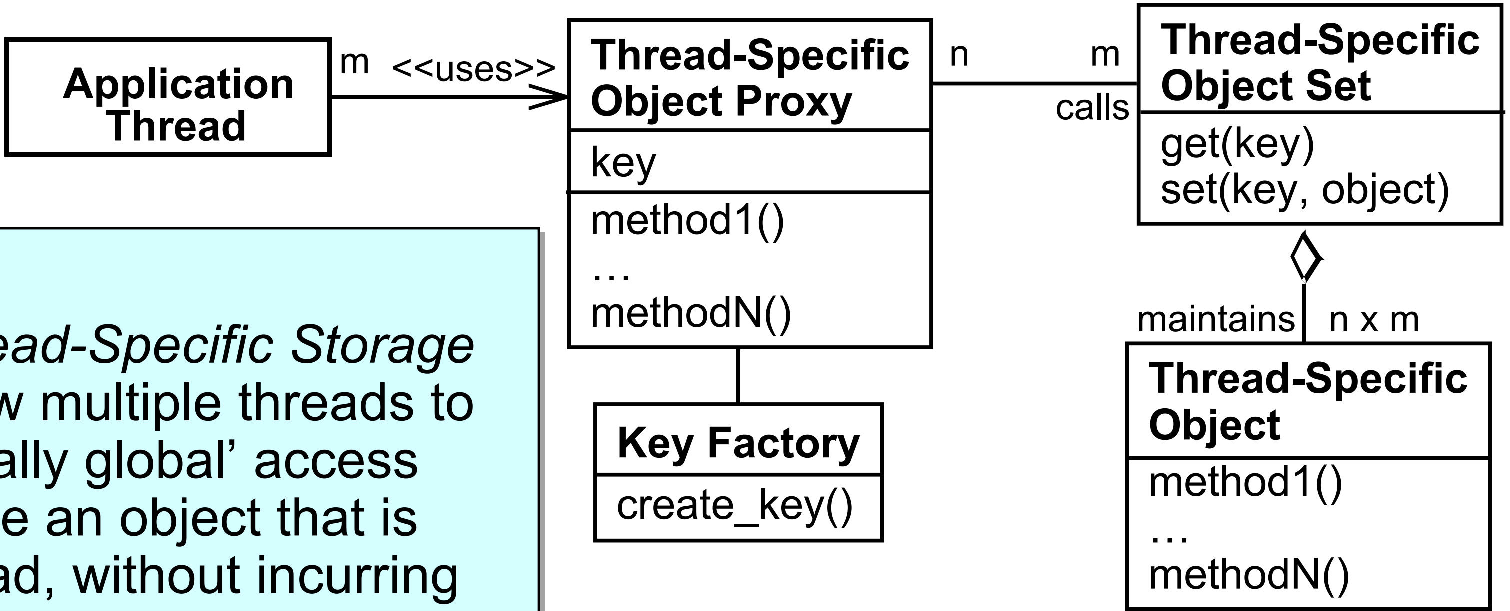
Logging Access Statistics Efficiently

Context

- Web servers often need to log certain information
  - *e.g.*, count number of times web pages are accessed

Problem

- Having a central logging object in a multi-threaded server process can become a bottleneck
  - *e.g.*, due to synchronization required to serialize access by multiple threads

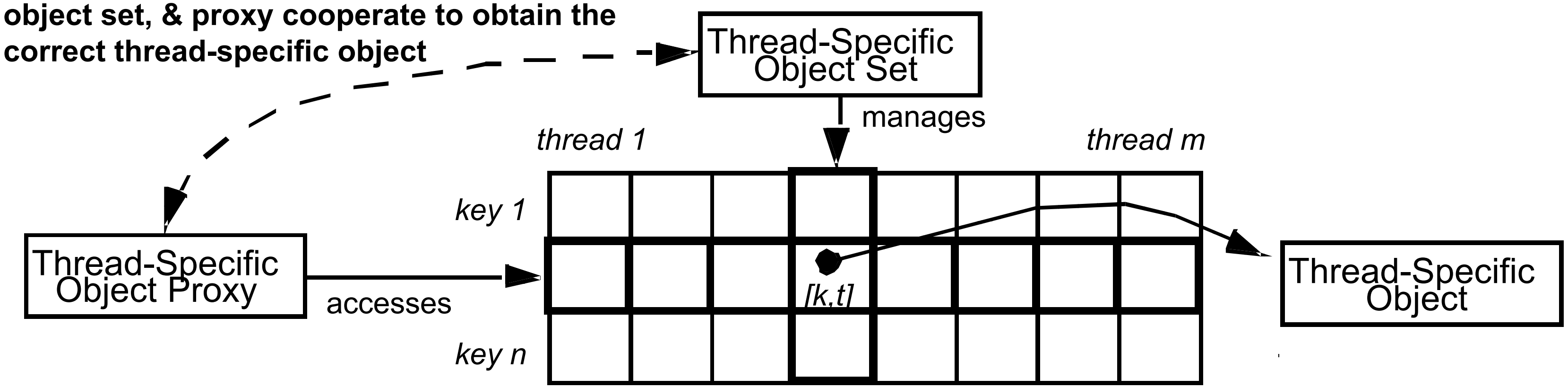


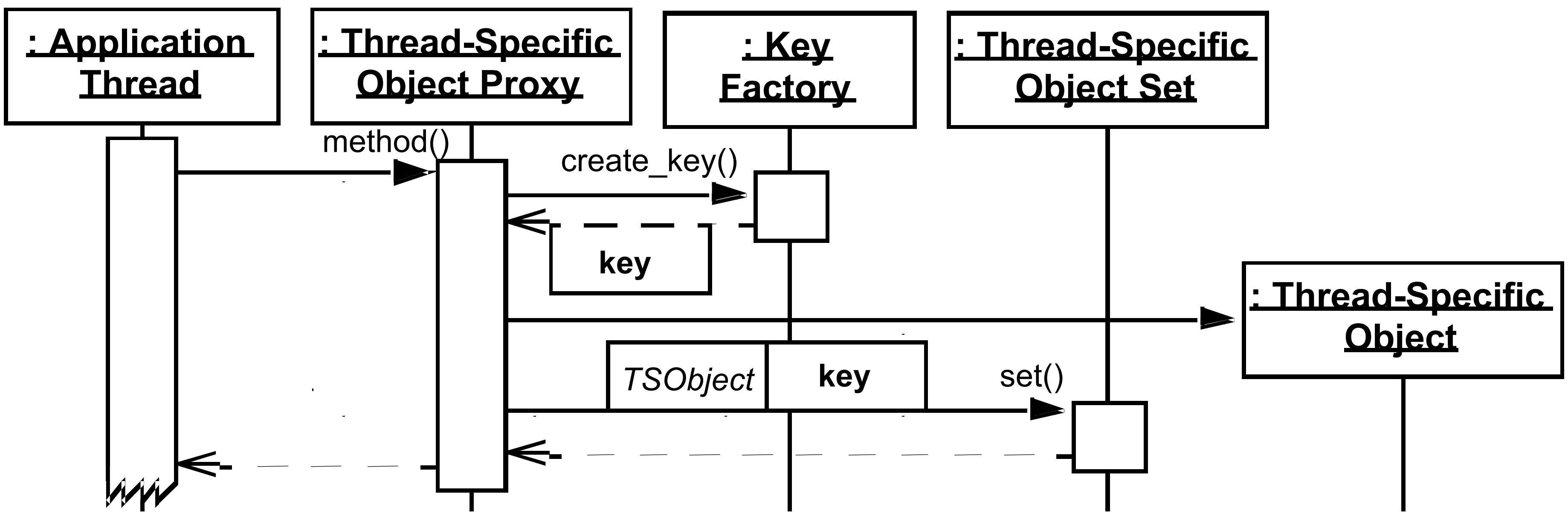
Solution

- Apply the *Thread-Specific Storage* pattern to allow multiple threads to use one 'logically global' access point to retrieve an object that is local to a thread, without incurring locking overhead on each object access

Thread-Specific Storage Pattern Dynamics

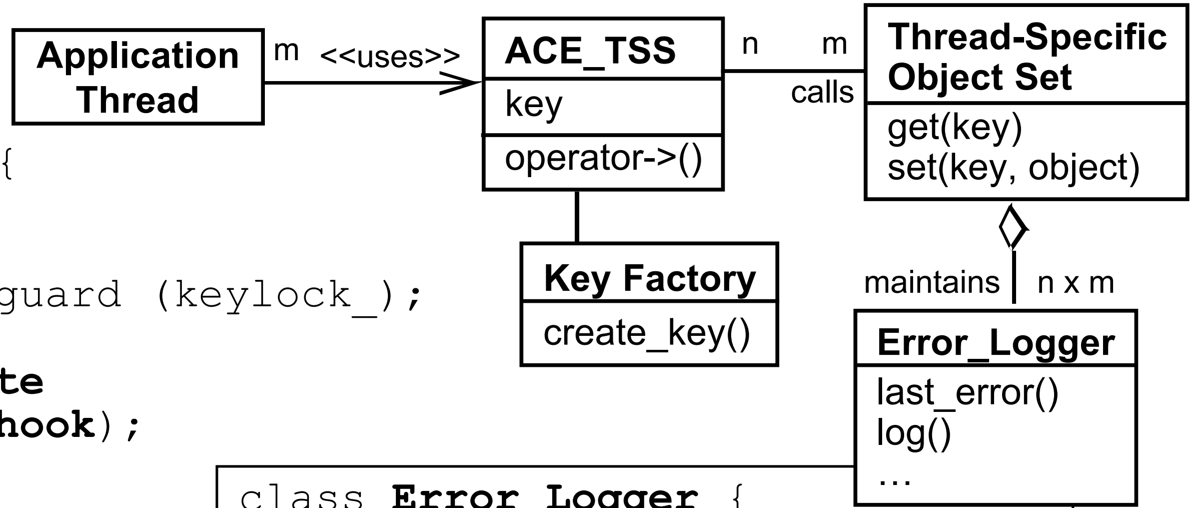
The application thread identifier, thread-specific object set, & proxy cooperate to obtain the correct thread-specific object





Applying the Thread-Specific Storage Pattern to JAWS

```
template <class TYPE>
Class ACE_TSS {
public:
  TYPE *operator->() const {
    TYPE *tss_data = 0;
    if (!once_) {
      Guard <Thread_Mutex> guard (keylock_);
      if (!once_) {
        ACE_OS::thr_keycreate
          (&key_, &cleanup_hook);
        once_ = true;
      }
    }
    ACE_OS::thr_getspecific
      (key, (void **) &tss_data);
    if (tss_data == 0) {
      tss_data = new TYPE;
      ACE_OS::thr_setspecific
        (key, (void *) tss_data);
    }
    return tss_data;
  }
private:
  mutable pthread_key_t key_;
  mutable bool once_;
  mutable Thread_Mutex keylock_;
  static void cleanup_hook (void *ptr);
};
```




```
class Error_Logger {
public:
  int last_error ();
  void log (const char *format,
            ...);
};
```

```
ACE_TSS <Error_Logger>
   my_logger;
// ...
if (recv (……) == -1 &&
    my_logger->last_error () !=
    EWOULDBLOCK)
   my_logger->log
    (“recv failed, errno = %d”,
     my_logger->last_error ());
};
```

Pros & Cons of the Thread-Specific Storage Pattern

This pattern has four **benefits**:

- ***Efficiency***
  - It's possible to implement this pattern so that no locking is needed to access thread-specific data
- ***Ease of use***
  - When encapsulated with wrapper facades, thread-specific storage is easy for application developers to use
- ***Reusability***
  - By combining this pattern with the Wrapper Façade pattern it's possible to shield developers from non-portable OS platform characteristics
- ***Portability***
  - It's possible to implement portable thread-specific storage mechanisms on most multi-threaded operating systems

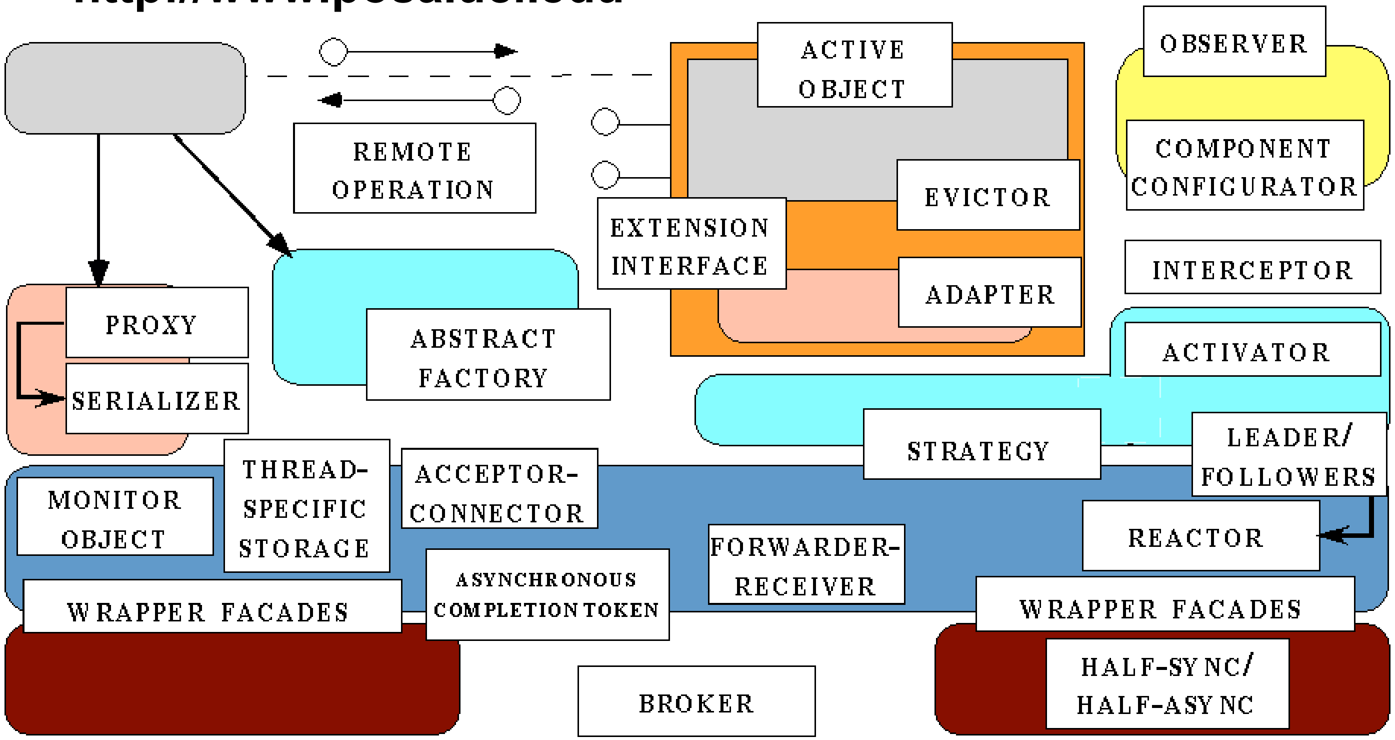
This pattern also has **liabilities**:

- ***It encourages use of thread-specific global objects***
  - Many applications do not require multiple threads to access thread-specific data via a common access point
  - In this case, data should be stored so that only the thread owning the data can access it
- ***It obscures the structure of the system***
  - The use of thread-specific storage potentially makes an application harder to understand, by obscuring the relationships between its components
- ***It restricts implementation options***
  - Not all languages support parameterized types or smart pointers, which are useful for simplifying the access to thread-specific data

Tutorial Example 3:
Applying Patterns to Real-time CORBA

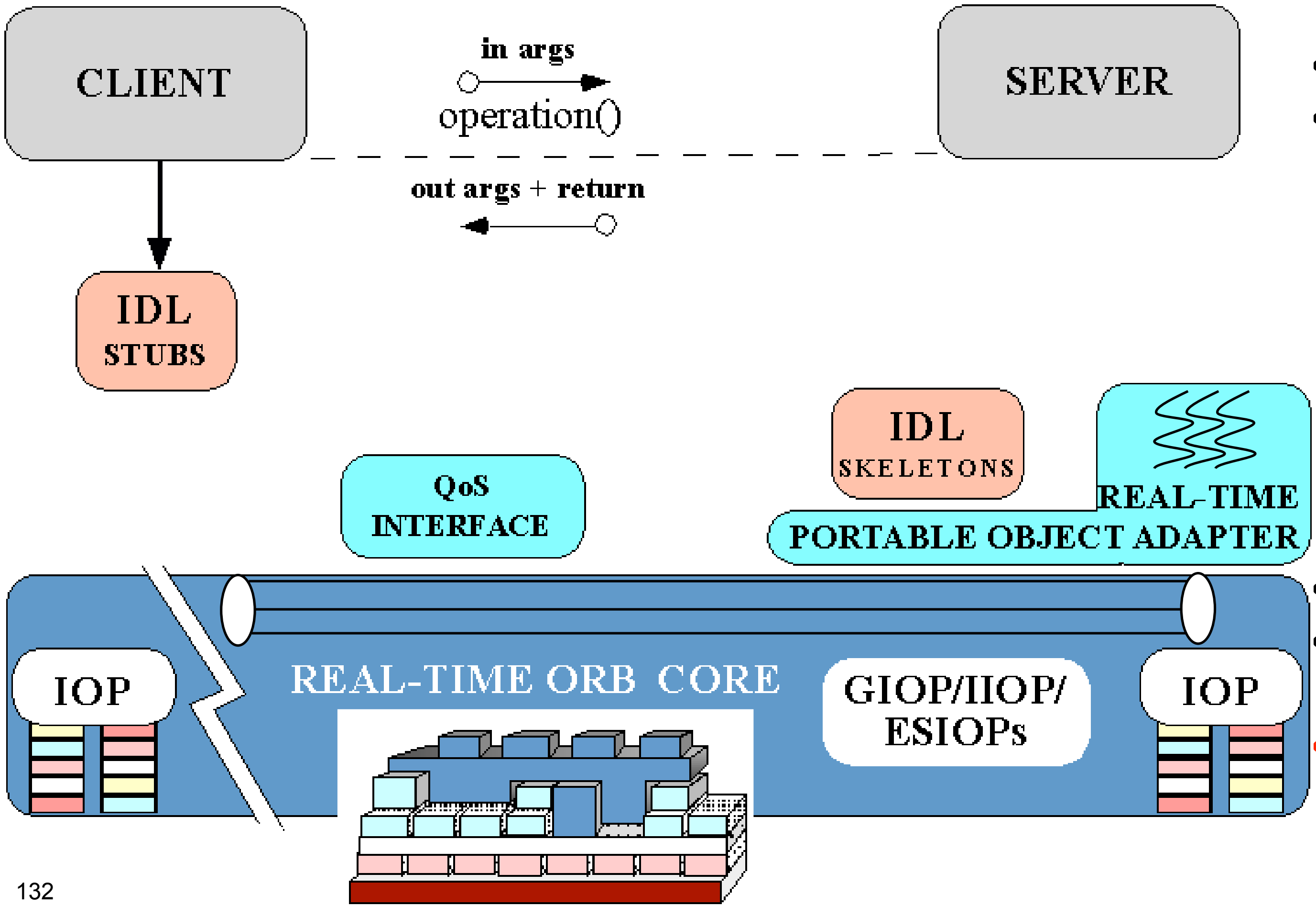
http://www.posa.uci.edu

UML models of a software architecture can illustrate *how* a system is designed, but not *why* the system is designed in a particular way



Patterns are used throughout *The ACE ORB (TAO)* Real-time CORBA implementation to codify expert knowledge & to generate the ORB's software architecture by capturing recurring structures & dynamics & resolving common design forces

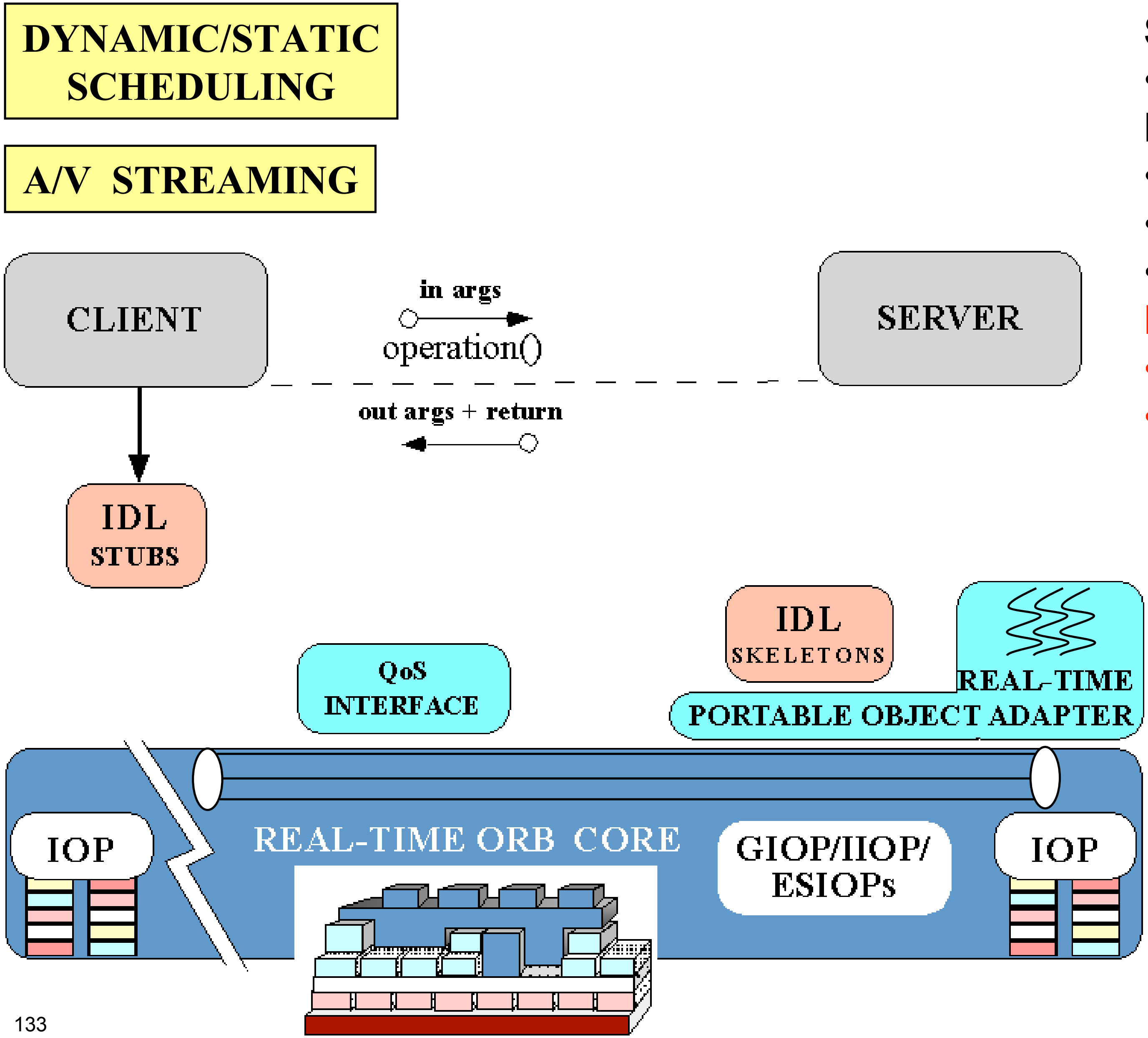
The Evolution of TAO



TAO ORB

- Compliant with CORBA 2.4 & some CORBA 3.0
  - AMI
  - INS
  - Portable Interceptors
- Pattern-centric design
- Key capabilities
  - QoS-enabled
  - Configurable
  - Pluggable protocols
    - IIOP
    - UIOP
    - Shared memory
    - SSL
    - VME
- Open-source
- Commercially supported
  - www.theaceorb.com
- Available now
  - ZEN (RT Java/RT CORBA)
  - www.zen.uci.edu

The Evolution of TAO



Static Scheduling
- Rate monotonic analysis

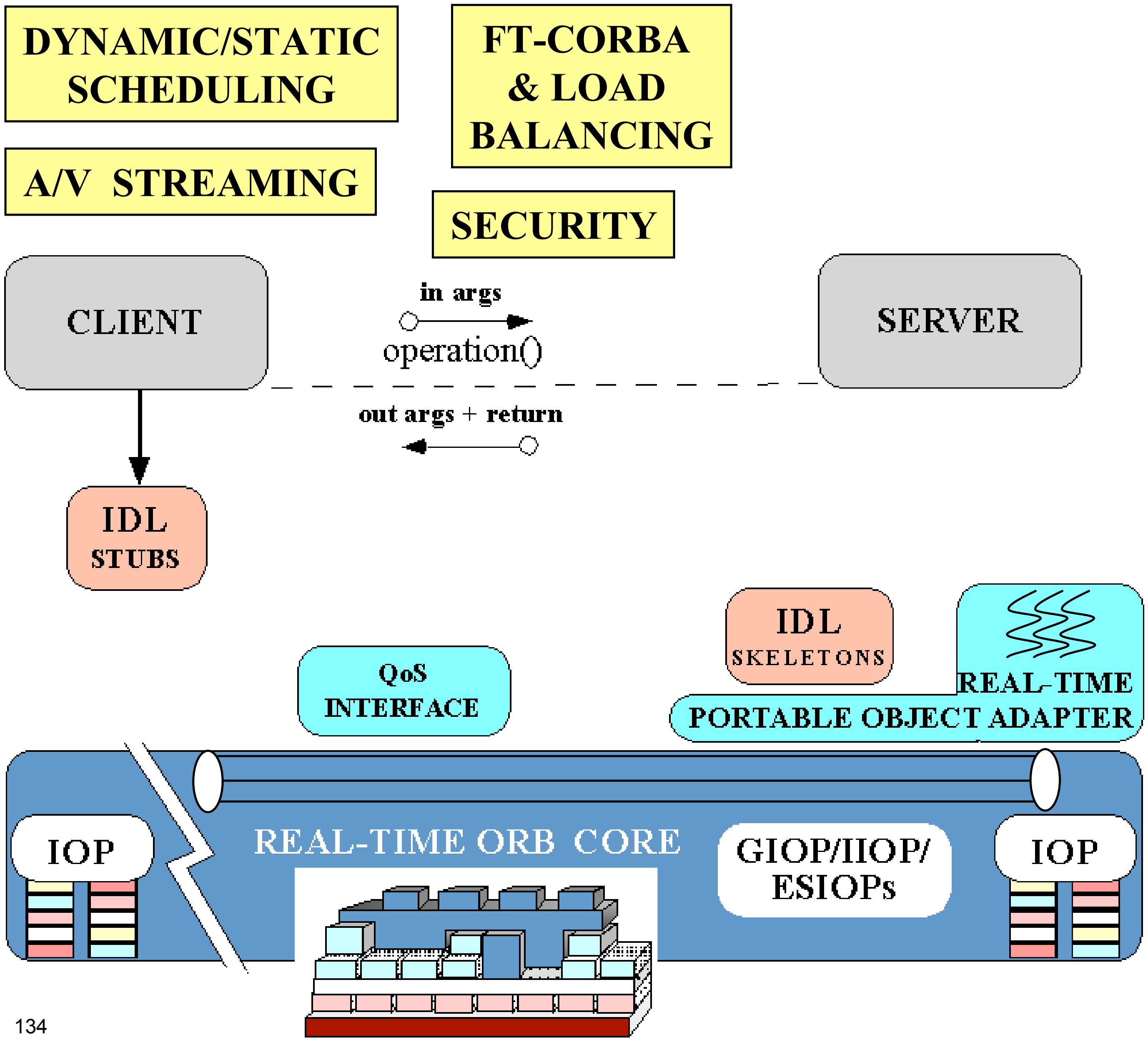
Dynamic Scheduling
- Earliest deadline first
- Minimum laxity first
- Maximal urgency first

Hybrid Dynamic/Static
- Demo in WSOA
- ETA Winter 2001

A/V Streaming Service
- QoS mapping
- QoS monitoring
- QoS adaptation

ACE QoS API (AQoSA)
- GQoS + RAPI
- Integration with A/V Streaming ETA Winter 2001

The Evolution of TAO



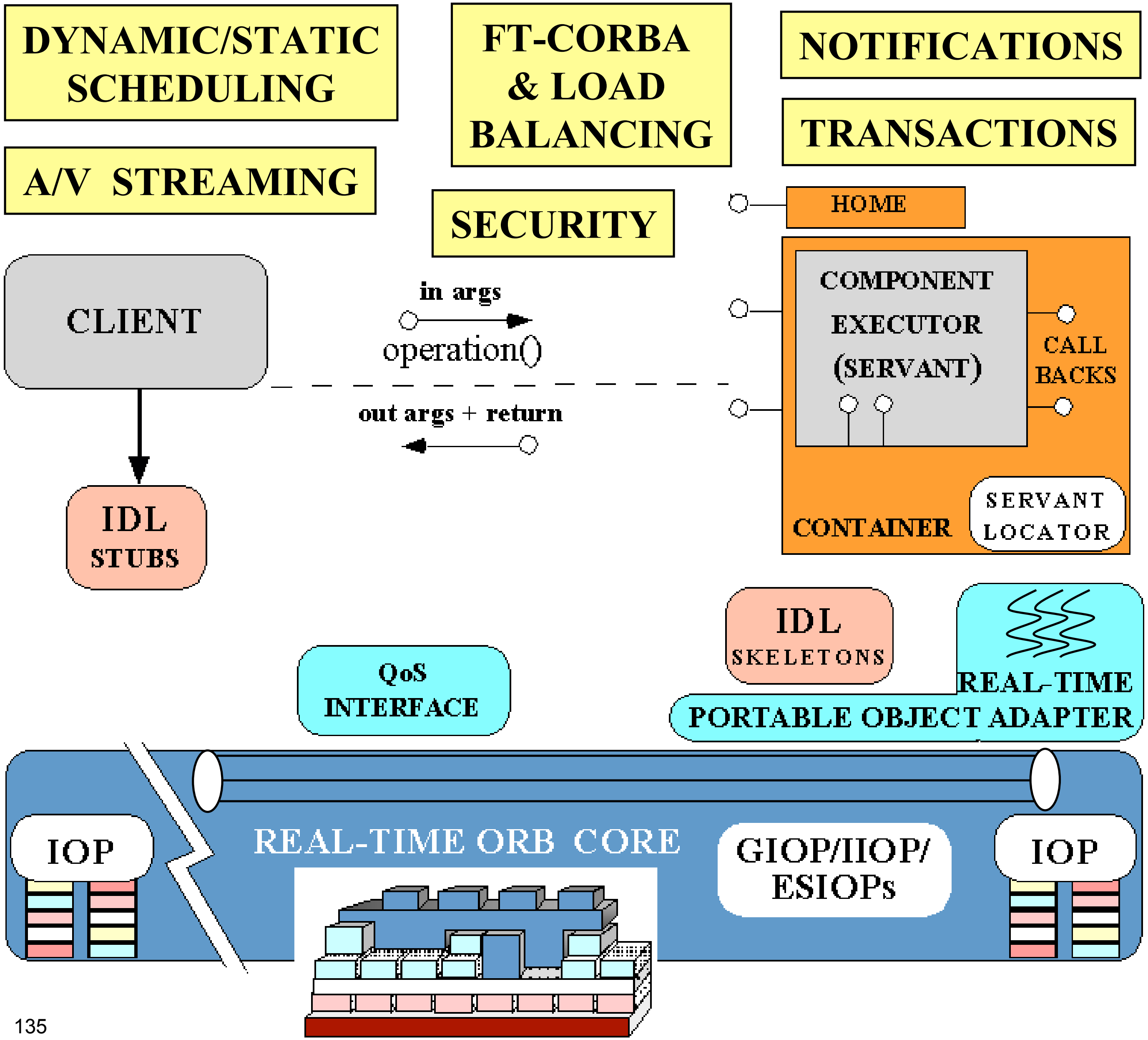
FT-CORBA
- Entity redundancy
- Multiple models
  - Cold passive
  - Warm passive
  - Active
- IOGR
- ETA Winter 2001

Load Balancing
- Static & dynamic
- LOCATION_FORWARDING

SSL Support
- Integrity
- Confidentiality
- Authentication (limited)

Security Service
- Authentication
- Access control
- Non-repudiation
- Audit
- ETA Winter 2001

The Evolution of TAO



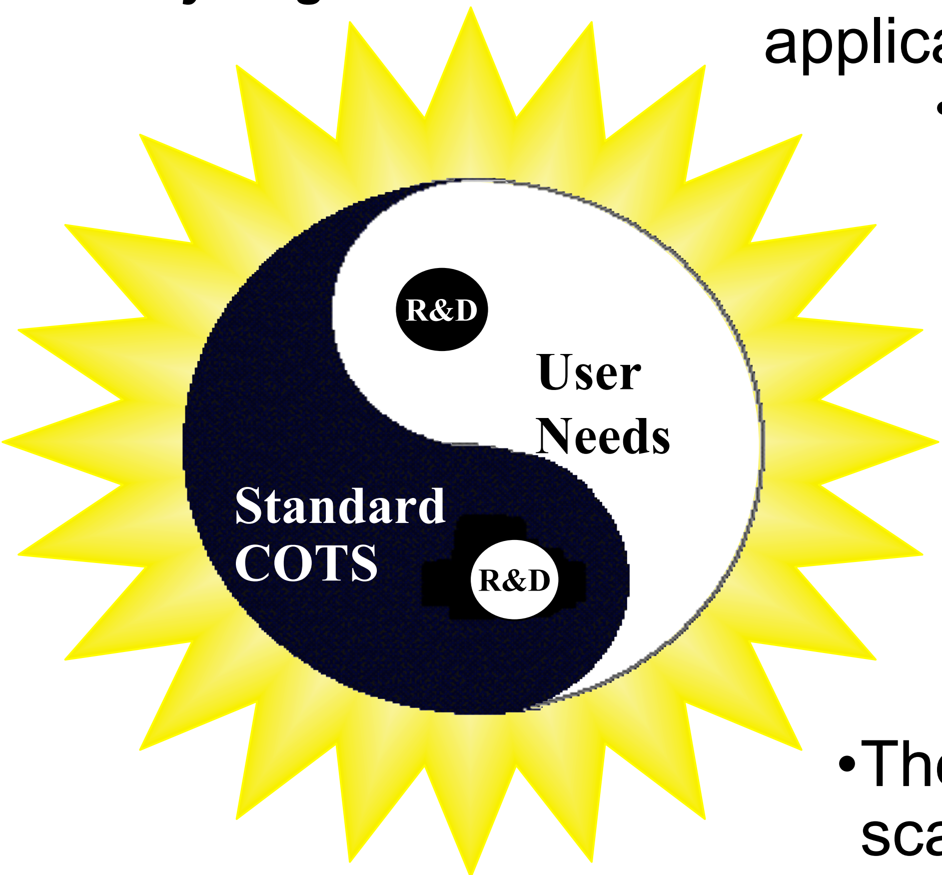
Notification Service
- Structured events
- Event filtering
- QoS properties
  - Priority
  - Expiry times
  - Order policy
- Compatible w/Events

Object Transaction Service
- Encapsulates RDBMs
- ETA Winter 2001

CORBA Component Model (CCM)
- Extension Interfaces
- Component navigation
- Standardized life-cycles
- Dynamic configuration
- QoS-enabled containers
- Reflective collocation
- ETA Winter 2001

Concluding Remarks

R&D Synergies



- Researchers & developers of distributed applications face common challenges
  - *e.g., connection management, service initialization, error handling, flow & congestion control, event demuxing, distribution, concurrency control, fault tolerance synchronization, scheduling, & persistence*
  - *Patterns*, *frameworks*, & *components* help to resolve these challenges
- These techniques can yield efficient, scalable, predictable, & flexible middleware & applications

"Secrets" to R&D success:

- Embrace & lead COTS standards
- Leverage open-source
- Be entrepreneurial & use the Web
- Solve "real" problems
- See ideas thru to completion
- Leave an enduring legacy