Pattern-Oriented Software Architectures
Patterns & Frameworks for
Concurrent & Distributed Systems

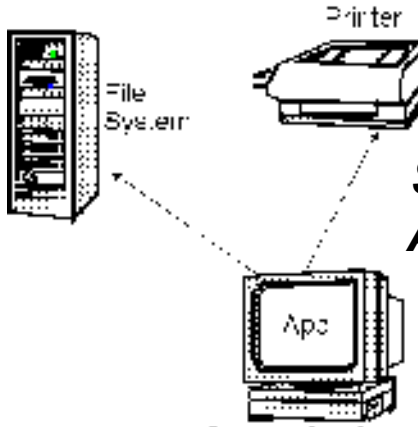
Dr. Douglas C. Schmidt
d.schmidt@vanderbilt.edu
http://www.dre.vanderbilt.edu/~schmidt/

Professor of EECS
Vanderbilt University
Nashville, Tennessee

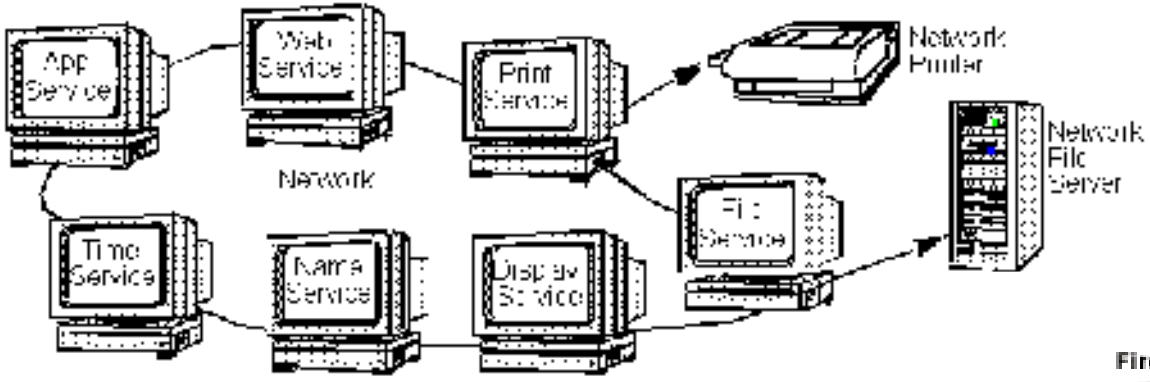
Tutorial Motivation

Observations

- Building robust, efficient, & extensible concurrent & networked applications is hard
  - *e.g.,* we must address many complex topics that are less problematic for non-concurrent, stand-alone applications

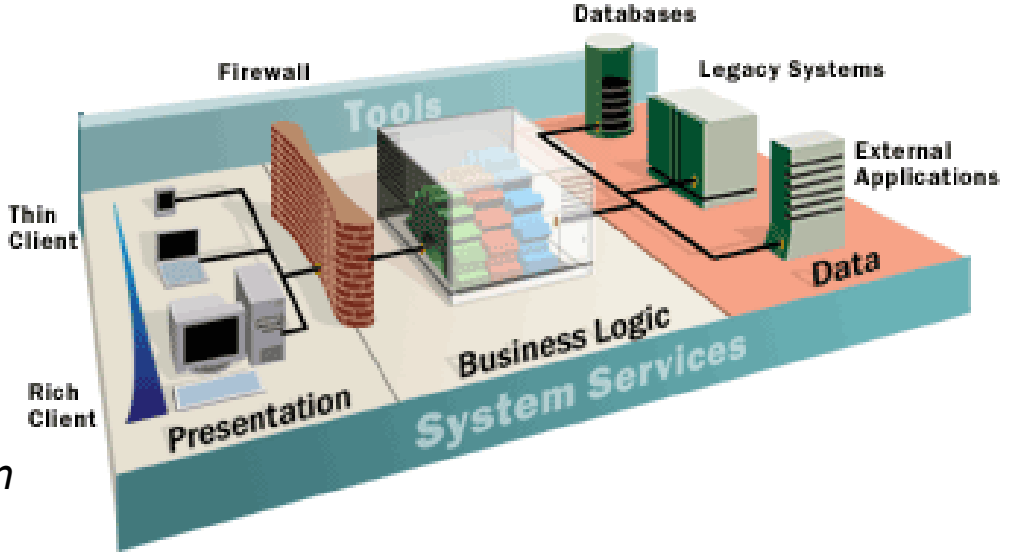


Stand-alone Architecture



Networked Architecture

- Fortunately, there are reusable solutions to many common challenges, e.g.:
  - *Connection mgmt & event demuxing*
  - *Service initialization*
  - *Error handling & fault tolerance_*
  - *Flow & congestion control*
  - *Distribution*
  - *Concurrency, scheduling, & synchronization*
  - *Persistence*

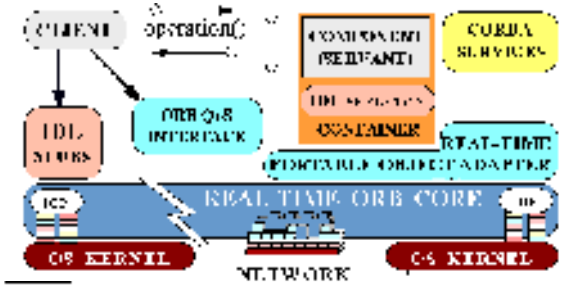


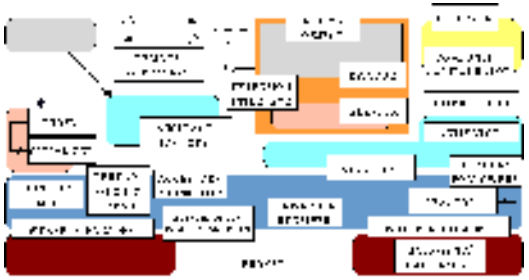
Tutorial Outline

Cover OO techniques & language features that enhance software quality

OO techniques & language features:

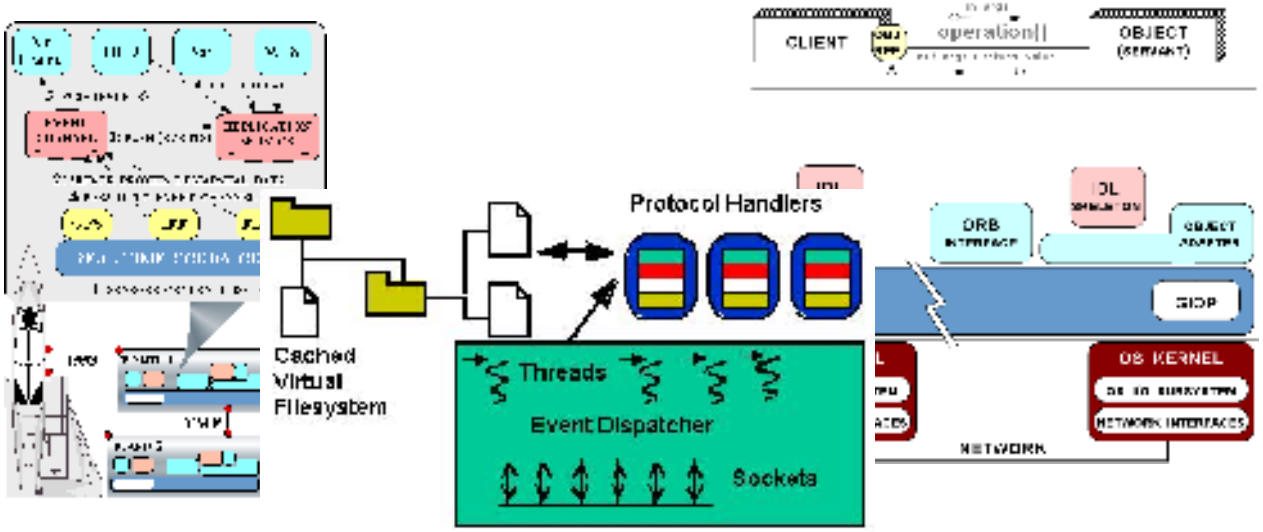
- ***Frameworks & components***, which embody reusable software middleware & application *implementations*
- ***Patterns*** *(25+)*, which embody reusable software *architectures* & *designs*
- ***OO language features****, e.g., classes, dynamic binding & inheritance, parameterized types*



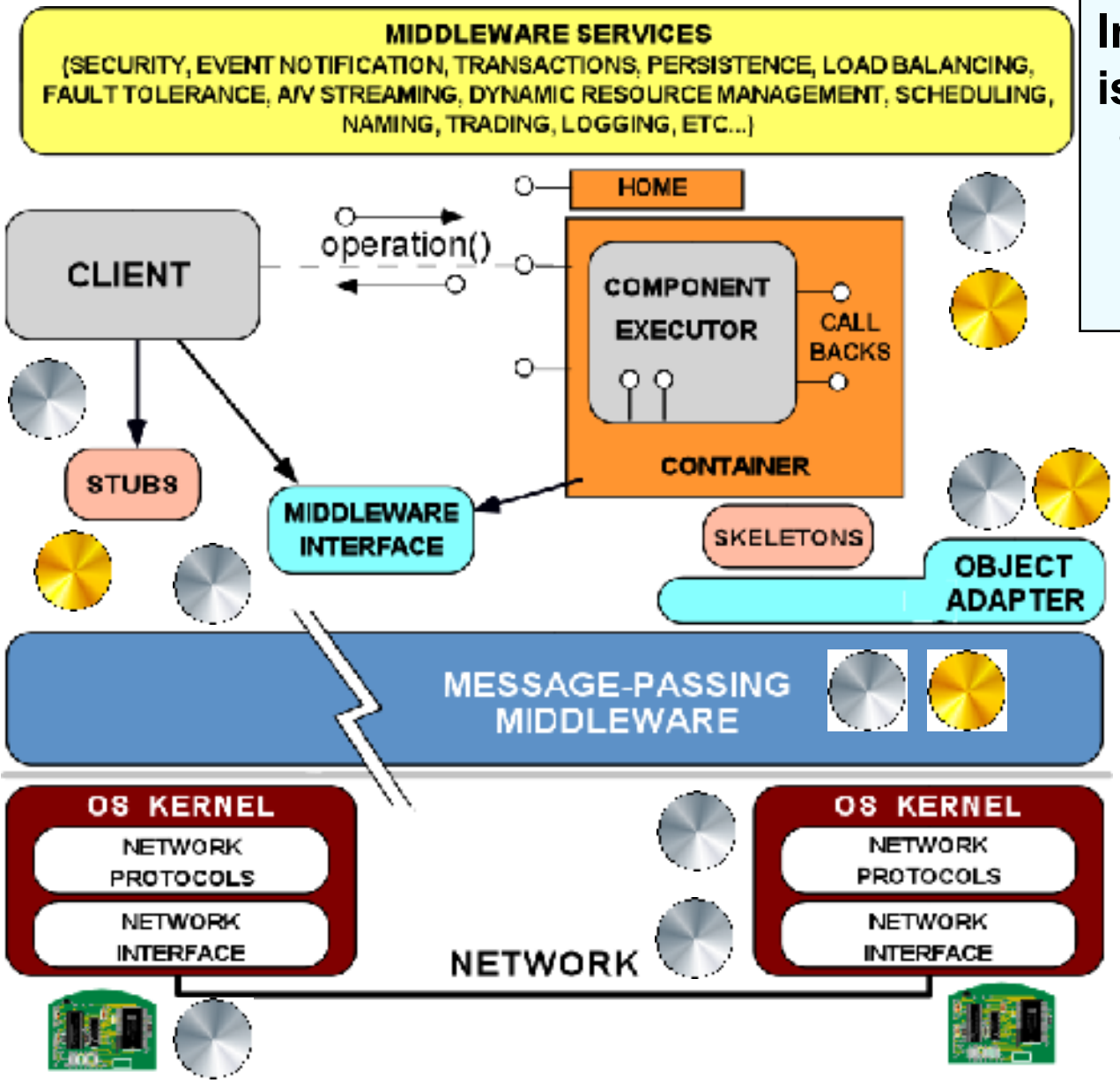


Tutorial Organization

2. *Technology trends & background*
3. *Concurrent & network challenges & solution approaches*
4. *Case studies*
5. *Wrap-up*



Technology Trends (1/4)



Information technology is being commoditized

- i.e., hardware & software are getting cheaper, faster, & (generally) better at a fairly predictable rate

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

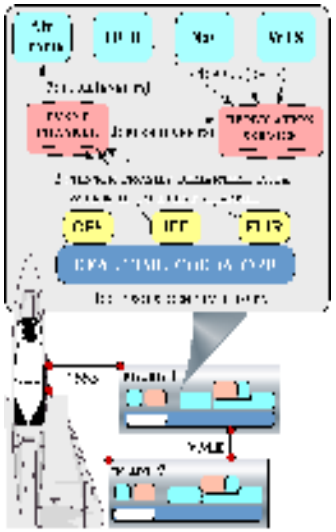
- Intel x86 & Power PC chipsets
- TCP/IP, GSM, Link16
- POSIX, Windows, & VMs
- Middleware & component models
- Quality of service (QoS) aspects

Technology Trends (2/4)

Growing acceptance of a network-centric component paradigm

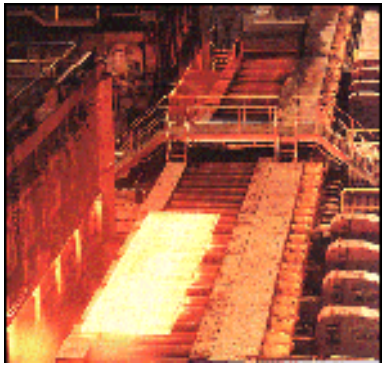
- i.e., distributed applications with a range of QoS needs are constructed by integrating components & frameworks via various communication mechanisms

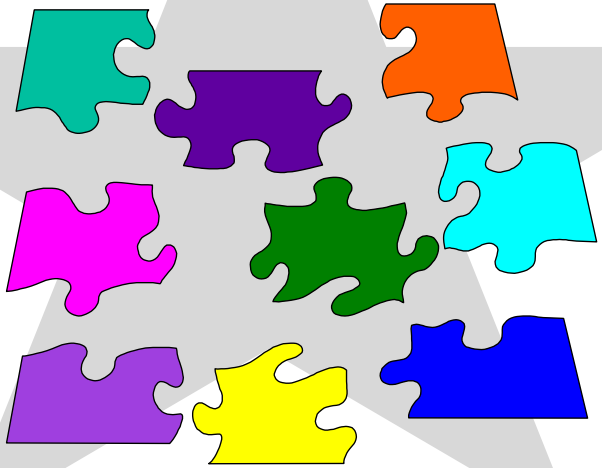
Avionics Mission Computing



Process Automation Quality Control

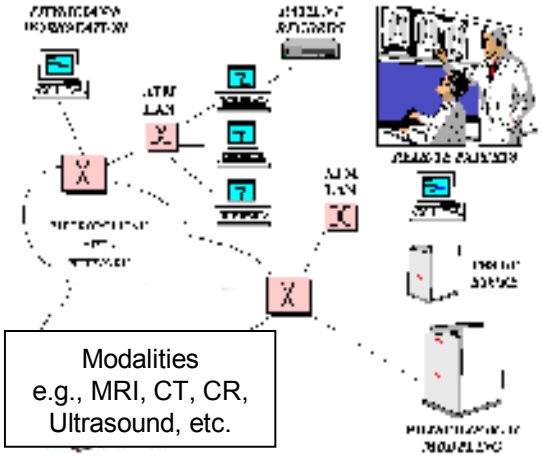
Hot Rolling Mills





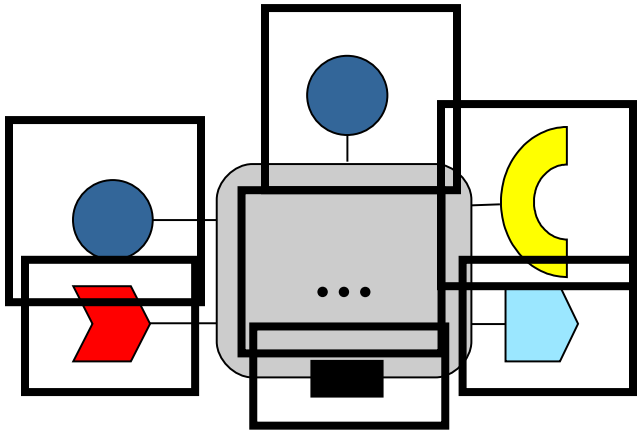
Software Defined Radio

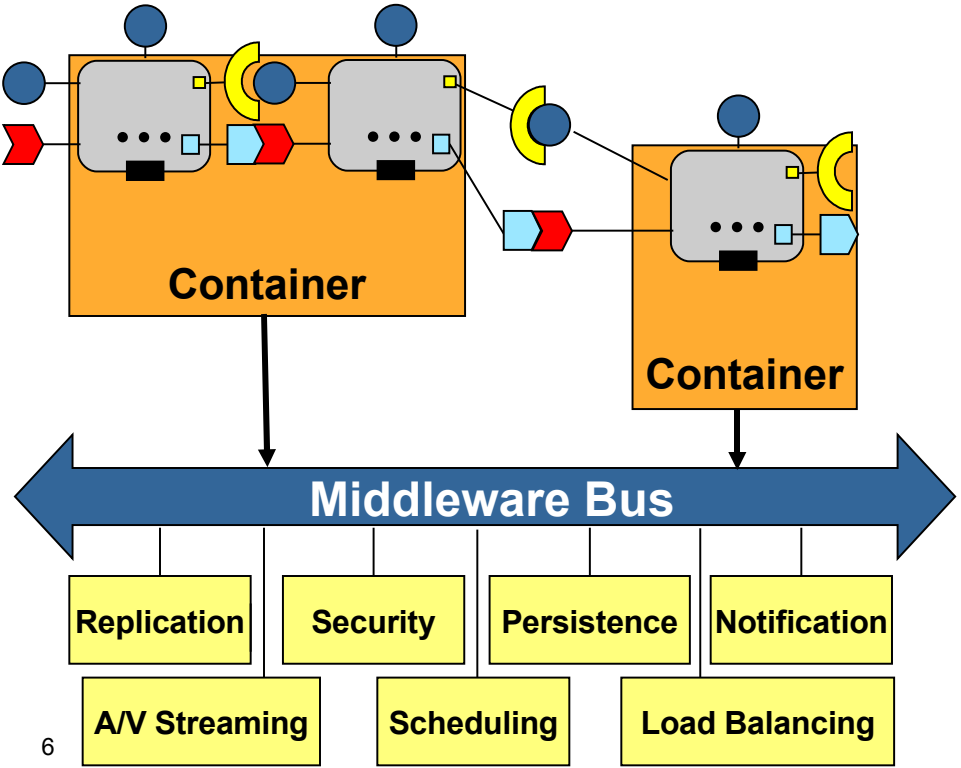
Electronic Medical Imaging



Modalities
e.g., MRI, CT, CR, Ultrasound, etc.

Technology Trends (3/4)



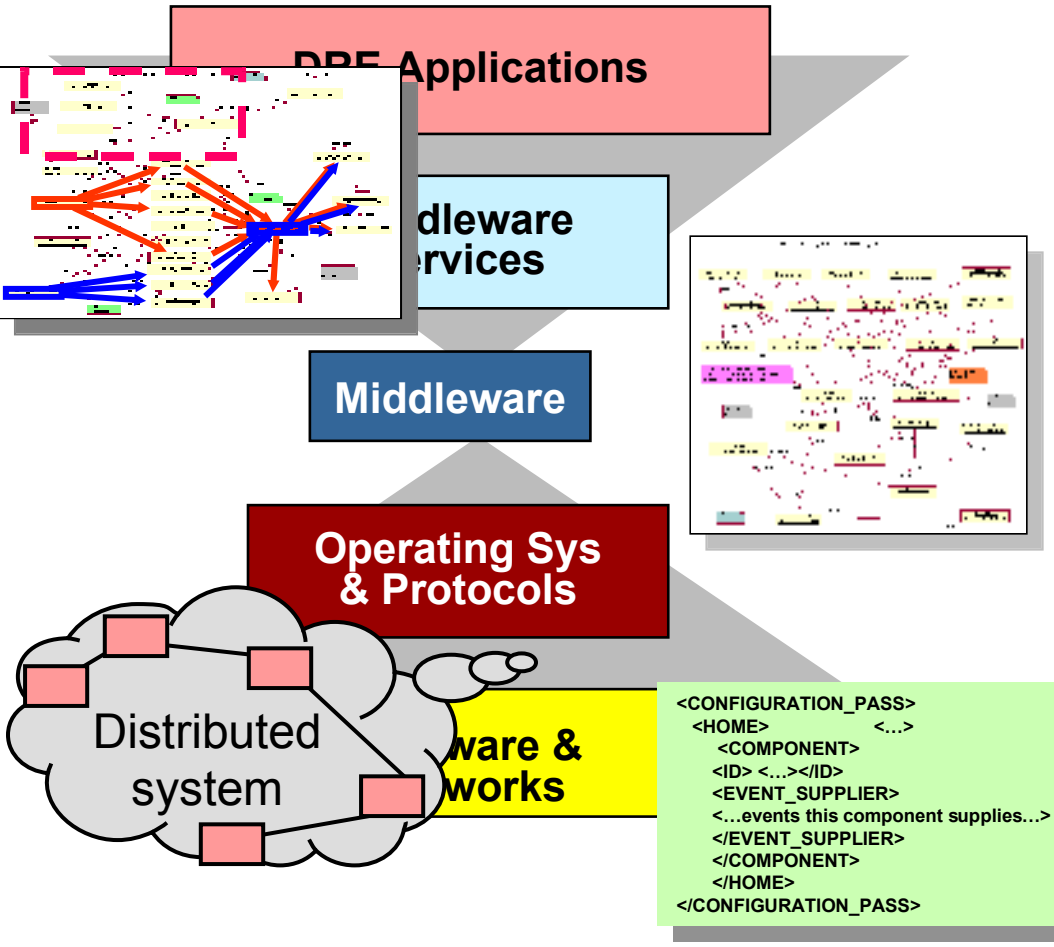


Component middleware is maturing & becoming pervasive

- *Components* encapsulate application "business" logic
- Components interact via *ports*
  - ***Provided interfaces***, e.g.,facets
  - ***Required connection points***, e.g., receptacles
  - ***Event sinks & sources***
  - ***Attributes***
- *Containers* provide execution environment for components with common operating requirements
- Components/containers can also
  - Communicate via a ***middleware bus*** and
  - Reuse ***common middleware services***

Technology Trends (4/4)

Model driven development is integrating generative software technologies with QoS-enabled component middleware

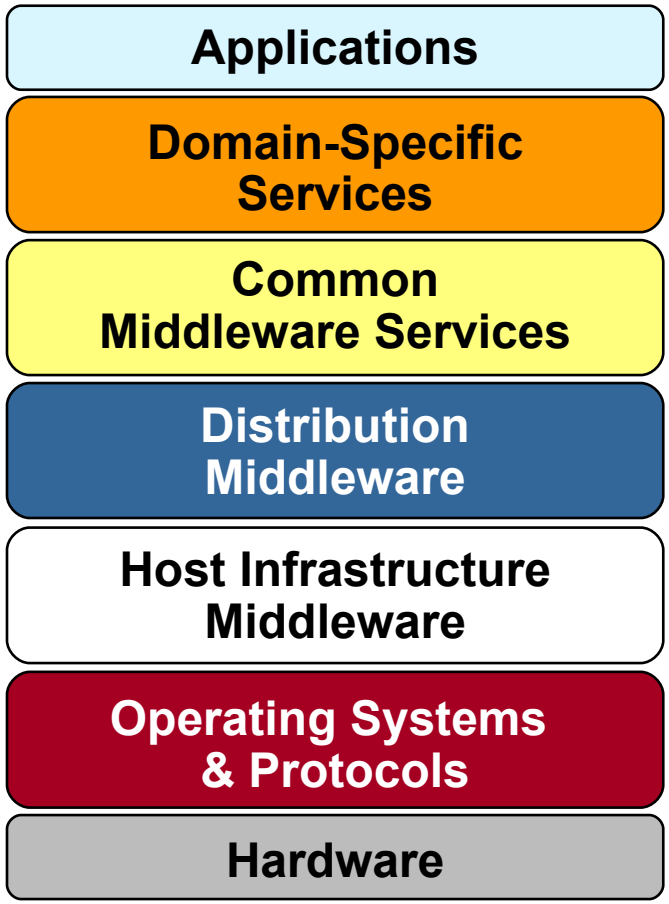


- e.g., standard technologies are emerging that can:
  1. ***Model***
  2. ***Analyze***
  3. ***Synthesize & optimize***
  4. ***Provision & deploy***

  multiple layers of QoS-enabled middleware & applications
- These technologies are guided by patterns & implemented by component frameworks
- Partial specialization is essential for inter-/intra-layer optimization

Goal is ***not*** to replace programmers per se – it is to provide ***higher-level domain-specific languages*** for middleware developers & users

The Evolution of Middleware



There are multiple COTS middleware layers & research/business opportunities

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

- Tedious, error-prone, & costly over lifecycles

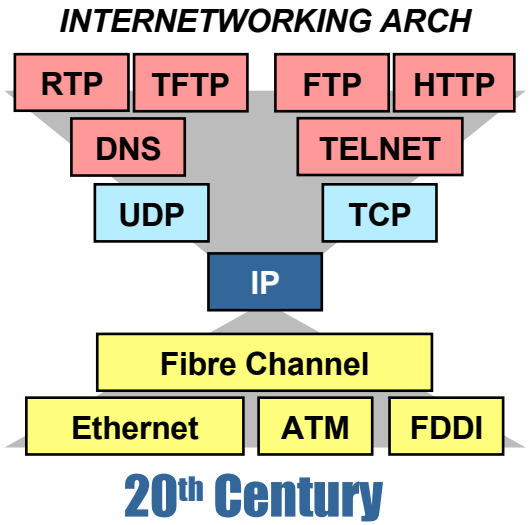
There are layers of middleware, just like there are layers of networking protocols

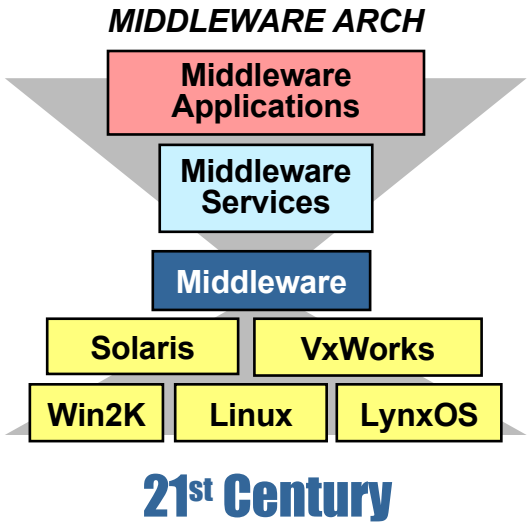
Standards-based COTS middleware helps:

- Control end-to-end resources & QoS
- Leverage hardware & software technology advances
- Evolve to new environments & requirements
- Provide a wide array of reusable, off-the-shelf developer-oriented services

Operating System & Protocols

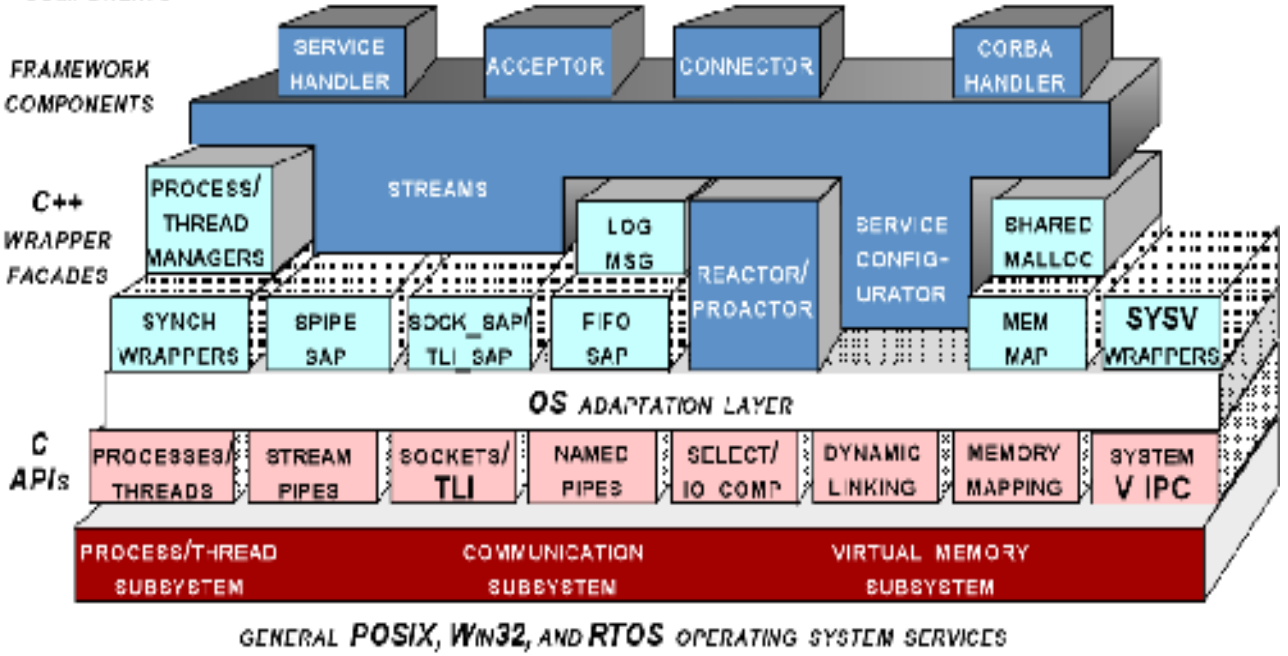
- **Operating systems & protocols** provide mechanisms to manage endsystem resources, e.g.,
  - CPU scheduling & dispatching
  - Virtual memory management
  - Secondary storage, persistence, & file systems
  - Local & remote interprocess communication (IPC)
- **OS examples**
  - UNIX/Linux, Windows, VxWorks, QNX, etc.
- **Protocol examples**
  - TCP, UDP, IP, SCTP, RTP, etc.

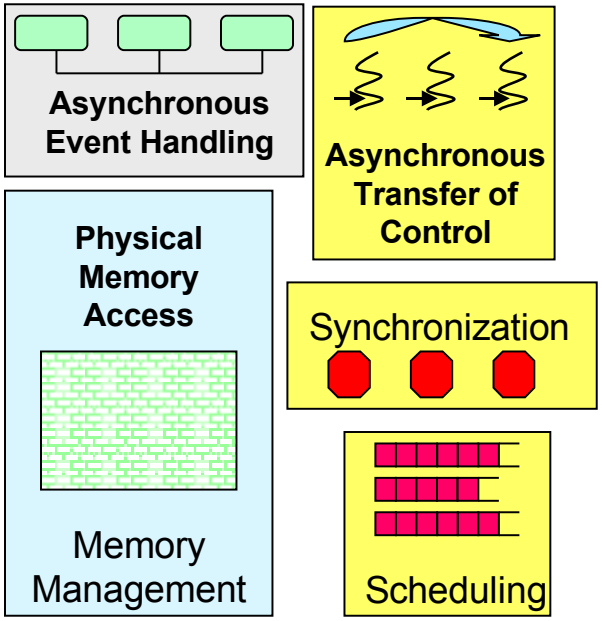




Host Infrastructure Middleware

- **Host infrastructure middleware** encapsulates & enhances native OS mechanisms to create reusable network programming components
  - These components abstract away many tedious & error-prone aspects of low-level OS APIs
- **Examples**
  - Java Virtual Machine (JVM), Common Language Runtime (CLR), ADAPTIVE Communication Environment (ACE)



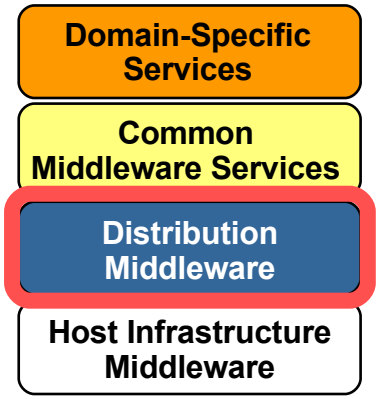


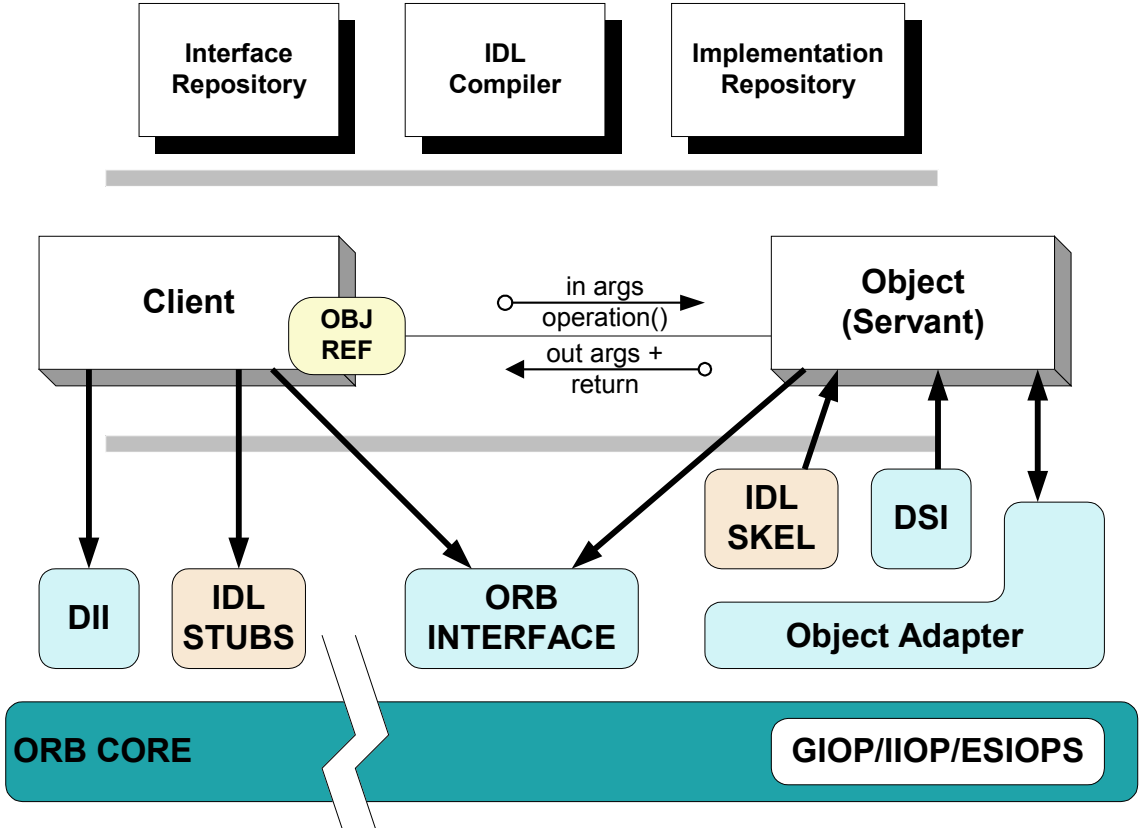
www.rtj.org

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

Distribution Middleware

- ***Distribution middleware*** defines higher-level distributed programming models whose reusable APIs & components automate & extend native OS capabilities
- **Examples**
  - OMG CORBA, Sun's Remote Method Invocation (RMI), Microsoft's Distributed Component Object Model (DCOM)

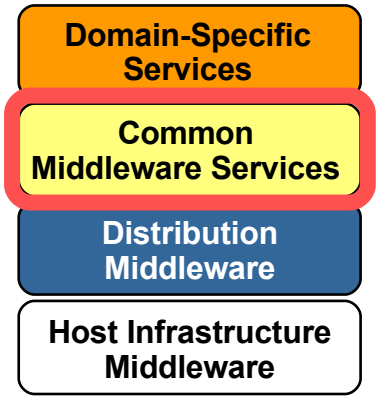


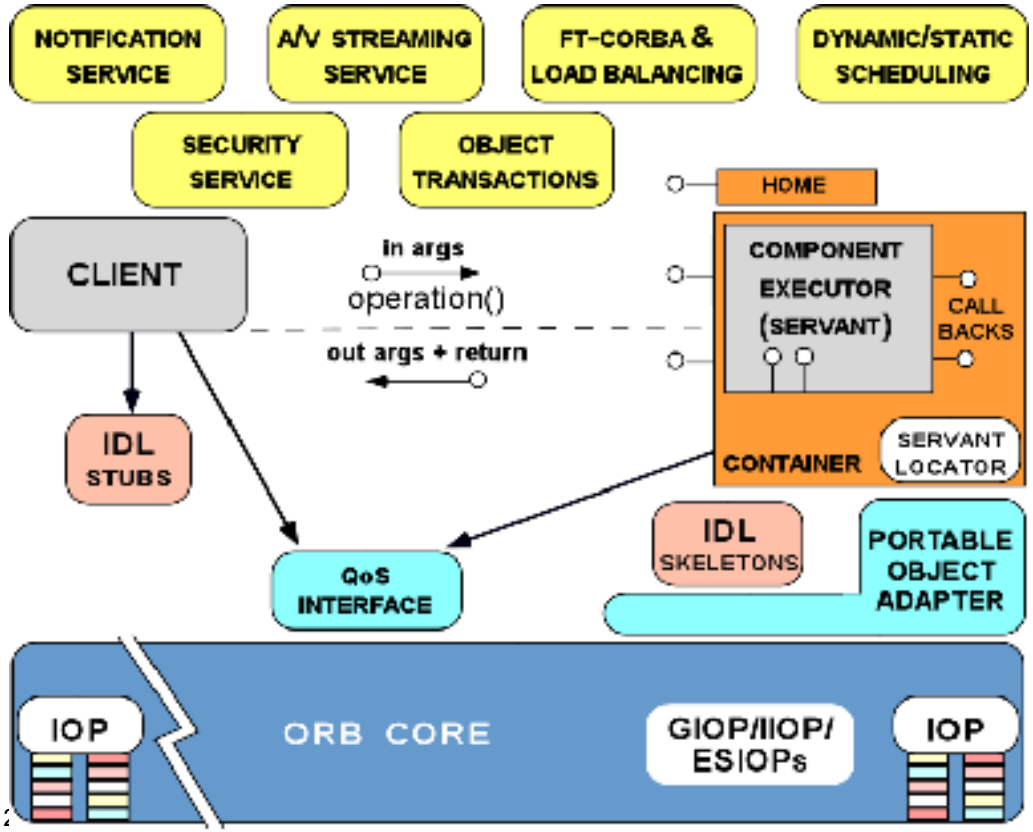


- Distribution middleware avoids hard-coding client & server application dependencies on object location, language, OS, protocols, & hardware

Common Middleware Services

- ***Common middleware services*** augment distribution middleware by defining higher-level domain-independent services that focus on programming "business logic"
- **Examples**
  - CORBA Component Model & Object Services, Sun's J2EE, Microsoft's .NET

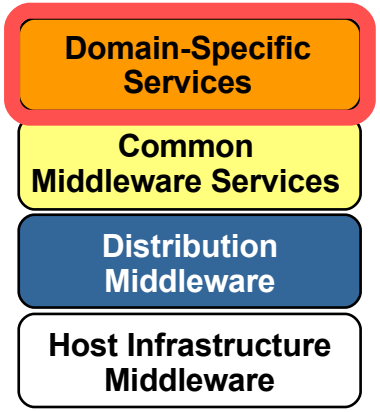




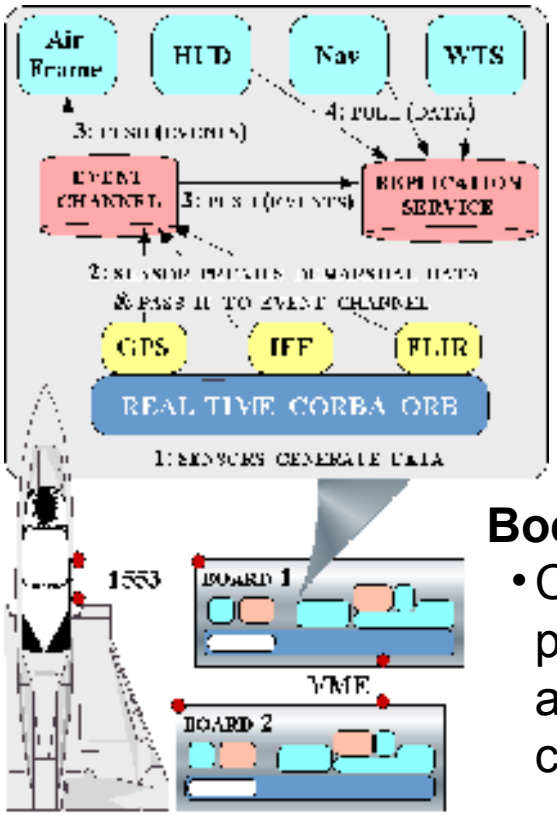
- Common middleware services support many recurring distributed system capabilities, e.g.,
  - Transactional behavior
  - Authentication & authorization,
  - Database connection pooling & concurrency control
  - Active replication
  - Dynamic resource management

Domain-Specific Middleware

- ***Domain-specific middleware services*** are tailored to the requirements of particular domains, such as telecom, e-commerce, health care, process automation, or aerospace



- **Examples**

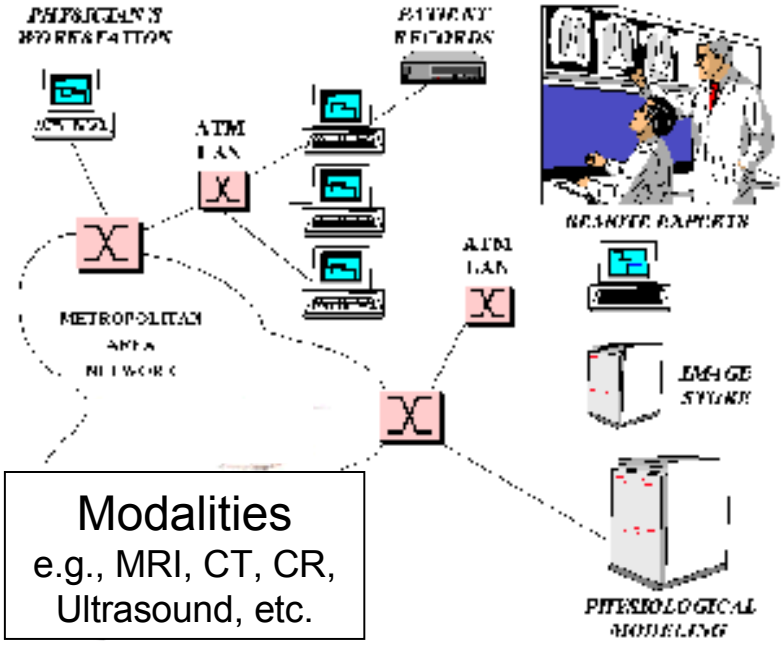


Siemens MED Syngo

- Common software platform for distributed electronic medical systems
- Used by all ~13 Siemens MED business units worldwide

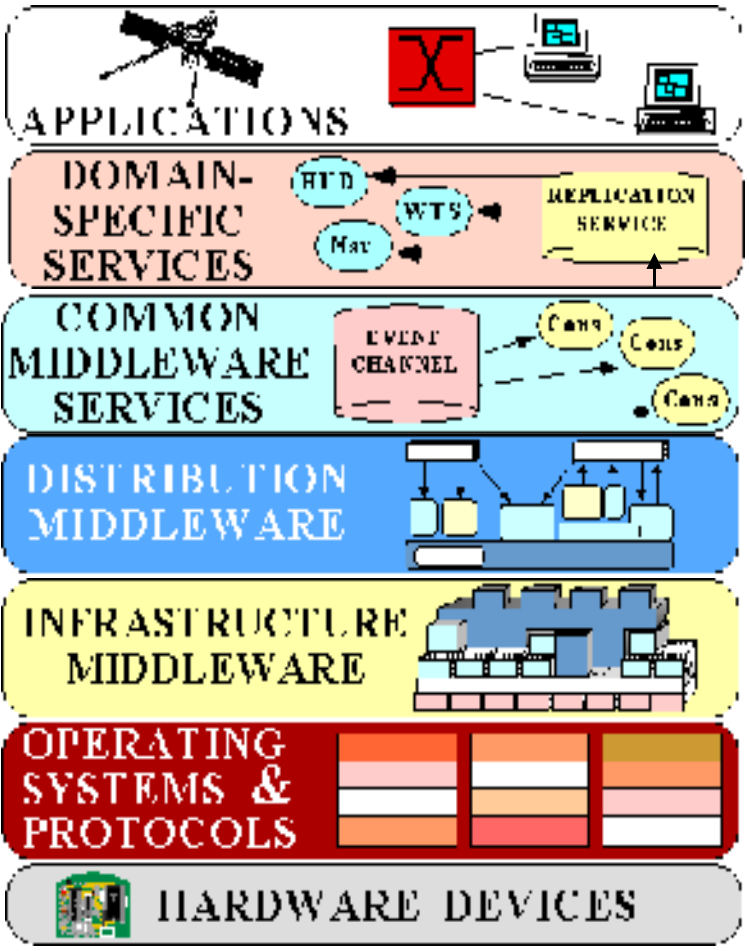
Boeing Bold Stroke

- Common software platform for Boeing avionics mission computing systems



Modalities
e.g., MRI, CT, CR, Ultrasound, etc.

Consequences of COTS & IT Commoditization



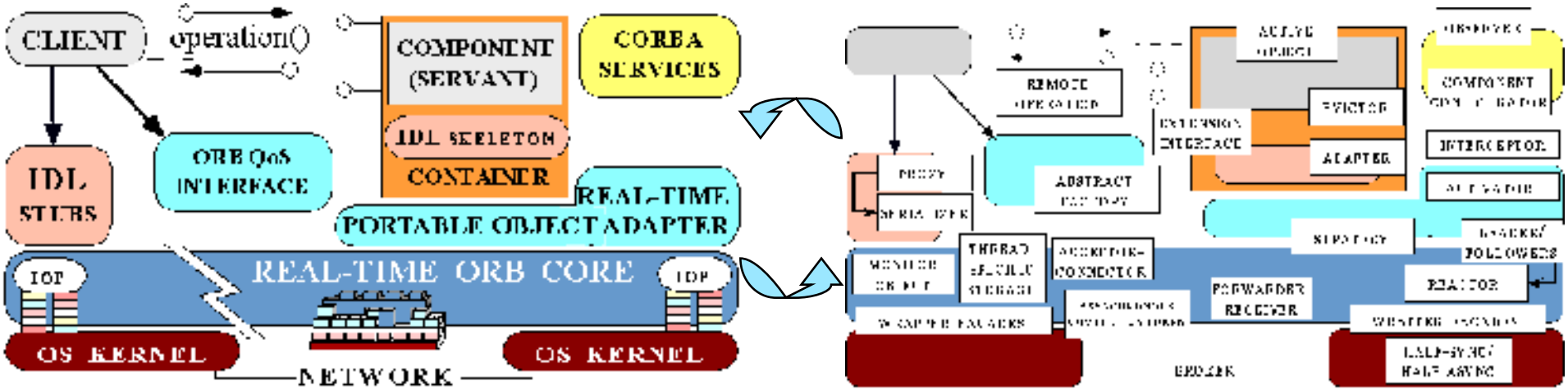
- More emphasis on integration rather than programming
- Increased technology convergence & standardization
- Mass market economies of scale for technology & personnel
- More disruptive technologies & global competition
- Lower priced--but often lower quality--hardware & software components
- The decline 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 depends on success of long-term R&D on *complex* distributed real-time & embedded (DRE) systems

Why We are Succeeding Now

Recent synergistic advances in fundamental technologies & processes:



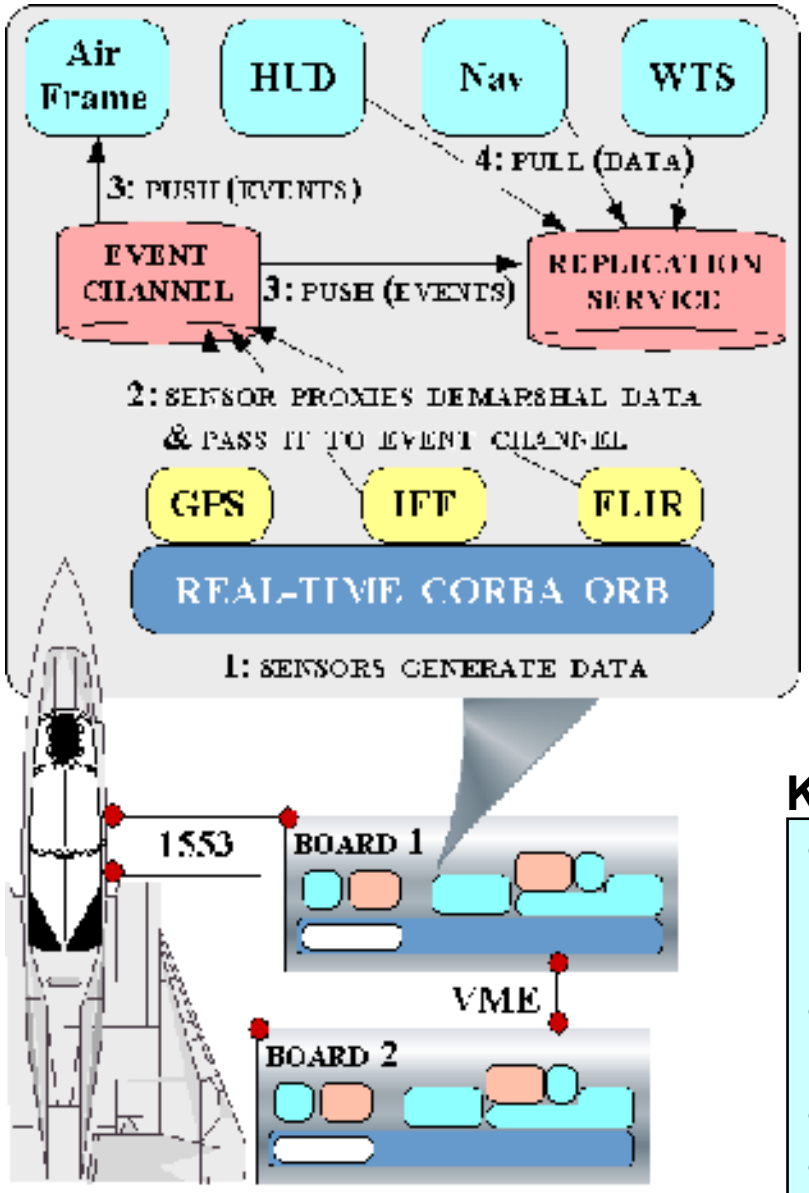
Standards-based QoS-enabled Middleware: Pluggable service & micro-protocol components & reusable "semi-complete" application frameworks

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

Revolutionary changes in software process & methods: Open-source, refactoring, agile methods, advanced V&V techniques, model-driven development

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

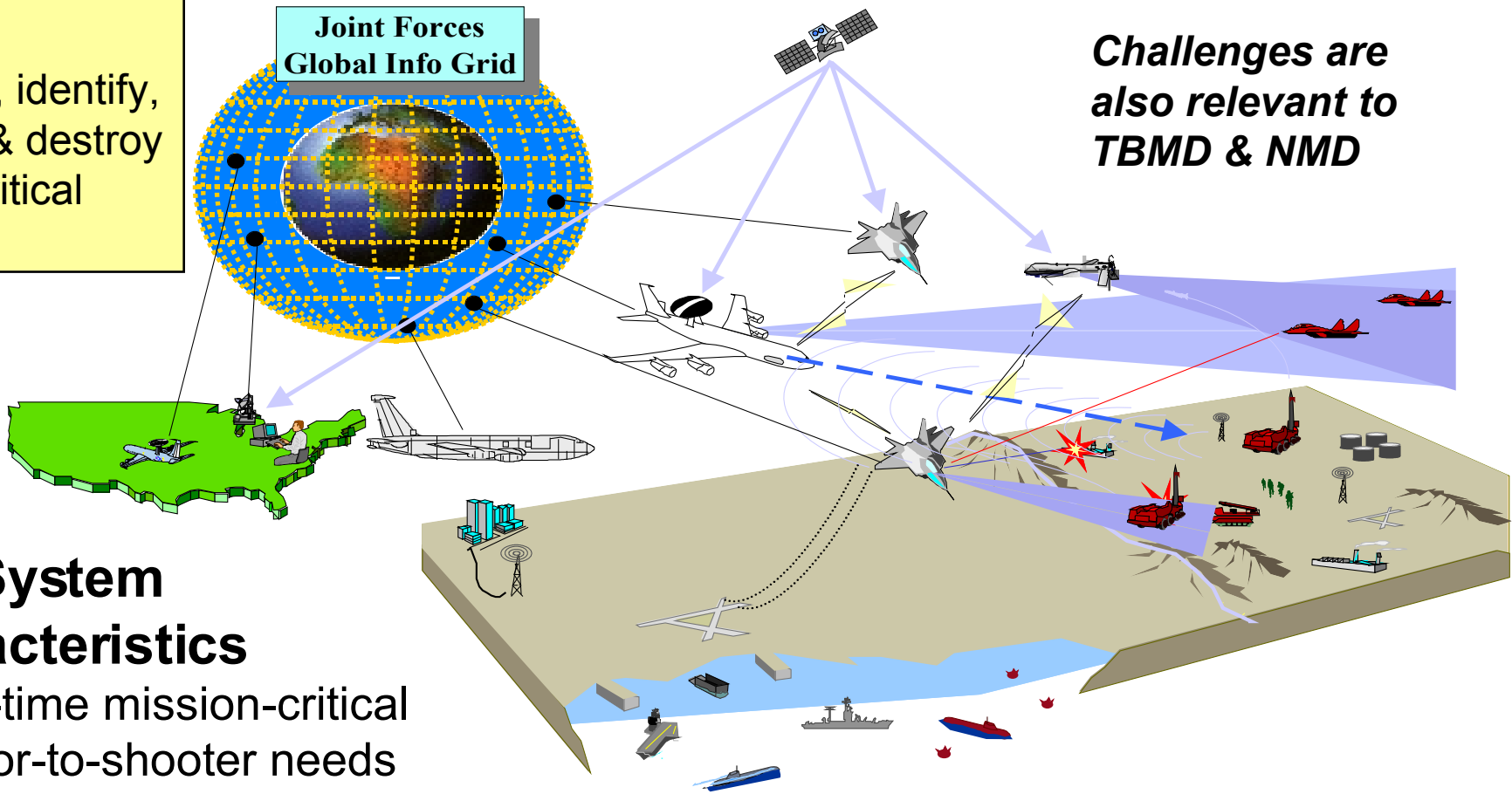
- Test flown at China Lake NAWS by Boeing OSAT II '98, funded by OS-JTF
  - www.cs.wustl.edu/~schmidt/TAO-boeing.html
- Also used on SOFIA project by Raytheon
  - sofia.arc.nasa.gov
- First use of RT CORBA in mission computing
- Drove Real-time CORBA standardization

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

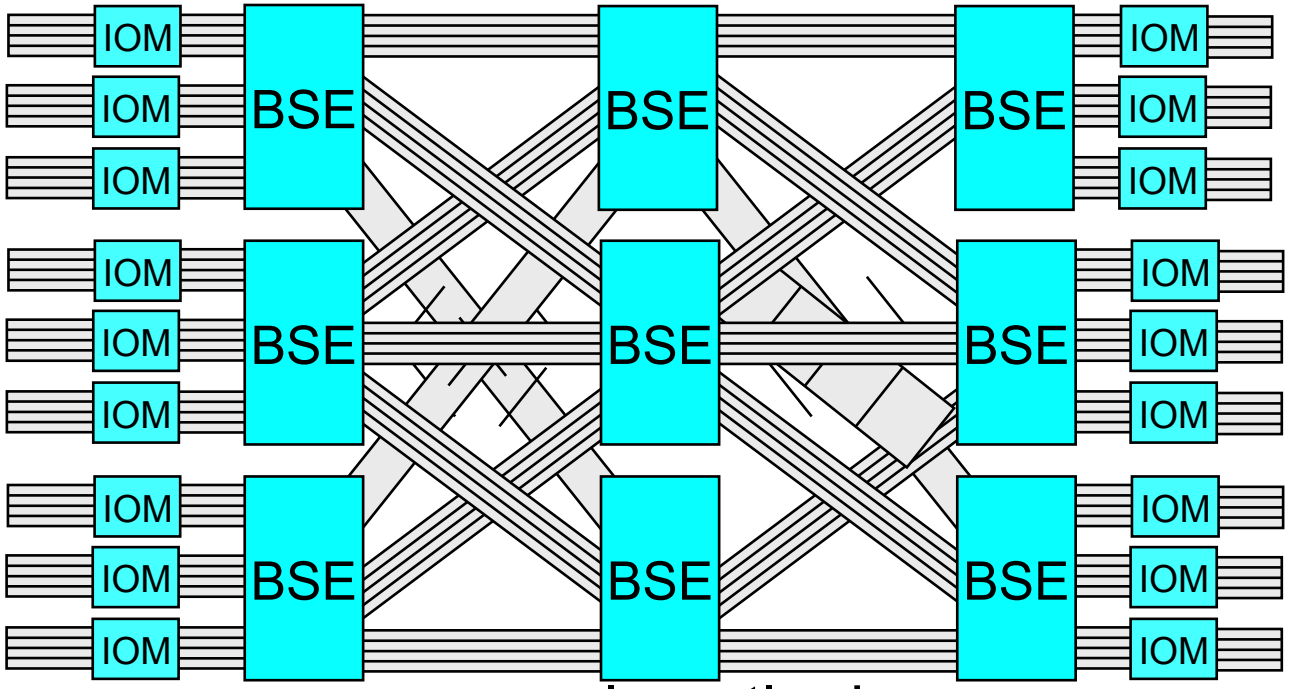
- Real-time mission-critical sensor-to-shooter needs
- Highly dynamic QoS requirements & environmental conditions
- Multi-service & asset coordination

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

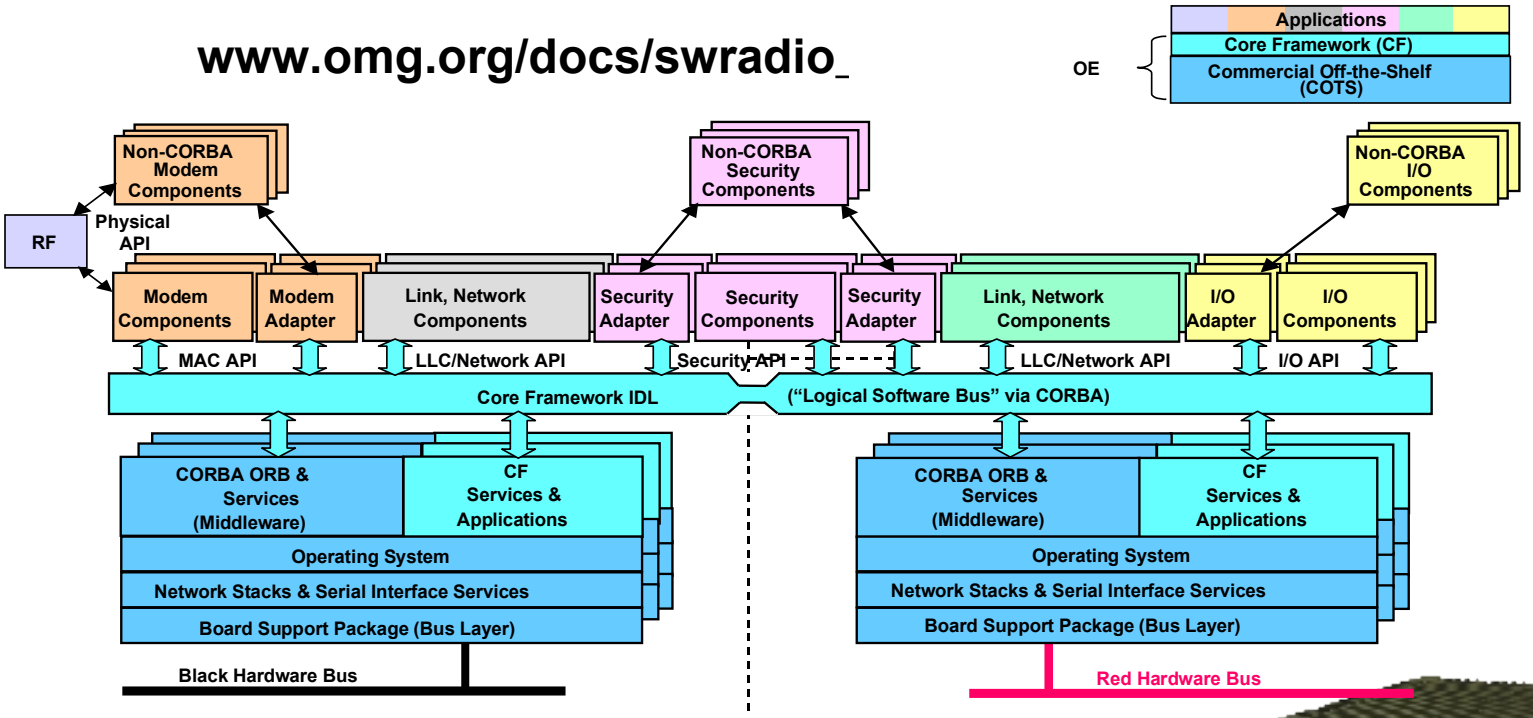
- Very high-speed WDM links
- $10^2/10^3$ line cards
- Stringent requirements for availability
- Multi-layer load balancing, *e.g.:*
  - Layer 3+4
  - Layer 5

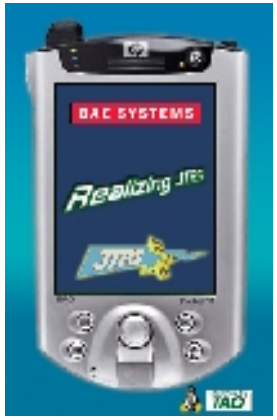
Key Software Solution Characteristics

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

Example:

Applying COTS to Software Defined Radios



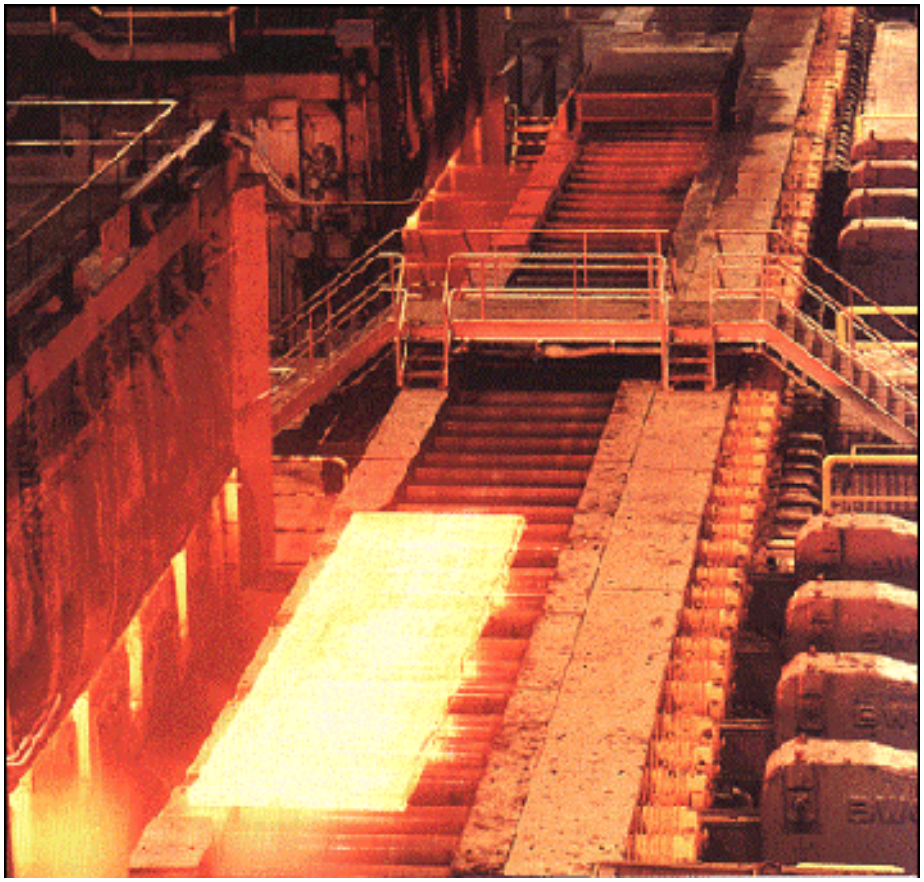


Key Software Solution Characteristics

- Transitioned to BAE systems for the Joint Tactical Radio Systems
- Programmable radio with waveform-specific components
- Uses CORBA component middleware based on ACE+TAO

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

www.siroll.de

Key Software Solution Characteristics

- 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

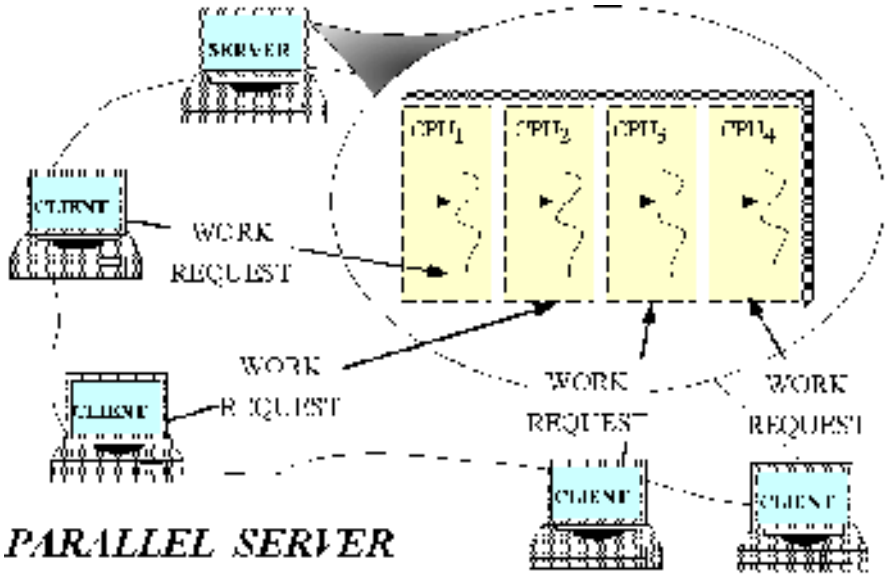
Key Software Solution Characteristics

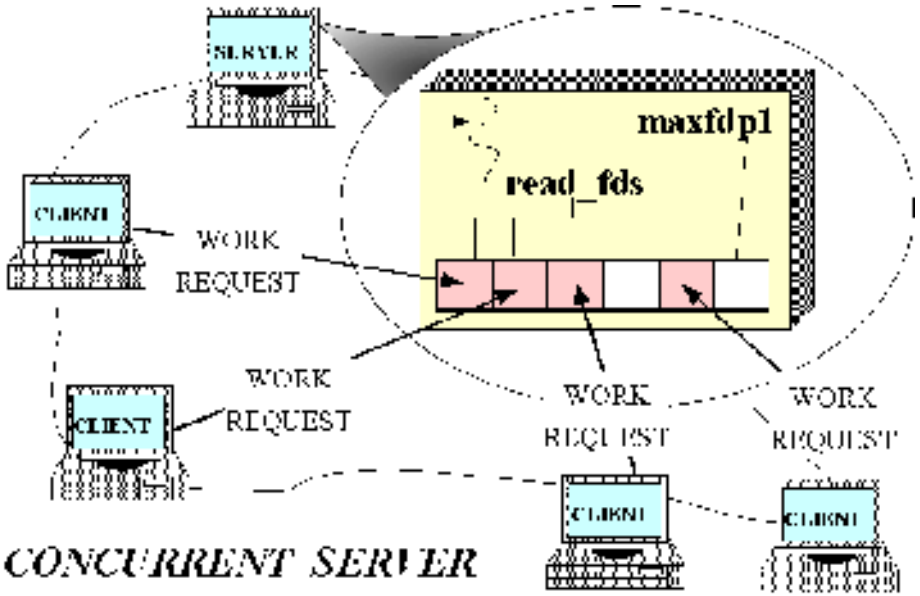
- Affordable, flexible, & COTS
  - Embedded Linux (Lem)
  - Compact PCI bus + Celeron processors
  - Remote booted by DHCP/TFTP
  - Real-time CORBA (ACE+TAO)

Key Opportunities & Challenges in Concurrent Applications

Motivations

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





Accidental Complexities

- Low-level APIs
- Poor debugging tools

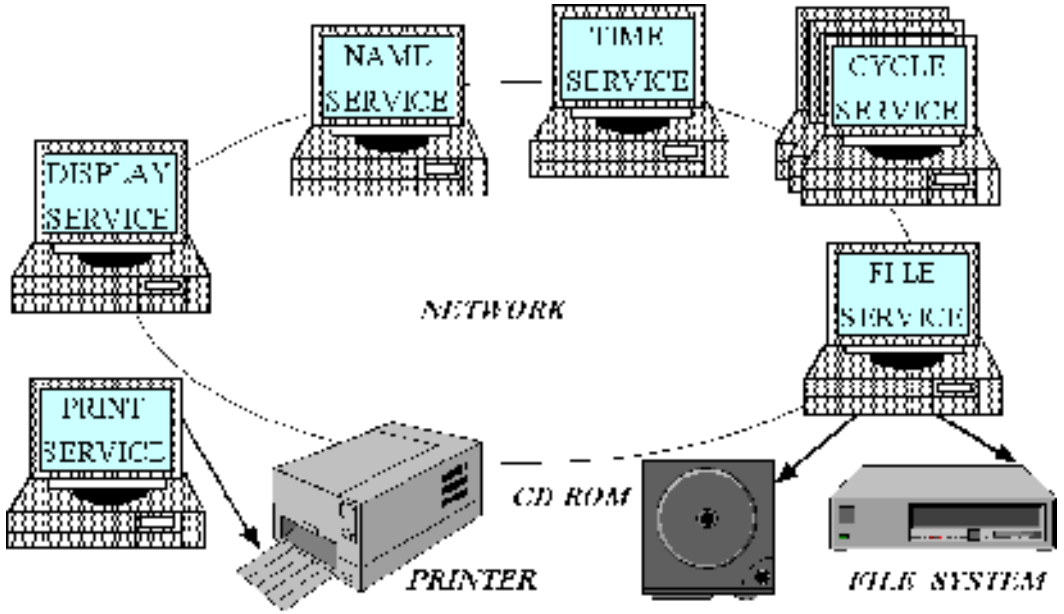
Inherent Complexities

- Scheduling
- Synchronization
- Deadlocks

Key Opportunities & Challenges in Networked & Distributed Applications

Motivations

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



Accidental Complexities

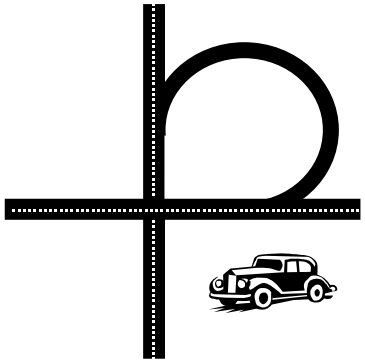
- Algorithmic decomposition
- Continuous re-invention & re-discovery of core concepts & components

Inherent Complexities

- Latency
- Reliability
- Load balancing
- Causal ordering

Overview of Patterns

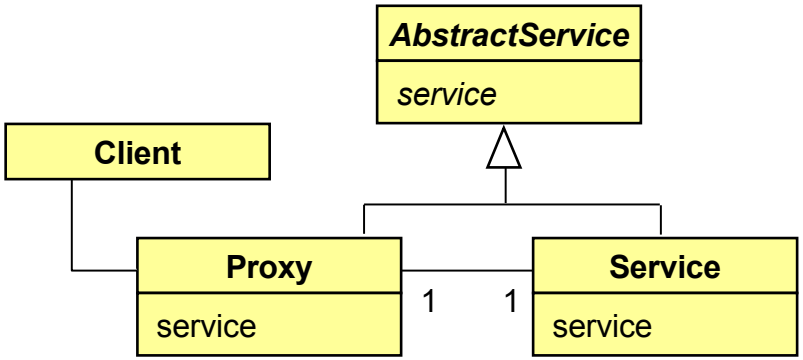
- Present *solutions* to common software *problems* arising within a certain *context*



- Help resolve key software design forces

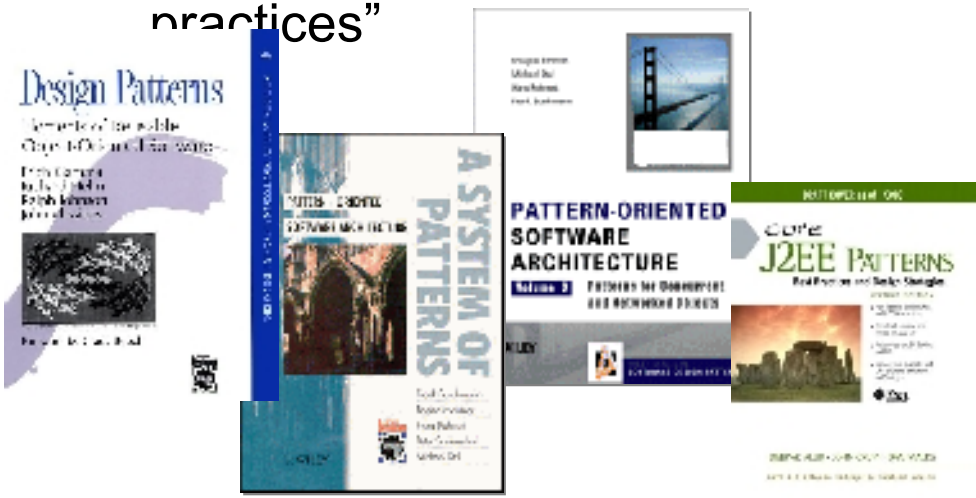
- ***Flexibility***
- ***Extensibility***
- ***Dependability***
- ***Predictability***
- ***Scalability***
- ***Efficiency***

- Capture recurring structures & dynamics among software participants to facilitate reuse of successful designs



The Proxy Pattern

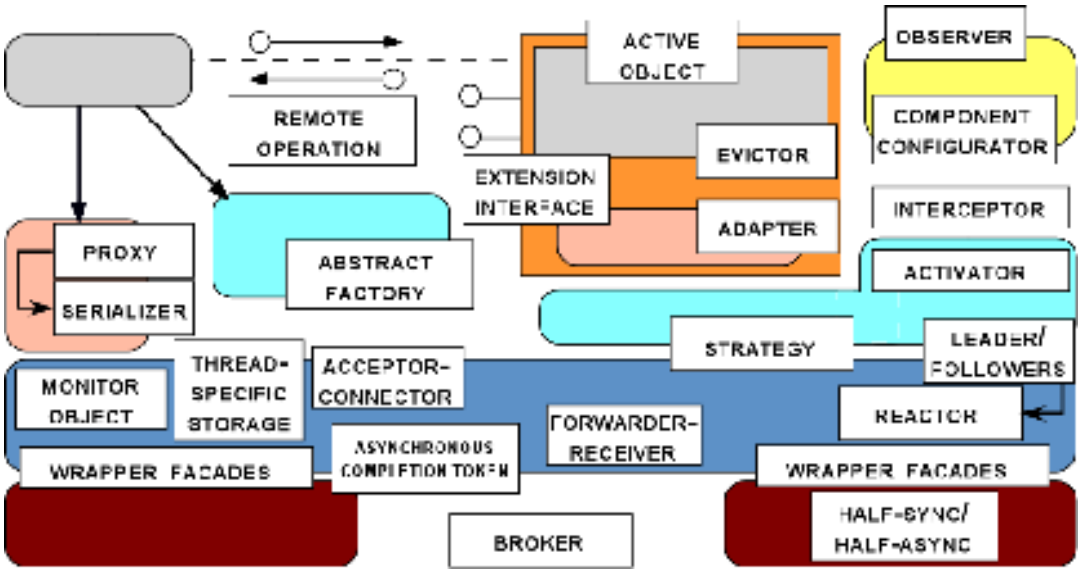
- Generally codify expert knowledge of design strategies, constraints & "best practices"



Overview of Pattern Languages

Motivation

- Individual patterns & pattern catalogs are insufficient
- Software modeling methods & tools largely just illustrate ***how*** – not ***why*** – systems are designed



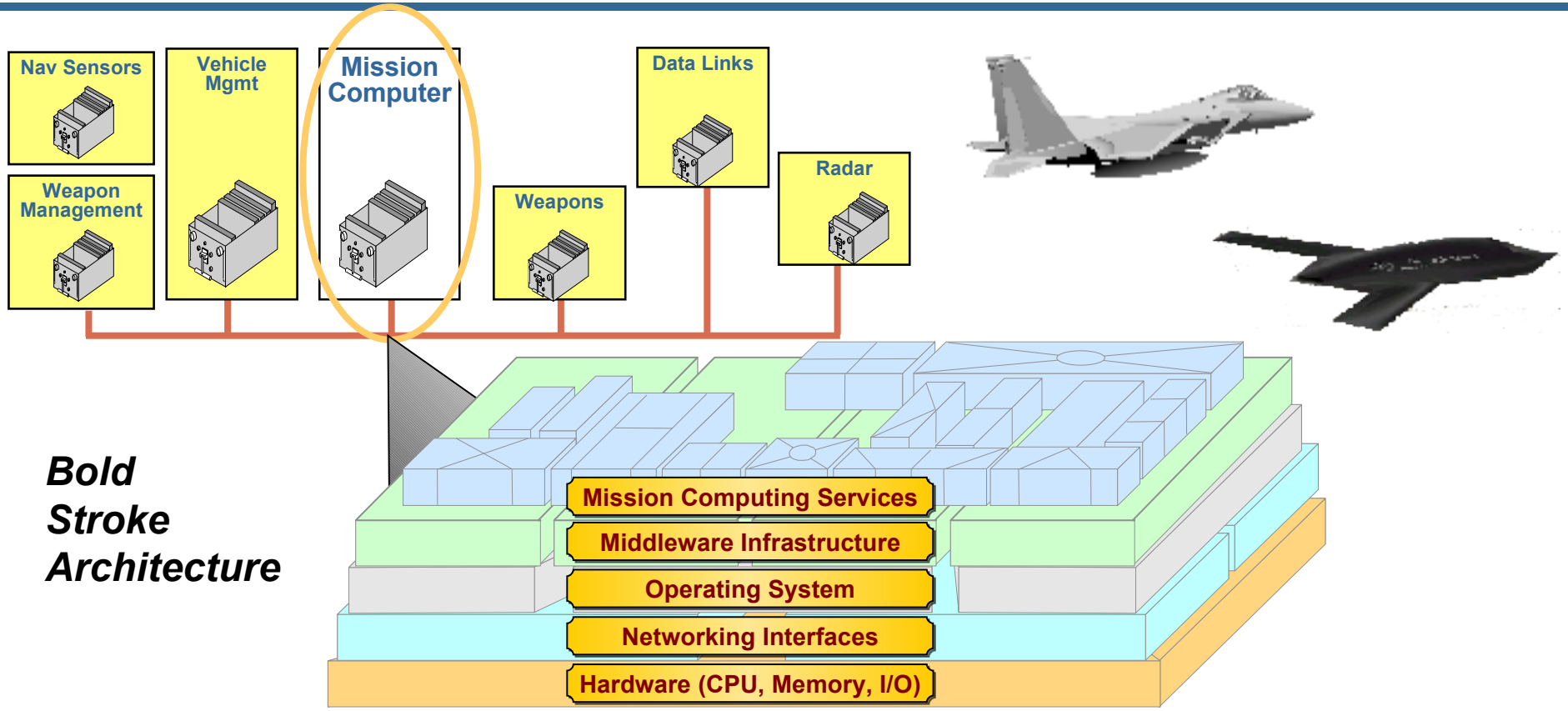
Benefits of Pattern Languages

- 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*

Taxonomy of Patterns & Idioms

Type	Description	Examples
Idioms	Restricted to a particular language, system, or tool	Scoped locking
Design patterns	Capture the static & dynamic roles & relationships in solutions that occur repeatedly	Active Object, Bridge, Proxy, Wrapper Façade, & Visitor
Architectural patterns	Express a fundamental structural organization for software systems that provide a set of predefined subsystems, specify their relationships, & include the rules and guidelines for organizing the relationships between them	Half-Sync/Half-Async, Layers, Proactor, Publisher-Subscriber, & Reactor
Optimization principle patterns	Document rules for avoiding common design & implementation mistakes that degrade performance	Optimize for common case, pass information between layers

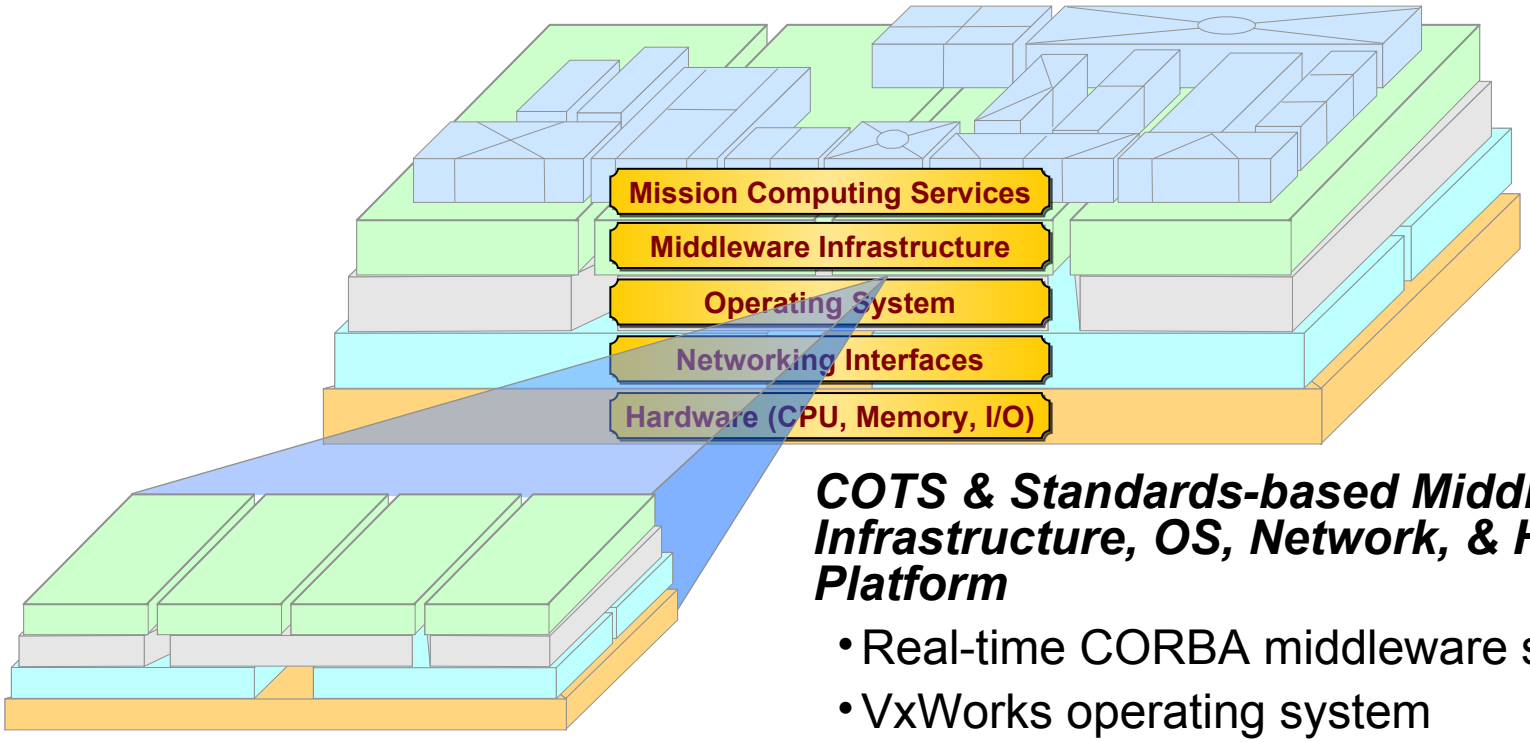
Tutorial Example: Boeing Bold Stroke



Bold Stroke Architecture

- Avionics mission computing product-line architecture for Boeing military aircraft, e.g., F-18 E/F, 15E, Harrier, UCAV
- DRE system with 100+ developers, 3,000+ software components, 3-5 million lines of C++ code
- Based on COTS hardware, networks, operating systems, & middleware
- Used as Open Experimention Platform (OEP) for DARPA IXO PCES, MoBIES, SEC, MICA programs

Tutorial Example: Boeing Bold Stroke



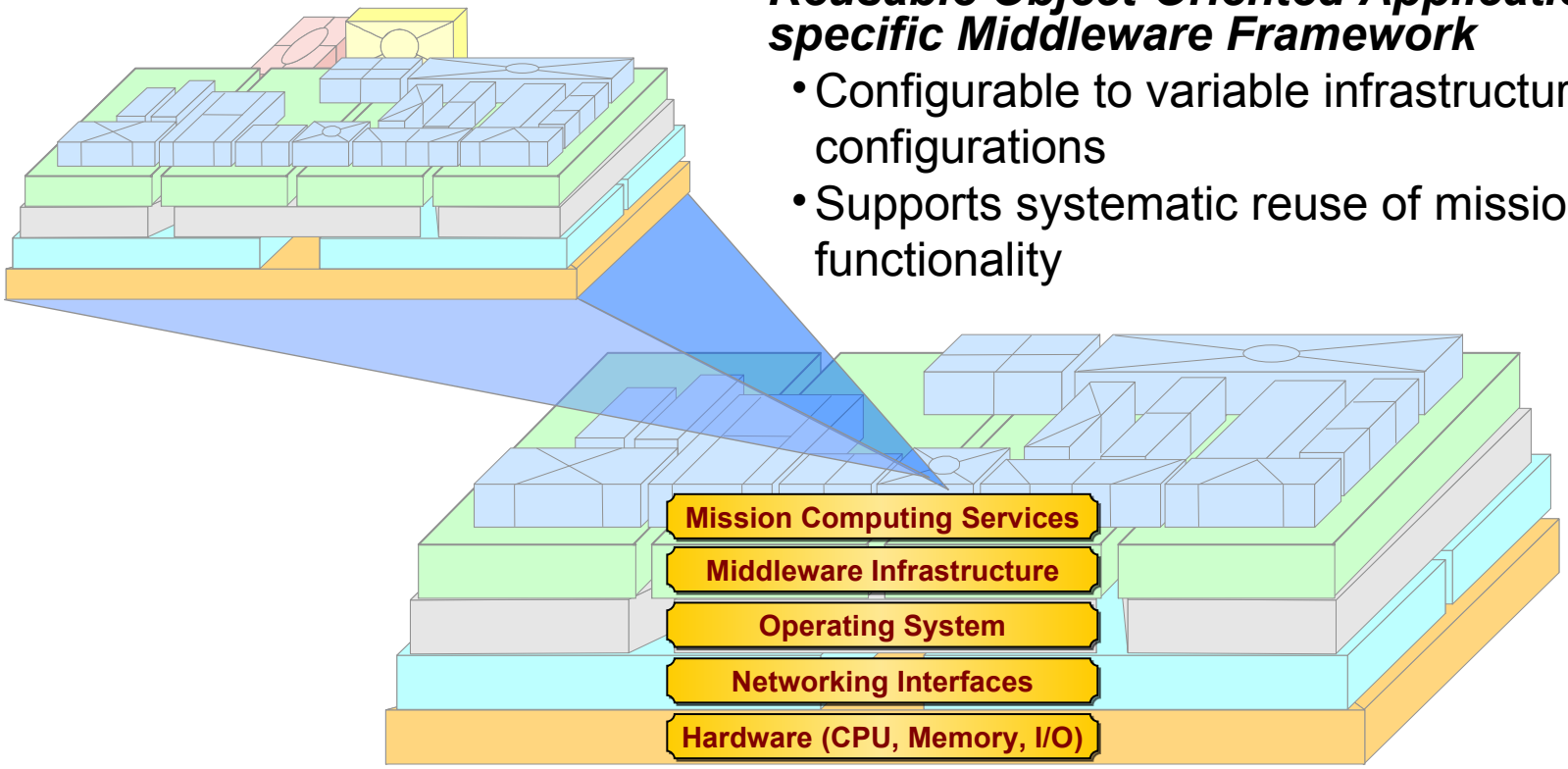
COTS & Standards-based Middleware Infrastructure, OS, Network, & Hardware Platform

- Real-time CORBA middleware services
- VxWorks operating system
- VME, 1553, & Link16
- PowerPC

Tutorial Example: Boeing Bold Stroke

Reusable Object-Oriented Application Domain-specific Middleware Framework

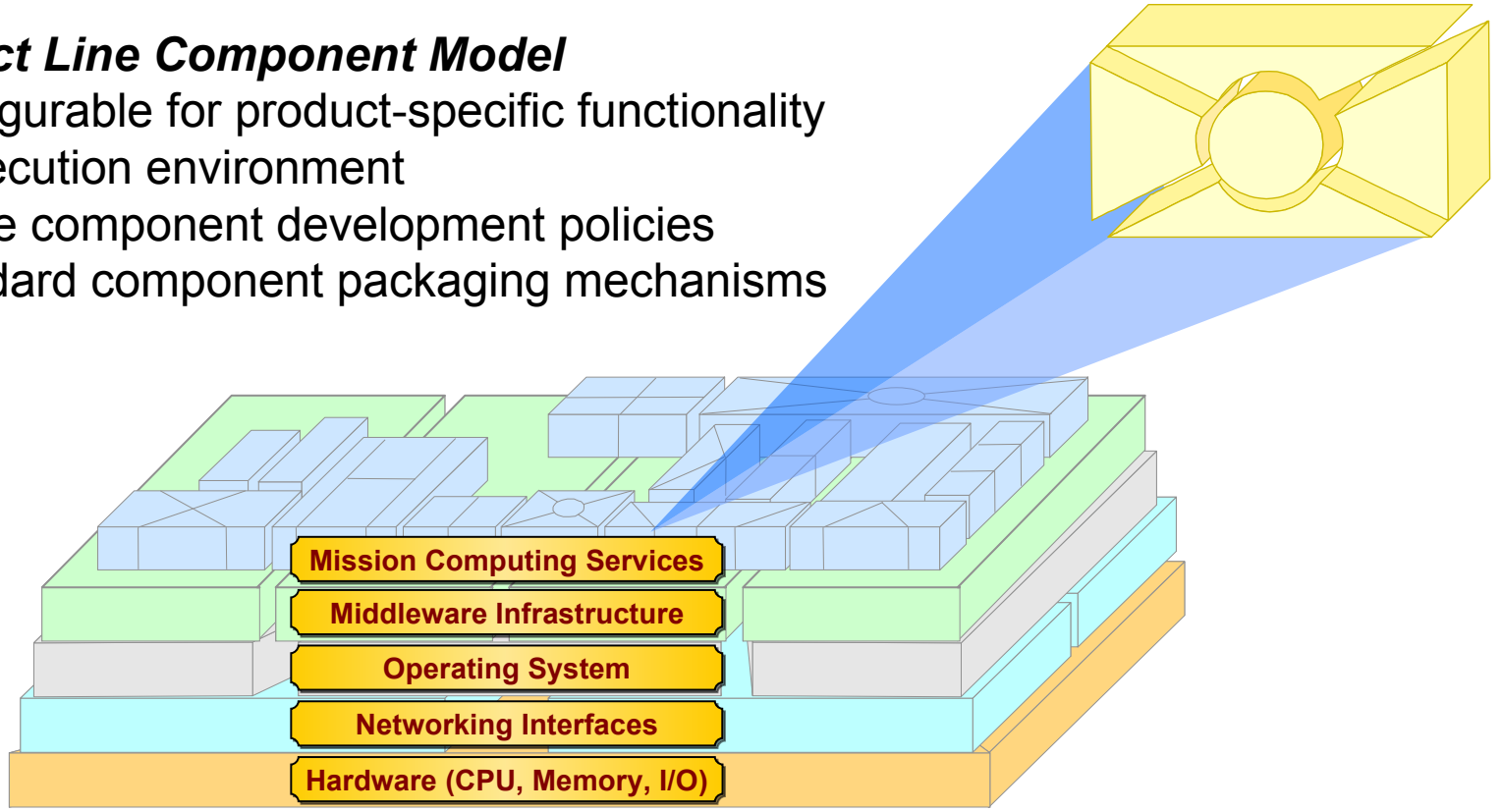
- Configurable to variable infrastructure configurations
- Supports systematic reuse of mission computing functionality



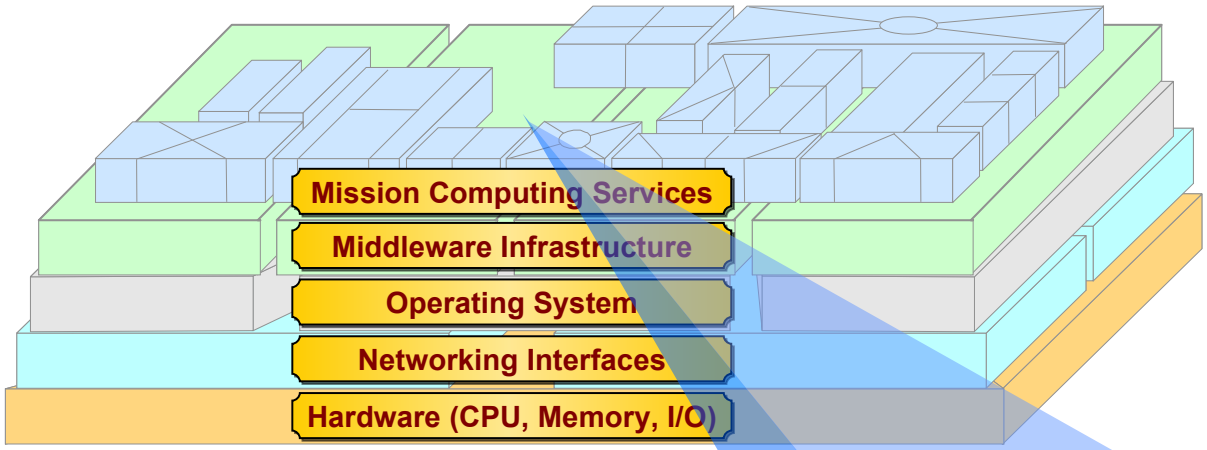
Tutorial Example: Boeing Bold Stroke

Product Line Component Model

- Configurable for product-specific functionality & execution environment
- Single component development policies
- Standard component packaging mechanisms

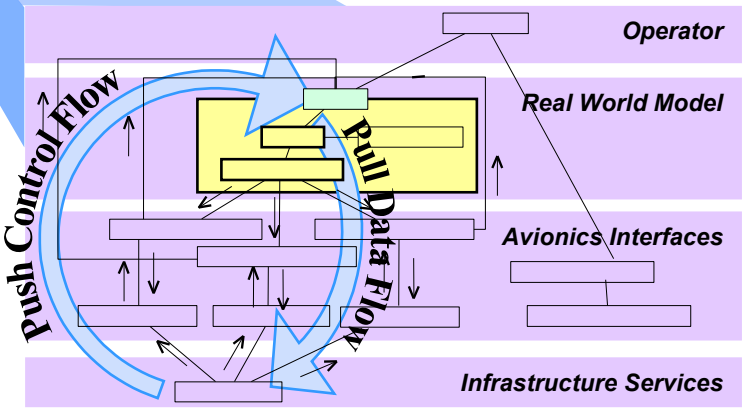


Tutorial Example: Boeing Bold Stroke



Component Integration Model

- Configurable for product-specific component assembly & deployment environments
- Model-based component integration policies



Legacy Avionics Architectures

Key System Characteristics

- Hard & soft real-time deadlines
  - ~20-40 Hz
- Low latency & jitter between boards
  - ~100 *u*secs
- Periodic & aperiodic processing
- Complex dependencies
- Continuous platform upgrades

Avionics Mission Computing Functions

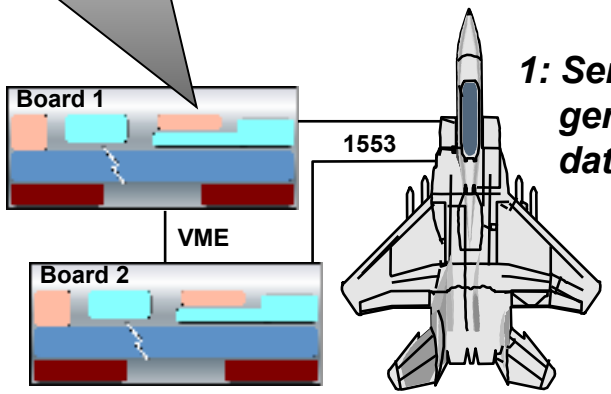
- Weapons targeting systems (WTS)
- Airframe & navigation (Nav)
- Sensor control (GPS, IFF, FLIR)
- Heads-up display (HUD)
- Auto-pilot (AP)

4: Mission functions perform avionics operations

3: Sensor proxies process data & pass to missions functions

2: I/O via interrupts

1: Sensors generate data



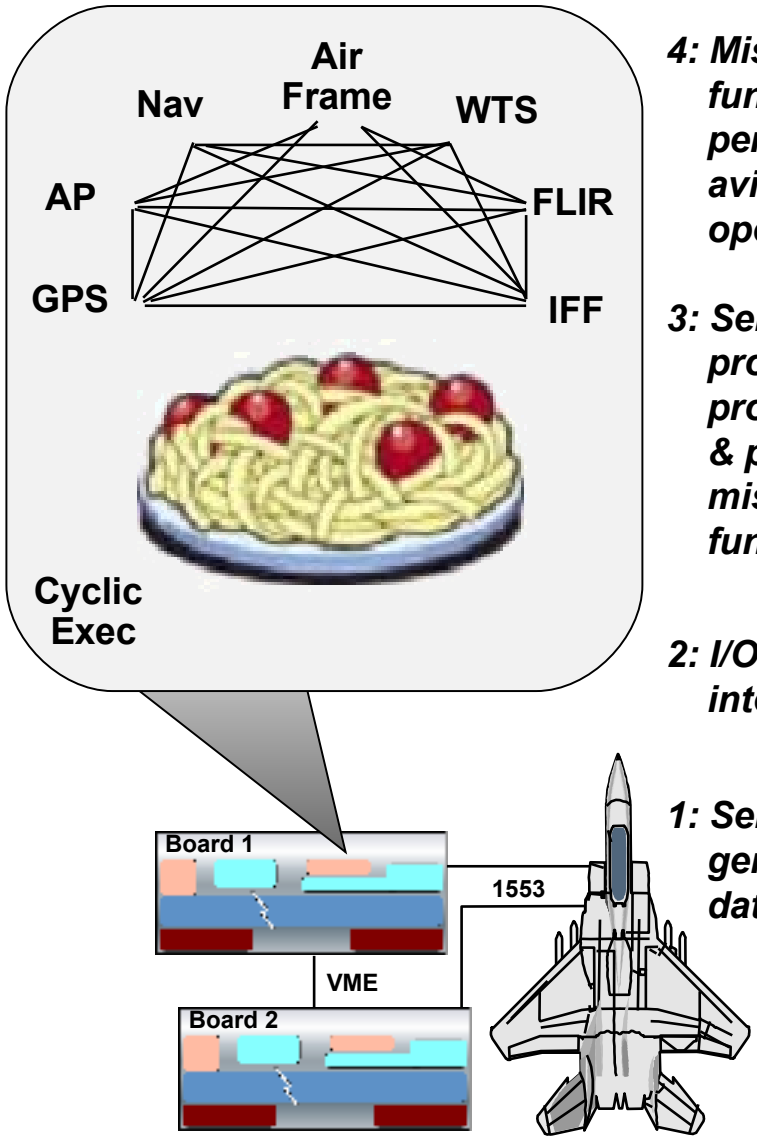
Legacy Avionics Architectures

Key System Characteristics

- Hard & soft real-time deadlines
  - ~20-40 Hz
- Low latency & jitter between boards
  - ~100 *u*secs
- Periodic & aperiodic processing
- Complex dependencies
- Continuous platform upgrades

Limitations with Legacy Avionics Architectures

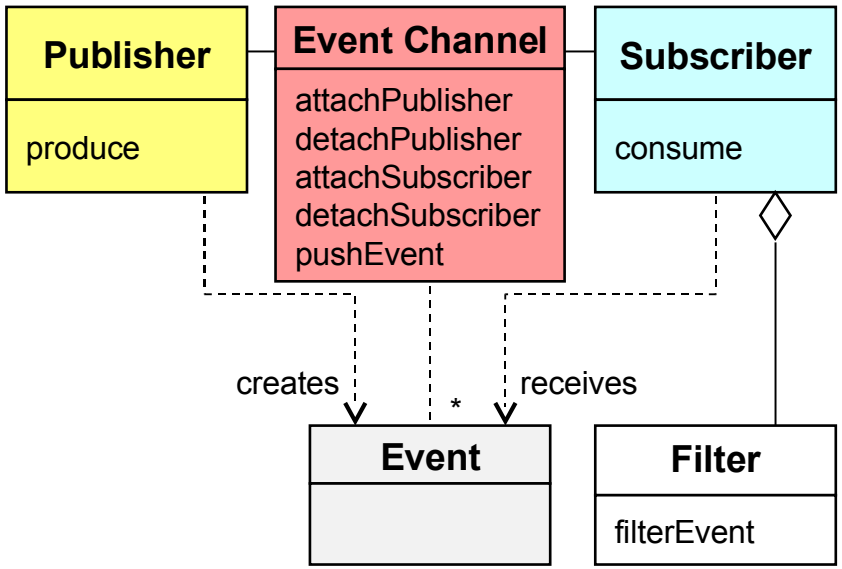
- Stovepiped
- Proprietary
- Expensive
- Vulnerable
- ***Tightly coupled***
- ***Hard to schedule***
- ***Brittle & non-adaptive***



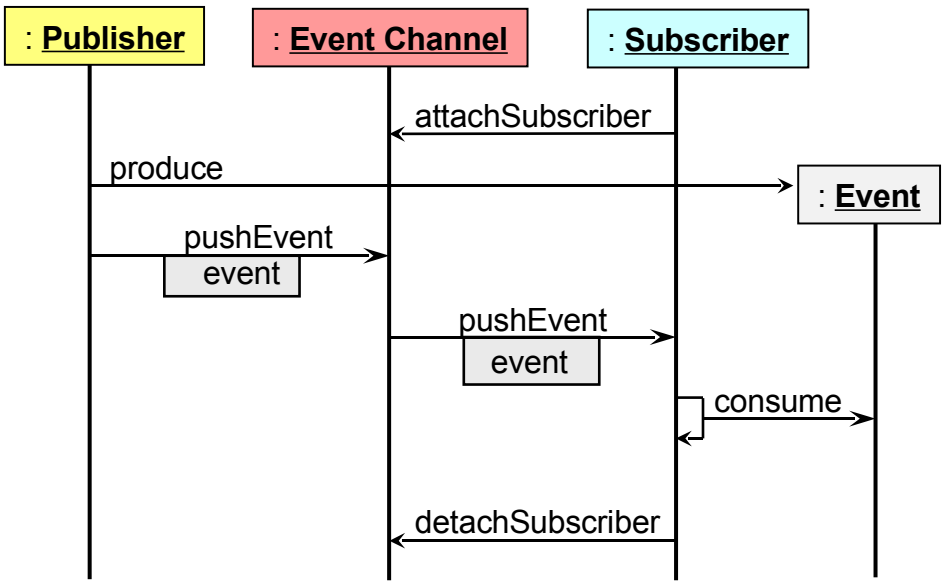
Decoupling Avionics Components

Context	Problems	Solution
• I/O driven DRE application • Complex dependencies • Real-time constraints	• Tightly coupled components • Hard to schedule • Expensive to evolve	• Apply the *Publisher-Subscriber* architectural pattern to distribute periodic, I/O-driven data from a single point of source to a collection of consumers

Structure



Dynamics



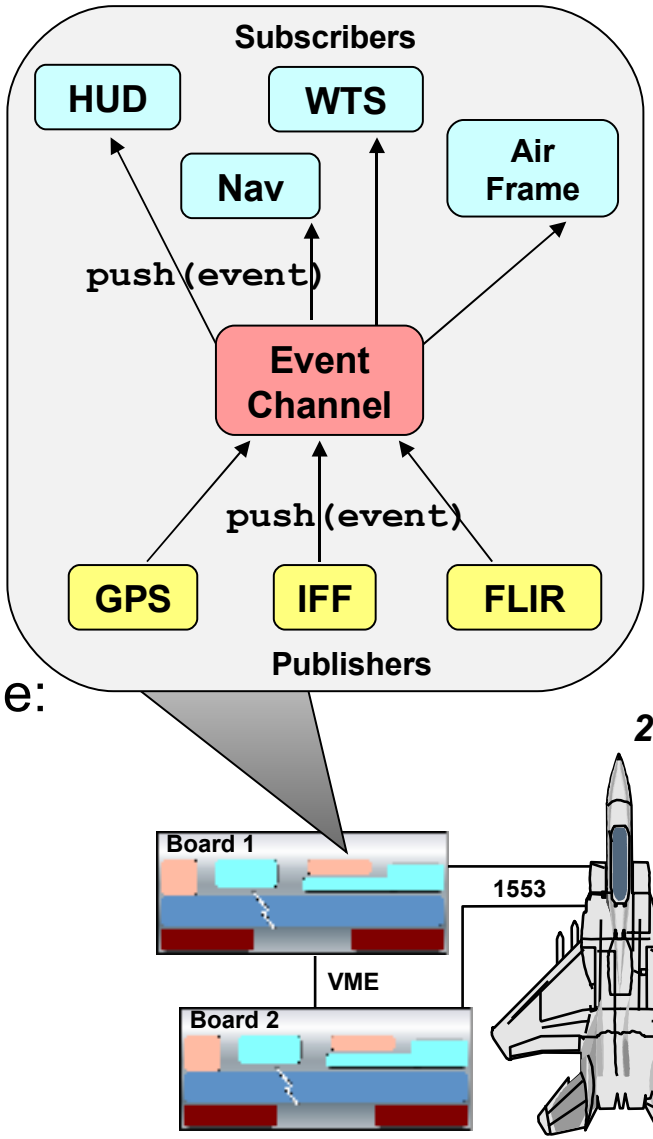
Applying the Publisher-Subscriber Pattern to Bold Stroke

Bold Stroke uses the ***Publisher-Subscriber*** pattern to decouple sensor processing from mission computing operations

- Anonymous publisher & subscriber relationships
- Group communication
- Asynchrony

Considerations for implementing the ***Publisher-Subscriber*** pattern for mission computing applications include:

- ***Event notification model***
  - Push control vs. pull data interactions
- ***Scheduling & synchronization strategies***
  - e.g., priority-based dispatching & preemption
- ***Event dependency management***
  - e.g.,filtering & correlation mechanisms



5: Subscribers perform avionics operations

4: Event Channel pushes events to subscribers(s)

3: Sensor publishers push events to event channel

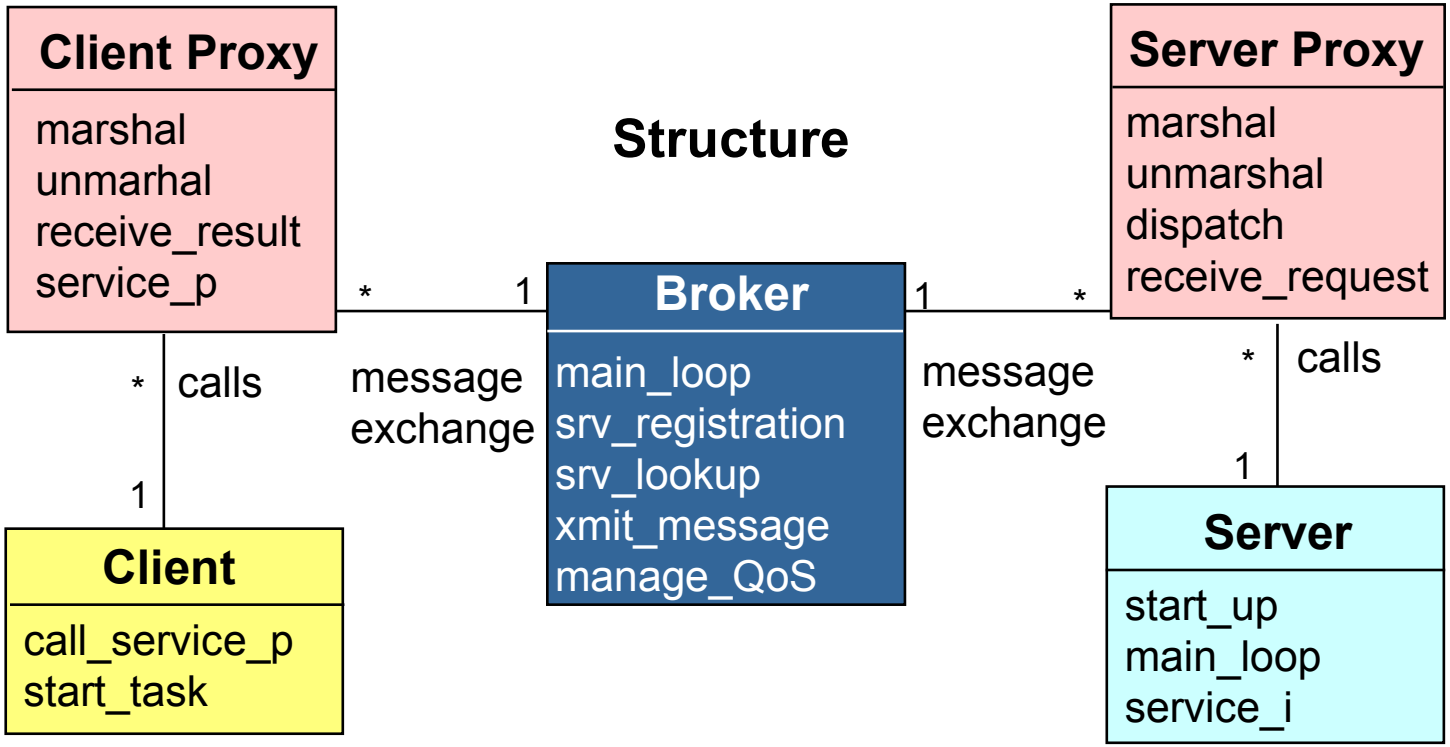
2: I/O via interrupts

1: Sensors generate data

Ensuring Platform-neutral & Network-transparent Communication

Context	Problems	Solution

• Mission computing requires remote IPC
• Stringent DRE requirements
	• Applications need capabilities to:
	  • Support remote communication
	  • Provide location transparency
	  • Handle faults
	  • Manage end-to-end QoS
	  • Encapsulate low-level system details
		• Apply the *Broker* architectural pattern to provide platform-neutral communication between mission computing boards

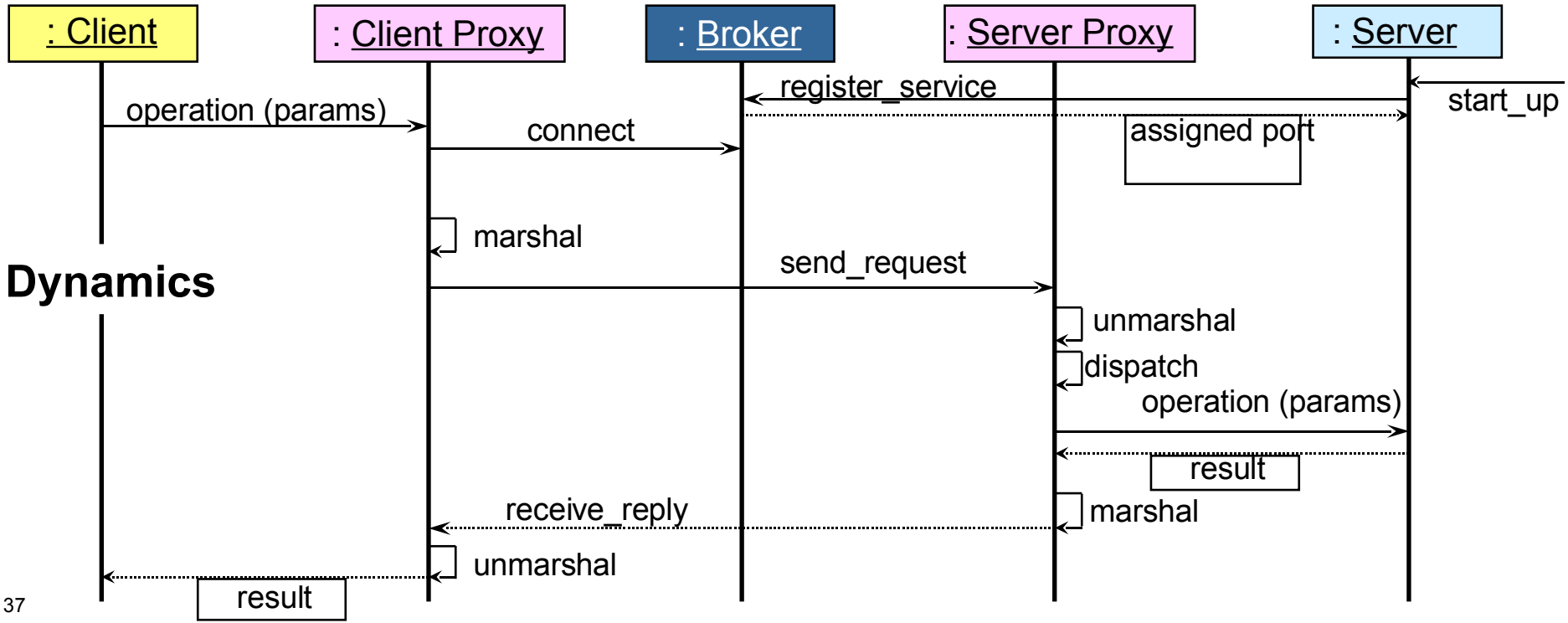


Ensuring Platform-neutral & Network-transparent Communication

Context	Problems	Solution

• Mission computing requires remote IPC
• Stringent DRE requirements
	• Applications need capabilities to:
	  • Support remote communication
	  • Provide location transparency
	  • Handle faults
	  • Manage end-to-end QoS
	  • Encapsulate low-level system details
		• Apply the *Broker* architectural pattern to provide platform-neutral communication between mission computing boards

Dynamics



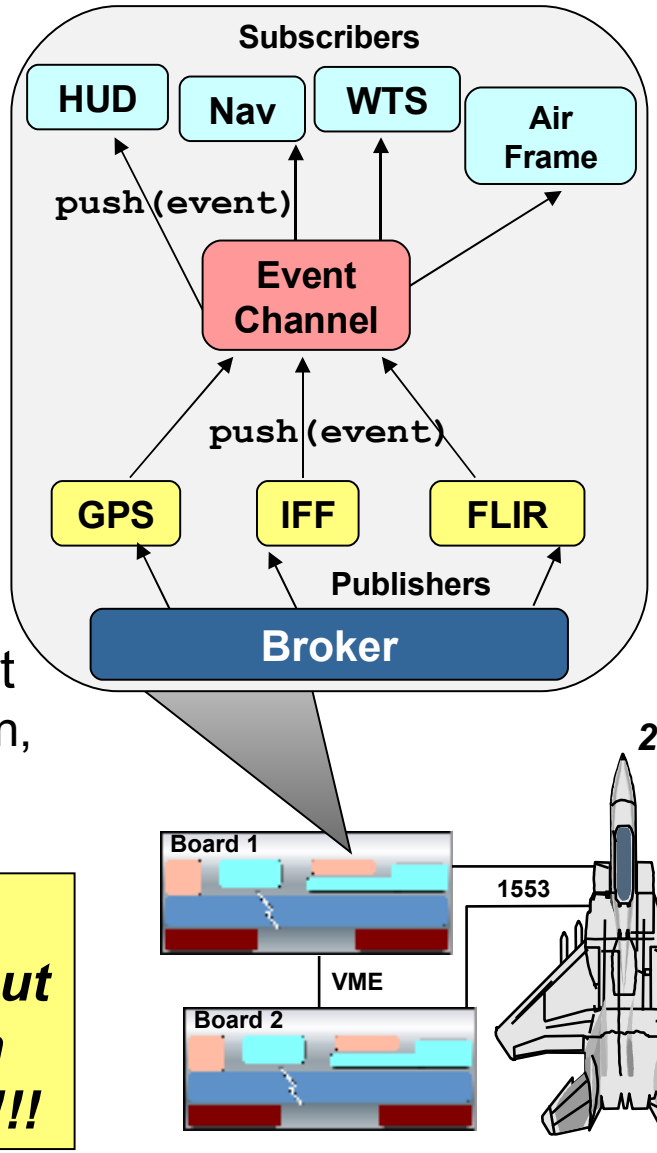
Applying the Broker Pattern to Bold Stroke

Bold Stroke uses the ***Broker*** pattern to shield distributed applications from environment heterogeneity, *e.g.,*

- Programming languages
- Operating systems
- Networking protocols
- Hardware

A key consideration for implementing the ***Broker*** pattern for mission computing applications is ***QoS*** support

- e.g., latency, jitter, priority preservation, dependability, security, etc.



6: Subscribers perform avionics operations

5: Event Channel pushes events to subscribers(s)

4: Sensor publishers push events to event channel

3: Broker handles I/O via upcalls

2: I/O via interrupts

1: Sensors generate data

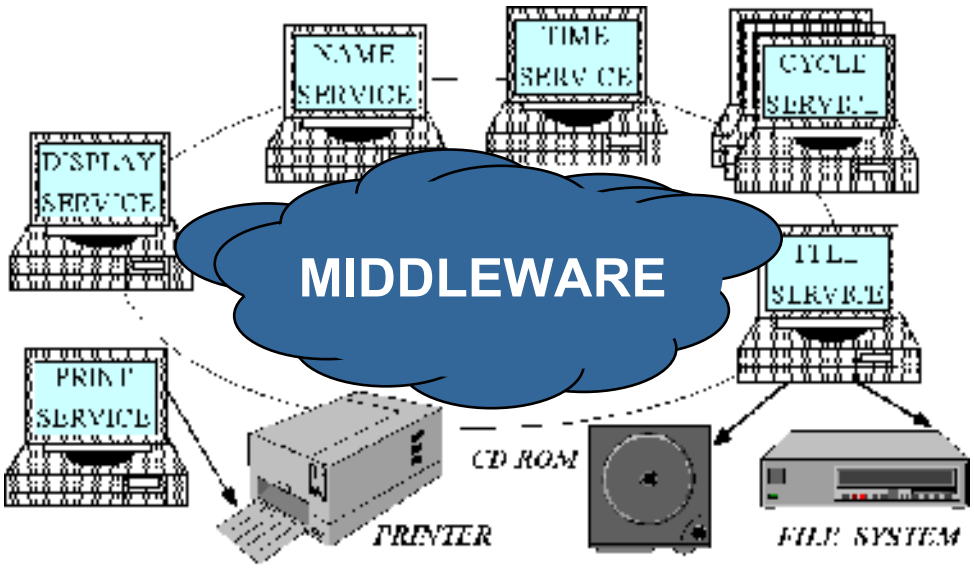
Caveat

These patterns are very useful, but having to implement them from scratch is tedious & error-prone!!!

Software Design Abstractions for Concurrent & Networked Applications

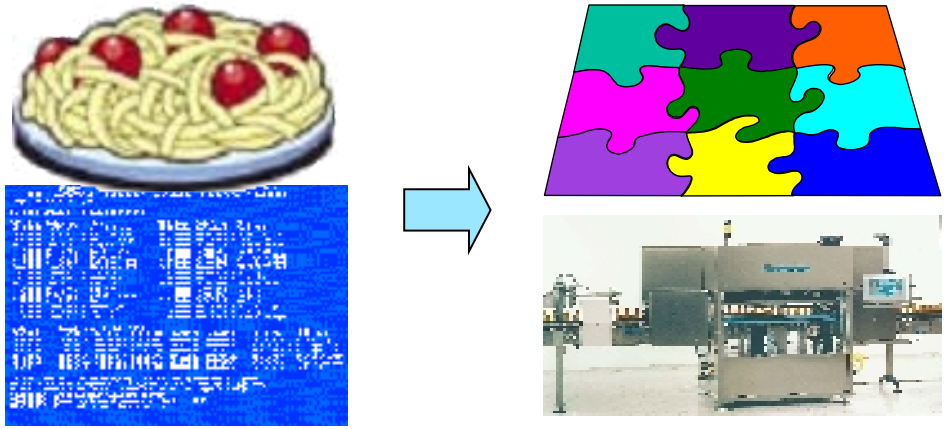
Problem

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



Solution

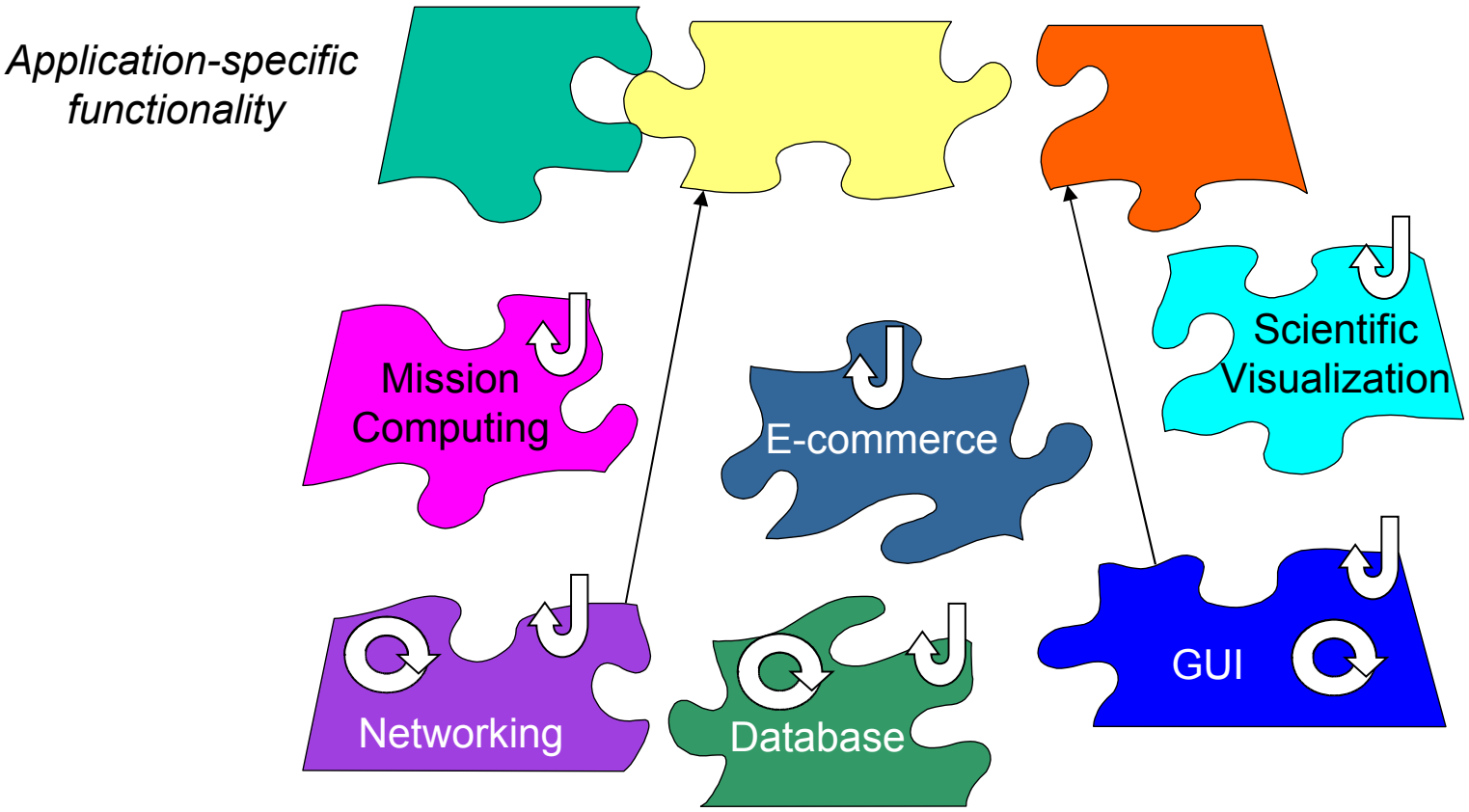
- Don't structure distributed applications & middleware as a monolithic spagetti
- Instead, decompose them into modular *classes*, *frameworks*, & *components*



Overview of Frameworks

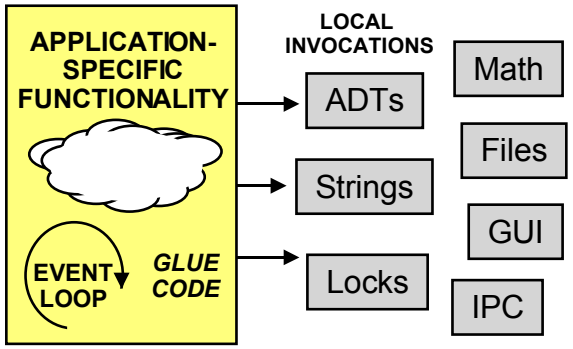
Framework Characteristics

- Frameworks exhibit "inversion of control" at runtime via callbacks
- Frameworks provide integrated domain-specific structures & functionality
- Frameworks are "semi-complete" applications



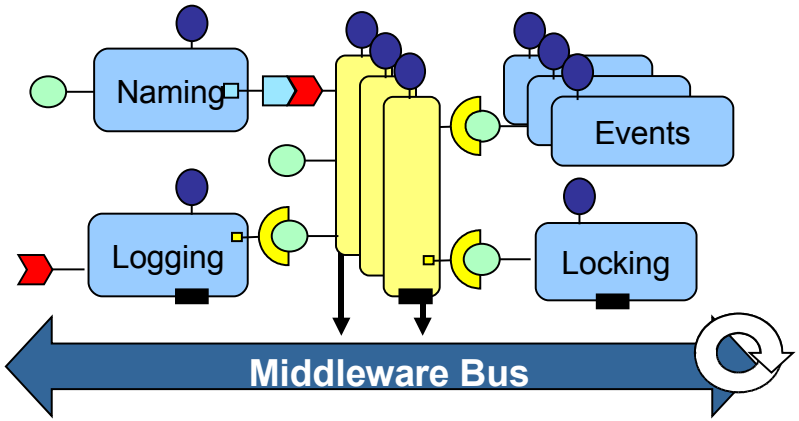
Comparing Class Libraries, Frameworks, & Components

Class Library Architecture



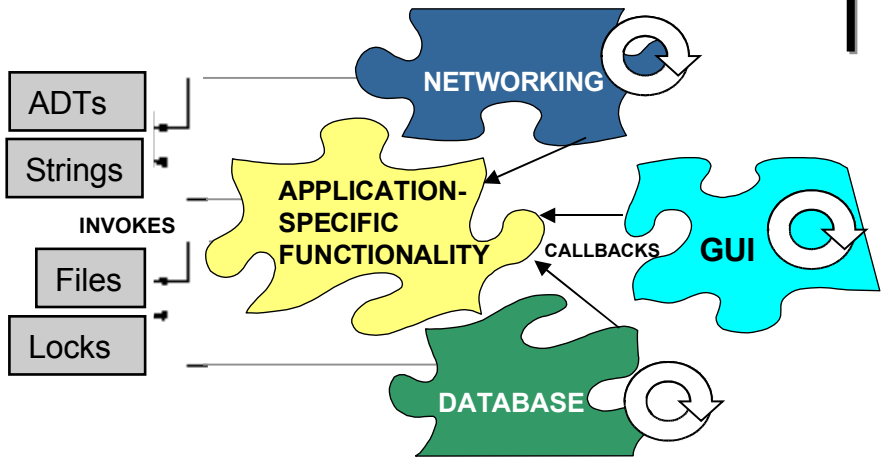
A ***class*** is a unit of abstraction & implementation in an OO programming language

Component Architecture



A ***component*** is an encapsulation unit with one or more interfaces that provide clients with access to its services

Framework Architecture



A ***framework*** is an integrated set of classes that collaborate to produce a reusable architecture for a family of applications

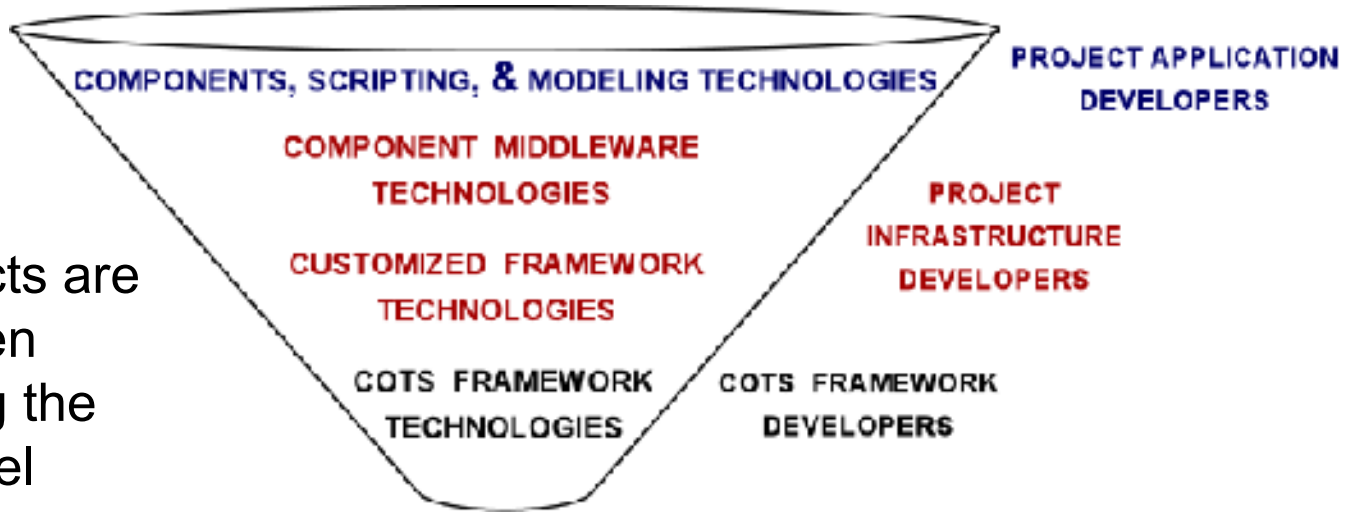
Class Libraries	Frameworks	Components
Micro-level	Meso-level	Macro-level
Stand-alone language entities	"Semi-complete" applications	Stand-alone composition entities
Domain-independent	Domain-specific	Domain-specific or Domain-independent
Borrow caller's thread	Inversion of control	Borrow caller's thread

Using Frameworks Effectively

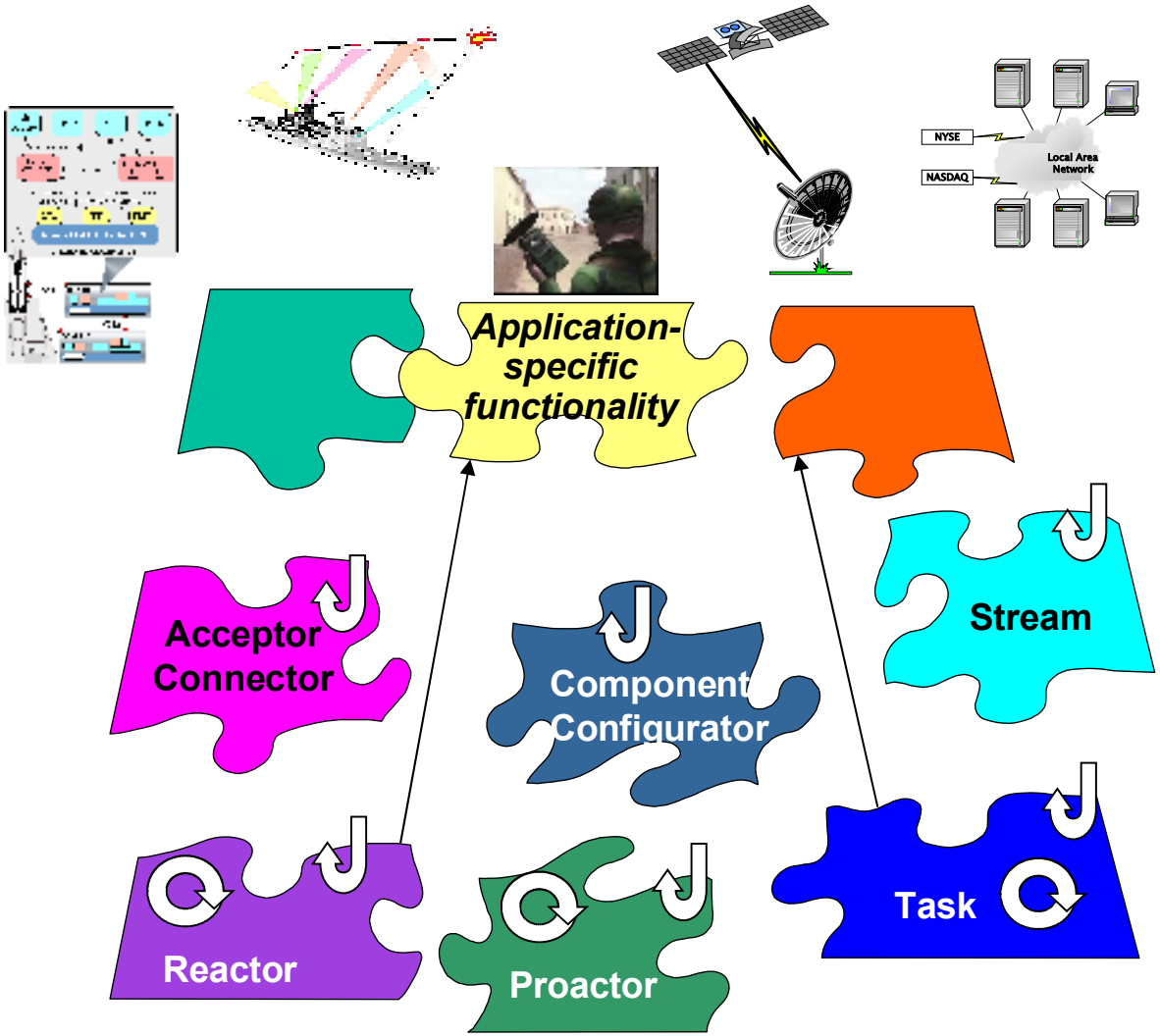
Observations

- Frameworks are powerful, but hard to develop & use effectively by application developers
  - It's often better to use & customize COTS frameworks than to develop in-house frameworks
- Components are easier for application developers to use, but aren't as powerful or flexible as frameworks

Successful projects are therefore often organized using the "funnel" model



Overview of the ACE Frameworks

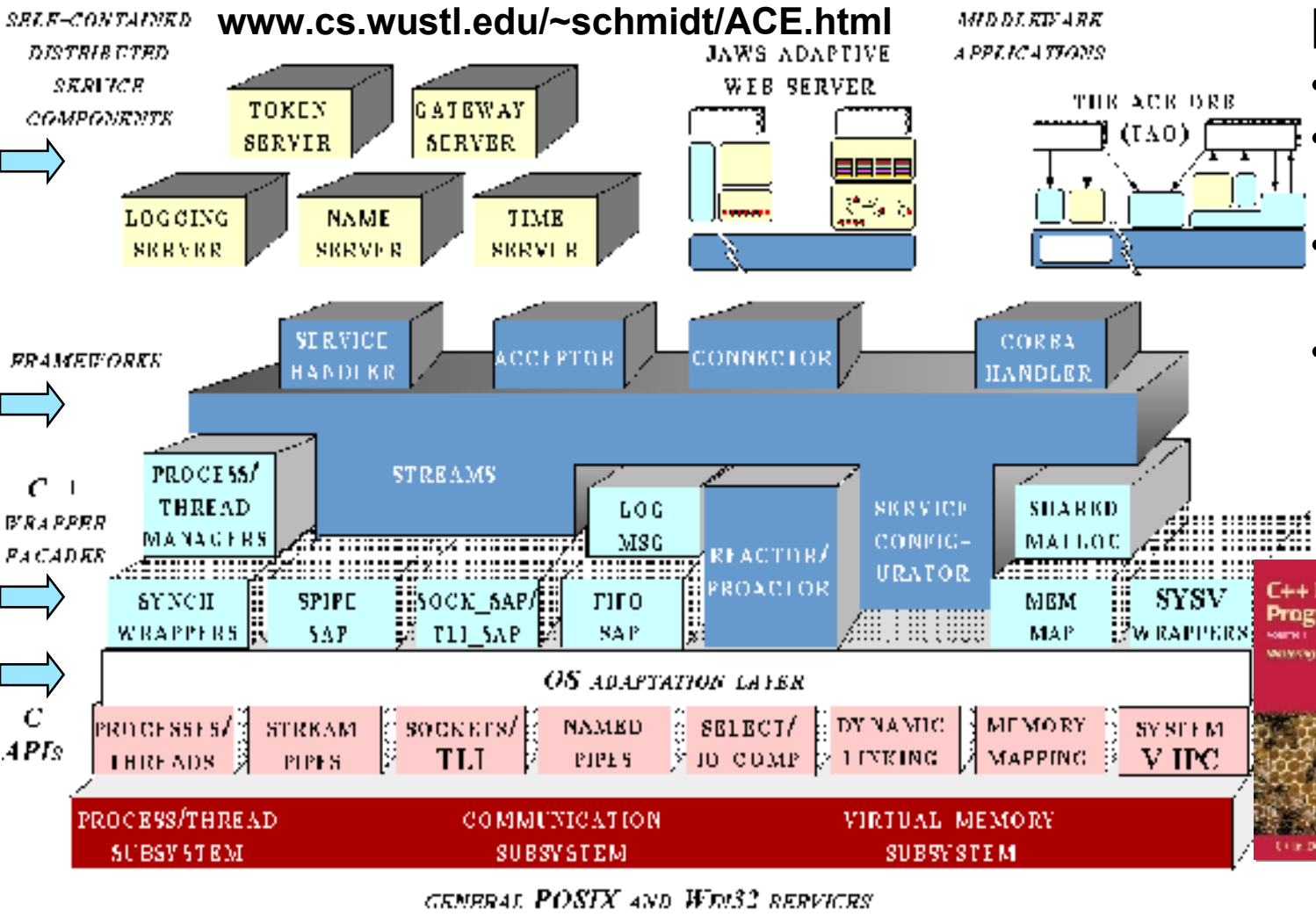


Features

- Open-source
- 6+ integrated frameworks
- 250,000+ lines of C++
- 40+ person-years of effort
- Ported to Windows, UNIX, & real-time operating systems
  - *e.g.,* VxWorks, pSoS, LynxOS, Chorus, QNX
- Large user community

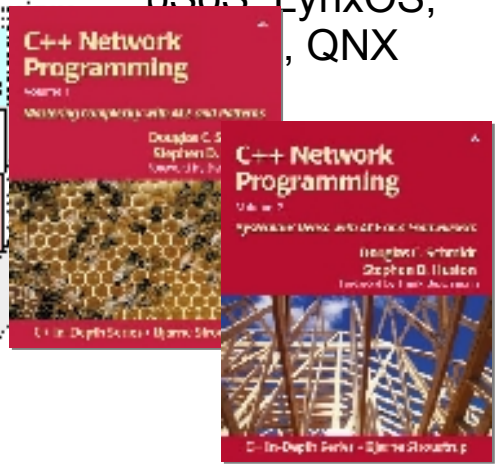
www.cs.wustl.edu/~schmidt/ACE.html

The Layered Architecture of ACE



Features

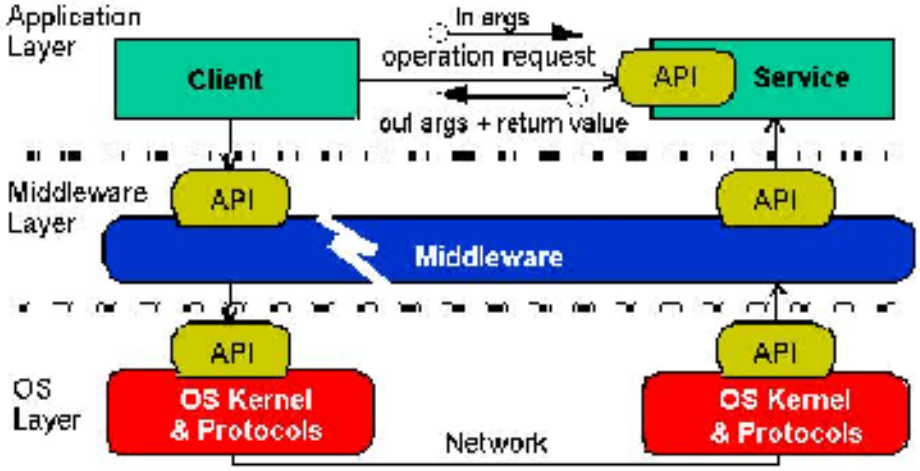
- Open-source
- 250,000+ lines of C++
- 40+ person-years of effort
- Ported to Win32, UNIX, & RTOSs
  - *e.g.*, VxWorks, pSoS, LynxOS, , QNX



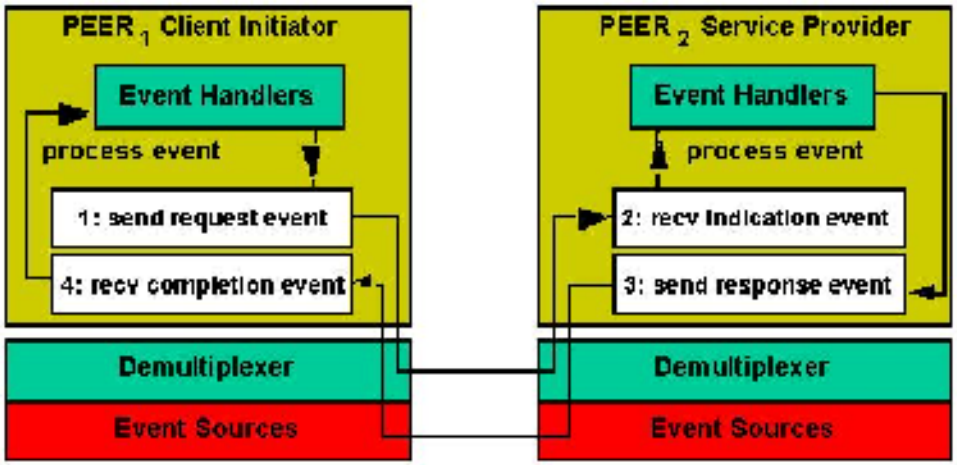
- **Large open-source user community**
  - www.cs.wustl.edu/~schmidt/ACE-users.html
- **Commercial support by Riverace**
  - www.riverace.com/

Key Capabilities Provided by ACE

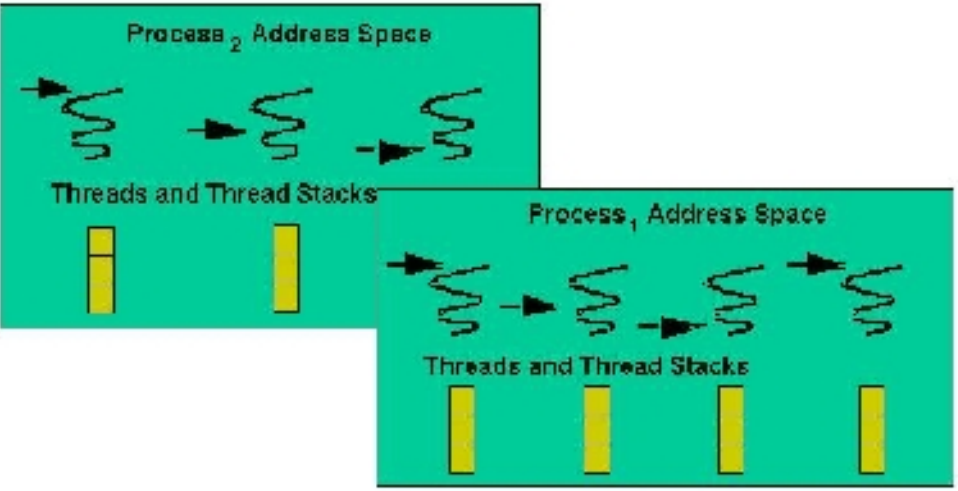
Service Access & Control



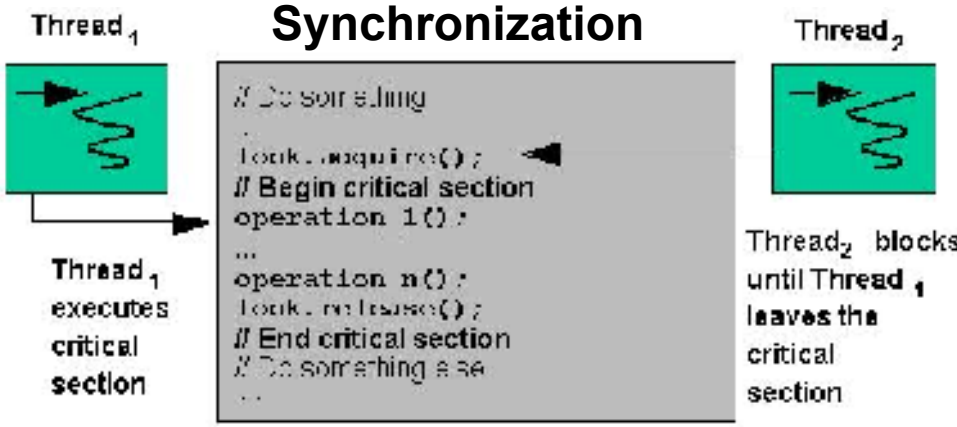
Event Handling



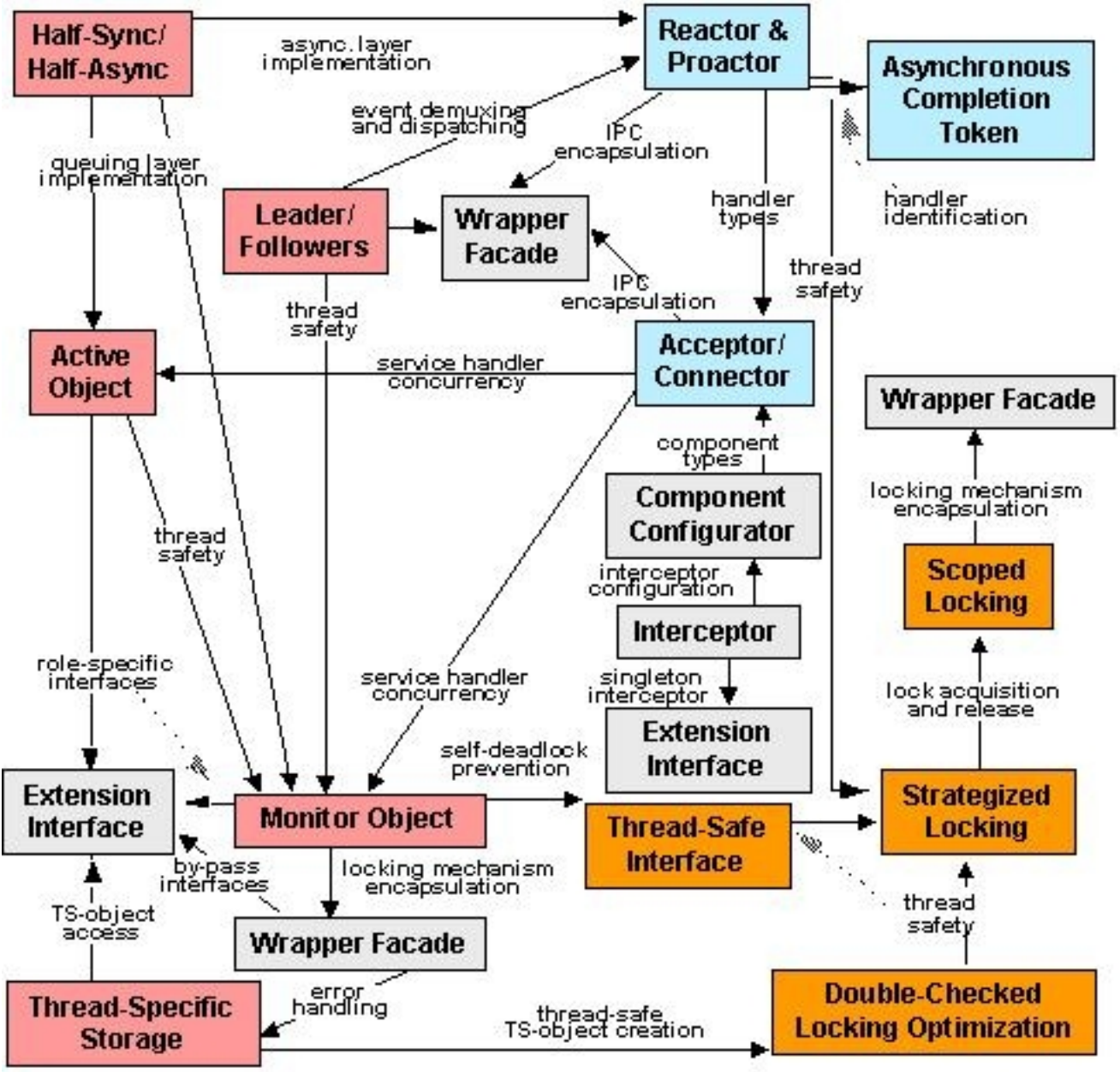
Concurrency



Synchronization



The POSA2 Pattern Language



Pattern Benefits

- Preserve crucial design information used by applications & middleware frameworks & components
- Facilitate reuse of proven software designs & architectures
- Guide design choices for application developers

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.

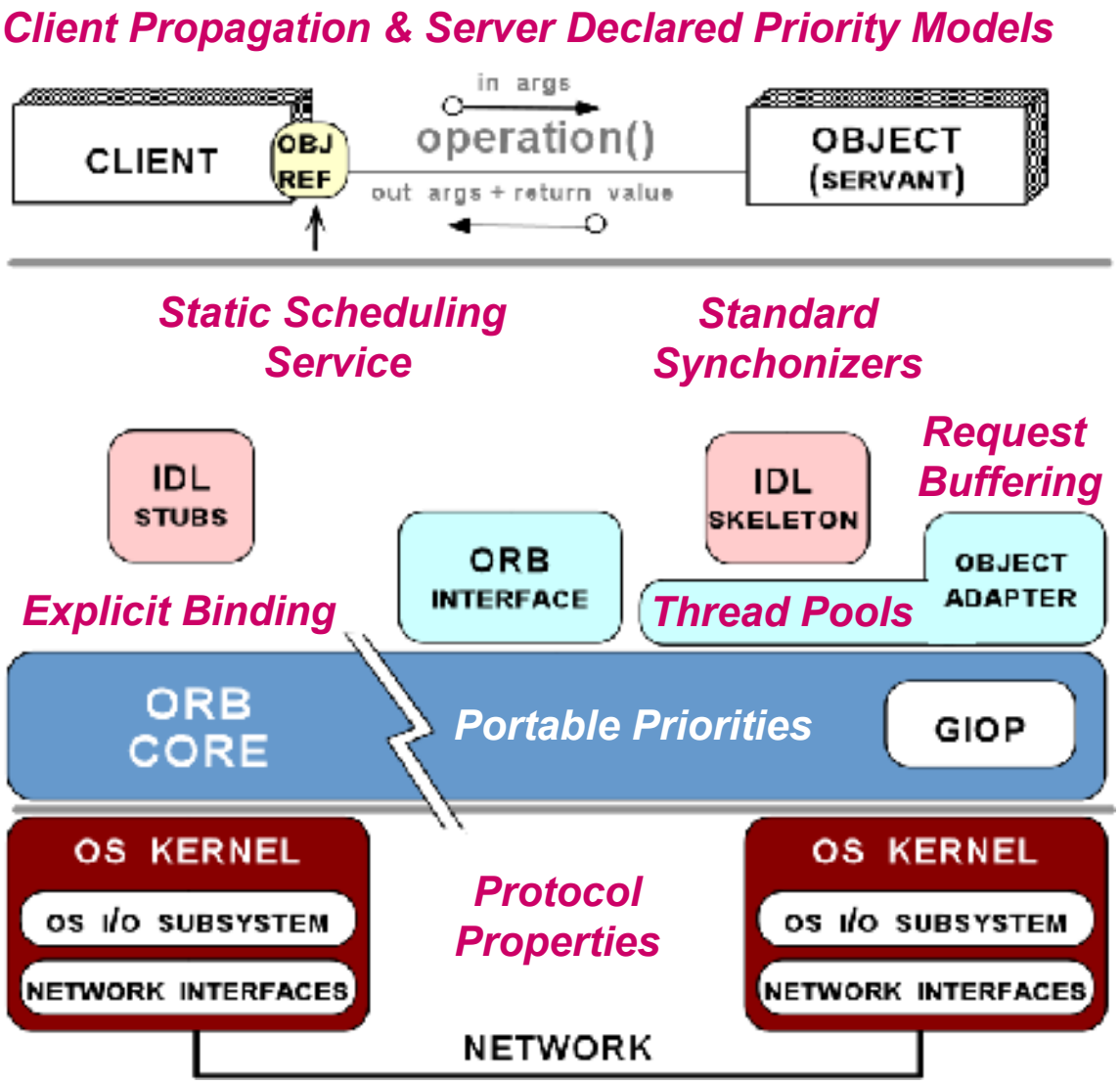
The *Monitor Object* design pattern synchronizes concurrent method execution to ensure that only one method at a time runs within an object. It also allows an object's methods to cooperatively schedule their execution sequences.

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.

Implementing the Broker Pattern for Bold Stroke Avionics



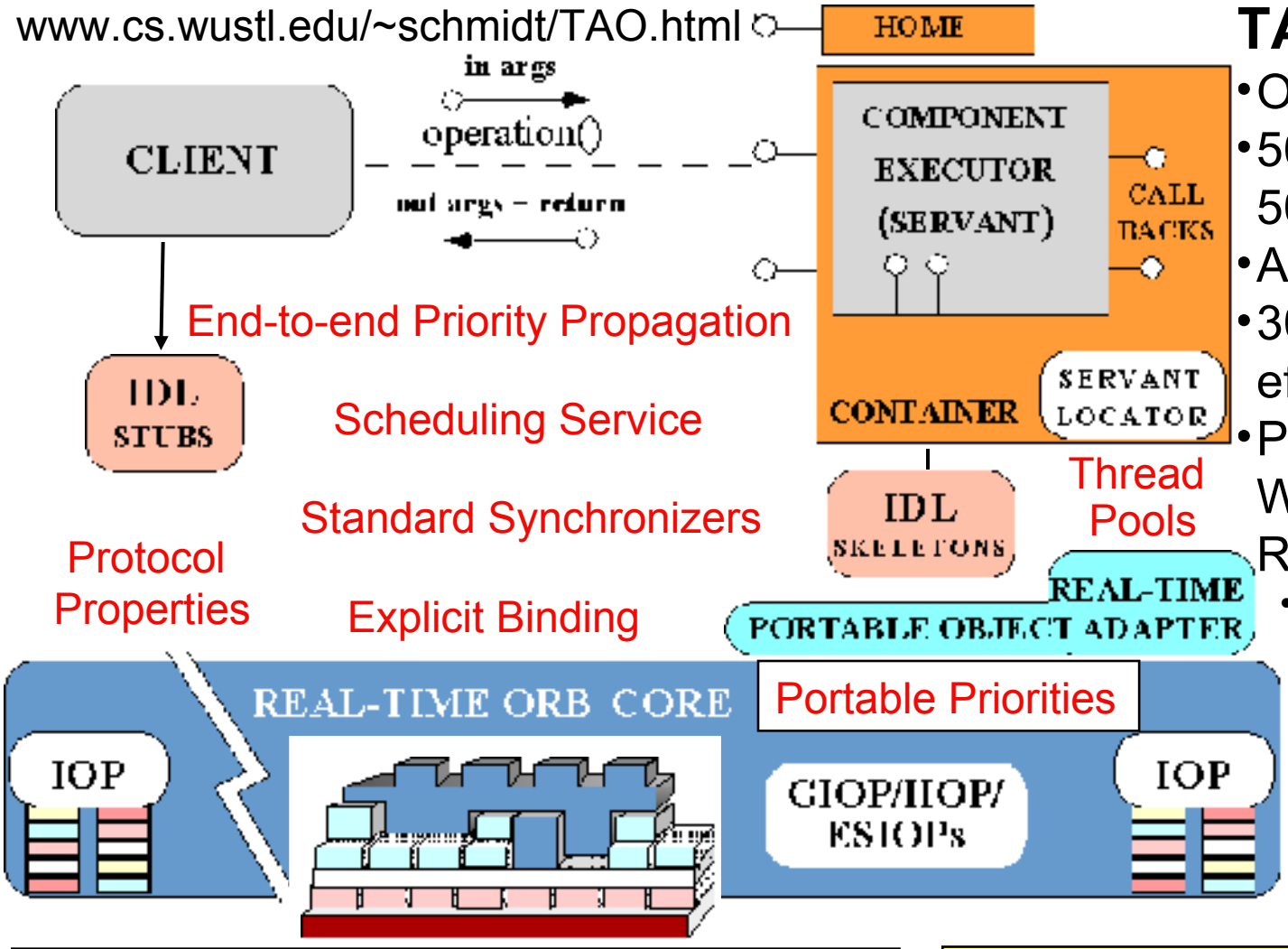
- **CORBA** is a distribution middleware standard
- ***Real-time CORBA*** adds QoS to classic CORBA to control:
  1. ***Processor Resources***
  2. ***Communication Resources***
  3. ***Memory Resources***
- These capabilities address some (but by no means all) important DRE application development & QoS-enforcement challenges

www.omg.org

Example of Applying Patterns & Frameworks to Middleware:

Real-time CORBA & The ACE ORB (TAO)

www.cs.wustl.edu/~schmidt/TAO.html



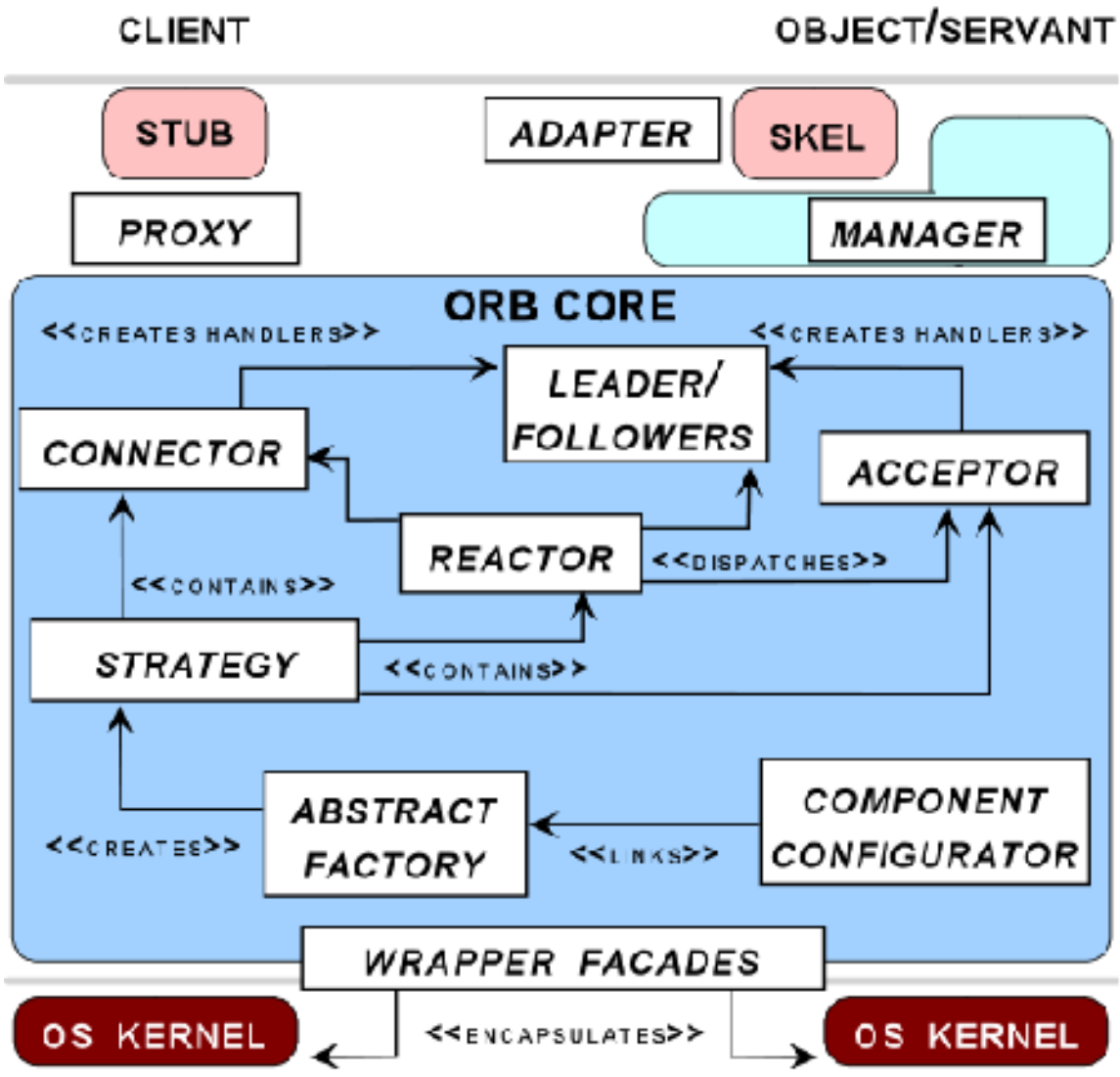
TAO Features

- 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

- **Commercially supported**
  - www.theaceorb.com
  - www.prismtechnologies.com

Key Patterns Used in TAO

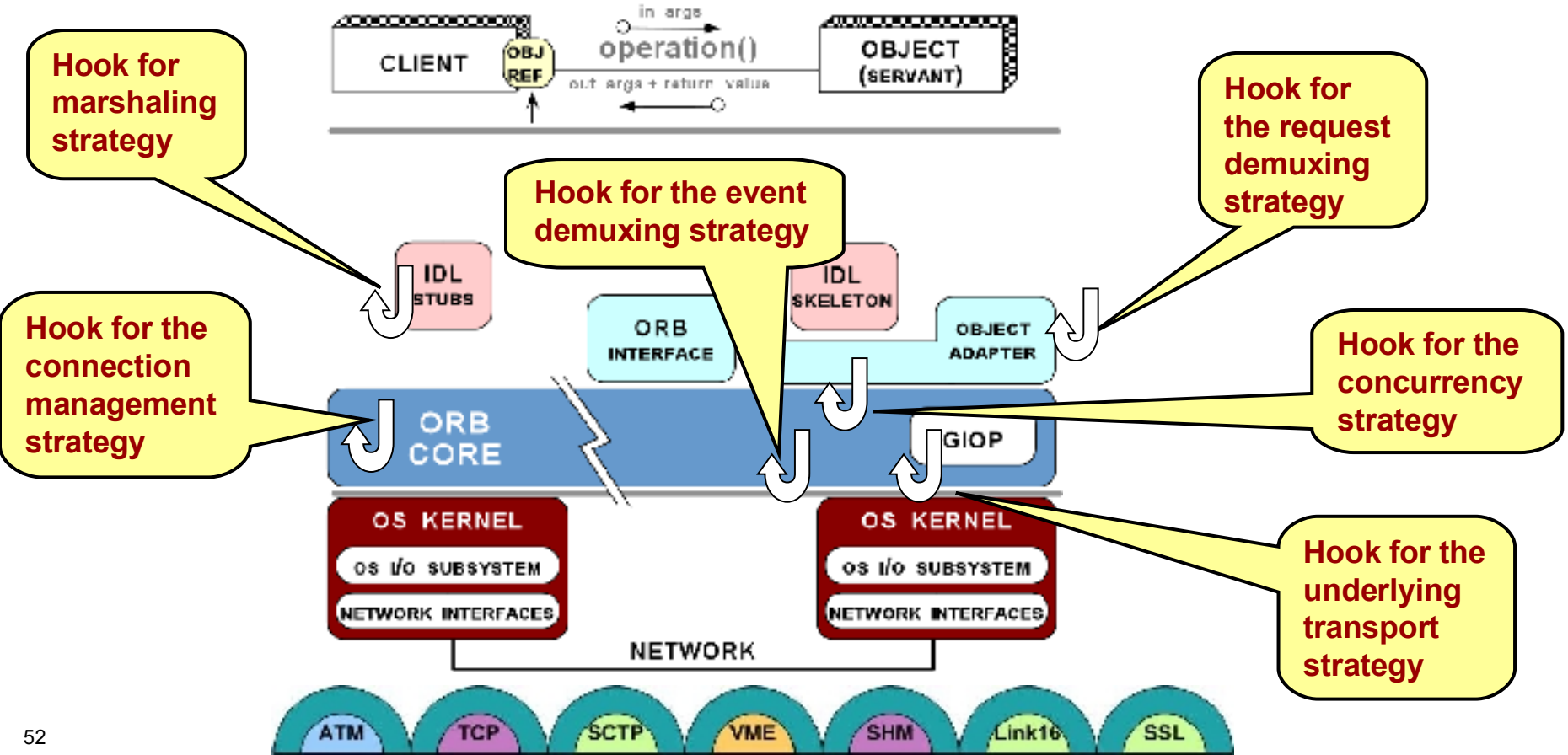


- ***Wrapper facades*** enhance portability
- ***Proxies & adapters*** simplify client & server applications, respectively
- ***Component Configurator*** dynamically configures ***Factories***
- ***Factories*** produce ***Strategies***
- ***Strategies*** implement interchangeable policies
- Concurrency strategies use ***Reactor*** & ***Leader/Followers***
- ***Acceptor-Connector*** decouples connection management from request processing
- ***Managers*** optimize request demultiplexing

www.cs.wustl.edu/~schmidt/PDF/ORB-patterns.pdf

Enhancing ORB Flexibility w/the Strategy Pattern

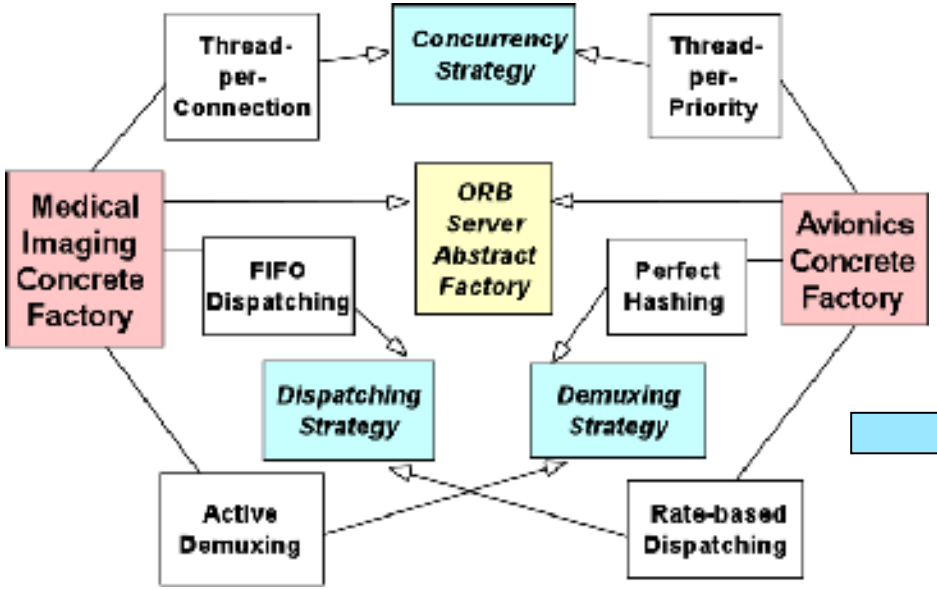
Context	Problem	Solution
• Multi-domain resuable middleware framework	• Flexible ORBs must support multiple event & request demuxing, scheduling, (de)marshaling, connection mgmt, request transfer, & concurrency policies	• Apply the *Strategy* pattern to factory out similarity amongst alternative ORB algorithms & policies

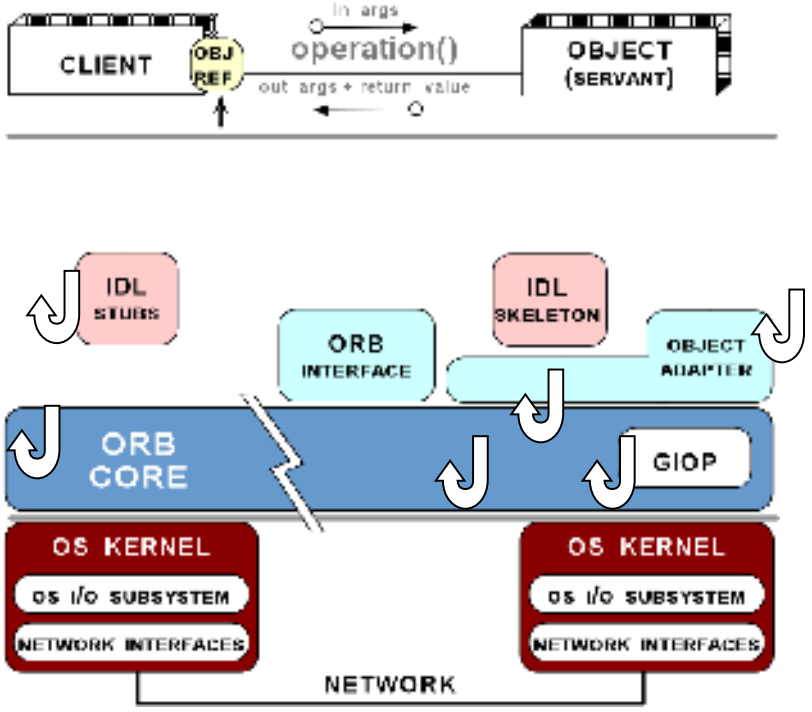


Consolidating Strategies with the Abstract Factory Pattern

Context	Problem	Solution

• A heavily strategized framework or application	• Aggressive use of Strategy pattern creates a configuration nightmare
	  • Managing many individual strategies is hard
	  • It's hard to ensure that groups of semantically compatible strategies are configured
		• Apply the *Abstract Factory* pattern to consolidate multiple ORB strategies into semantically compatible configurations



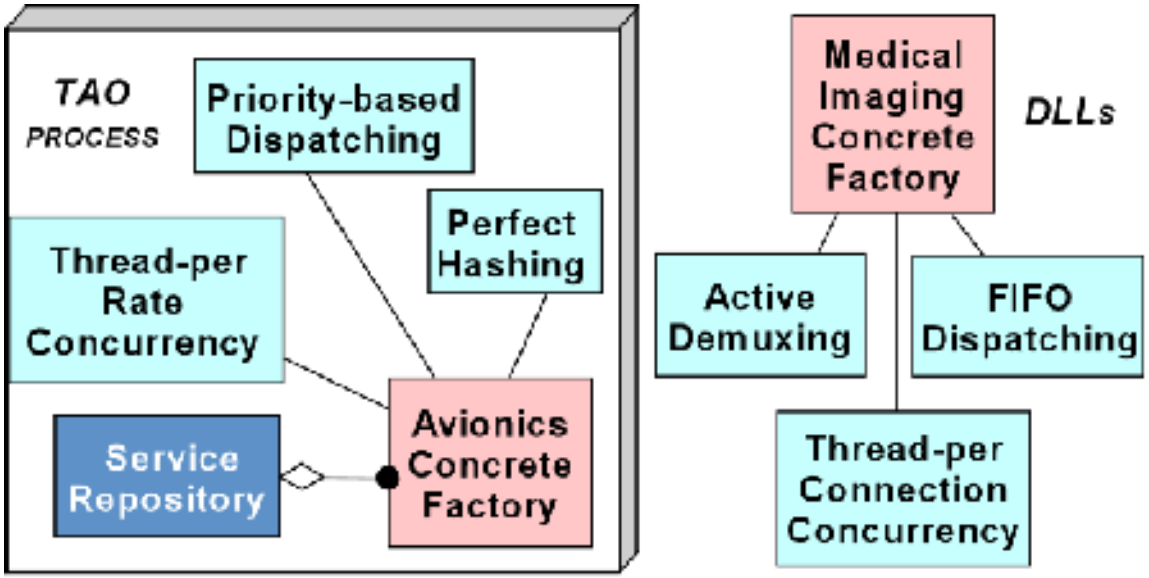


Concrete factories create groups of strategies

Dynamically Configuring Factories w/the Component Configurator Pattern

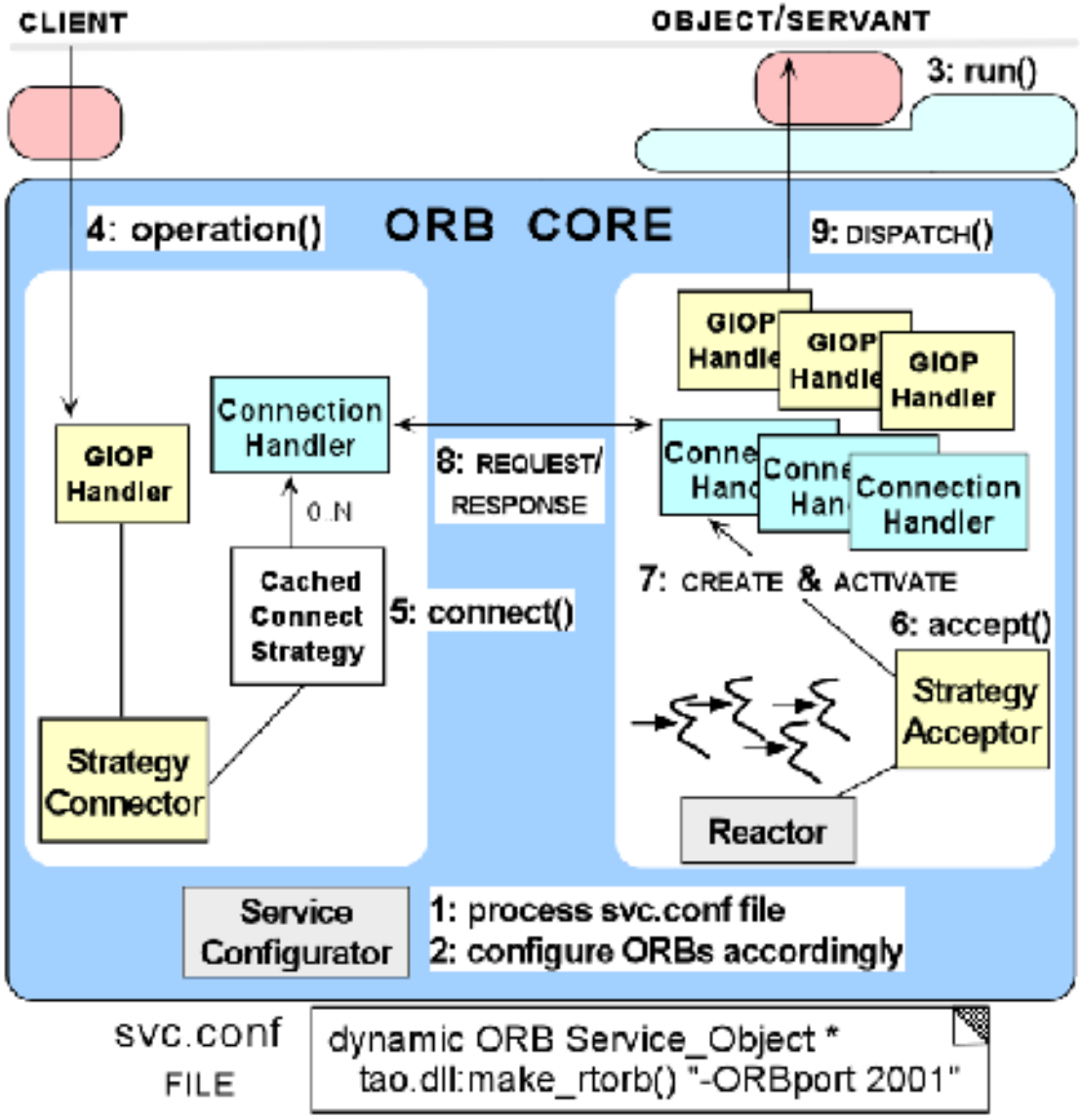
Context	Problem	Solution

• Resource constrained & highly dynamic environments	• Prematurely commiting to a particular ORB configuration is inflexible & inefficient
	  • Certain decisions can't be made until runtime
	  • Forcing users to pay for components that don't use is undesirable
		• Apply the *Component Configurator* pattern to assemble the desired ORB factories (& thus strategies) dynamically



- ORB strategies are decoupled from when the strategy implementations are configured into an ORB
- This pattern can reduce the memory footprint of an ORB

ACE Frameworks Used in TAO



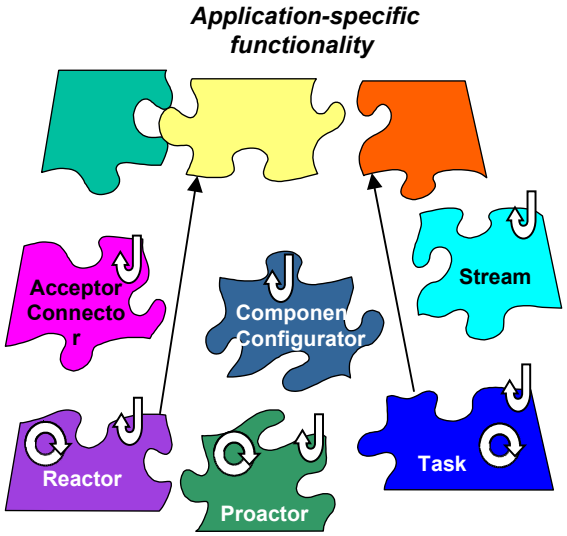
- ***Reactor*** drives the ORB event loop
  - Implements the ***Reactor*** & ***Leader/Followers*** patterns
- ***Acceptor-Connector*** decouples passive/active connection roles from GIOP request processing
  - Implements the ***Acceptor-Connector*** & ***Strategy*** patterns
- ***Service Configurator*** dynamically configures ORB strategies
  - Implements the ***Component Configurator*** & ***Abstract Factory*** patterns

www.cs.wustl.edu/~schmidt/PDF/ICSE-03.pdf

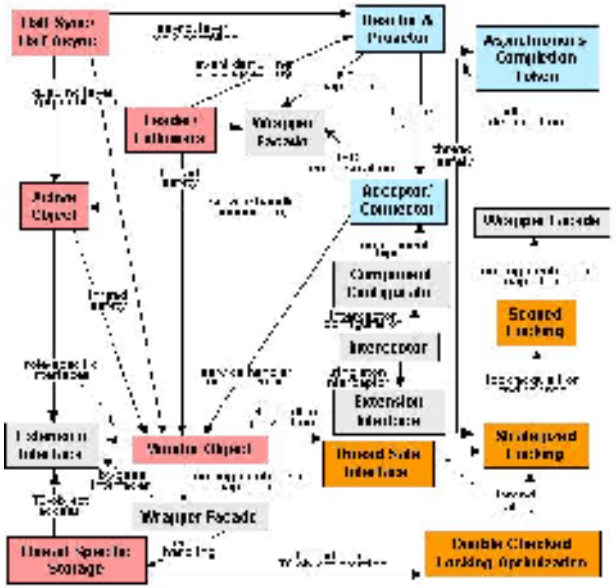
Summary of Pattern, Framework, & Middleware Synergies

The technologies codify expertise of experienced researchers & developers

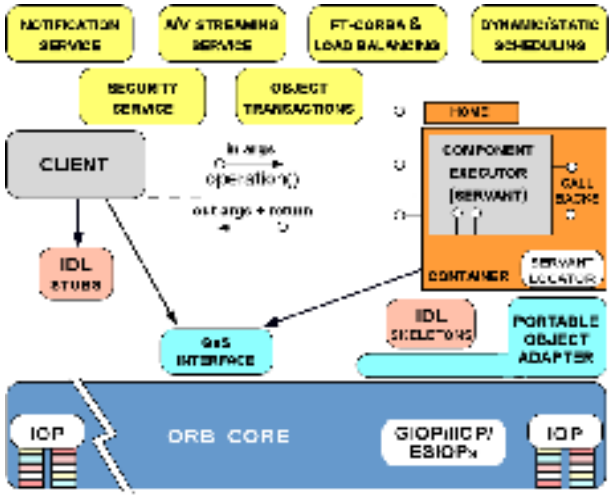
- Frameworks codify expertise in the form of reusable algorithms, component implementations, & extensible architectures



- Patterns codify expertise in the form of reusable architecture design themes & styles, which can be reused event when algorithms, components implementations, or frameworks cannot



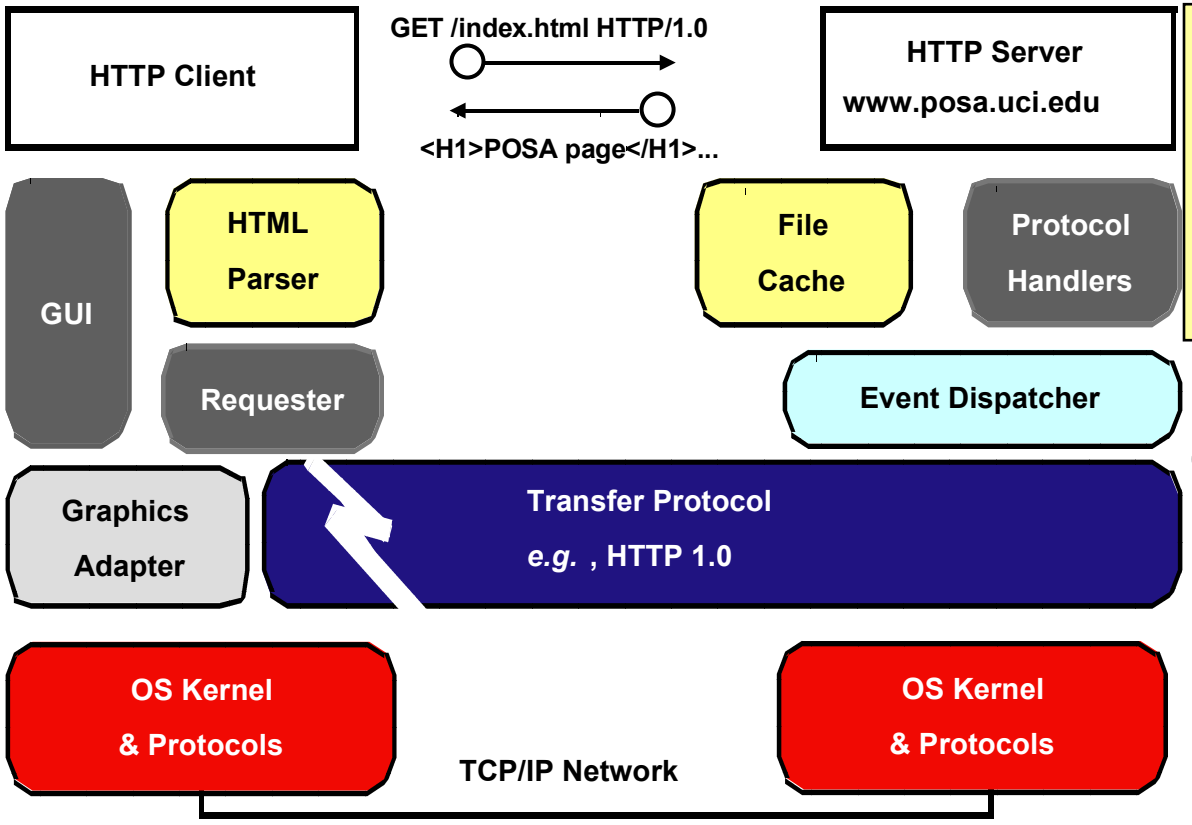
- Middleware codifies expertise in the form of standard interfaces & components that provide applications with a simpler façade to access the powerful (& complex) capabilities of frameworks



There are now powerful feedback loops advancing these technologies

Tutorial Example:

High-performance Content Delivery Servers



Goal

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

Key System Characteristics

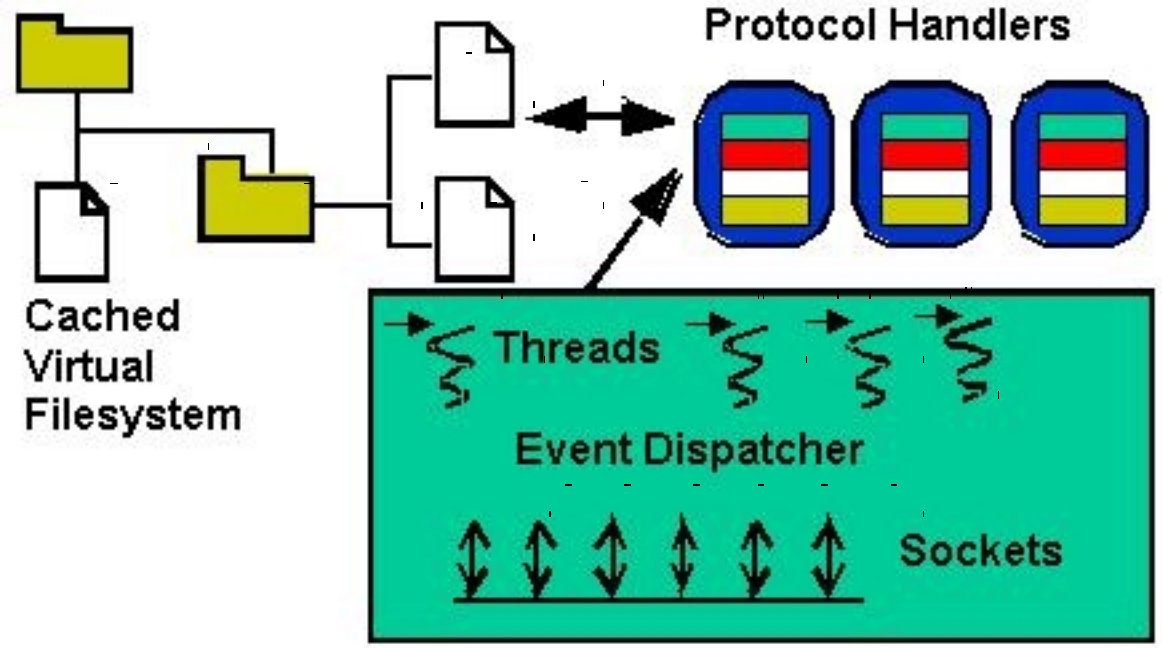
- Robust implementation
  - *e.g.,* stop malicious clients
- Extensible to other protocols
  - *e.g.,* HTTP 1.1, IIOP, DICOM
- Leverage advanced multi-processor hardware & software

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.

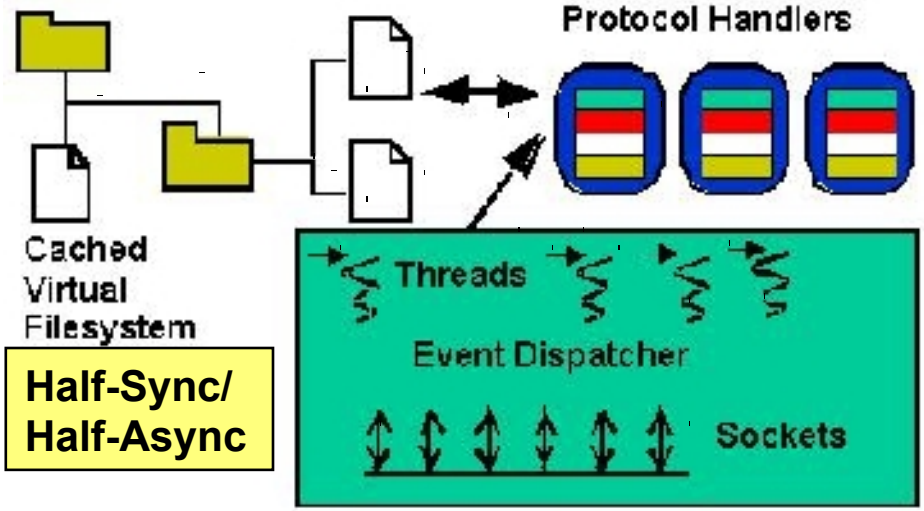
Protocol Handler

- 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



Component Configurator

Double-checked Locking Optimization

Thread-safe Interface

Strategized Locking

Scoped Locking

Half-Sync/ Half-Async

Acceptor-Connector

Leader/Followers

Proactor

Reactor

Monitor Object

Wrapper Facade

Thread-specific Storage

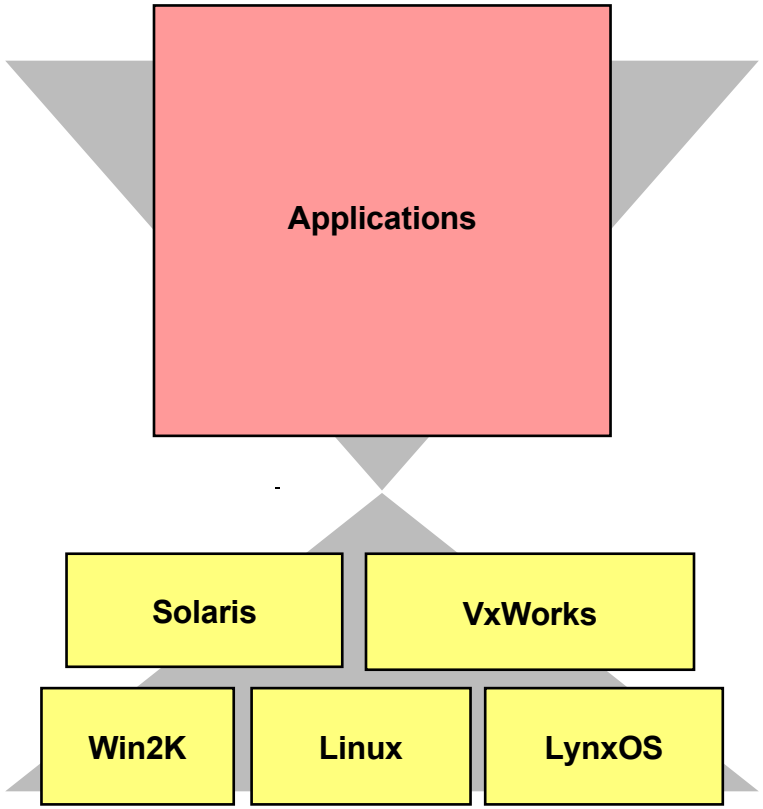
Patterns help resolve the following common design challenges:

- Encapsulating low-level OS APIs
- Decoupling event demuxing & 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
- Enhancing Server (Re)Configurability
- 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 (1/2)

Context

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



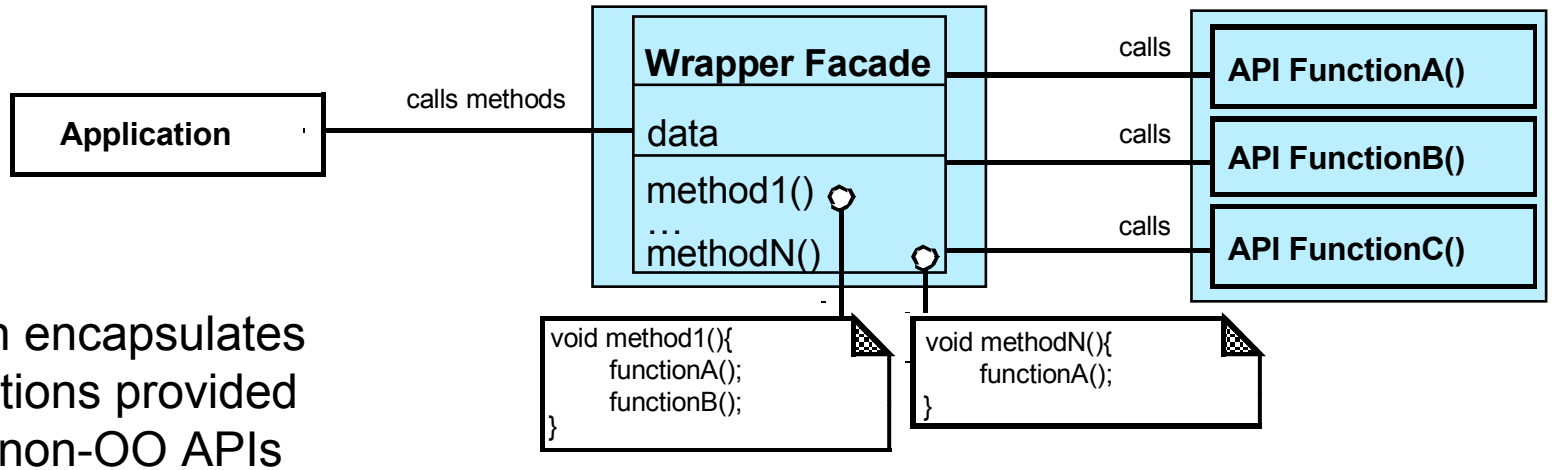
Problem

- The diversity of hardware & operating systems makes it hard to build portable & robust Web server software
- Programming directly to low-level OS APIs is tedious, error-prone, & non-portable

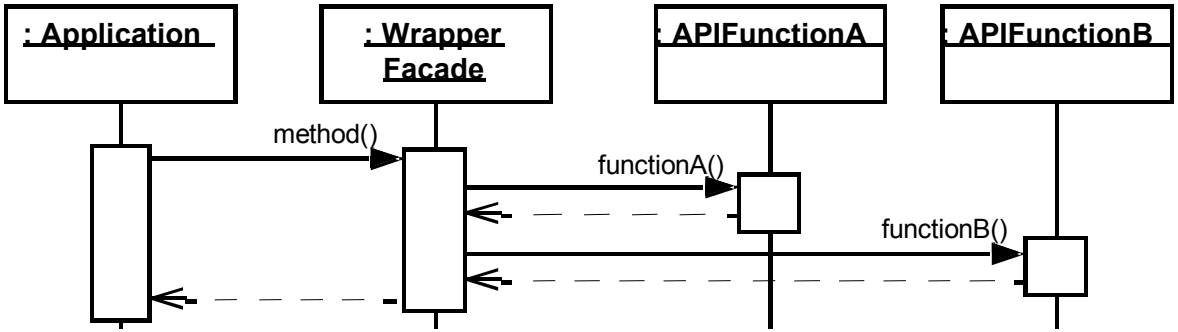
Encapsulating Low-level OS APIs (2/2)

Solution

- Apply the *Wrapper Facade* design pattern (P2) to avoid accessing low-level operating system APIs directly



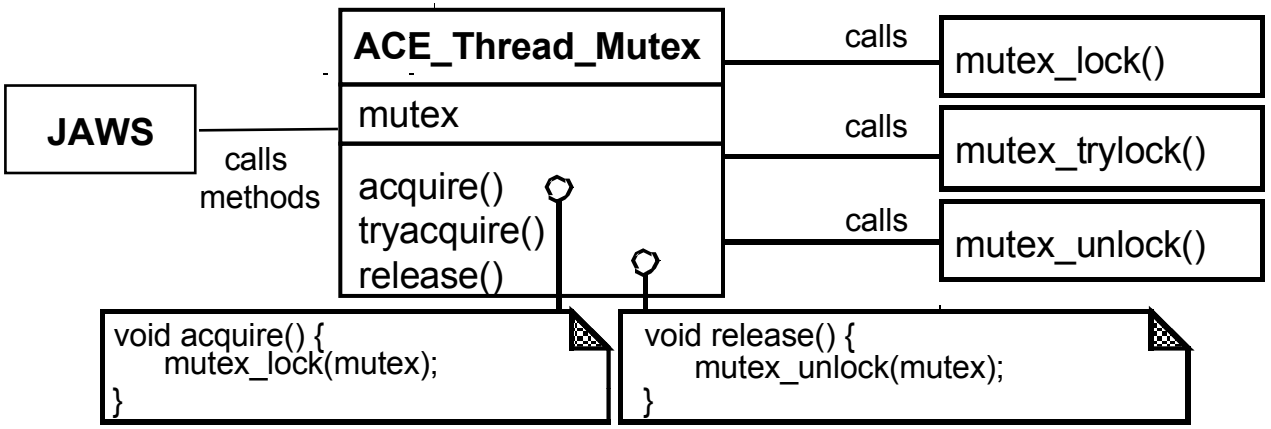
This pattern encapsulates data & functions provided by existing non-OO APIs within more concise, robust, portable, maintainable, & cohesive OO class interfaces



Applying the Wrapper Façade Pattern in JAWS

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

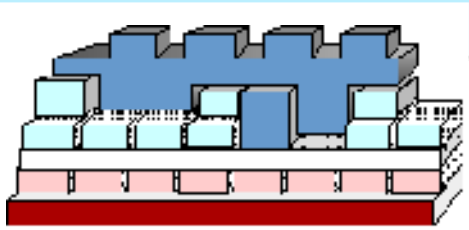
For example, JAWS uses the **`ACE_Thread_Mutex`** wrapper facade in ACE to provide a portable interface to OS mutual exclusion mechanisms



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

`ACE_Thread_Mutex` is also available for other threading APIs, *e.g.,* VxWorks, LynxOS, Windows, or POSIX threads

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

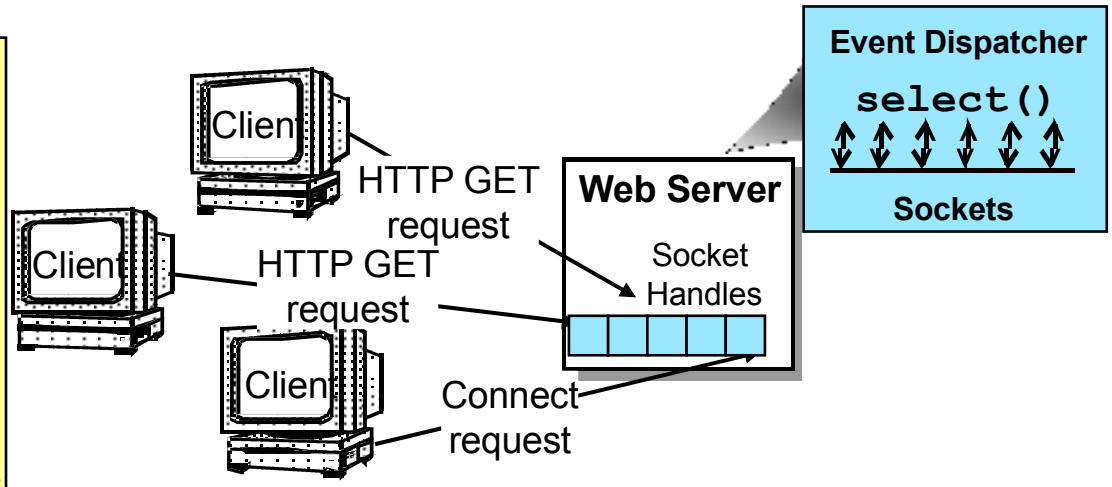
This pattern can incur **liabilities*:***

- ***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, & Protocol Processing (1/2)

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()`**



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

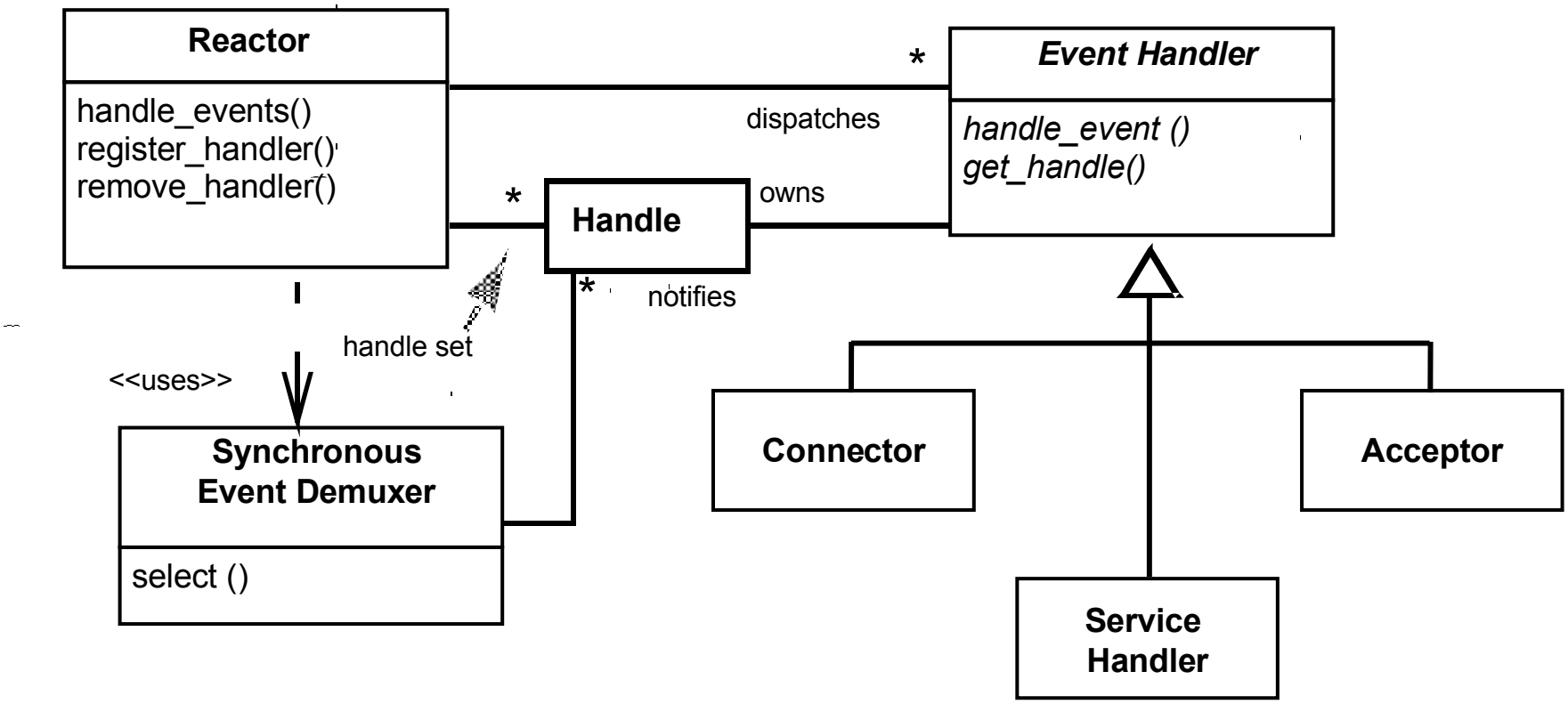
```
f (FD_ISSET (acceptor,  &ready_handles)) {
  int h;

  do {
    h = accept (acceptor, 0, 0);
    char buf[BUFSIZ];
    for (ssize_t i;  (i = read (h, buf, BUFSIZ)) > 0; )
       write (1, buf, i);
  } while (h != -1);
```

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

Decoupling Event Demuxing, Connection Management, & Protocol Processing (2/2)

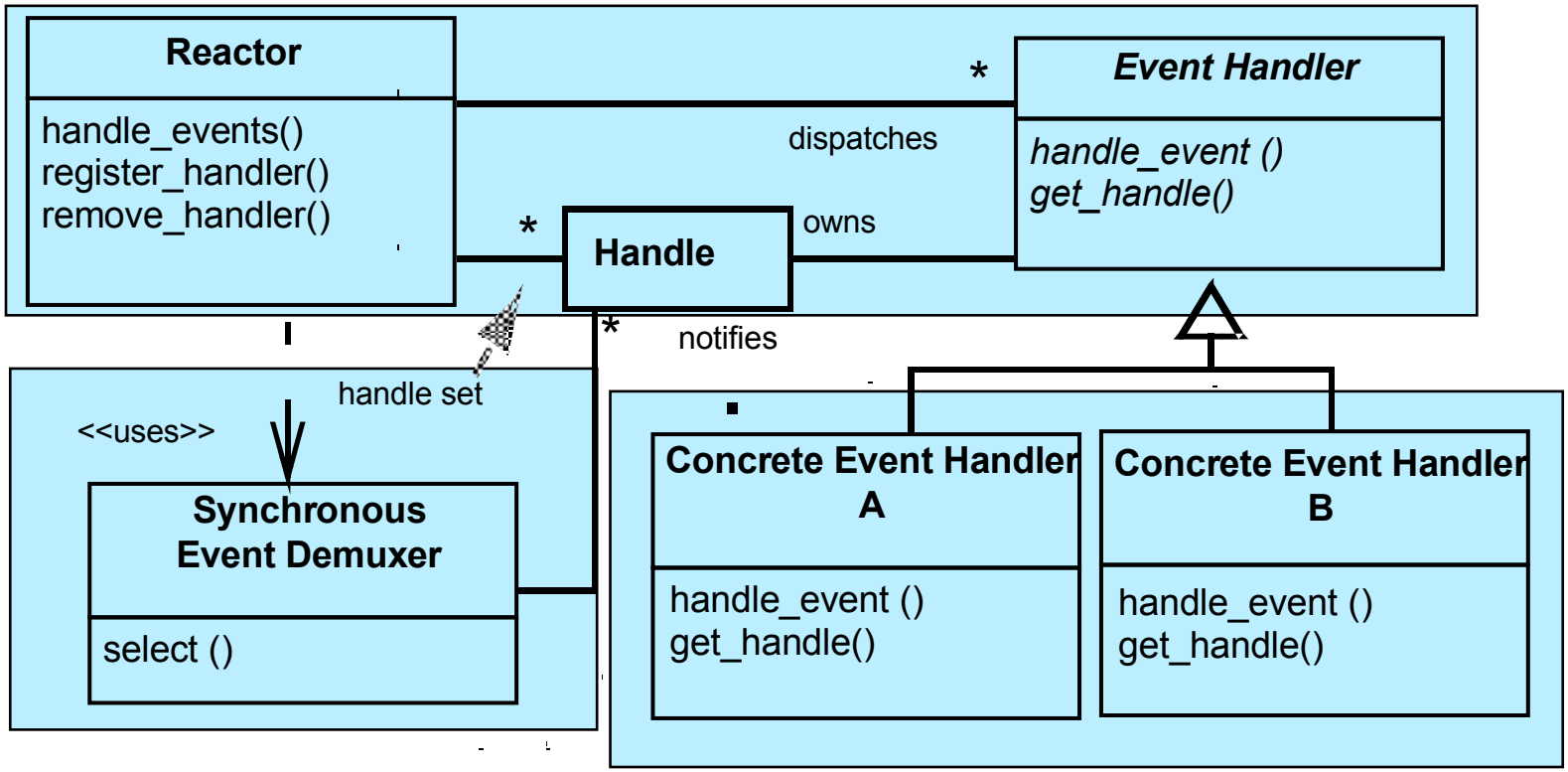
Solution

Apply the *Reactor* architectural pattern (P2) & the *Acceptor-Connector* design pattern (P2) to separate the generic event-demultiplexing & connection-management code from the web server's protocol code

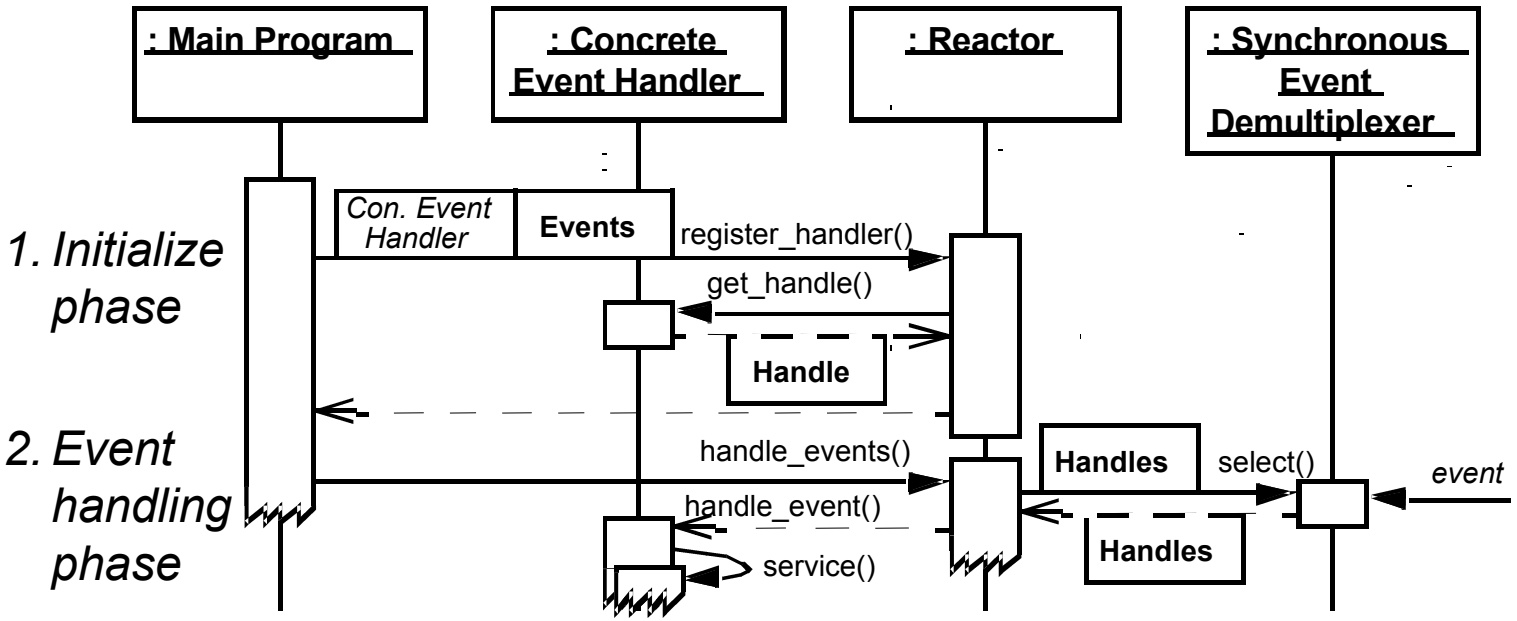


The Reactor Pattern

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



Reactor Pattern Dynamics

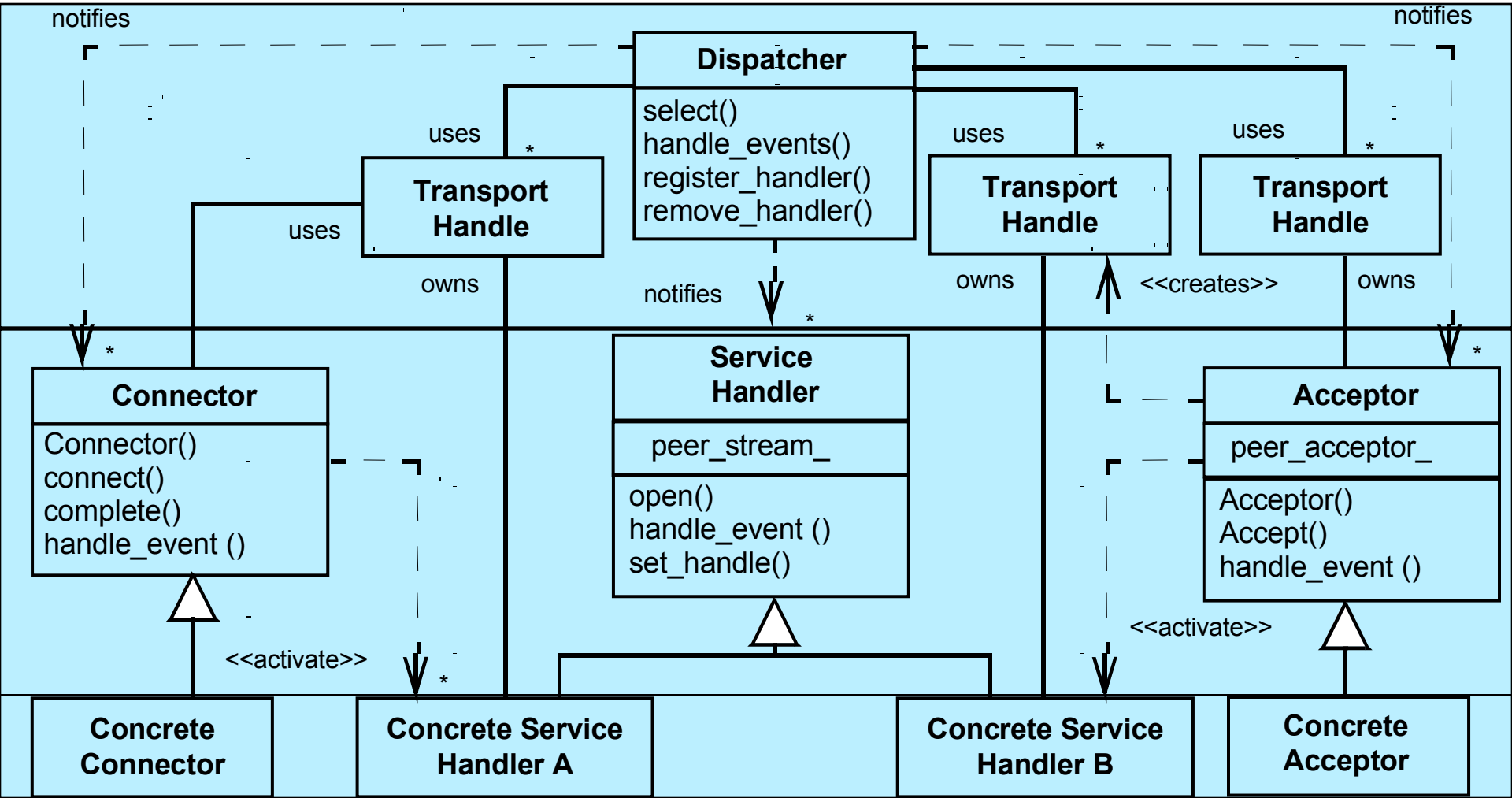


Observations

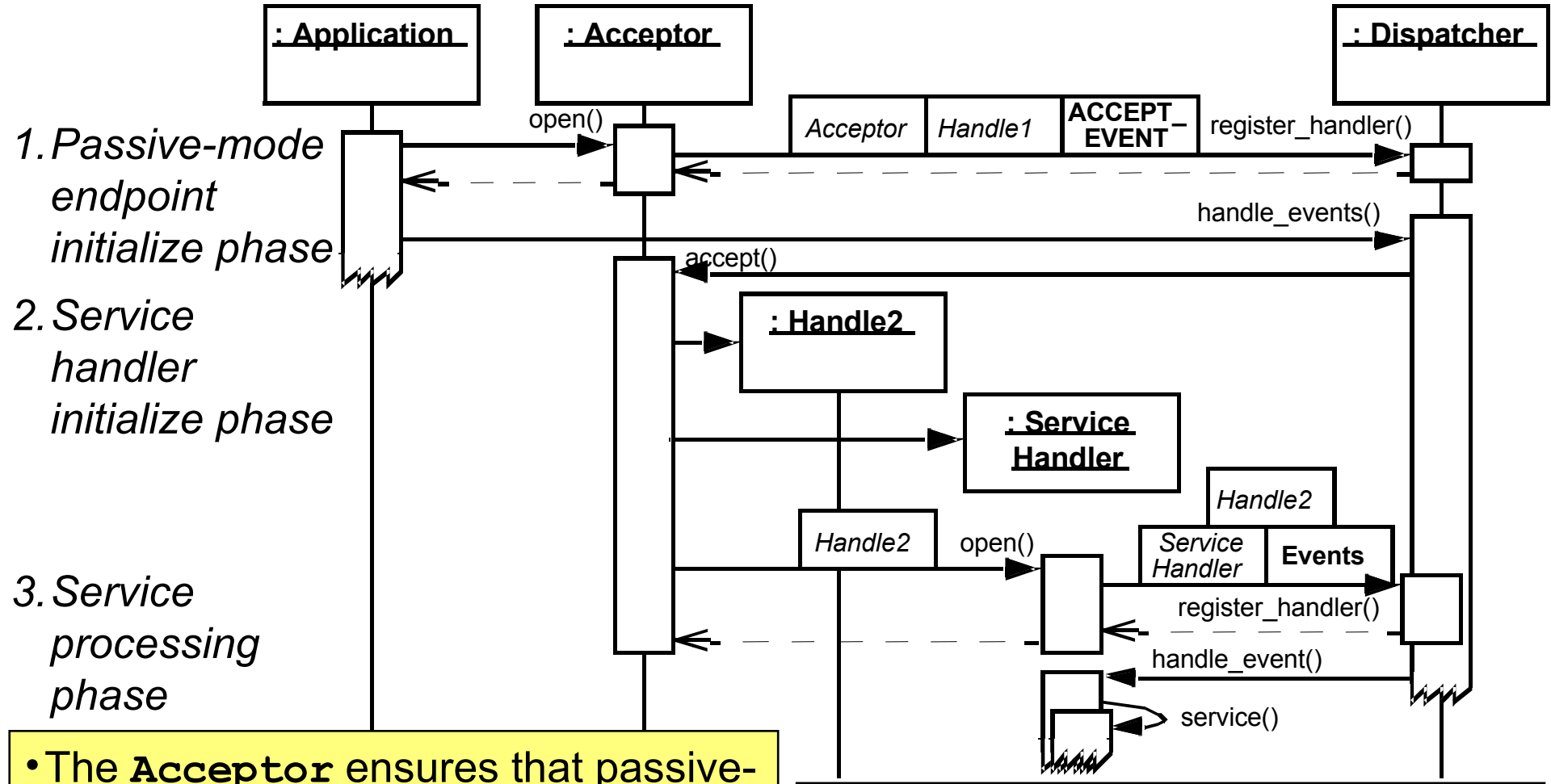
- Note inversion of control
- Also note how long-running event handlers can degrade the QoS since callbacks steal the reactor's thread!

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



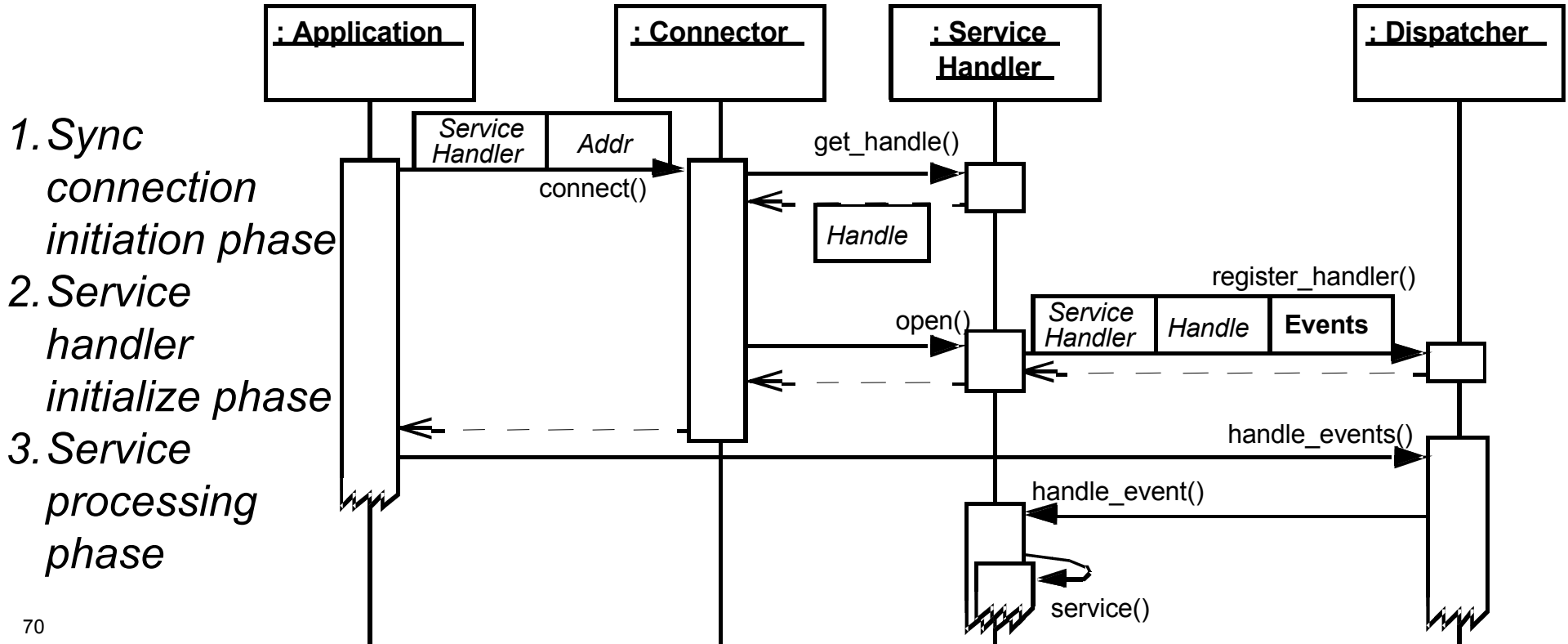
- The **`Acceptor`** ensures that passive-mode transport endpoints aren't used to read/write data accidentally
  - And vice versa for data transport endpoints…

- 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

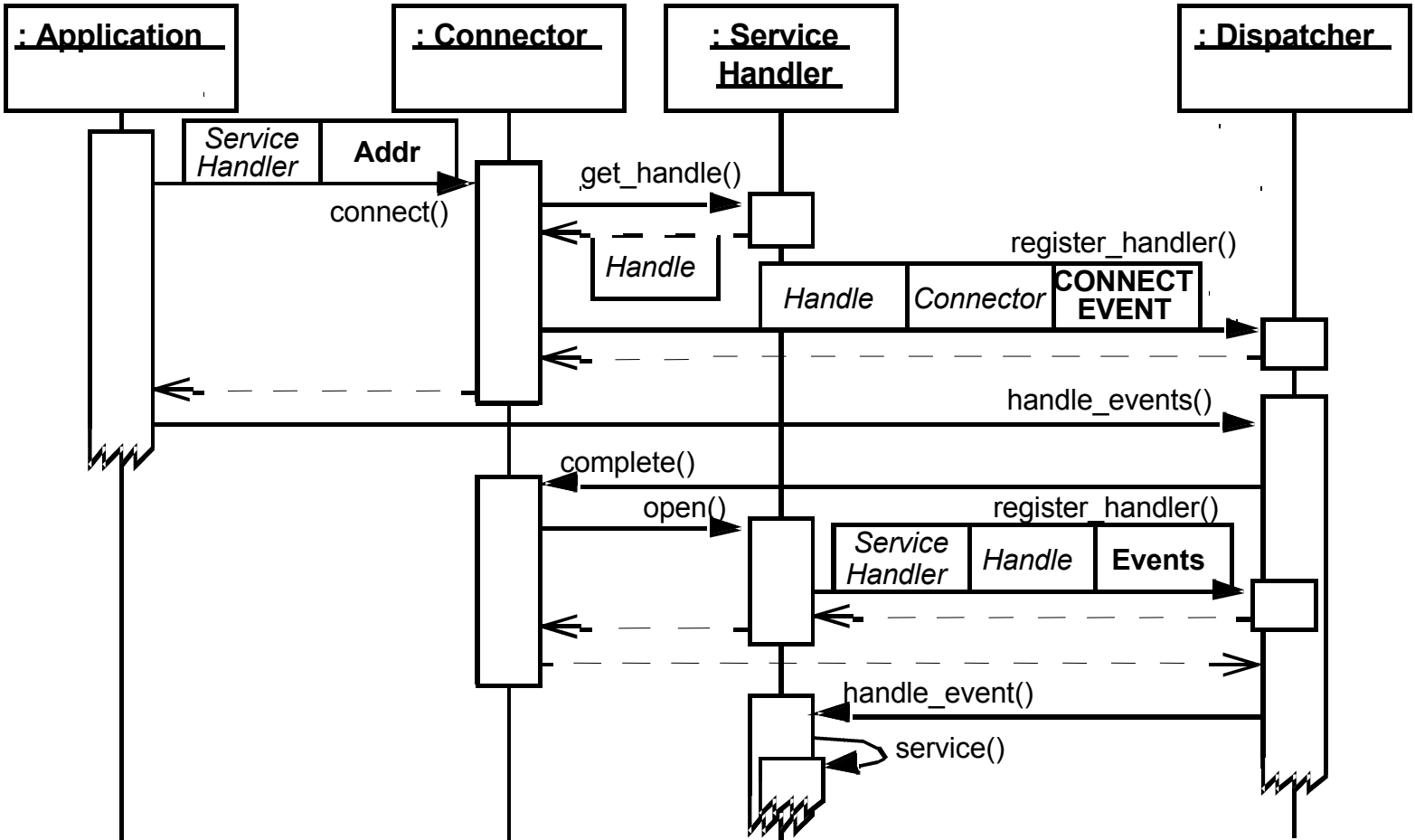


Asynchronous Connector Dynamics

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

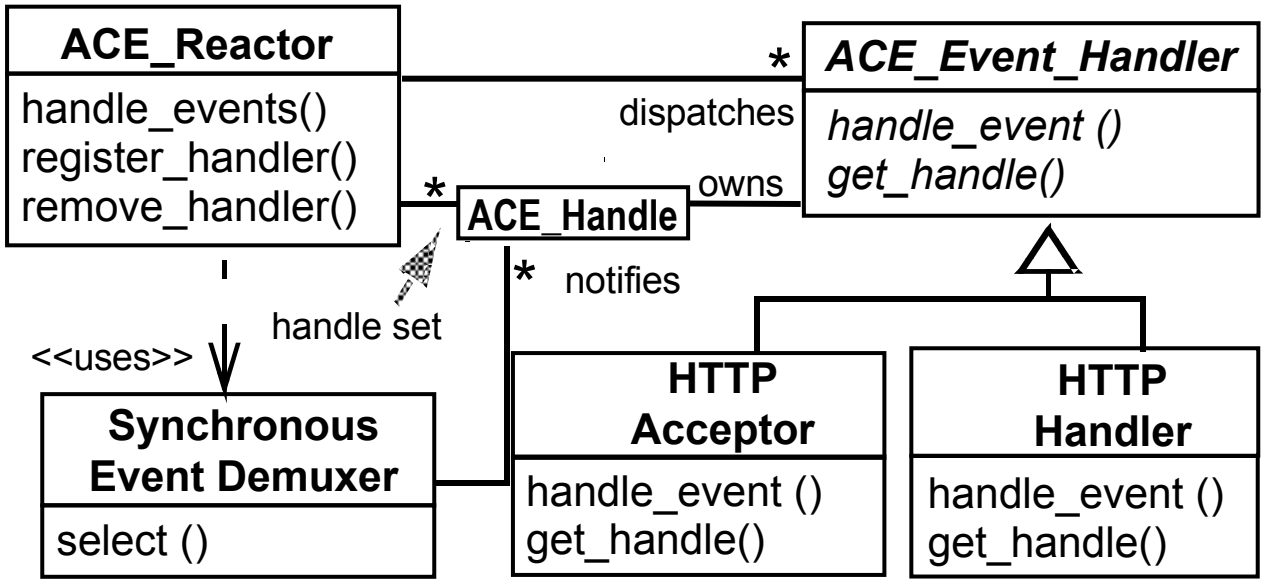
1. *Async connection initiation phase*
2. *Service handler initialize phase*
3. *Service processing phase*



Applying the Reactor and Acceptor-Connector Patterns in JAWS

The Reactor architectural pattern decouples:

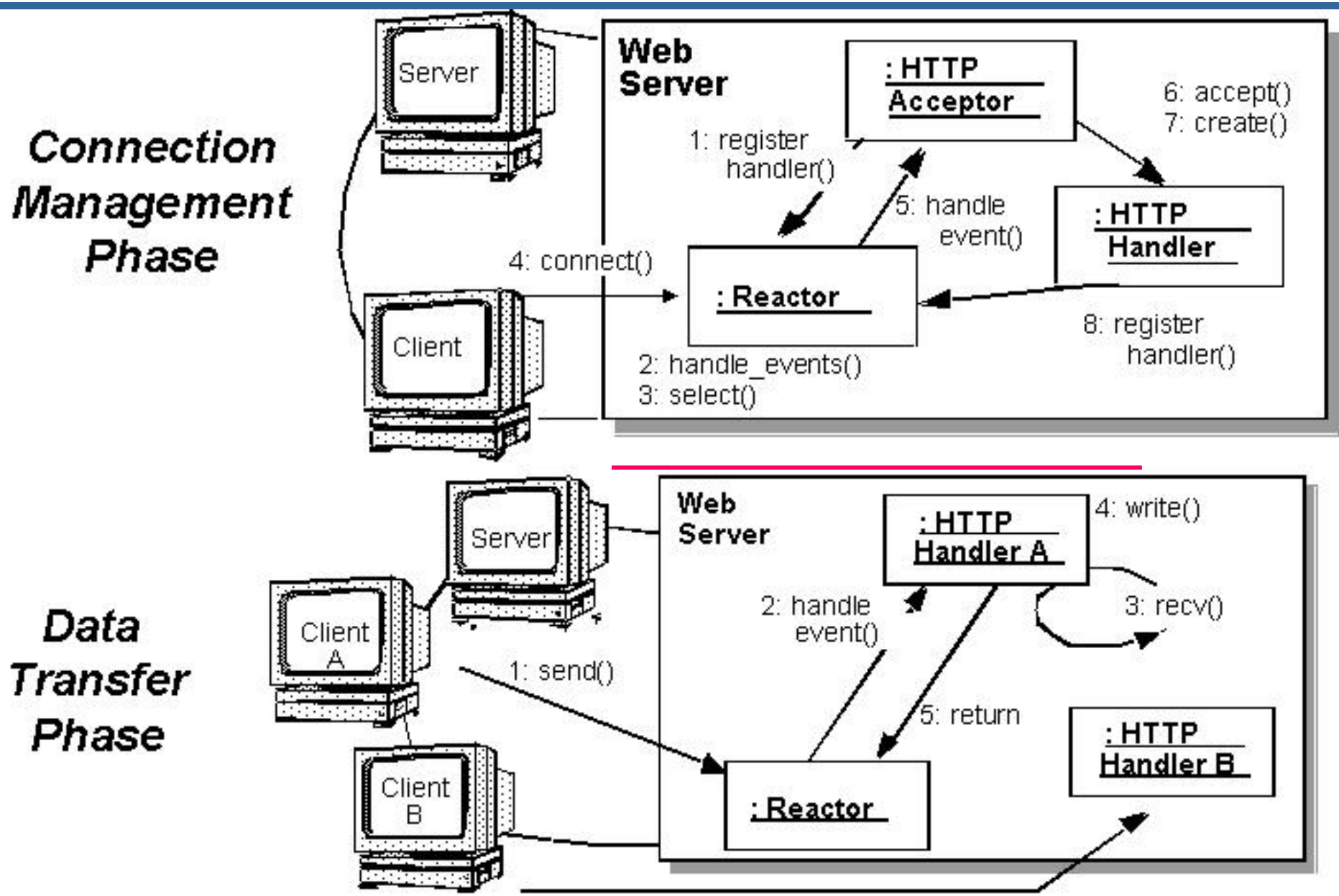
1. JAWS generic synchronous event demultiplexing & dispatching logic from
2. The HTTP protocol processing it performs in response to events



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 after 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 & Cons of 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:**

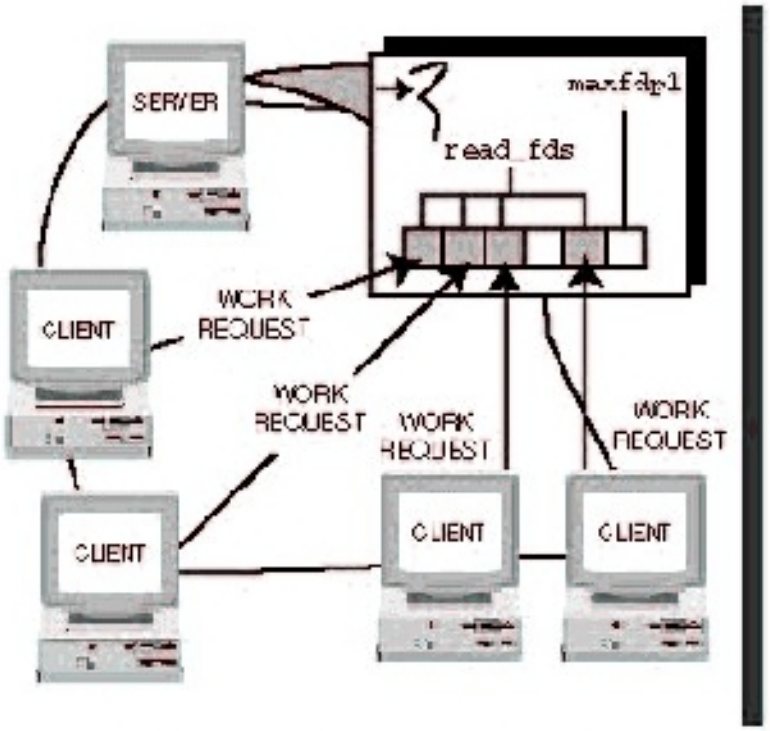
- ***Additional indirection***
  - The Acceptor-Connector pattern can incur additional indirection compared to using the underlying network programming interfaces directly
- ***Additional complexity***
  - The Acceptor-Connector pattern may add unnecessary complexity for simple client applications that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

Overview of Concurrency & Threading

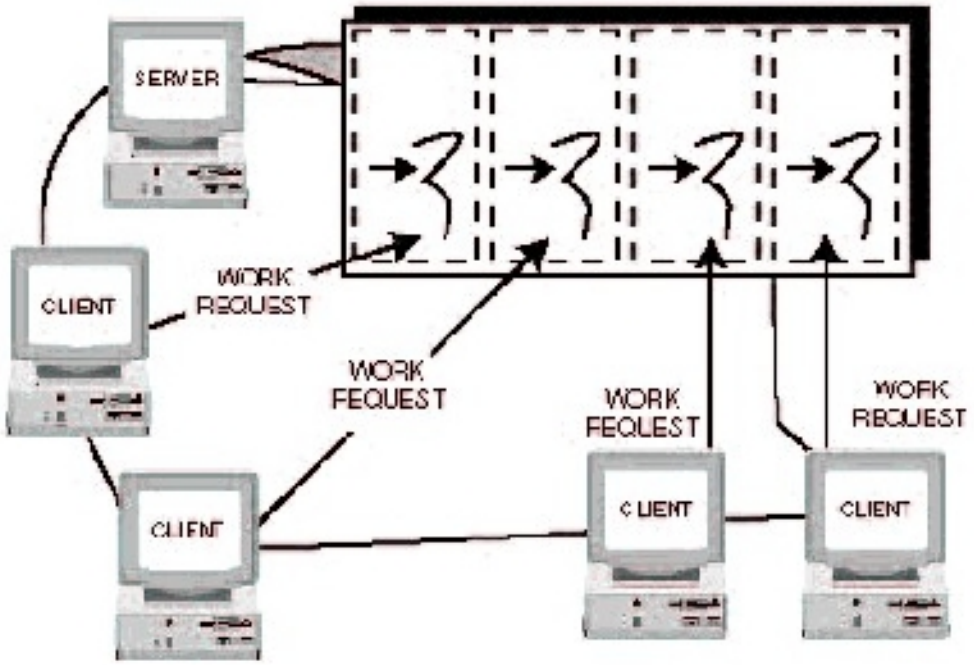
- Thus far, our web server has been entirely reactive, which can be a bottleneck for scalable systems
- Multi-threading is essential to develop scalable & robust networked applications, particularly servers
- The next group of slides present a domain analysis of concurrency design dimensions that address the policies & mechanisms governing the proper use of processes, threads, & synchronizers

- We outline the following design dimensions in this discussion:
  - Iterative versus concurrent versus reactive servers
  - Processes versus threads
  - Process/thread spawning strategies
  - User versus kernel versus hybrid threading models
  - Time-shared versus real-time scheduling classes

Iterative vs. Concurrent Servers



(1) ITERATIVE/REACTIVE SERVER

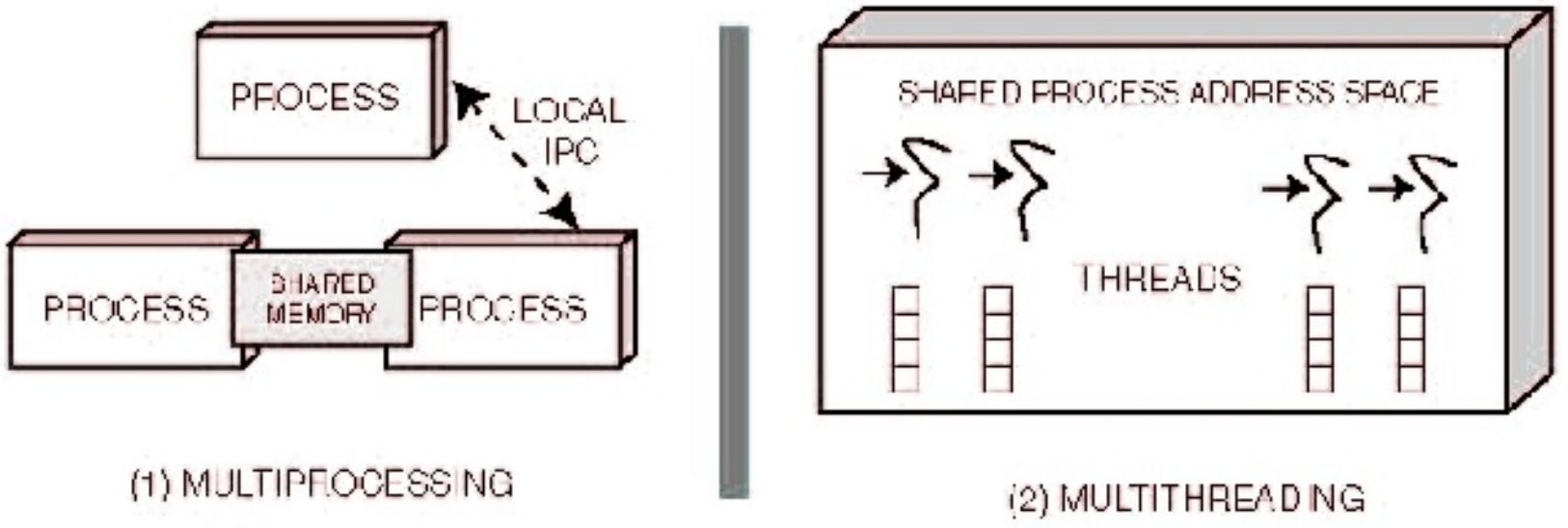


(2) CONCURRENT SERVER

- Iterative/reactive servers handle each client request in its entirety before servicing subsequent requests
- Best suited for short-duration or infrequent services

- Concurrent servers handle multiple requests from clients simultaneously
- Best suited for I/O-bound services or long-duration services
- Also good for busy servers

Multiprocessing vs. Multithreading

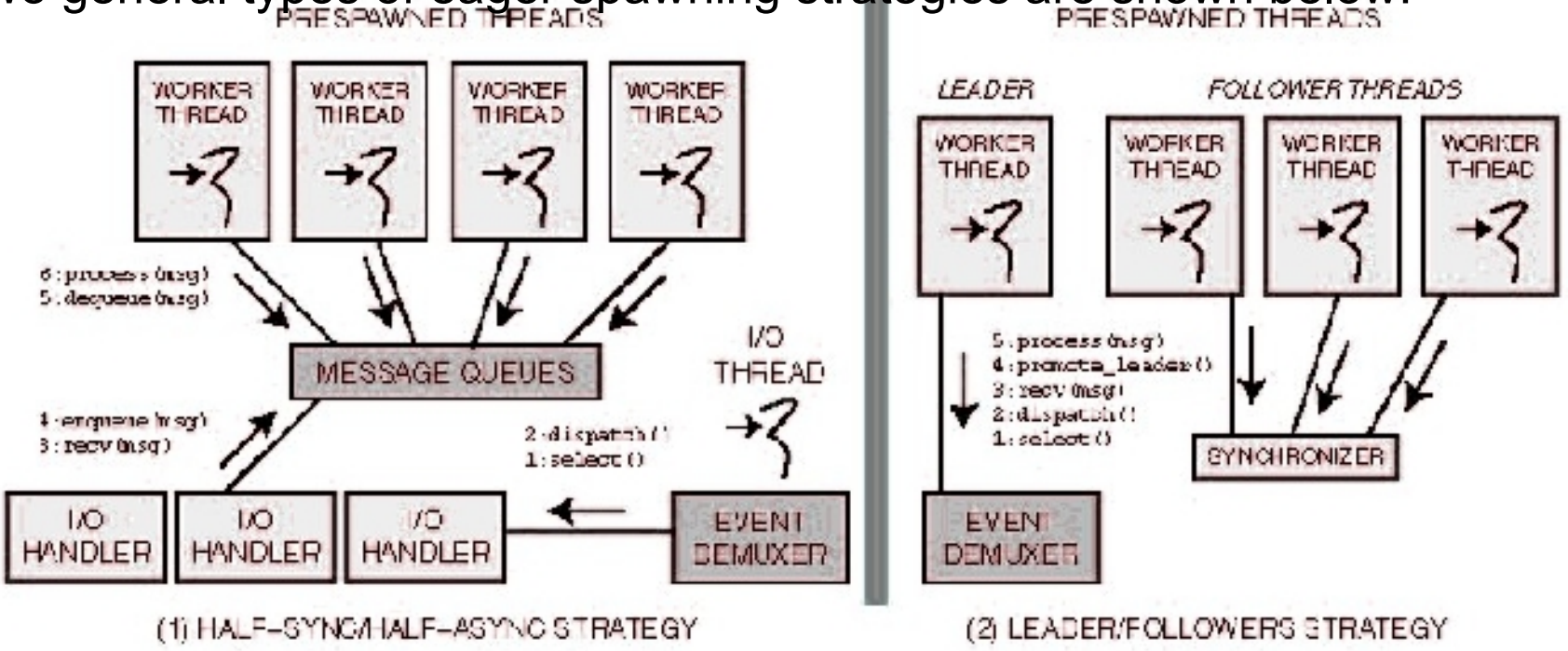


- A process provides the context for executing program instructions
- Each process manages certain resources (such as virtual memory, I/O handles, and signal handlers) & is protected from other OS processes via an MMU
- IPC between processes can be complicated & inefficient

- A thread is a sequence of instructions in the context of a process
- Each thread manages certain resources (such as runtime stack, registers, signal masks, priorities, & thread-specific data)
- Threads are not protected from other threads
- IPC between threads can be more efficient than IPC between processes

Thread Pool Eager Spawning Strategies

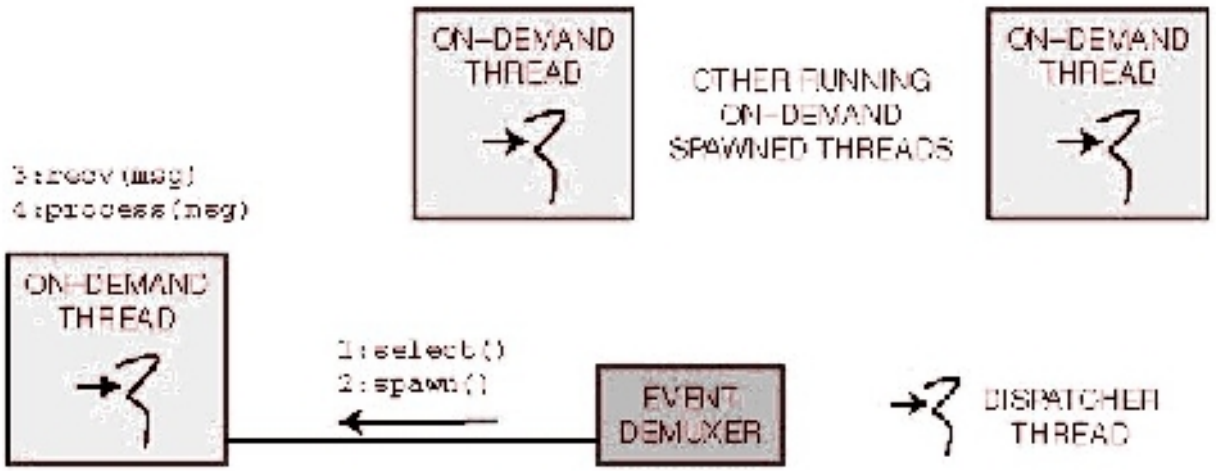
- This strategy prespawns one or more OS processes or threads at server creation time
- These``warm-started'' execution resources form a pool that improves response time by incurring service startup overhead before requests are serviced
- Two general types of eager spawning strategies are shown below:



- These strategies based on Half-Sync/Half-Async & Leader/Followers patterns

Thread-per-Request On-demand Spawning Strategy

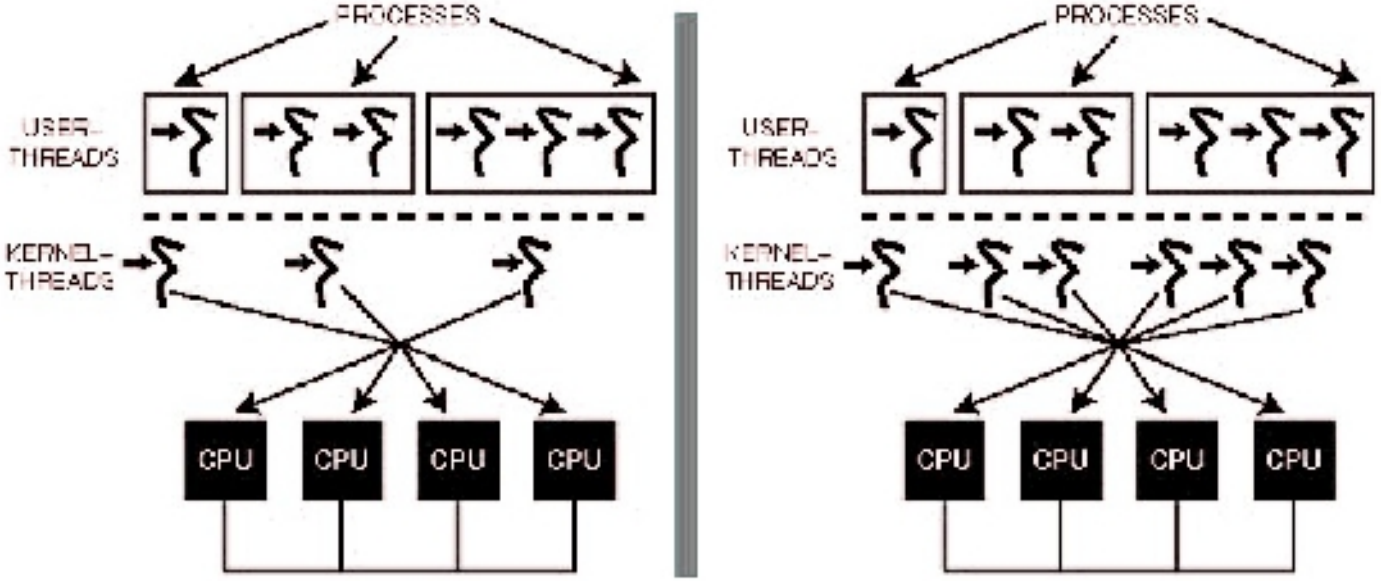
- On-demand spawning creates a new process or thread in response to the arrival of client connection and/or data requests
- Typically used to implement the thread-per-request and thread-per-connection models



- The primary benefit of on-demand spawning strategies is their reduced consumption of resources
- The drawbacks, however, are that these strategies can degrade performance in heavily loaded servers & determinism in real-time systems due to costs of spawning processes/threads and starting services

The N:1 & 1:1 Threading Models

- OS scheduling ensures applications use host CPU resources suitably
- Modern OS platforms provide various models for scheduling threads
- A key difference between the models is the *contention scope* in which threads compete for system resources, particularly CPU time
- The two different contention scopes are shown below:



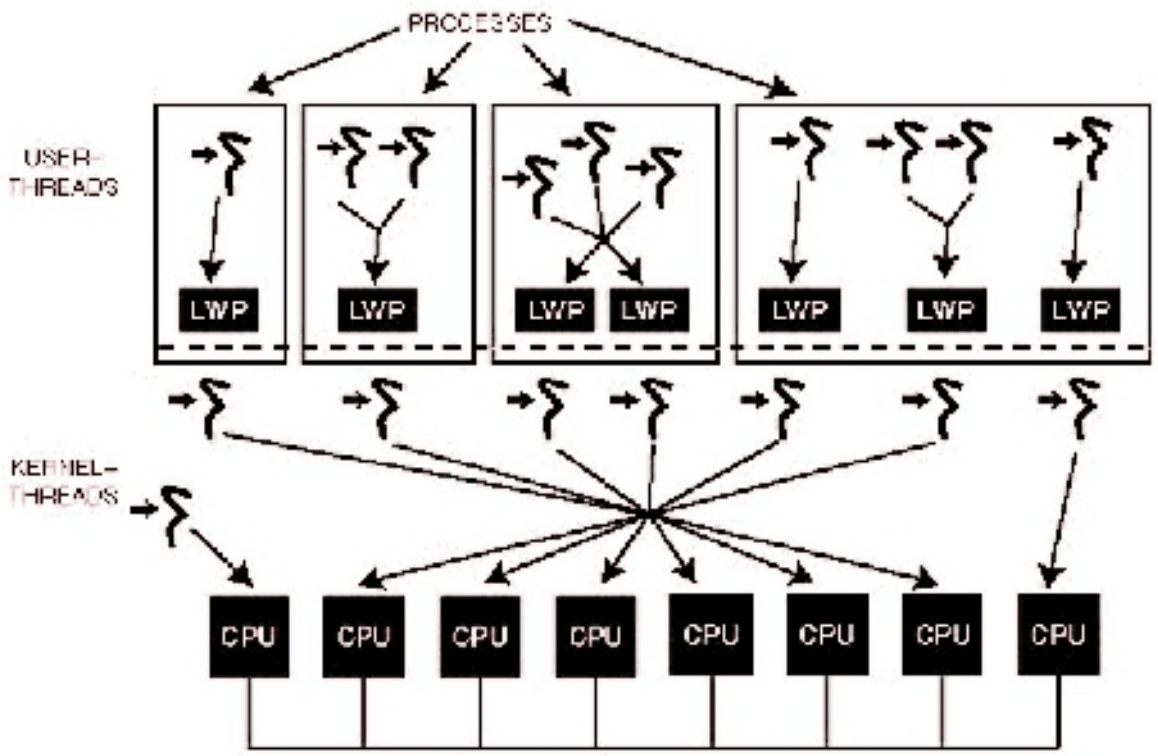
(1) N:1 USER THREADING MODEL

(2) 1:1 KERNEL THREADING MODEL

- ***Process contention scope*** (aka "user threading") where threads in the same process compete with each other (but not directly with threads in other processes)
- ***System contention scope*** (aka "kernel threading") where threads compete directly with other system-scope threads, regardless of what process they're in

The N:M Threading Model

- Some operating systems (such as Solaris) offer a combination of the N:1 & 1:1 models, referred to as the ``N:M'' hybrid-threading model
- When an application spawns a thread, it can indicate in which contention scope the thread should operate
- The OS threading library creates a user-space thread, but only creates a kernel thread if needed or if the application explicitly requests the system contention scope

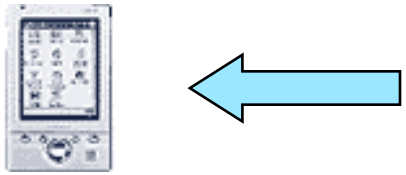


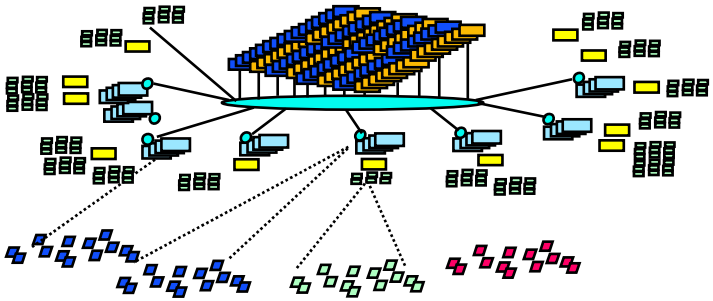
- When the OS kernel blocks an LWP, all user threads scheduled onto it by the threads library also block
- However, threads scheduled onto other LWPs in the process can continue to make progress

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

The Half-Sync/Half-Async Pattern

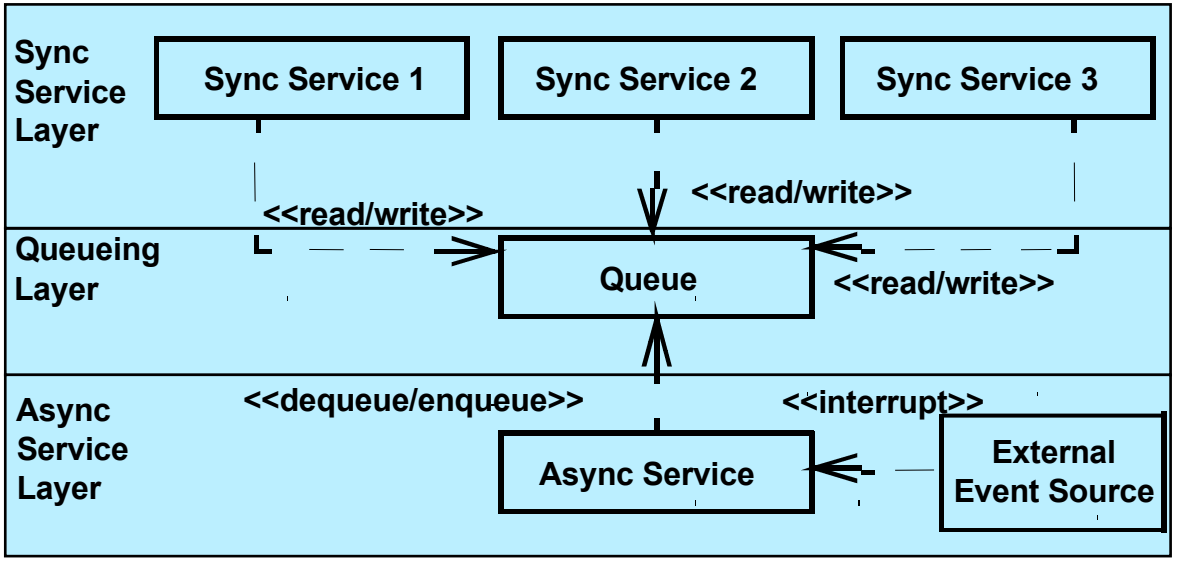
Solution

- Apply the *Half-Sync/Half-Async* architectural pattern (P2) to scale up server performance by processing different HTTP requests concurrently in multiple threads

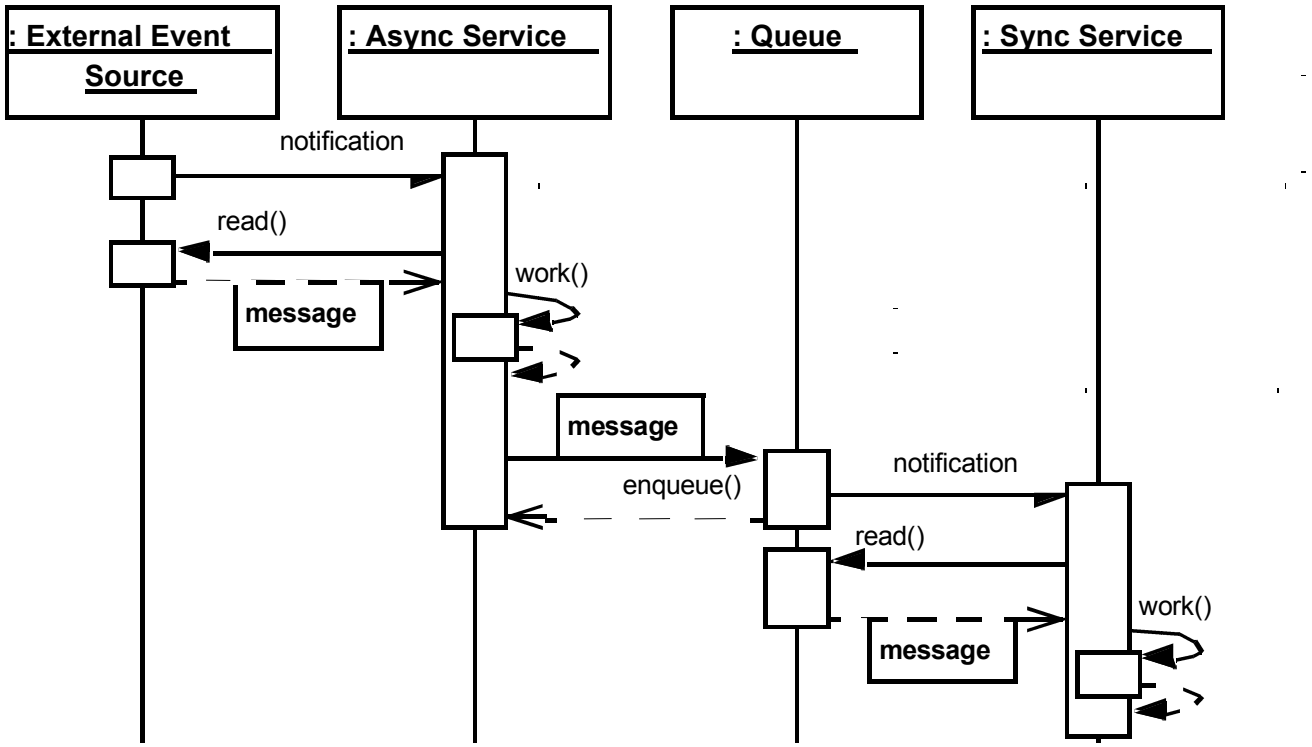
This solution yields two benefits:

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

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

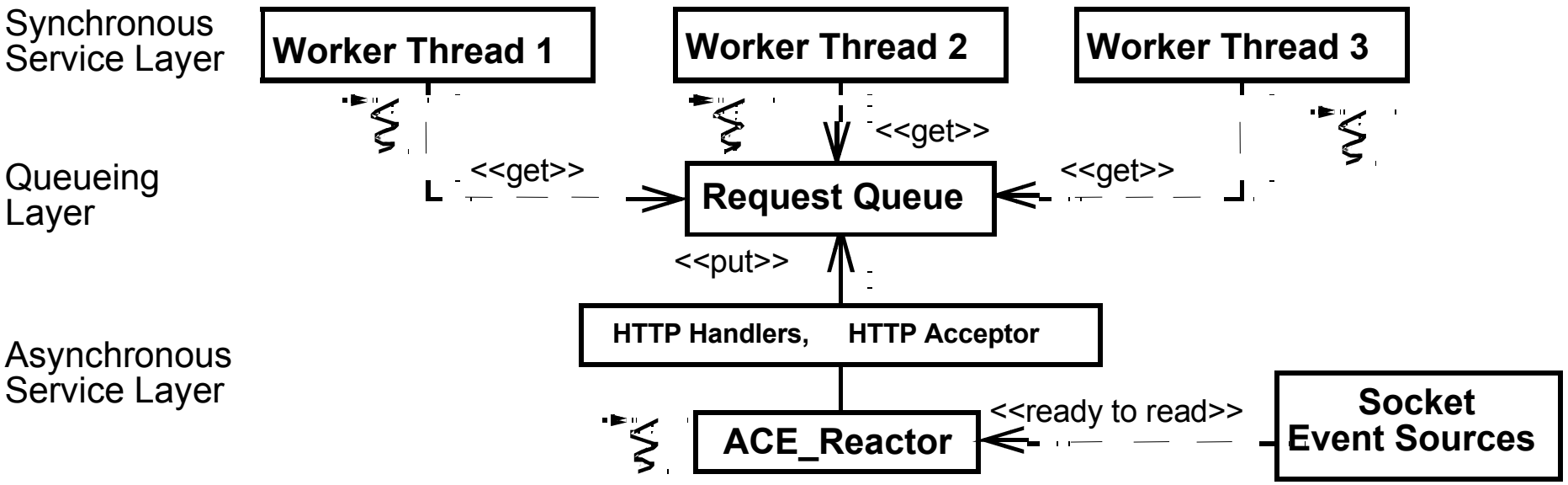


Half-Sync/Half-Async Pattern Dynamics



- 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 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 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

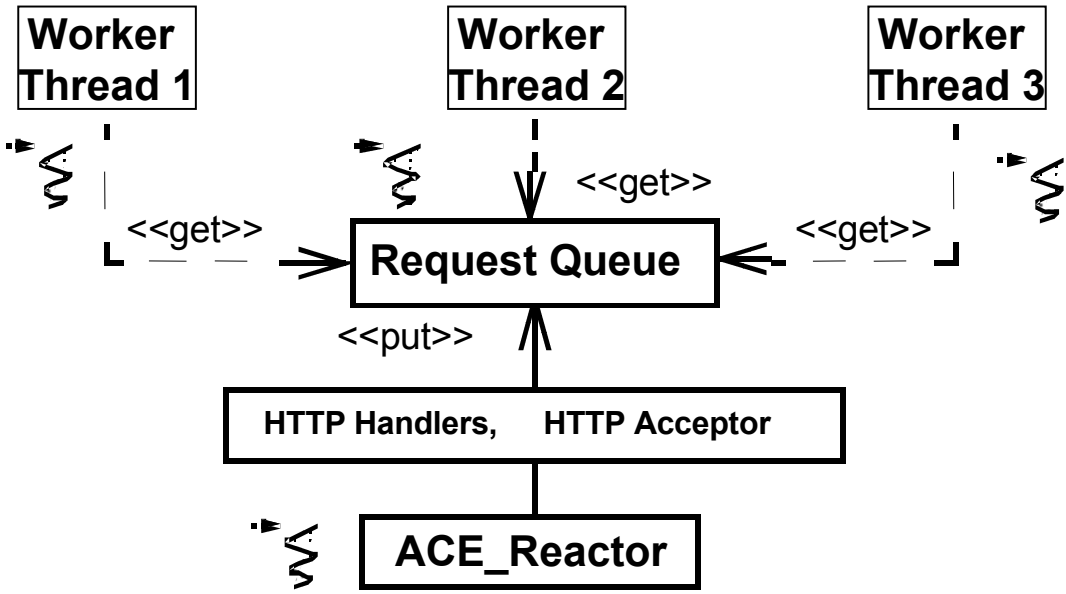
This pattern also incurs **liabilities**:

- ***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

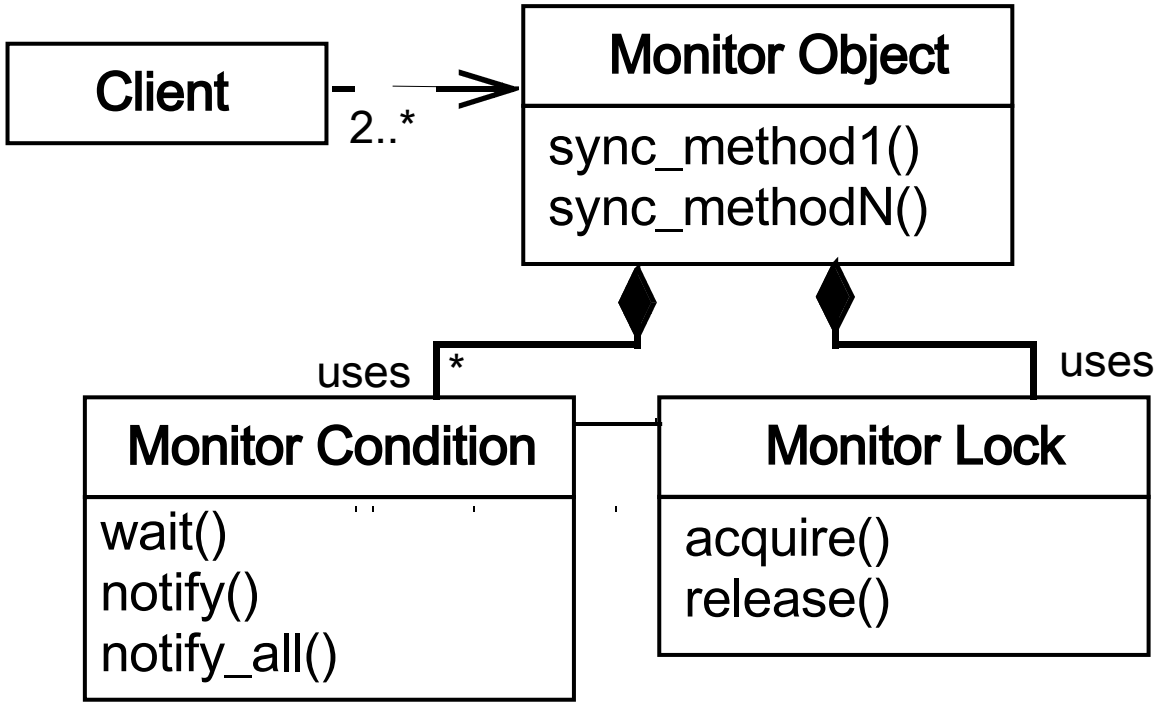
- A naive implementation of a request queue will incur race conditions or 'busy waiting' when multiple threads insert & remove requests
  - *e.g.,* multiple concurrent producer & consumer threads can corrupt the queue's internal state if it is not synchronized properly
  - Similarly, these threads will '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 (P2) to synchronize the queue efficiently & conveniently

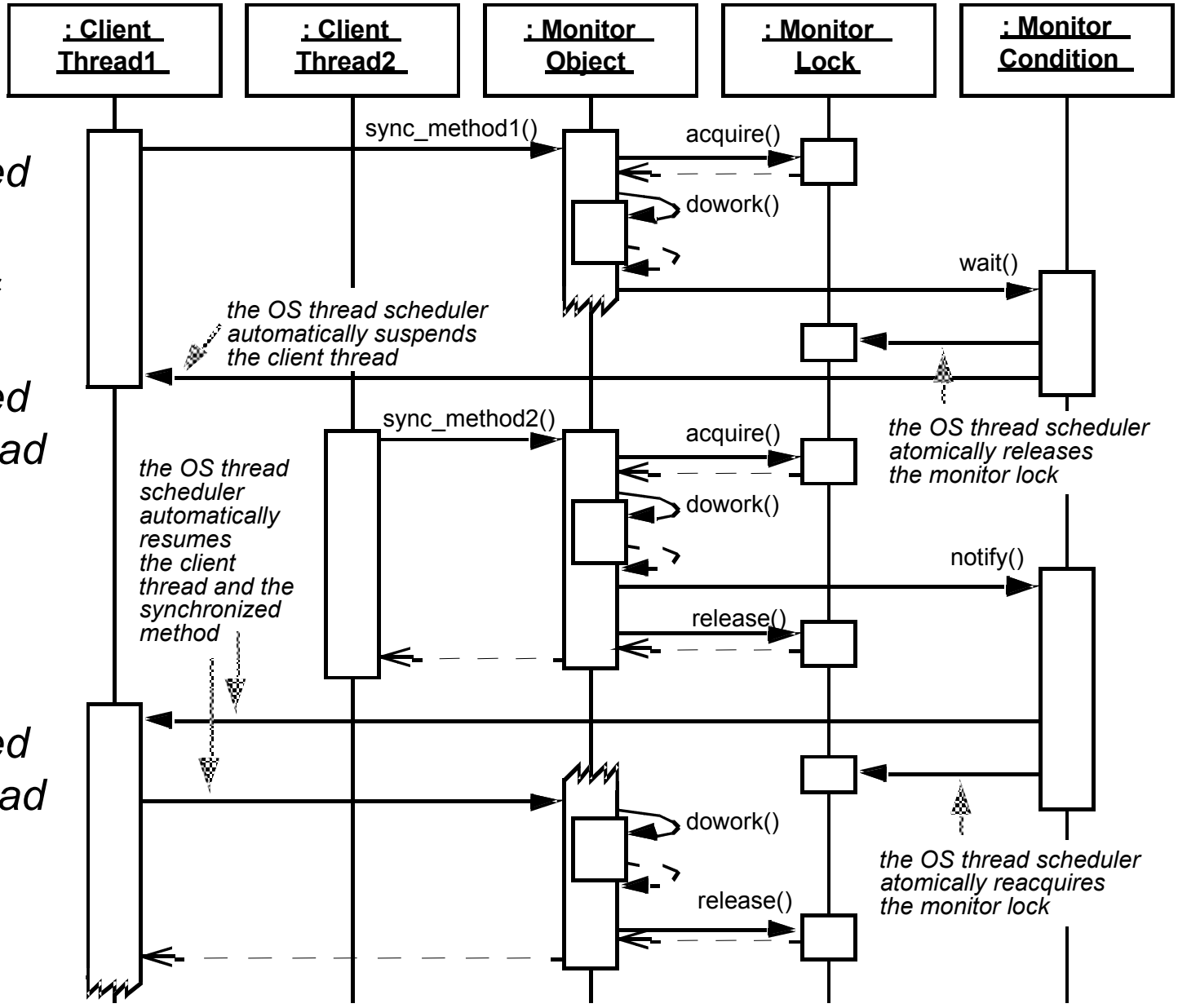
- This pattern synchronizes concurrent method execution to ensure that only one method at a time runs within an object
- It also allows an object's methods to cooperatively schedule their execution sequences



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

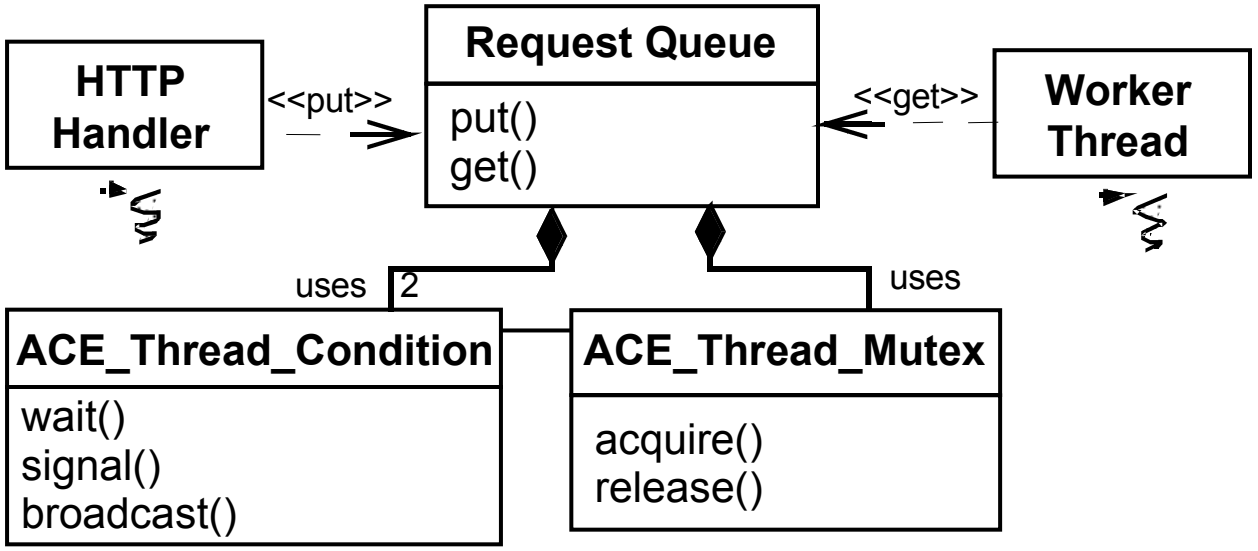
Monitor Object Pattern Dynamics

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



Applying 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 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

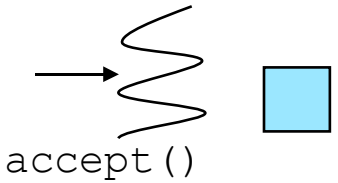
This pattern can also incur **liabilities**:

- 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

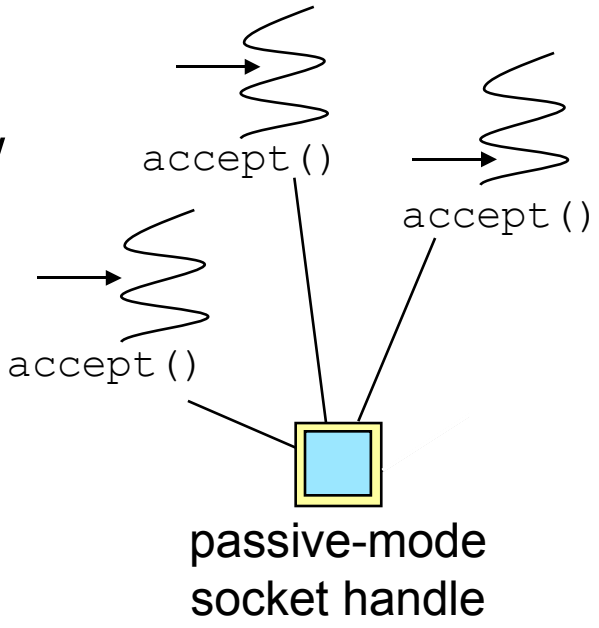
Minimizing Server Threading Overhead

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:



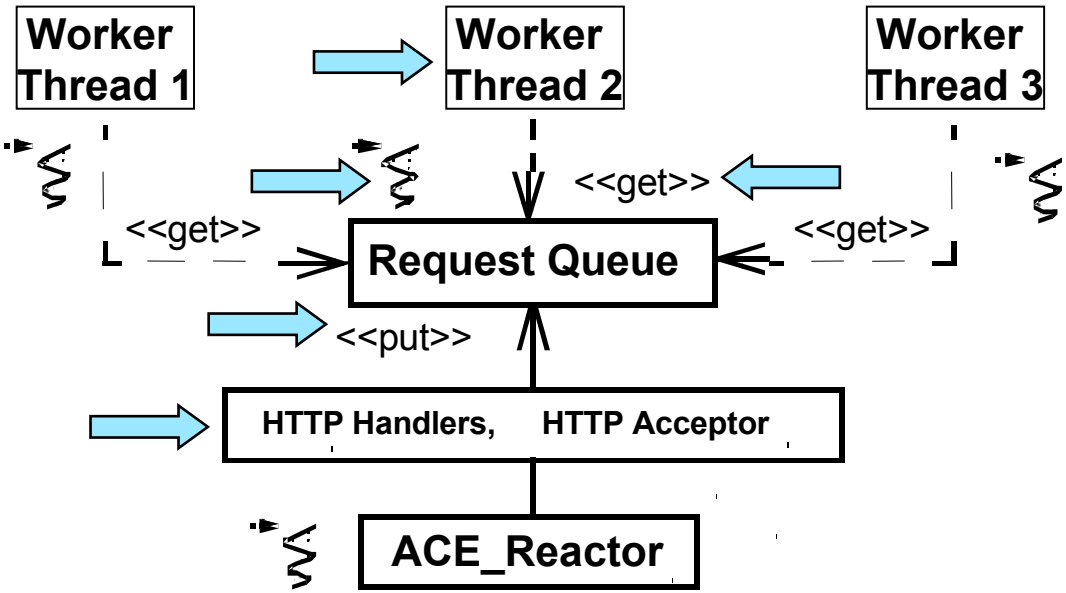
- The OS allows a pool of threads in a Web server to call **`accept()`** on the same passive-mode socket handle
- When a connection request arrives, the operating system's transport layer creates a new connected transport endpoint, encapsulates this new endpoint with a data-mode socket handle & passes 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 Half-Sync/Half-Async

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



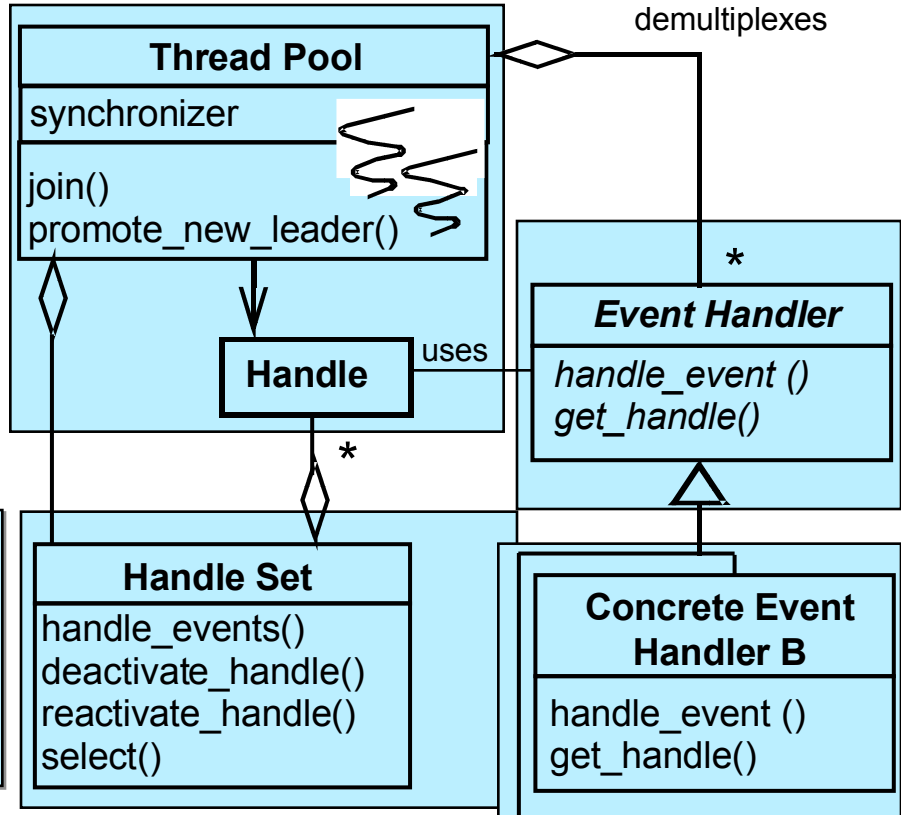
Solution

- Apply the *Leader/Followers* architectural pattern (P2) to minimize server threading overhead

The Leader/Followers Pattern

The Leader/Followers architectural pattern (P2) 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

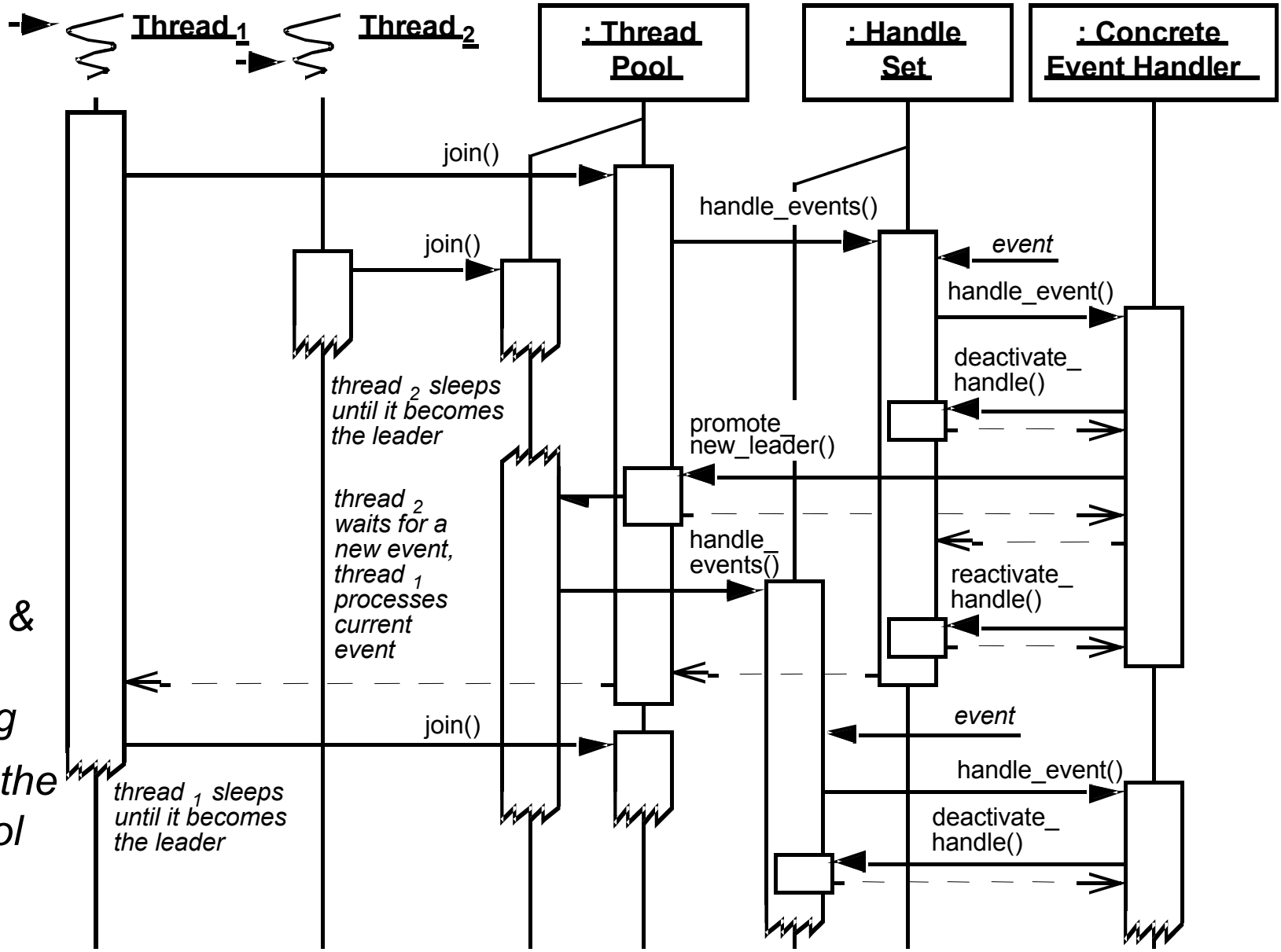
This pattern eliminates the need for—& the overhead of—a separate Reactor thread & synchronized request queue used in the Half-Sync/Half-Async pattern



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

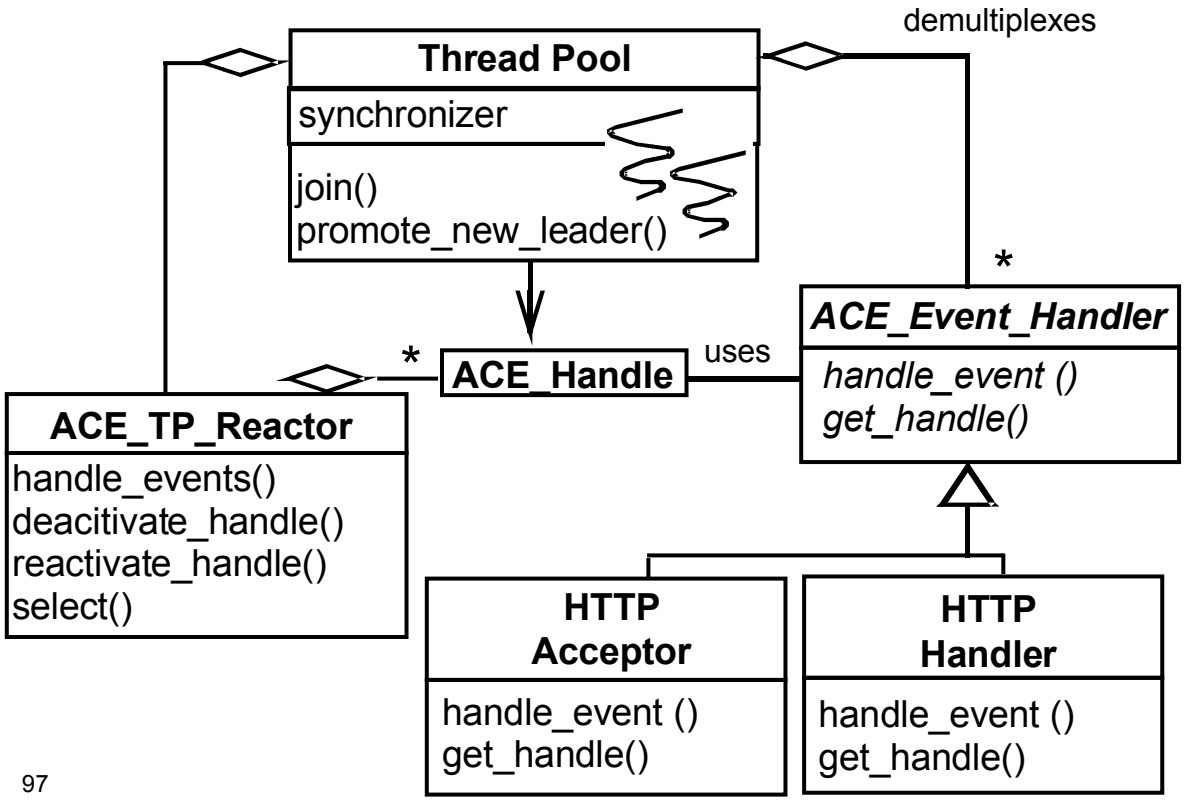
1. *Leader thread demuxing*
2. *Follower thread promotion*
3. *Event handler demuxing & event processing*
4. *Rejoining the thread pool*



Applying Leader/Followers Pattern in JAWS

Two options:

2. If platform supports **`accept()`** optimization then the Leader/Followers pattern can be implemented by the OS
3. 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 & Cons of 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**:

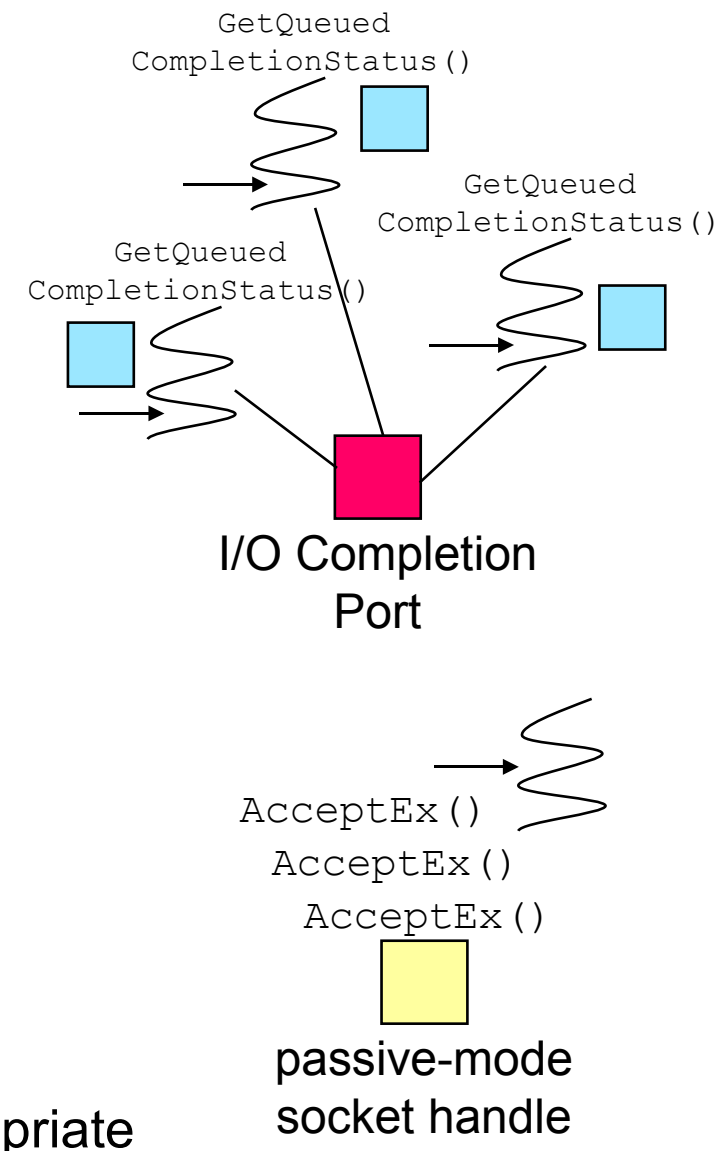
- ***Implementation complexity***
  - The advanced variants of the Leader/ Followers pattern are hard to implement
- ***Lack of flexibility***
  - In the Leader/ Followers model it is hard to discard or reorder events because there is no explicit queue
- ***Network I/O bottlenecks***
  - The Leader/Followers 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

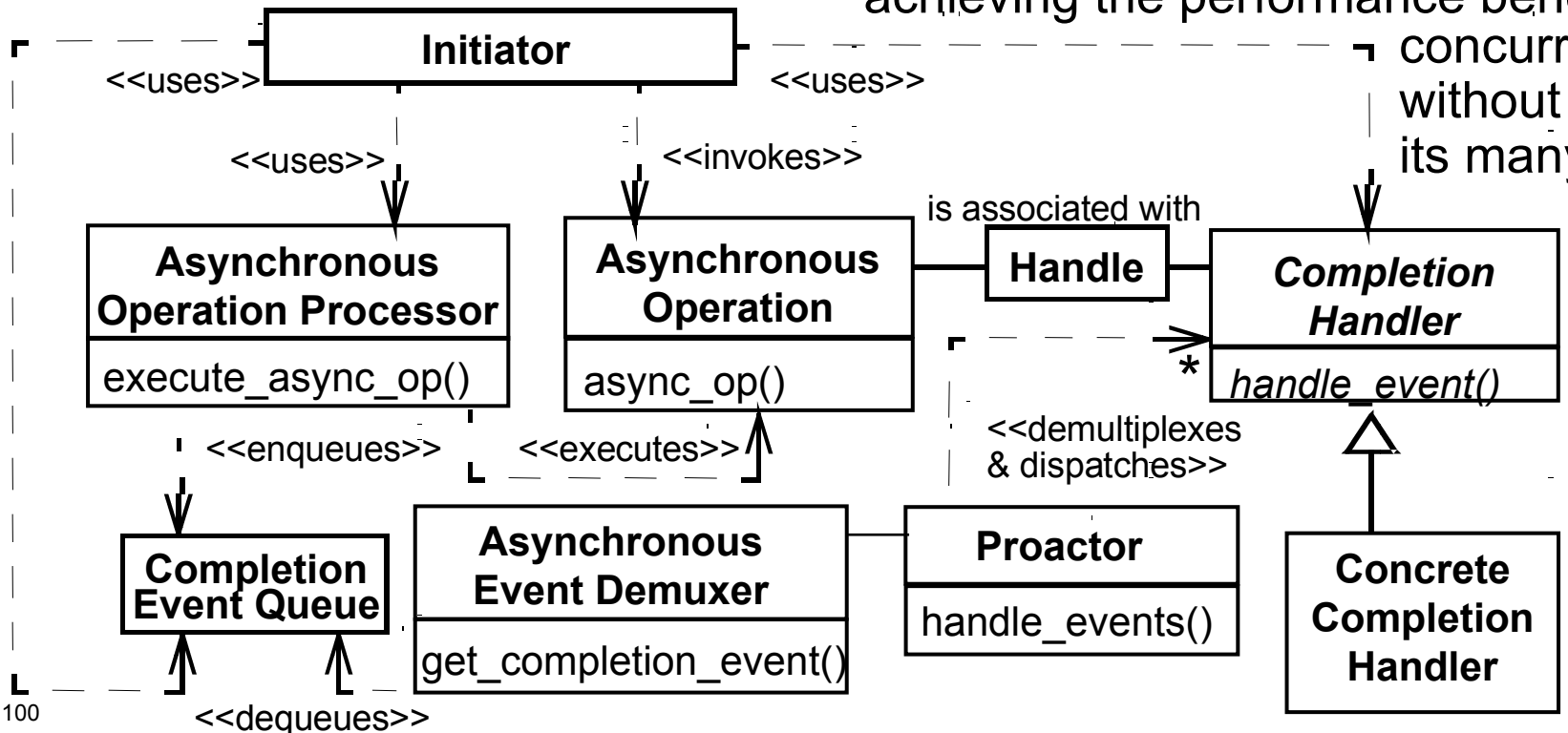
Problem

- 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

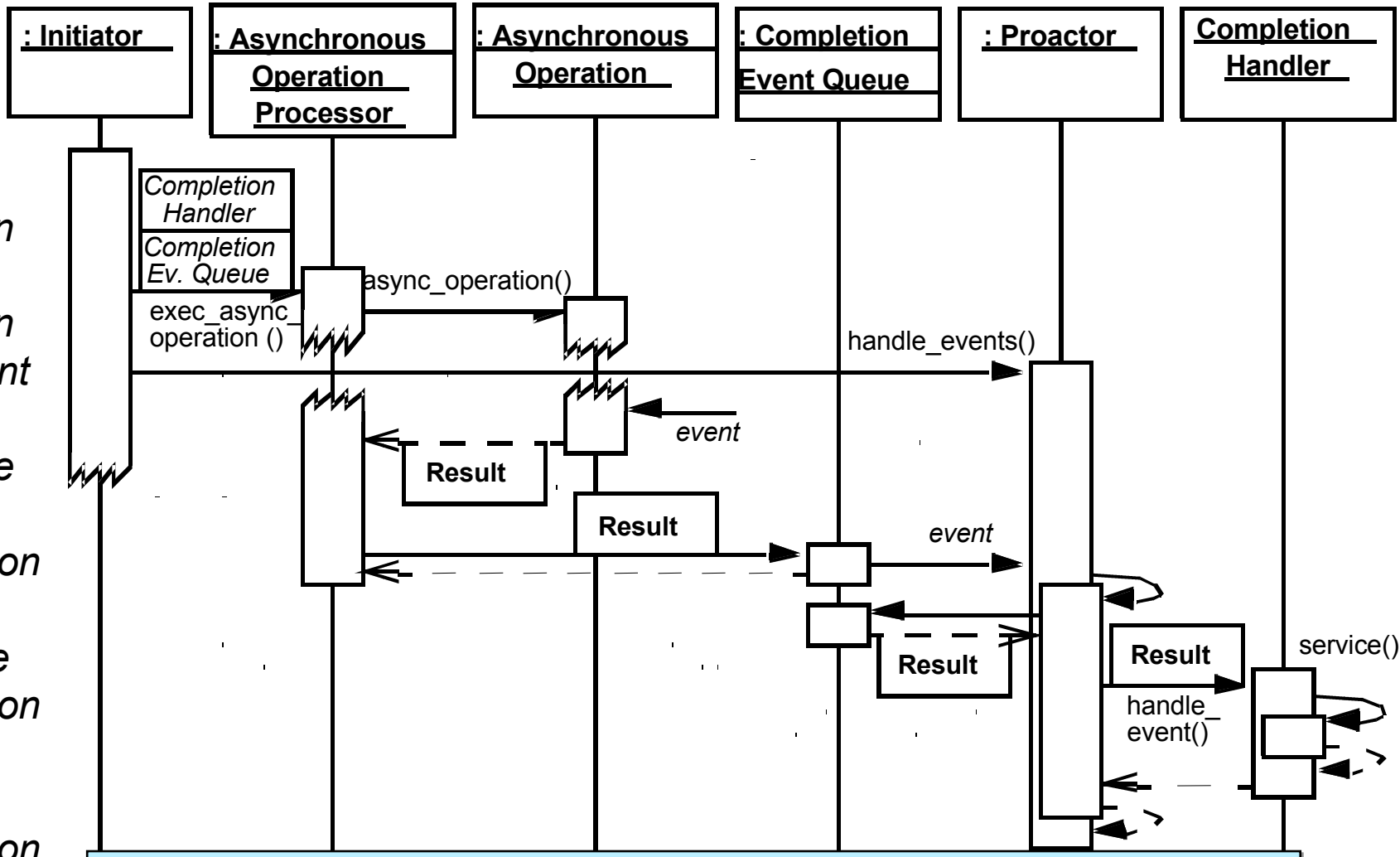
- Apply the *Proactor* architectural pattern (P2) to make efficient use of async I/O

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



Proactor Pattern Dynamics

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.:*

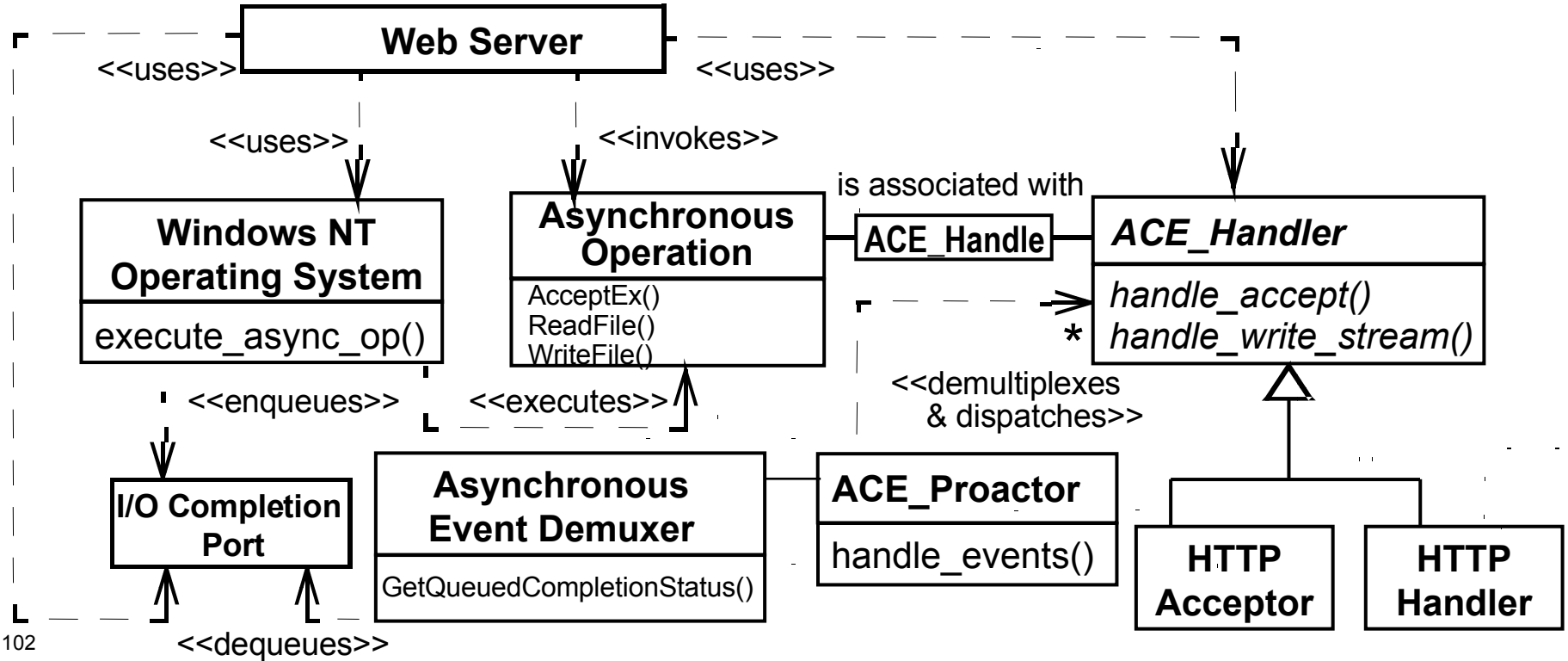
- Both process events via callbacks
- However, it's generally easier to multi-thread a proactor

Applying the Proactor Pattern in JAWS

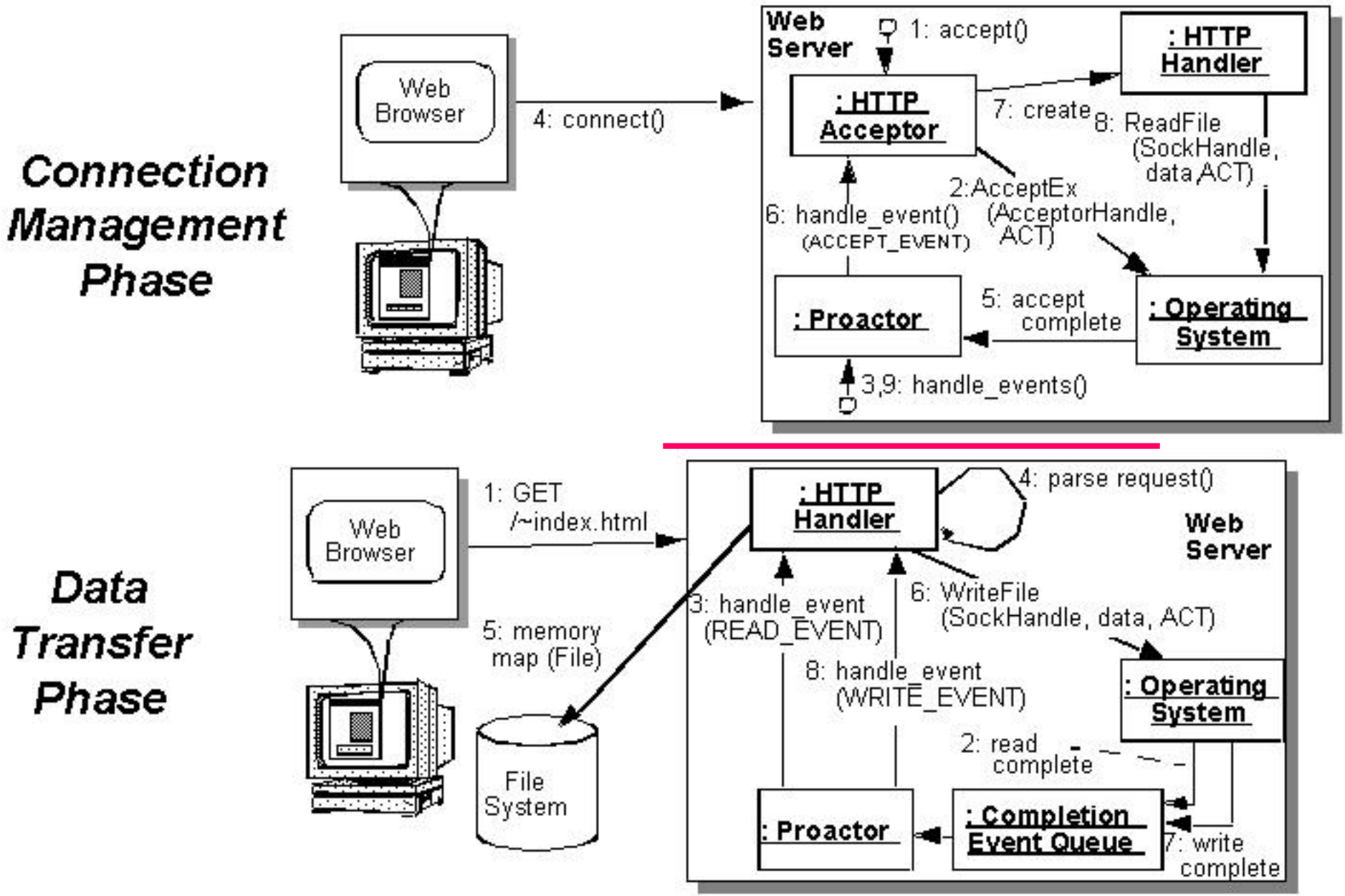
The Proactor pattern structures the JAWS concurrent server to receive & process requests from multiple clients asynchronously

JAWS HTTP components are split into two parts:

2. Operations that execute asynchronously
   - *e.g.,* to accept connections & receive client HTTP GET requests
3. 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



Pros & Cons of 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

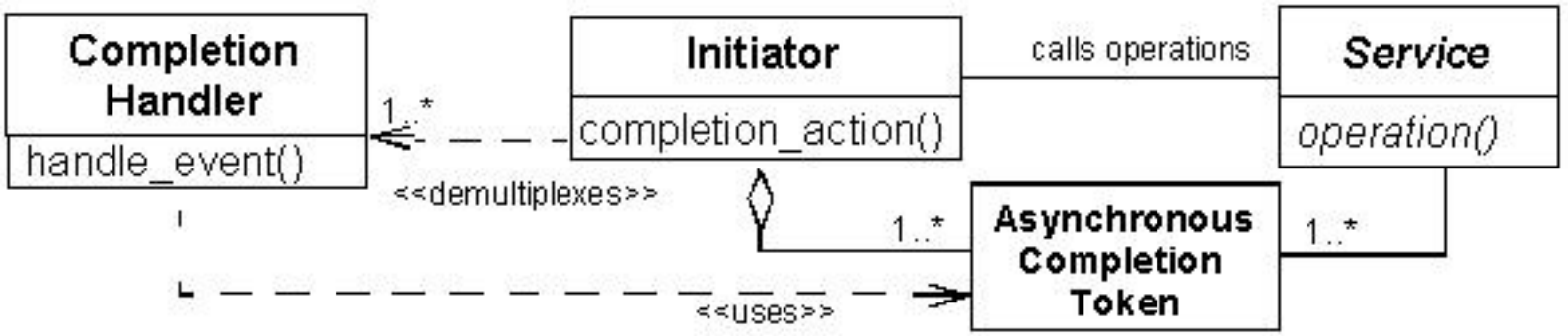
Problem
- As little overhead as possible should be incurred to determine how the completion handler will demux & process completion events after async operations finish executing
- When a response arrives, the application should spend as little time as possible demultiplexing the completion event to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

Solution
- Apply the *Asynchronous Completion Token* design pattern (P2) to demux & process the responses of asynchronous operations efficiently

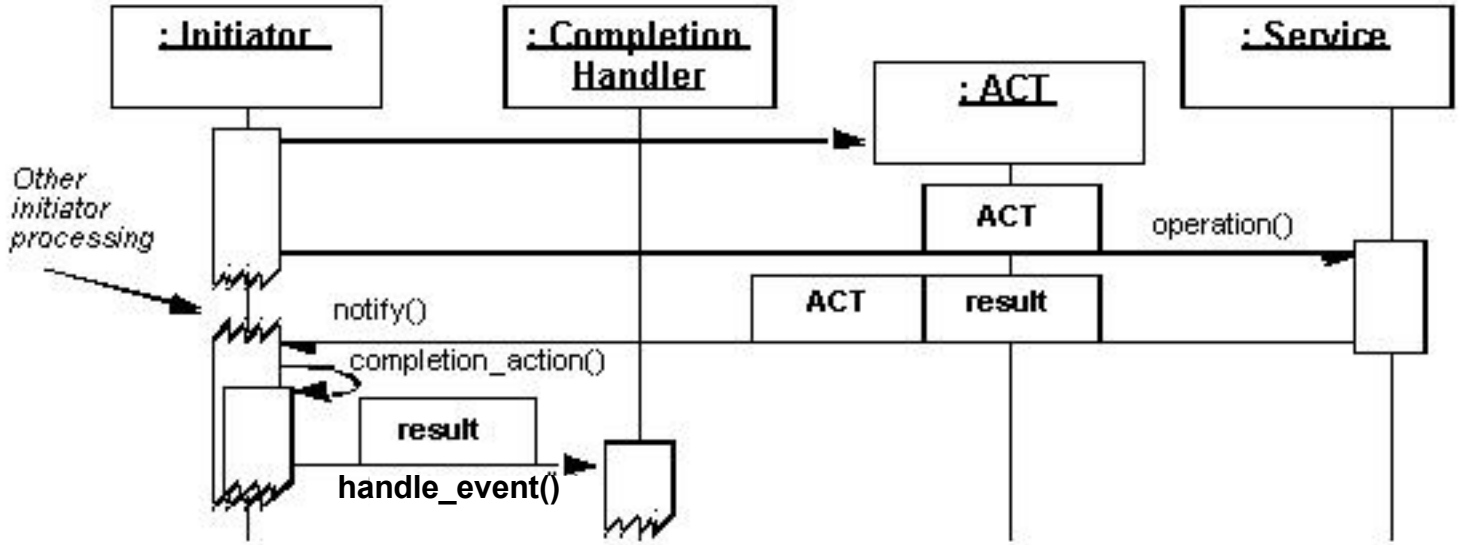
- Together with each async operation that a client *initiator* invokes on a *service*, transmit information that identifies how the initiator should process the service's response
- Return this information to the initiator when the operation finishes, so that it can be used to demux the response efficiently,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

Asynchronous Completion Token Pattern

Structure and Participants

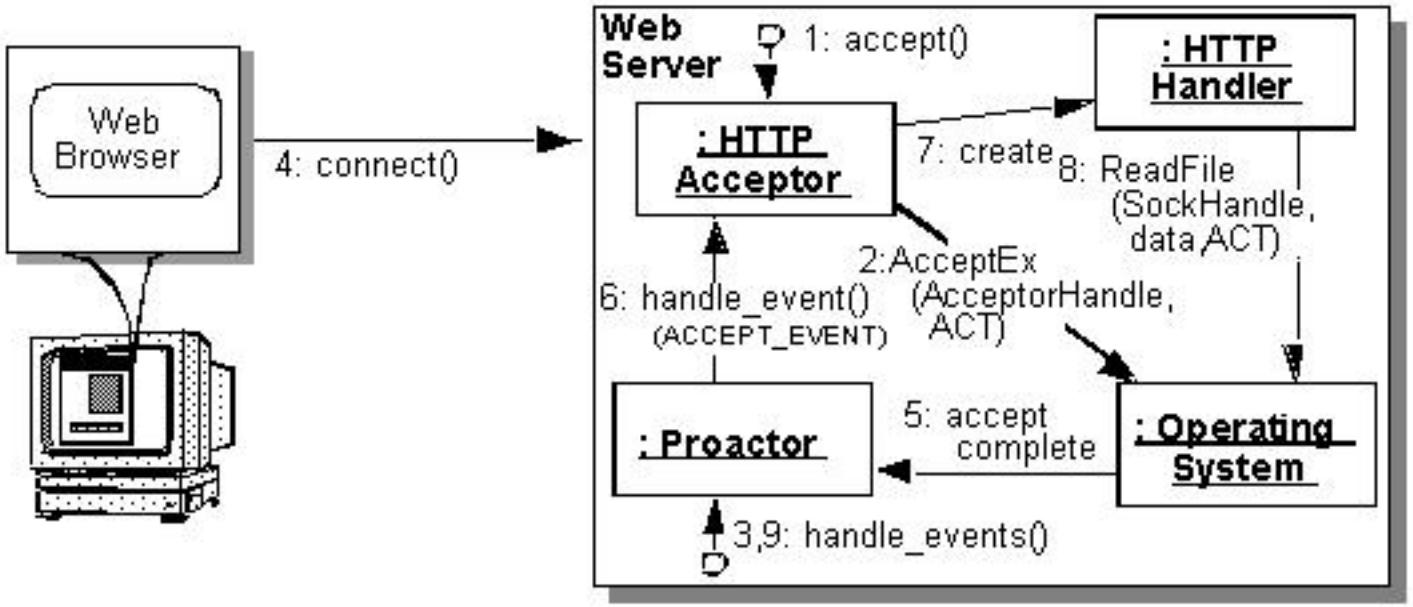


Dynamic Interactions

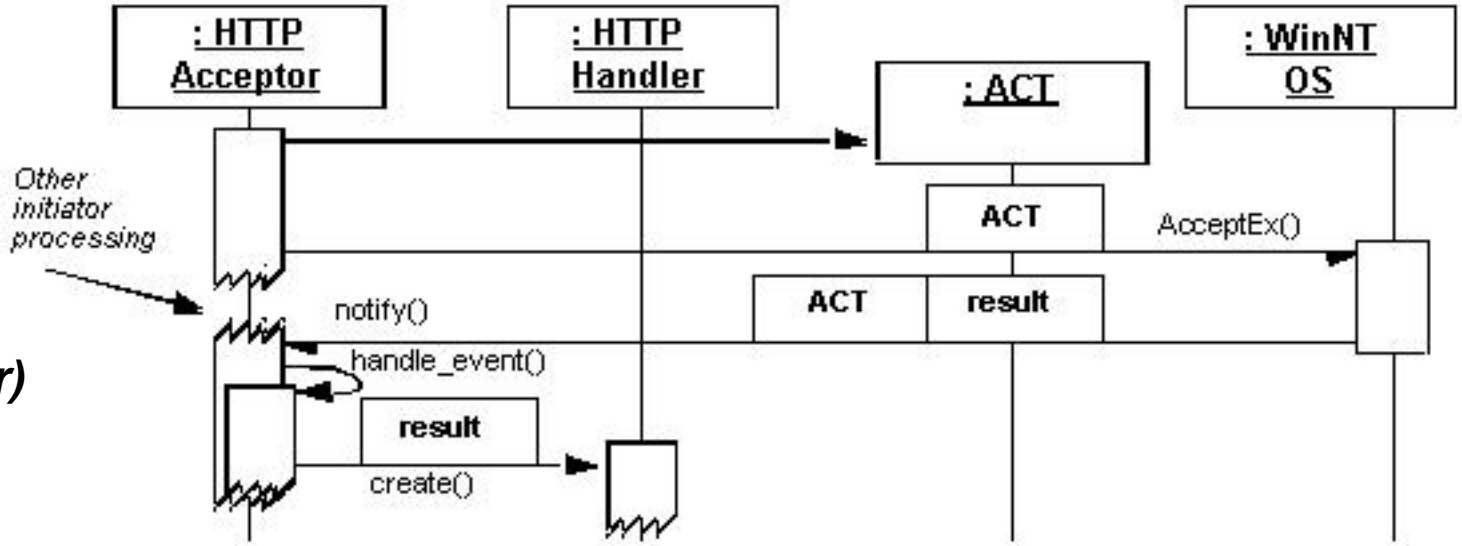


Applying the Asynchronous Completion Token Pattern in JAWS

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

Enhancing Server (Re)Configurability (1/2)

Context

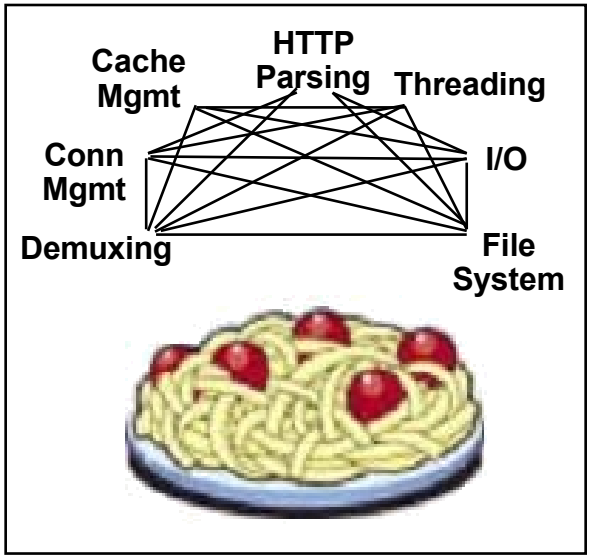
The implementation of certain web server components depends on a variety of factors:

- 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 web server component configuration is inflexible & inefficient:

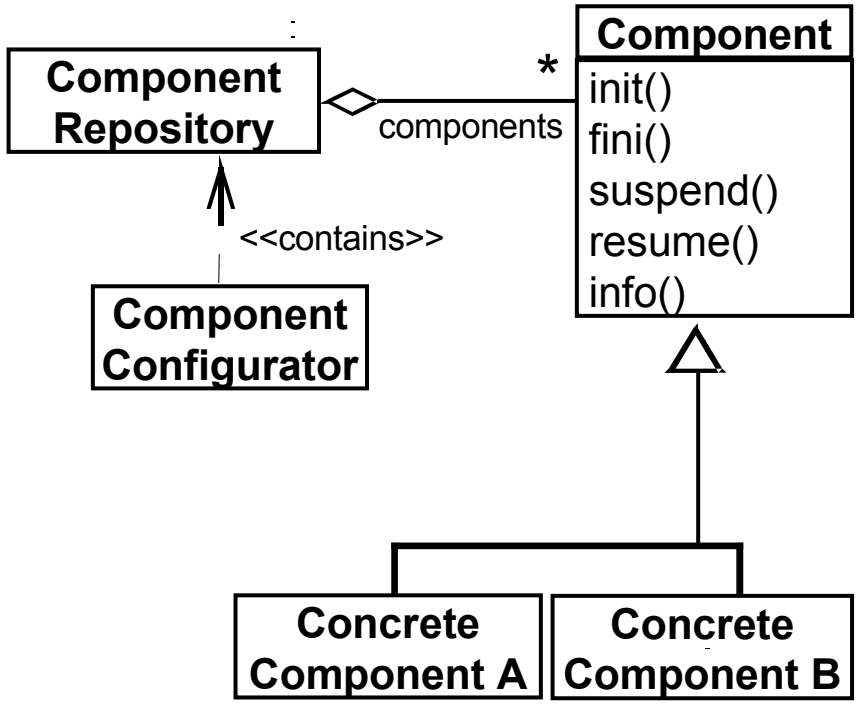
- No single web server configuration is optimal for all use cases
- Certain design decisions cannot be made efficiently until run-time



Enhancing Server (Re)Configurability (2/2)

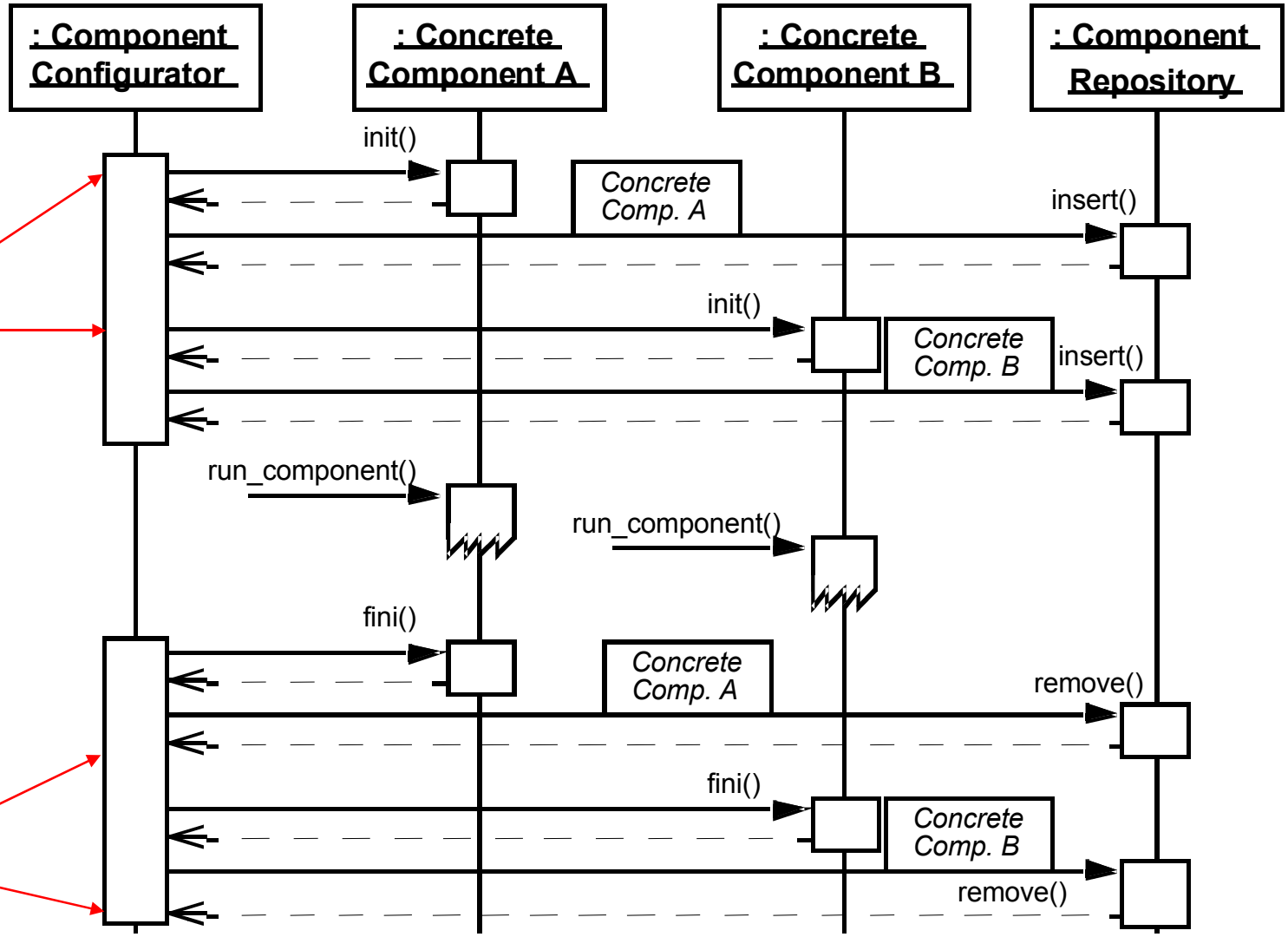
Solution
- Apply the *Component Configurator* design pattern (P2) to enhance server configurability

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



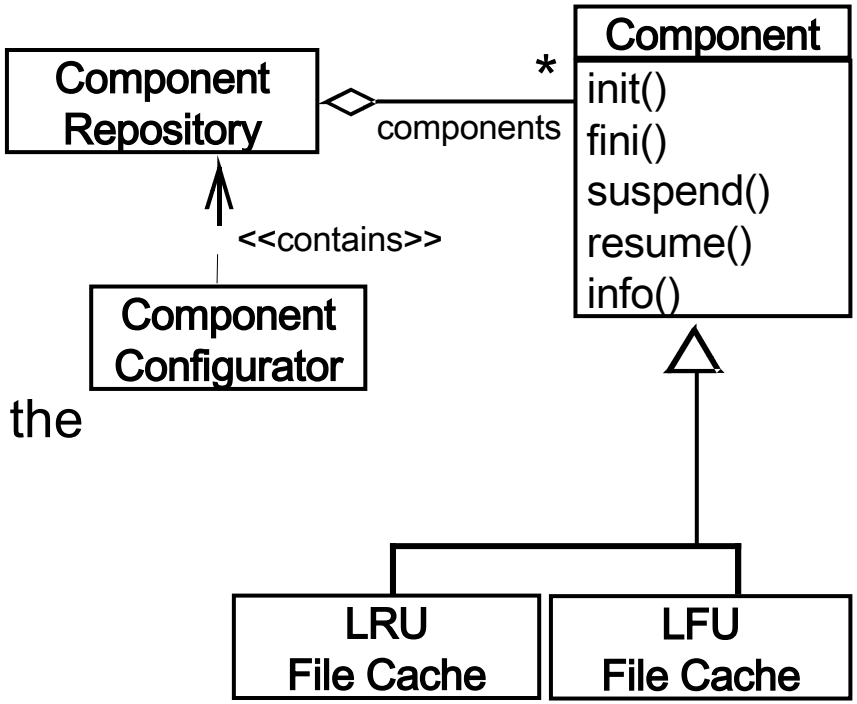
Component Configurator Pattern Dynamics

1. *Component initialization & dynamic linking*
2. *Component processing*
3. *Component termination & dynamic unlinking*



Applying the Component Configurator Pattern to Web Servers

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



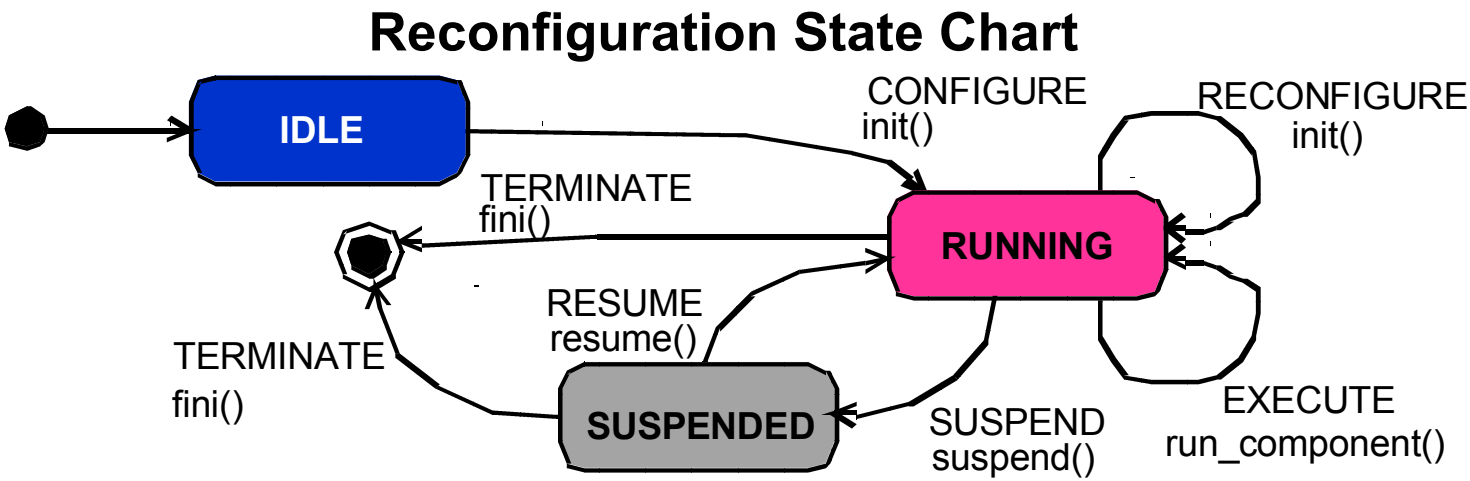
- For example, a web server can apply the Component Configurator pattern to configure various *Cached Virtual Filesystem* strategies
  - *e.g.,* least-recently used (LRU) or least-frequently used (LFU)

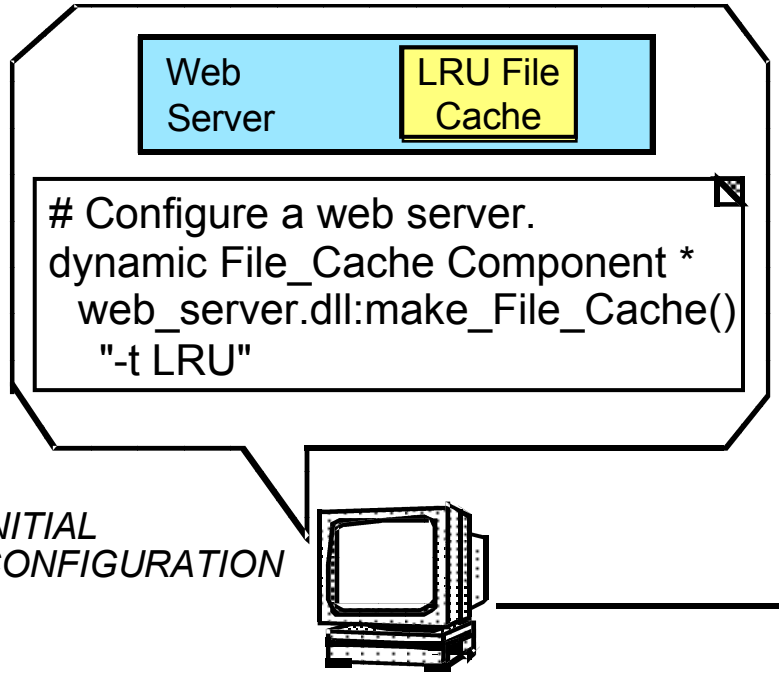
Concrete components can be packaged into a suitable unit of configuration, such as a dynamically linked library (DLL)

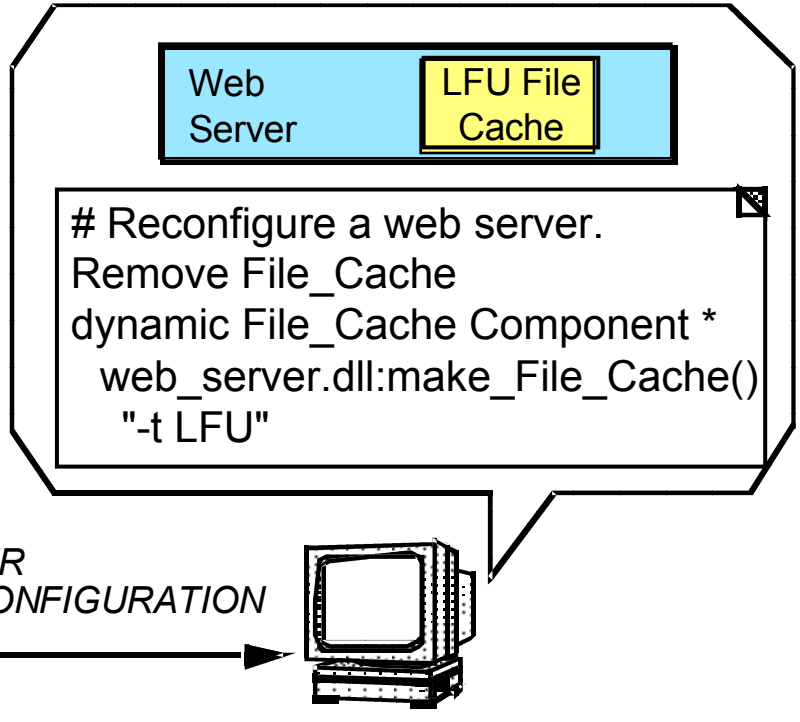
Only the components that are currently in use need to be configured into a web server

Reconfiguring JAWS

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







Pros & Cons of 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

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 (P2) 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<ACE_Null_Mutex>
        Content_Cache;
```

- Multi-threaded file cache using a thread mutex.

```
typedef File_Cache<ACE_Thread_Mutex>
        Content_Cache;
```

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

```
typedef File_Cache<ACE_RW_Mutex>
        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 (P2) 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 `ACE_Guard` Wrapper Facade

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

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

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

Applying the `ACE_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.

    ACE_Guard<LOCK> guard (*lock);
    // Implement the <lookup> method.
    // lock_ released automatically…
  }
```

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 (1/2)

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"

```
template <class LOCK>
class File_Cache {
public:
  const void *lookup
   (const string &path) const {
    ACE_Guard<LOCK> guard (lock_);
    const void *file_pointer =
      check_cache (path);
    if (file_pointer == 0) {
      insert (path);
      file_pointer =
        check_cache (path);
    }
    return file_pointer;
  }
  void insert (const string &path) {
    ACE_Guard<LOCK> guard (lock_);
    // ... insert <path> into
cache...
  }
private:
  mutable LOCK lock_;
  const void *check_cache
};
```

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

Minimizing Unnecessary Locking (2/2)

Solution

- Apply the *Thread-safe Interface* design pattern (P2) 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.

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 Scoped Locking to acquire/release lock automatically.
    ACE_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> isn't in cache, insert it & 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 (P2) 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 TRUE)
{
  acquire the mutex
  // Perform double-check to
  // avoid race condition.
  if (first_time_in is TRUE)
  {
    execute the critical section
    set first_time_in to FALSE
  }
  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.
      ACE_Guard<ACE_Thread_Mutex> guard (lock_);
      // Double check instance_.
      if (instance_ == 0)
        instance_ = new TYPE;
    }
    return instance_;
  }
private:
  static TYPE *instance_;
  static ACE_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

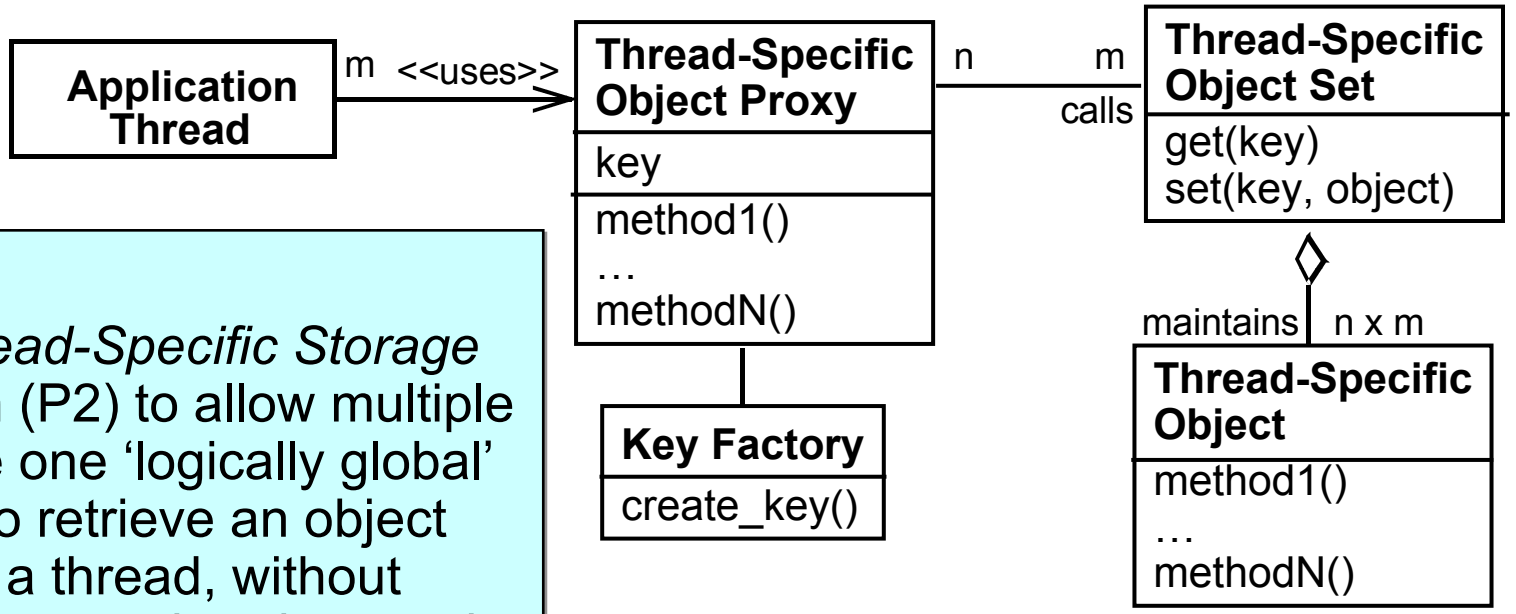
Logging Access Statistics Efficiently

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

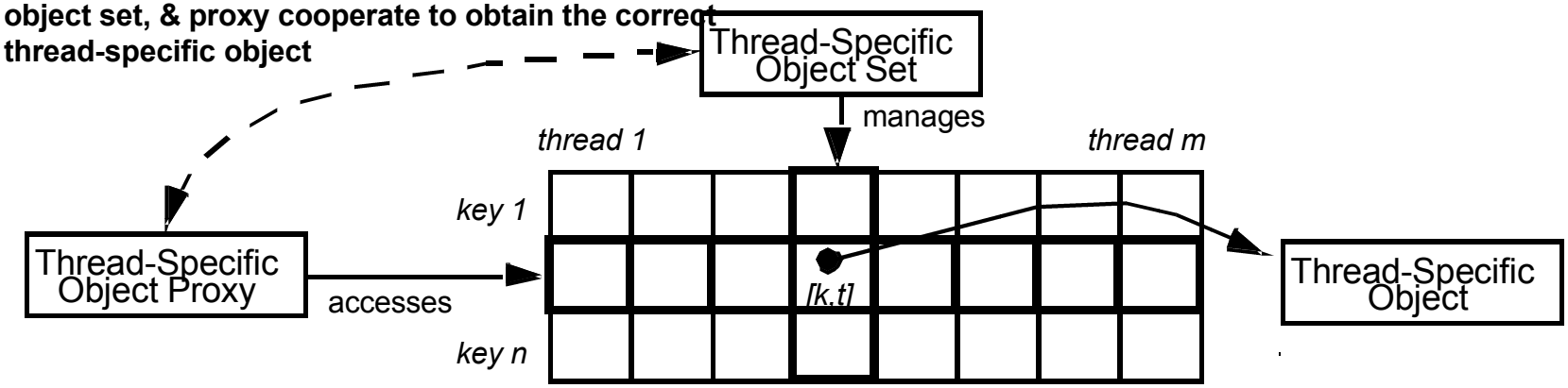


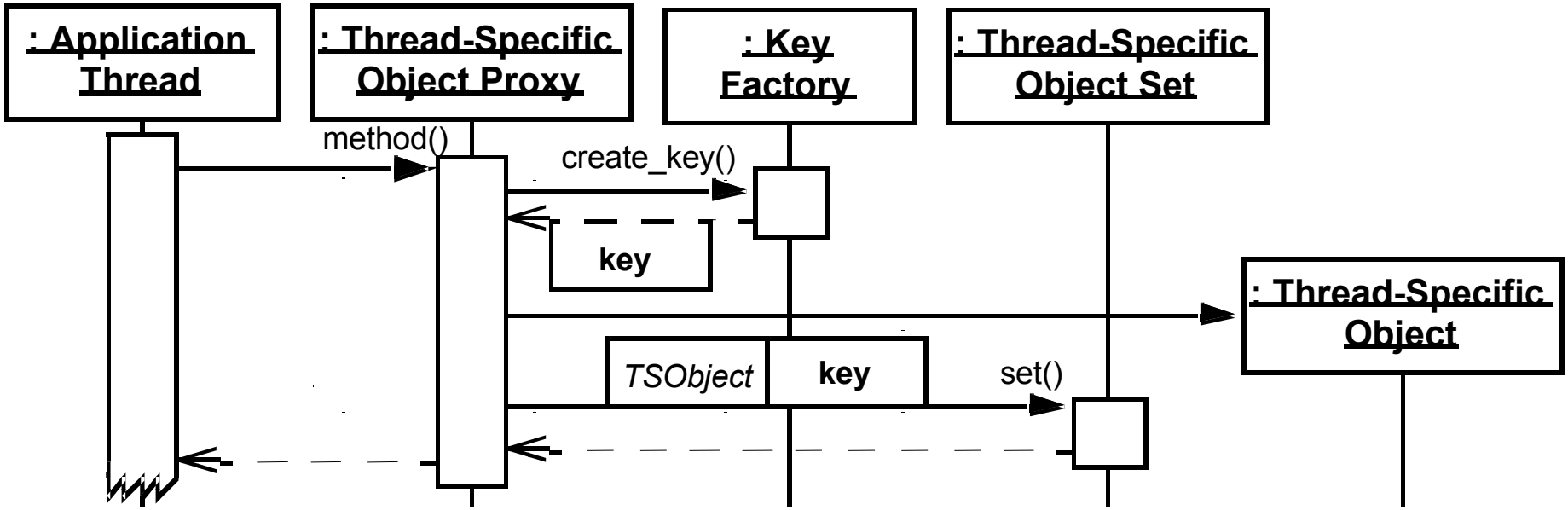
Solution

- Apply the *Thread-Specific Storage* design pattern (P2) 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

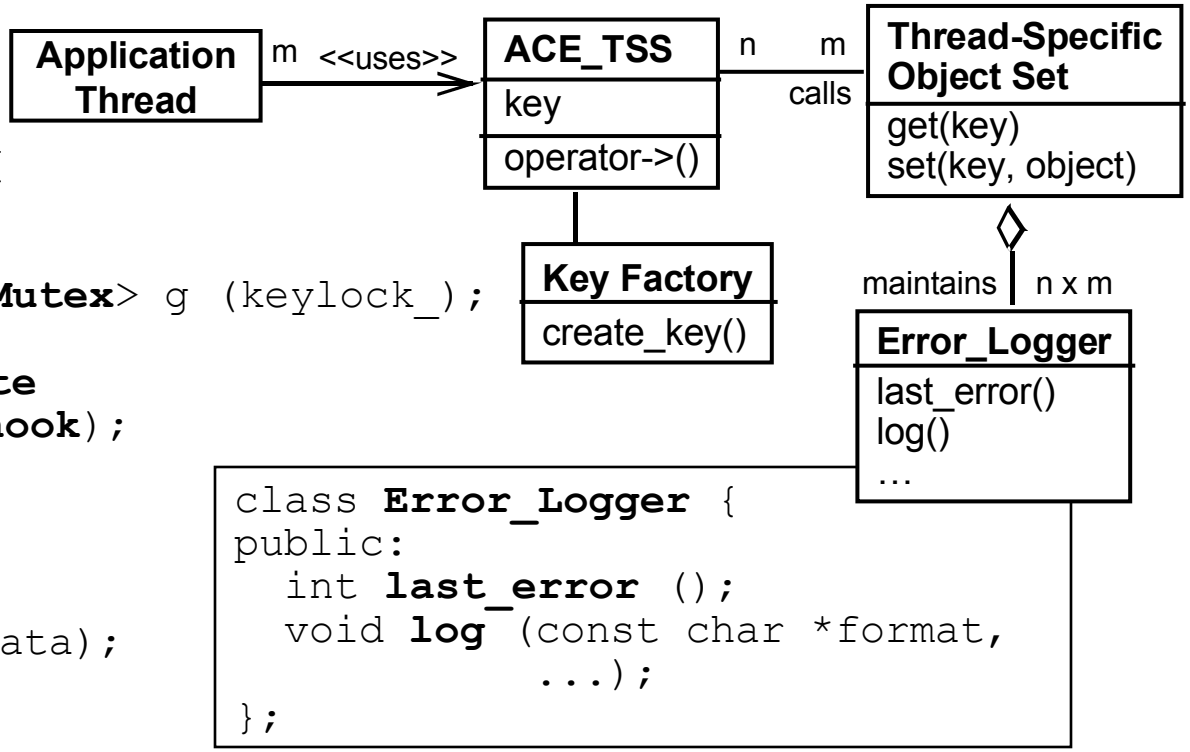
The application thread identifier, thread-specific object set, & proxy cooperate to obtain the correct thread-specific object





Applying the Thread-Specific Storage Pattern to JAWS

```
template <class TYPE>
Class ACE_TSS {
public:
  TYPE *operator->() const {
    TYPE *tss_data = 0;
    if (!once_) {
      ACE_Guard<ACE_Thread_Mutex> g (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 ACE_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**:

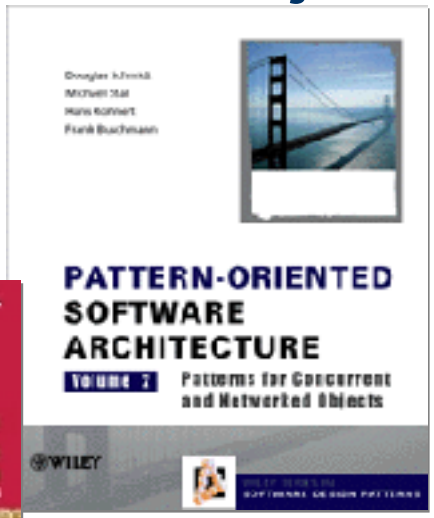
- ***Efficiency***
  - It's possible to implement this pattern so that no locking is needed to access thread-specific data
- ***Ease of use***
  - When encapsulated with wrapper facades, thread-specific storage is easy for application developers to use
- ***Reusability***
  - By combining this pattern 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

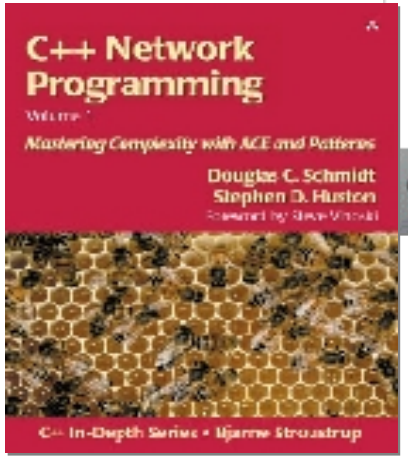
This pattern also has **liabilities**:

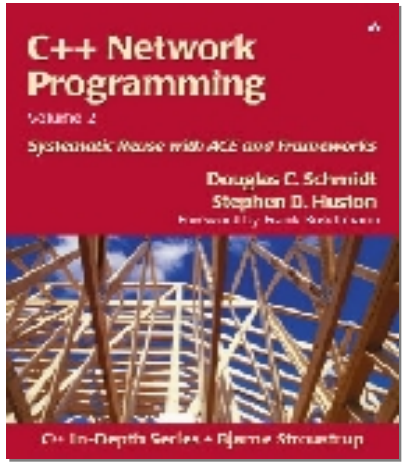
- ***It encourages use of thread-specific global objects***
  - Many applications do not require multiple threads to access thread-specific data via a common access point
  - In this case, data should be stored so that only the thread owning the data can access it
- ***It obscures the structure of the 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

Additional Information

- Patterns & frameworks for concurrent & networked objects
  - www.posa.uci.edu
- ACE & TAO open-source middleware
  - www.cs.wustl.edu/~schmidt/ACE.html
  - www.cs.wustl.edu/~schmidt/TAO.html







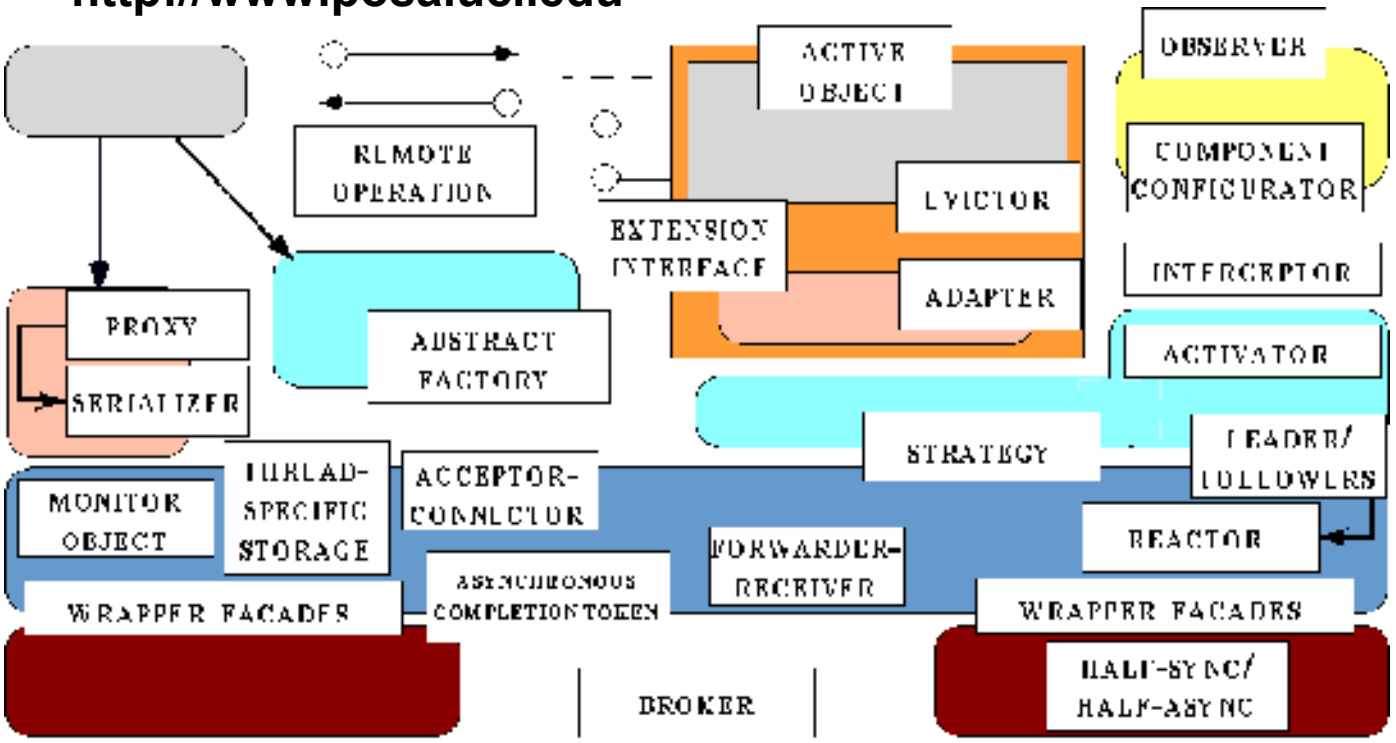
- ACE research papers
  - www.cs.wustl.edu/~schmidt/ACE-papers.html
- Extended ACE & TAO tutorials
  - UCLA extension, Jan 19-21, 2005
  - www.cs.wustl.edu/~schmidt/UCLA.html
- ACE books
  - www.cs.wustl.edu/~schmidt/ACE/

Tutorial Example:

Applying Patterns to Real-time CORBA

http://www.posa.uci.edu

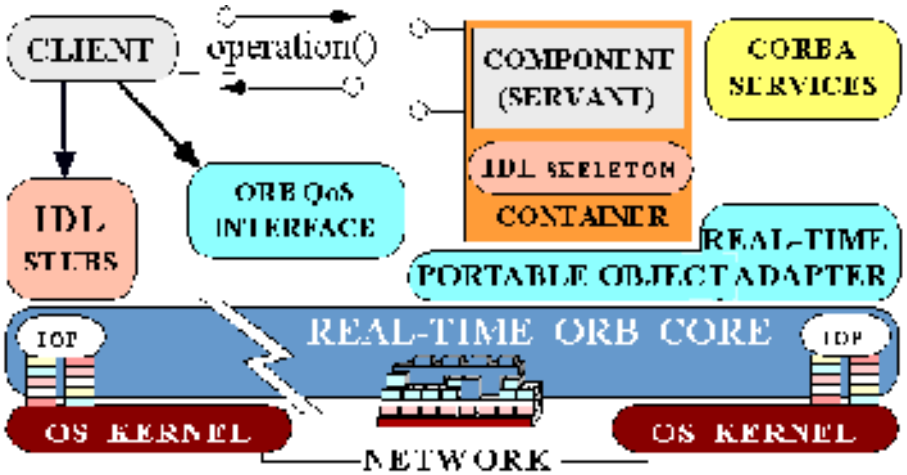
UML models of a software architecture can illustrate *how* a system is designed, but not *why* the system is designed in a particular way



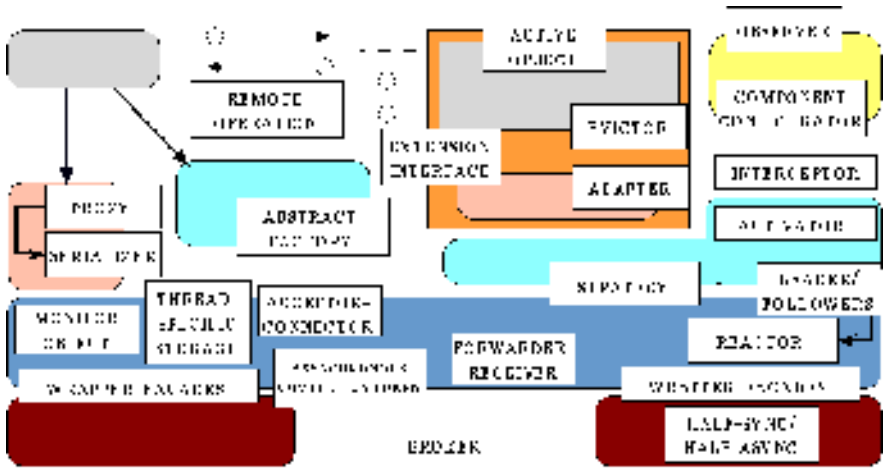
Patterns are used throughout *The ACE ORB (TAO)* Real-time CORBA implementation to codify expert knowledge & to generate the ORB's software architecture by capturing recurring structures & dynamics & resolving common design forces

R&D Context for ACE+TAO+CIAO

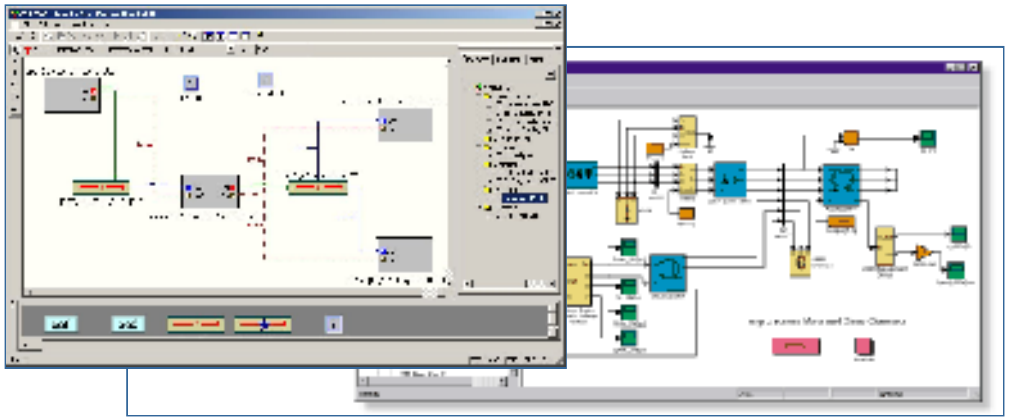
Our R&D focus: Advancing distruptive technologies to commoditize distributed real-time & embedded (DRE) systems



Standards-based QoS-enabled Middleware



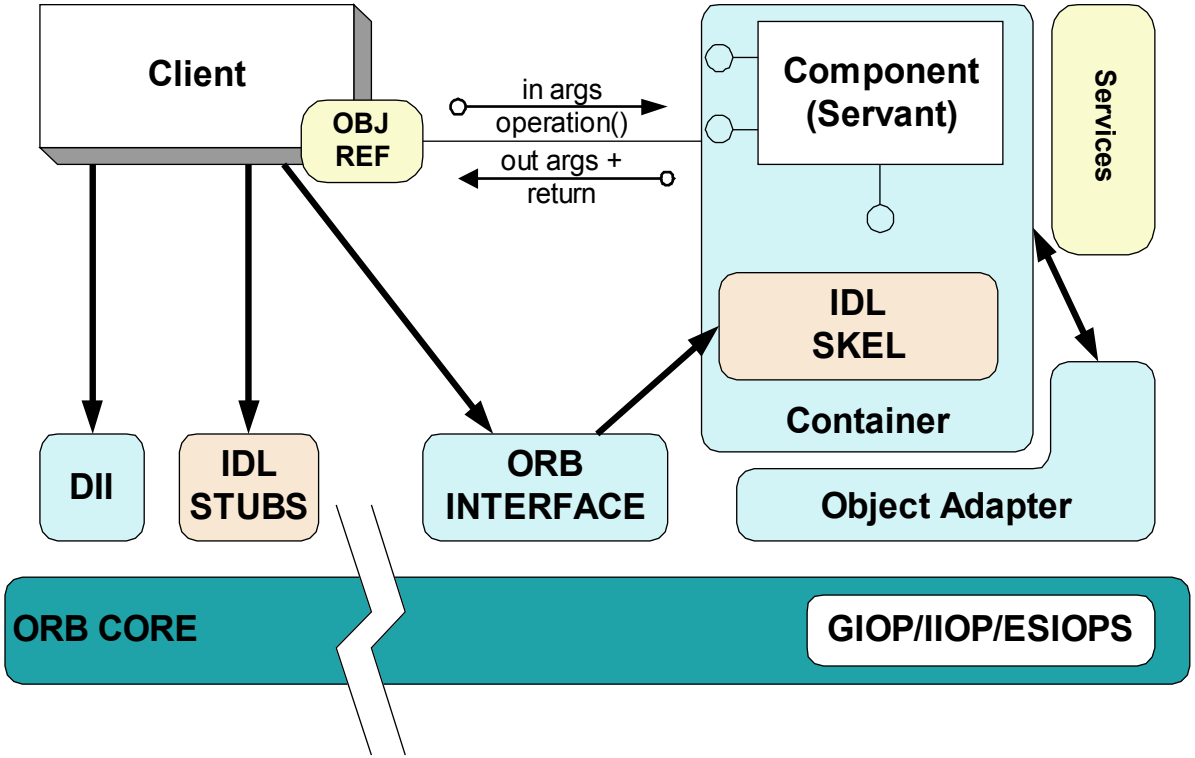
Patterns & Pattern Languages



Model-based Software Development & Domain-specific Languages

Open-source Standards-based COTS

TAO–The ACE ORB



- More than 500 Ksloc (C++)
- Open-source
- Based on ACE wrapper facades & frameworks
- Available on Unix, Win32, MVS, QNX, VxWorks, LynxOS, VMS, etc.
- Thousands of users around the world

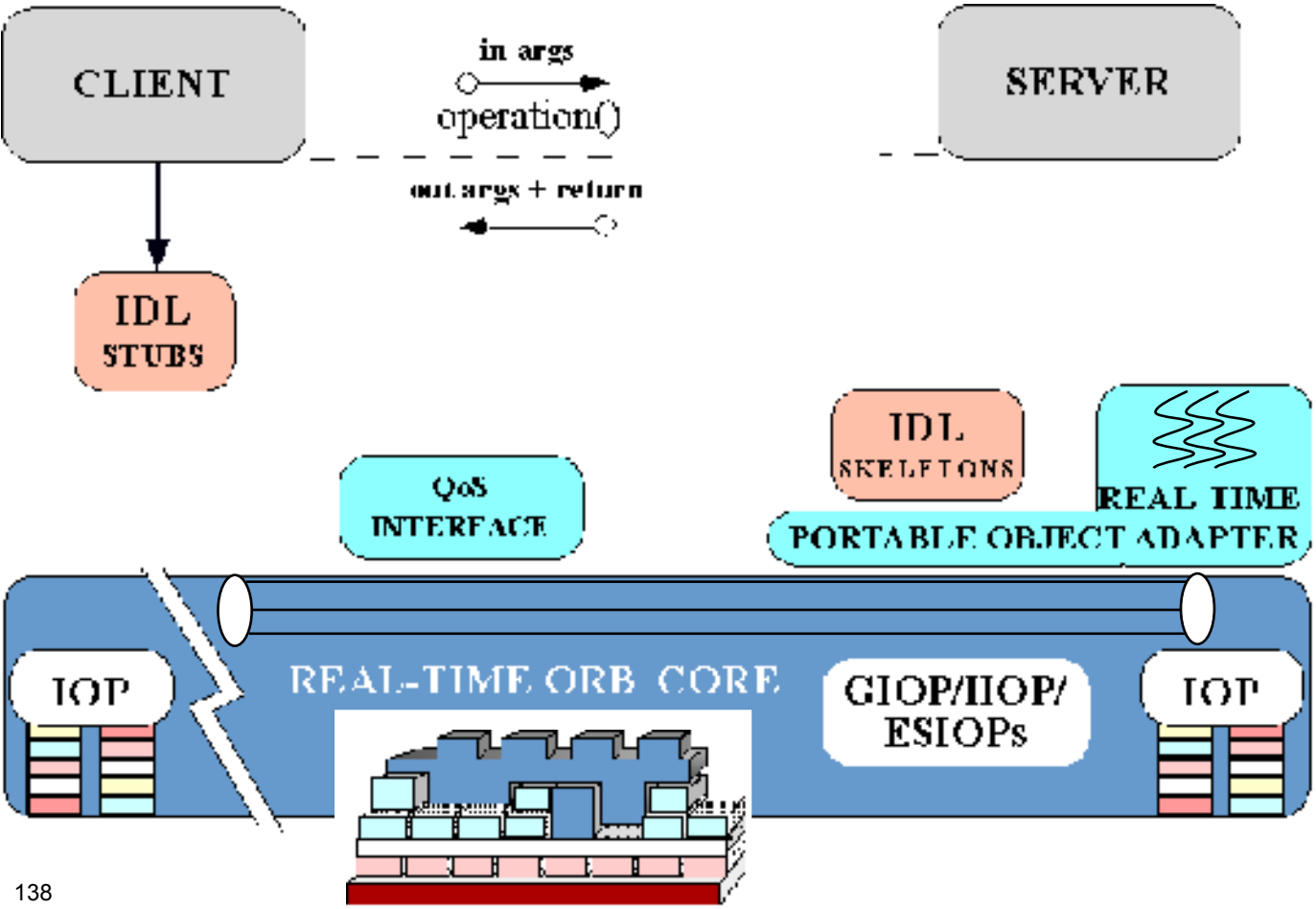
Objective: Advance technology to simplify the development of embedded & real-time systems

Approach: Use standard OO techology & patterns

- Commercially supported by many companies
  - OCI (www.theaceorb.com)
  - PrismTech (www.prismtechnologies.com)
  - And many more
    - www.cs.wustl.edu/~schmidt/commercial-support.html

The Evolution of TAO

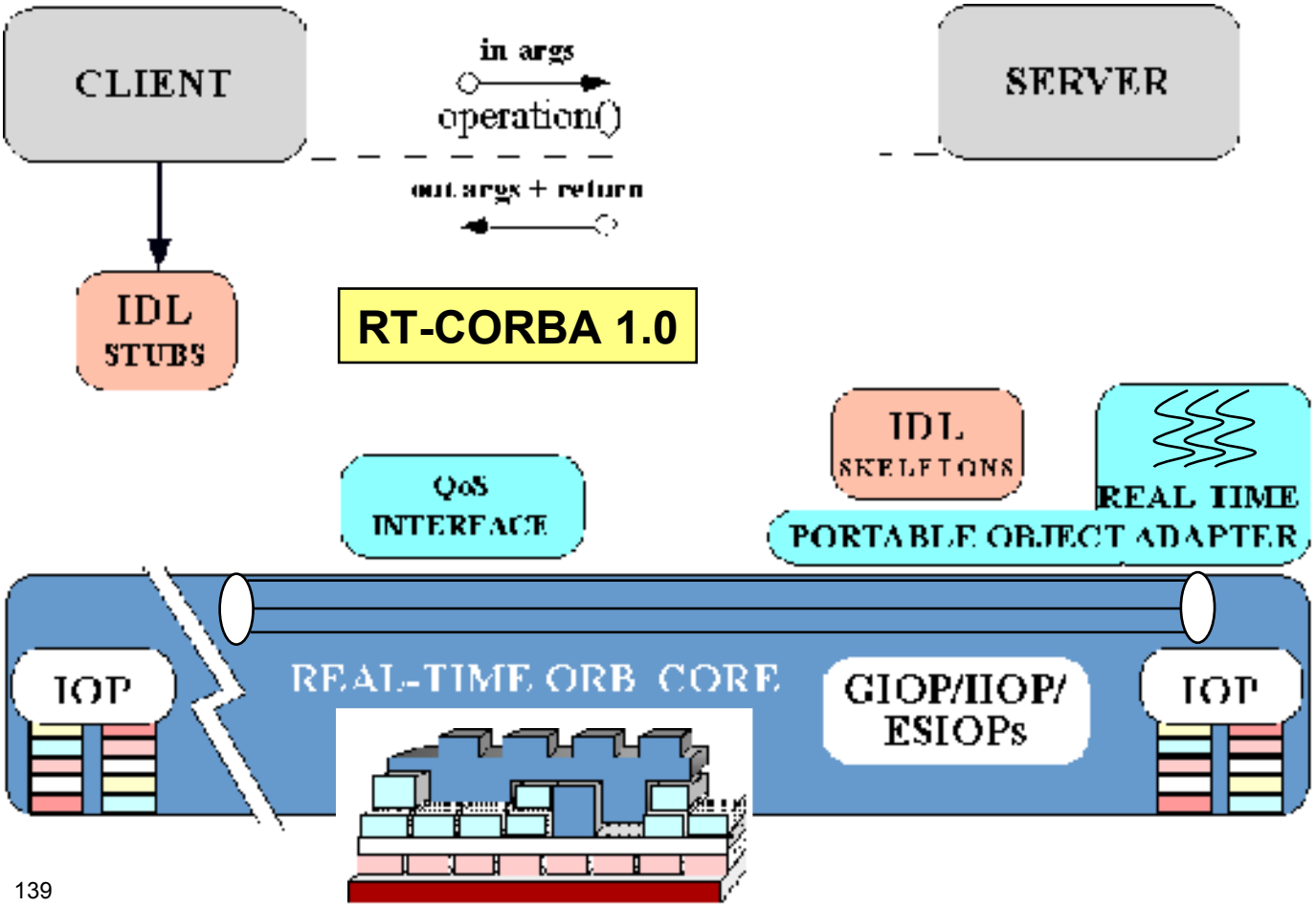
- TAO can be downloaded from
  - deuce.doc.wustl.edu/Download.html



TAO ORB

- Largely compliant with CORBA 3.0
  - No DCOM bridge ;-)
- Pattern-oriented software architecture
  - www.posa.uci.edu
- Key capabilities
  - QoS-enabled
  - Highly configurable
  - Pluggable protocols
    - IIOP/UIOP
    - DIOP
    - Shared memory
    - SSL
    - MIOP
    - SCIOP

The Evolution of TAO



RT-CORBA

- Portable priorities
- Protocol properties
- Standard synchronizers
- Explicit binding mechanisms
- Thread pools

TAO 1.4 (Jan '04)

- Current "official" release of TAO
- Heavily tested & optimized
- Baseline for next OCI & PrismTech supported releases
- www.dre.vanderbilt.edu/scoreboard

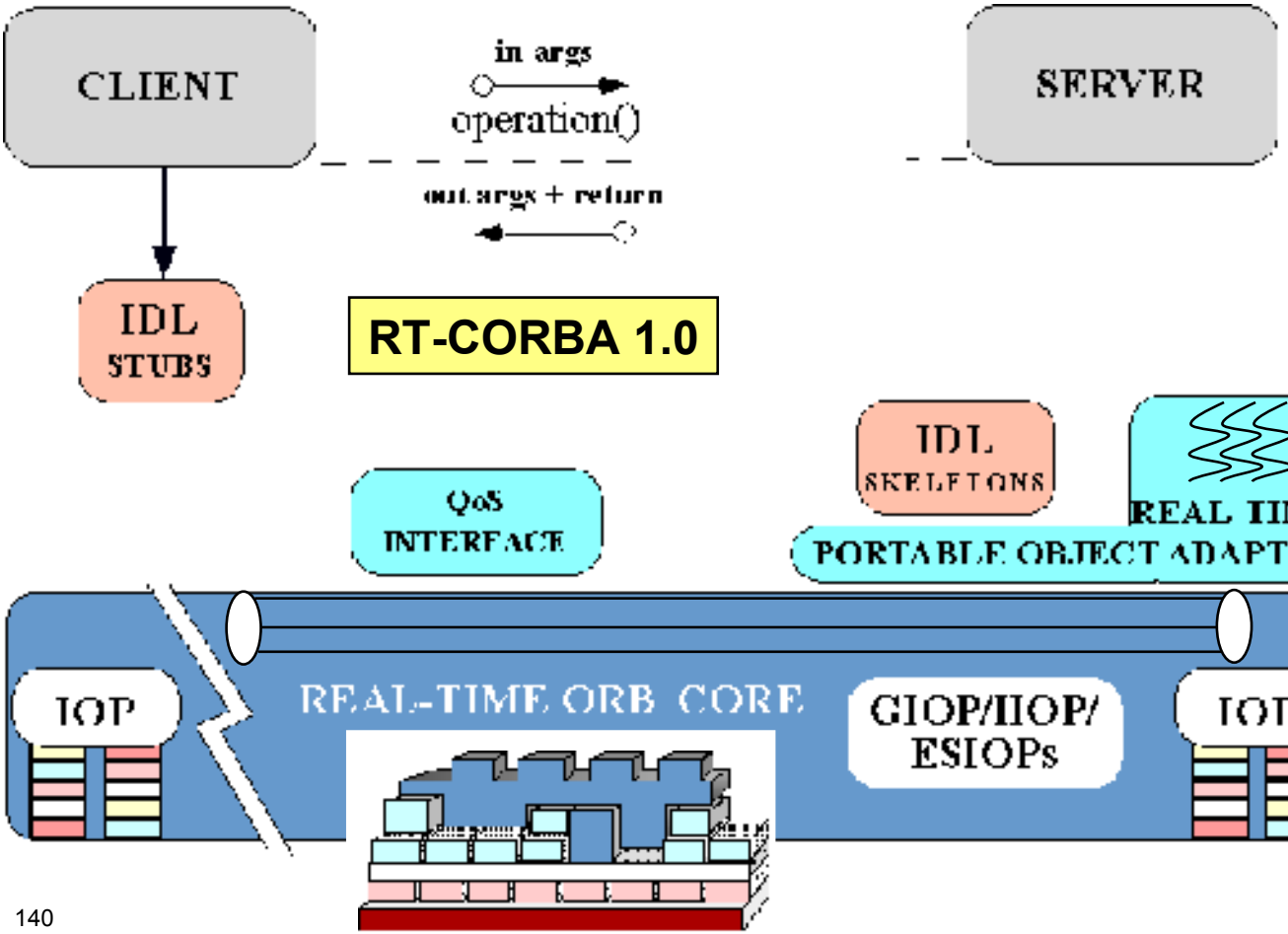
ZEN

- RT-CORBA/RT-Java
- Alpha now available www.zen.uci.edu

The Evolution of TAO

DYNAMIC/STATIC SCHEDULING

A/V STREAMING



Static Scheduling (1.0)
- Rate monotonic analysis

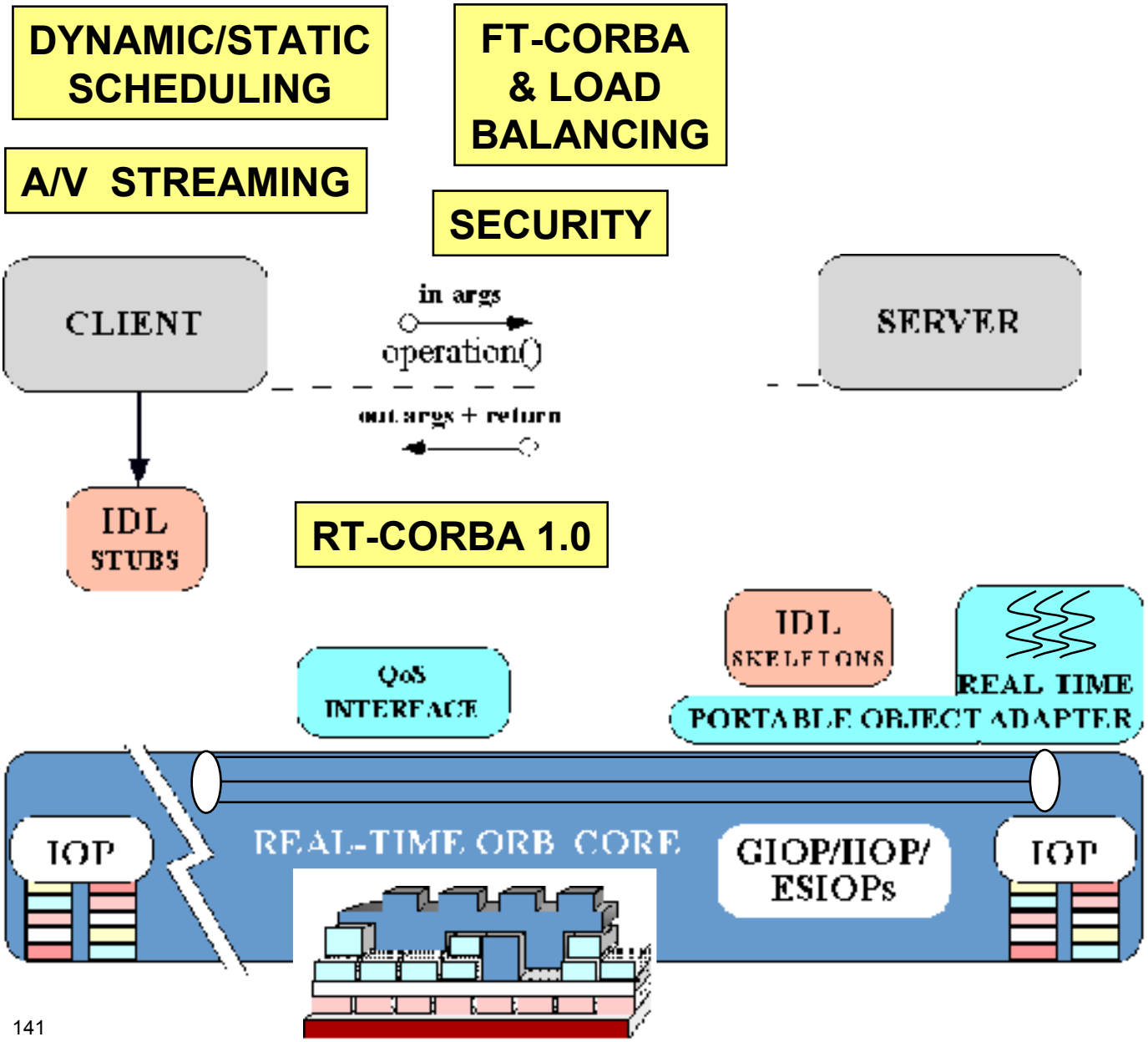
Dynamic Scheduling (2.0)
- Earliest deadline first
- Minimum laxity first
- Maximal urgency first

Hybrid Dynamic/Static
- Demo in WSOA
- Kokyu integrated in Summer 2003

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

ACE QoS API (AQoSA)
- GQoS/RAPI & DiffServ
- IntServ integrated with A/V Streaming & QuO
- DiffServ integrated with ORB

The Evolution of TAO



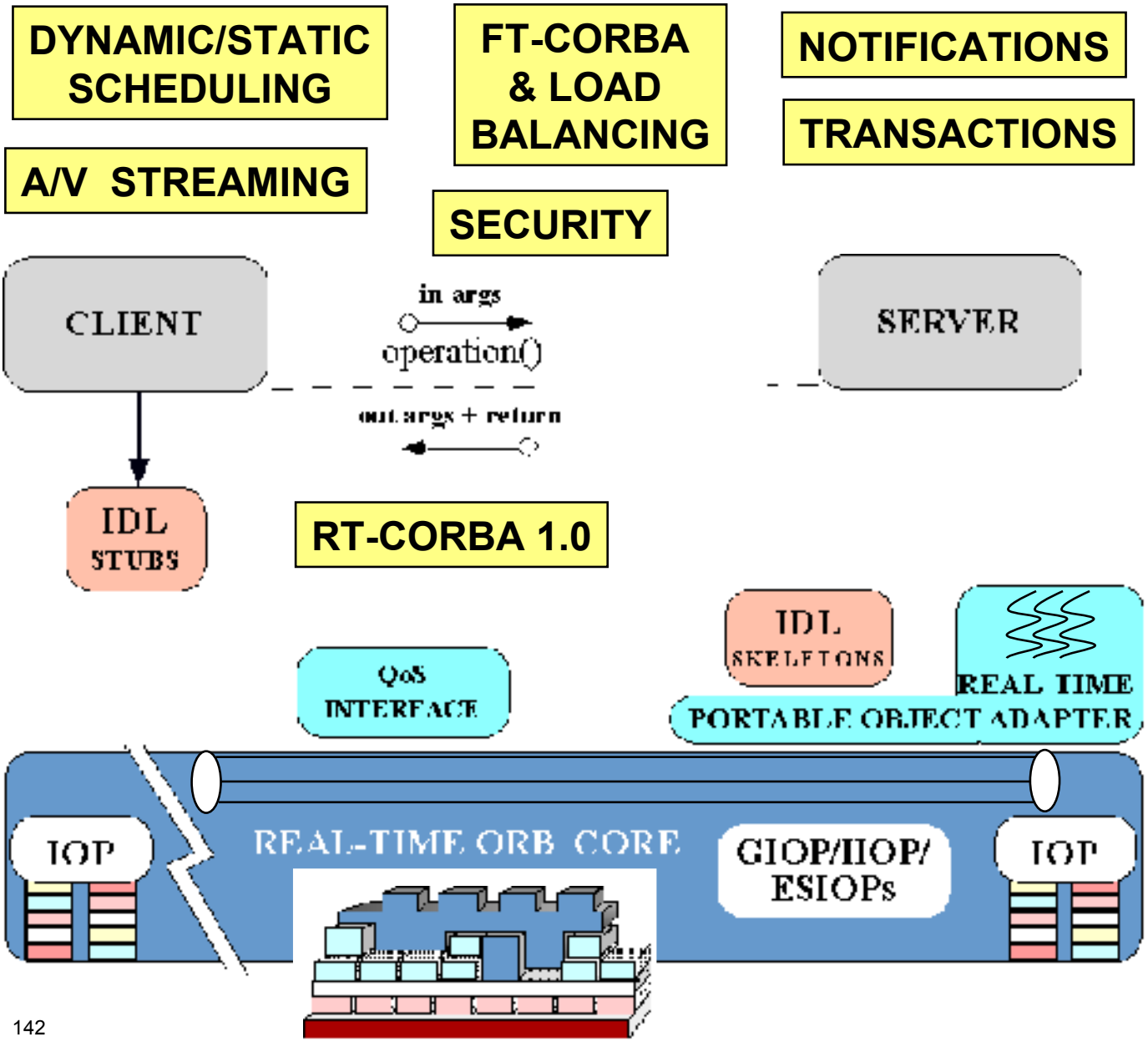
FT-CORBA (DOORS)
- Entity redundancy
- Multiple models
  - Cold passive
  - Warm passive
- IOGR
- HA/FT integrated by Winter 2004

Load Balancing
- Static & dynamic
- Integrated in TAO 1.3
- De-centralized LB
- OMG LB specification

SSL Support
- Integrity
- Confidentiality
- Authentication (limited)

Security Service (CSIv2)
- Authentication
- Access control
- Non-repudiation
- Audit
- Beta by Winter 2004

The Evolution of TAO

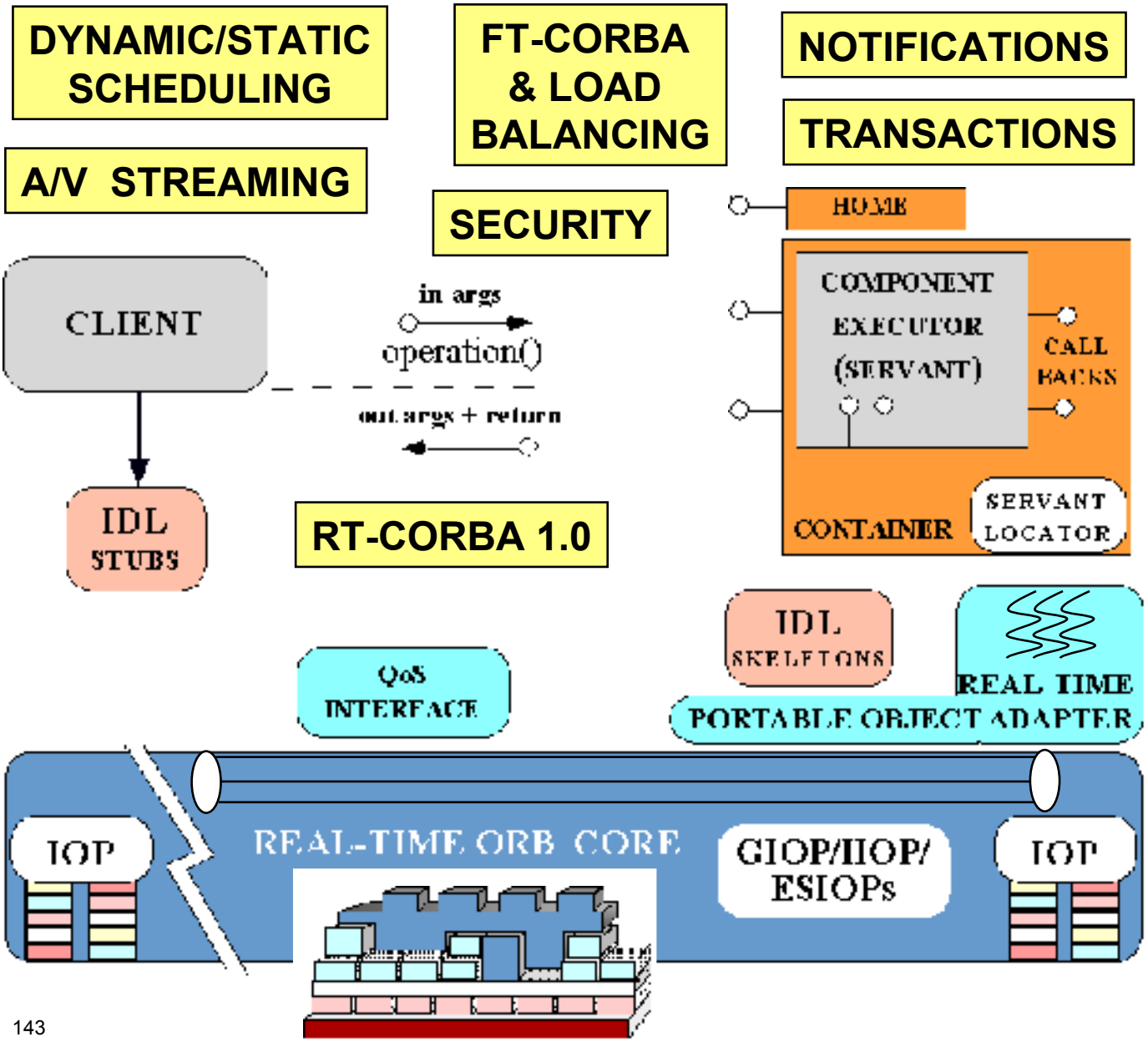


Notification Service
- Structured events
- Event filtering
- QoS properties
  - Priority
  - Expiry times
  - Order policy
- Compatible w/Events

Real-time Notification Service
- Summer 2003

Object Transaction Service
- Encapsulates RDBMs
- www.xots.org

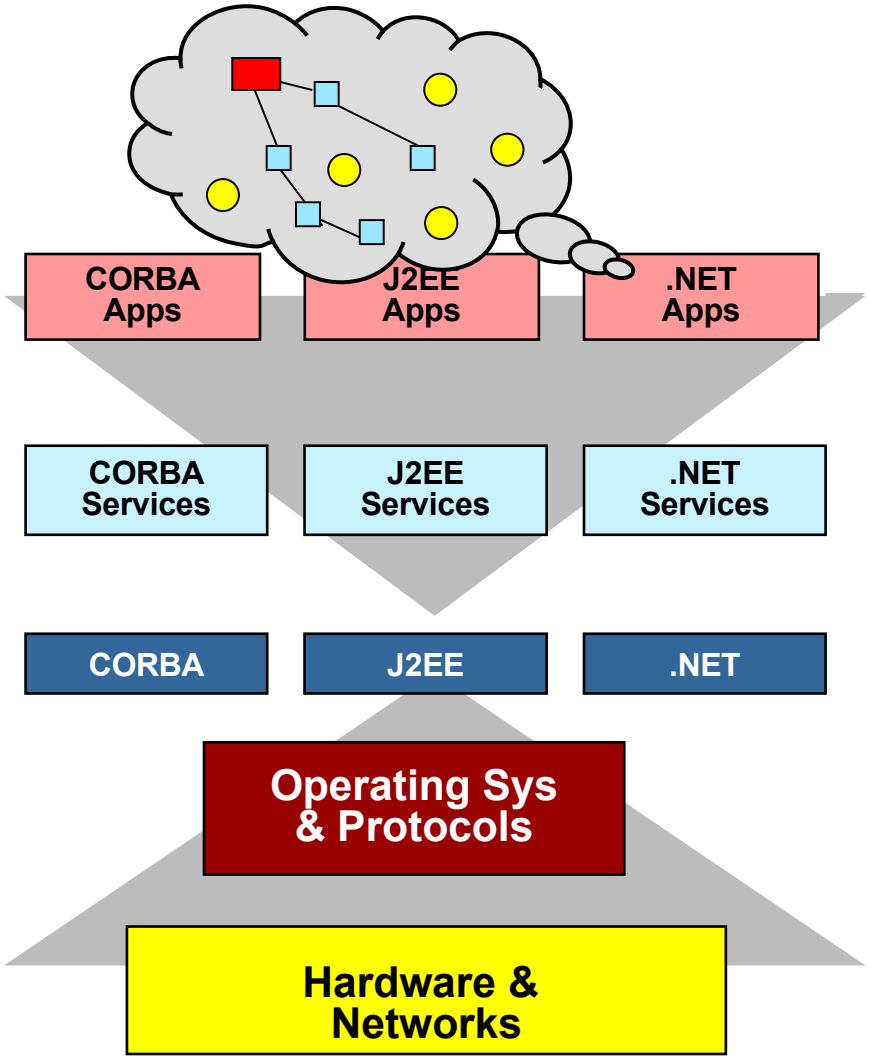
The Evolution of TAO



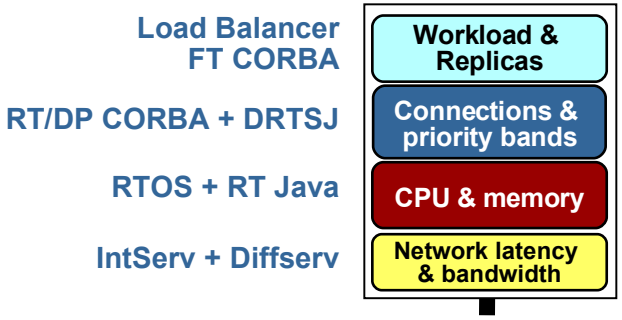
CORBA Component Model (CIAO)

- Extension Interfaces
- Component navigation
- Standardized life-cycles
- QoS-enabled containers
- Reflective collocation
- Implements the OMG Deployment & Configuration specification
- 1.0 release by Winter 2004

The Road Ahead (1/3)

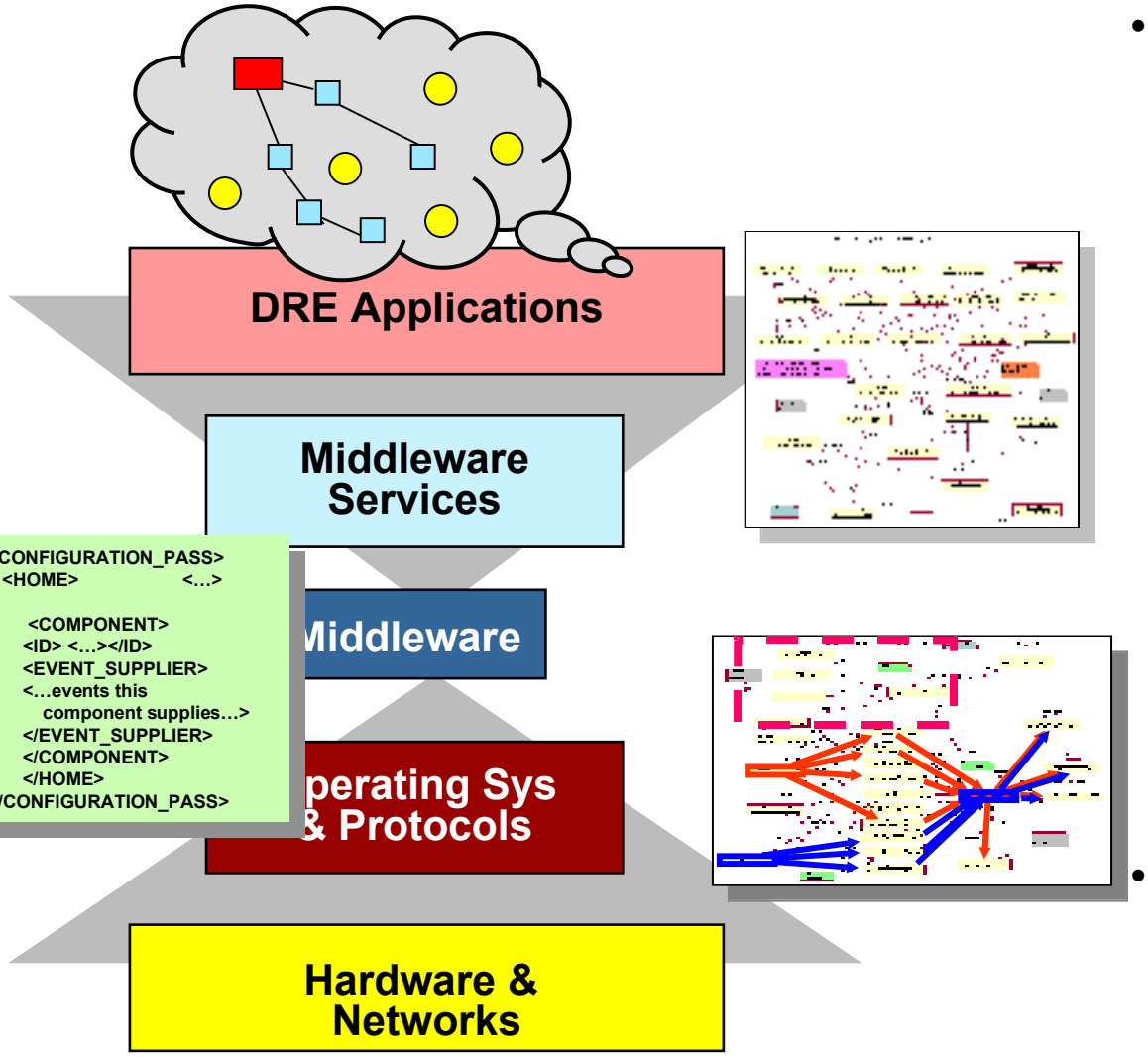


- Limit to how much application functionality can be factored into reusable COTS middleware, which impedes product-line architectures
- Middleware itself has become extremely complicated to use & provision statically & dynamically



- Component-based DRE systems are very complicated to deploy & configure
- There are now multiple middleware technologies to choose from

The Road Ahead (2/3)



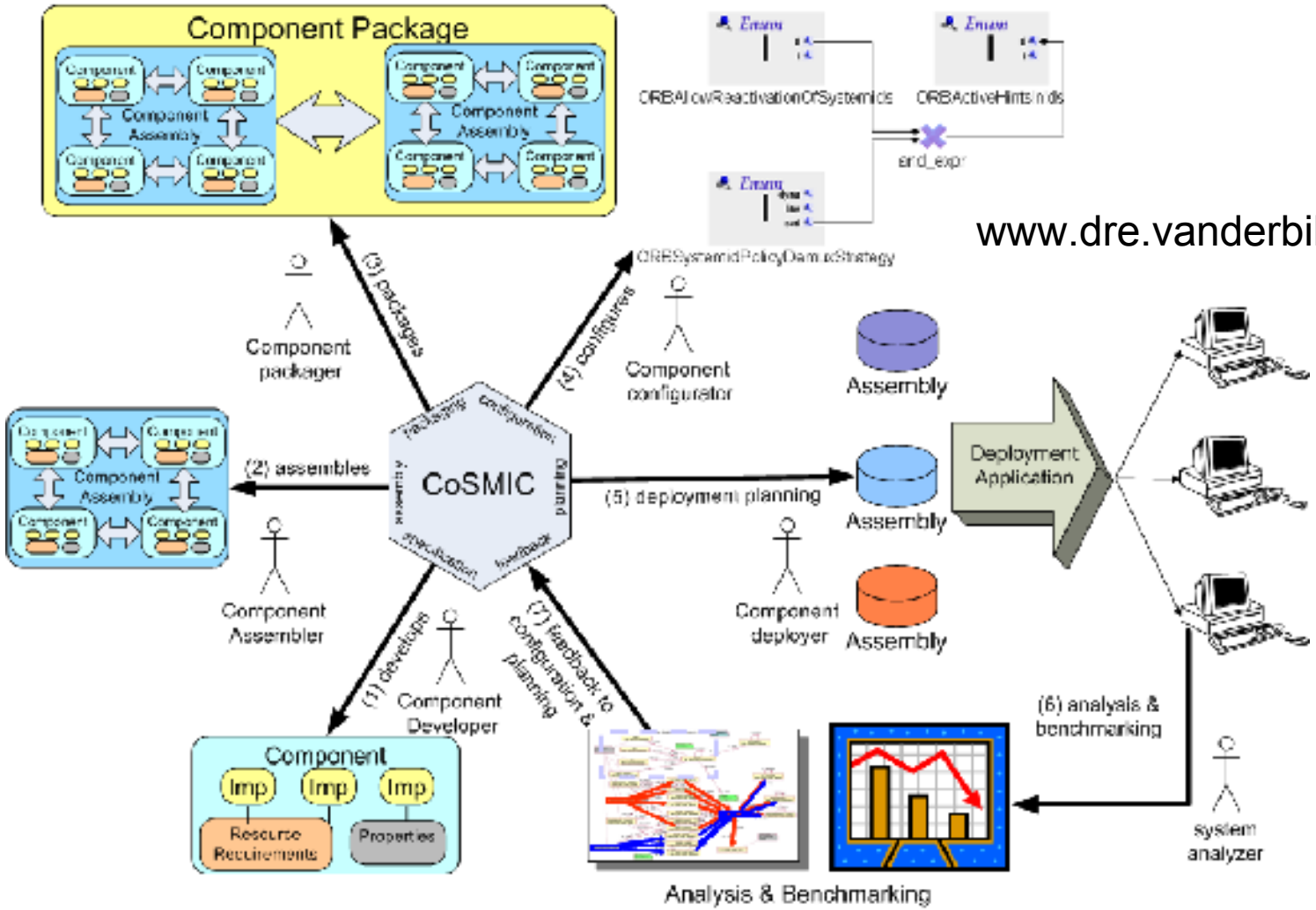
- Develop, validate, & standardize model-driven development (MDD) software technologies that:
  1. ***Model***
  2. ***Analyze***
  3. ***Synthesize &***
  4. ***Provision***

  multiple layers of middleware & ***application components*** that require ***simultaneous control*** of ***multiple quality of service properties end-to-end***
- Partial specialization is essential for inter-/intra-layer optimization & advanced product-line architectures

Goal is ***not*** to replace programmers per se – it is to provide ***higher-level domain-specific languages*** for middleware/application developers & users

The Road Ahead (3/3)

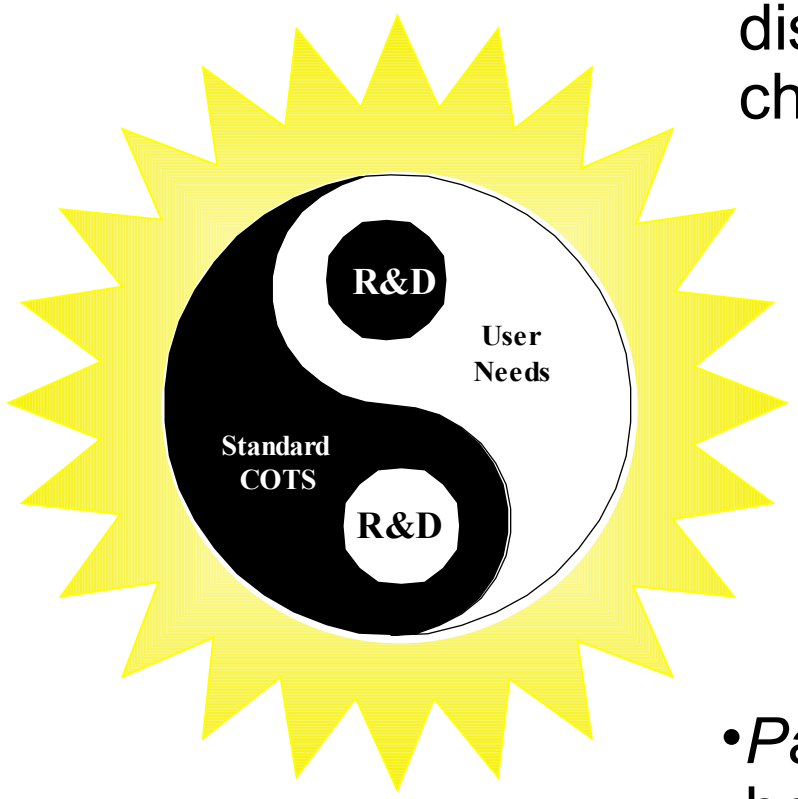
Our MDD toolsuite is called *CoSMIC* ("Component Synthesis using Model Integrated Computing")



www.dre.vanderbilt.edu/cosmic

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