Networked Embedded System
Patterns for C Developers
Part IV: Pattern Case Studies

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

Professor of EECS
Vanderbilt University
Nashville, Tennessee
The Road Ahead

CPU & network performance has increased by orders of magnitude in past decades

10 Megahertz to 3+ Gigahertz

1,200 bits/sec to 10+ Gigabits/sec

Extrapolating these trends another decade or so yields

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

Unfortunately, software quality & productivity hasn’t improved as rapidly or predictably as hardware – especially for networked embedded systems
Why Hardware Improves Consistently

Advances in hardware & networks stem largely from R&D on standardized & reusable APIs & protocols

x86 & Power PC chipsets

TCP/IP
In general, software has not been as standardized or **reusable** as hardware, especially for networked embedded systems.

**Proprietary & Stovepiped Application & Infrastructure Software**

**Standard/COTS Hardware & Networks**
What’s Hard About Software (& Software Reuse)?

**Human Nature**
- Organizational impediments
- Economic impediments
- Administrative impediments
- Political impediments
- Psychological impediments

**Technical Complexities**

**Accidental Complexities**
- Low-level APIs & debug tools
- Algorithmic decomposition

**Inherent Complexities**
- Quality of service (QoS)
- Causal ordering
- Scheduling & synchronization
- Deadlock

www.cs.wustl.edu/~schmidt/reuse-lessons.html
What’s Hard(er) About Networked Embedded SW?

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

• 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

Key challenge: How can we reuse these solutions effectively in a changing, heterogeneous world?
Promising Networked Embedded SW Technologies

Product-line Architectures
- Mission Computing Services
- Middleware Infrastructure
- Operating System
- Networking Interfaces
- Hardware (CPU, Memory, I/O)

Frameworks
- ADTs
- Strings
- Files
- Locks
- GUI
- DATABASE
- NETWORKING
- APPLICATION-SPECIFIC FUNCTIONALITY
- CALLBACKS

Patterns & Pattern Languages
- PROXY
- MONITOR OBJECT
- THREAD-SPECIFIC STORAGE
- ACCEPTOR-CONNECTOR
- FORWARDER-RECEIVER
- WRAPPER FACADES
- REMOTE OPERATION
- EXTENSION INTERFACE
- ACTIVE OBJECT
- OBSERVER
- COMPONENT CONFIGURATOR
- INTERCEPTOR
- ACTIVATOR
- STRATEGY
- LEADER/FOLLOWERS
- REACTOR
- BROKER
- HALF-SYNC

Component Middleware
- Naming
- Events
- Logging
- Locking

Model-Driven Engineering Tools
Example:

**Boeing Bold Stroke Product-line 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/Ada/C++/Java code
- **Based on COTS hardware, networks, operating systems, & middleware**
- **Used as Open Experimentation Platform (OEP)** for DARPA IXO PCES, MoBIES, SEC, MICA programs
Example: Boeing Bold Stroke Product-line Architecture

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

- Real-time CORBA middleware services
- VxWorks operating system
- VME, 1553, & Link16
- PowerPC
Separating Concerns In Bold Stroke

Context
• A complex networked embedded system like Bold Stroke with millions of lines of software, 1,000’s of components, & 100’s of developers

Problem
• Building networked embedded systems is hard due to complexity stemming from capabilities at many levels of abstraction

Solution
• Apply the Layers pattern (P1) to create a multi-tier architecture that separates concerns between tasks in different system layers
Pros & Cons of the Layers Pattern

This pattern has four **benefits:**

- **Reuse of layers**
  - If an individual layer embodies a well-defined abstraction & has a well-defined & documented interface, the layer can be reused in multiple contexts

- **Support for standardization**
  - Clearly-defined & commonly-accepted levels of abstraction enable the development of standardized tasks & interfaces

- **Dependencies are localized**
  - Standardized interfaces between layers usually confine the effect of code changes to the layer that is changed

- **Exchangeability**
  - Individual layer implementations can be replaced by semantically-equivalent implementations without undue effort

This pattern also has **liabilities:**

- **Cascades of changing behavior**
  - If layer interfaces & semantics aren’t abstracted properly then changes can ripple when behavior of a layer is modified

- **Higher overhead**
  - A layered architecture can be less efficient than a monolithic architecture

- **Unnecessary work**
  - If some services performed by lower layers perform excessive or duplicate work not actually required by the higher layer, performance can suffer

- **Difficulty of establishing the correct granularity of layers**
  - It’s important to avoid too many & too few layers
### Core Distribution Infrastructure Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Communication Style</th>
<th>Communication Relationship</th>
<th>Component Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broker</td>
<td>Remote Method Invocation</td>
<td>One-to-One</td>
<td>Component Interfaces</td>
</tr>
<tr>
<td>Messaging</td>
<td>Message</td>
<td>Many-to-One</td>
<td>Communication Endpoints &amp; Message Formats</td>
</tr>
<tr>
<td>Publisher/Subscriber</td>
<td>Events</td>
<td>One-to-Many</td>
<td>Event Formats</td>
</tr>
</tbody>
</table>

- **Broker** makes invocations on remote component objects look & act as much as possible like invocations on component objects in the same address space as their clients.

- **Messaging & Publisher-Subscriber** are most appropriate for integration scenarios where multiple, independently developed & self-contained services or applications must collaborate & form a coherent software system.
Reusable Application Domain-specific Middleware Framework

- Configurable to variable infrastructure configurations
- Supports systematic reuse of mission computing functionality
- Based on a pattern language for distributed computing
The POSA4 Pattern Language

Overview of POSA4

- The POSA4 pattern language for distributed computing includes 114 patterns grouped into thirteen problem areas.
- Each problem area addresses a specific technical topic related to building distributed systems.
Example:
Boeing Bold Stroke Product-line Architecture

**Product Line Component Model**
- Configurable for product-specific functionality & execution environment
- Single component development policies
- Standard component packaging mechanisms
**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 usecs
- 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)

1: Sensors generate data
2: I/O via interrupts
3: Sensor proxies process data & pass to missions functions
4: Mission functions perform avionics operations
Legacy Avionics Architectures

Key System Characteristics
• Hard & soft real-time deadlines
  • ~20-40 Hz
• Low latency & jitter between boards
  • ~100 usecs
• 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

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

### Structure

- **Publisher**
  - produce
  - attachPublisher
  - detachPublisher
  - attachSubscriber
  - detachSubscriber
  - pushEvent

- **Event Channel**
  - create

- **Subscriber**
  - consume

- **Event**
  - receive

- **Filter**
  - filterEvent

**Publisher-Subscriber** (P1) provides a change propagation mechanism that allows publishers in a distributed application to disseminate events that convey information that may be of interest to subscribers.
Decoupling Avionics Components

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

**Publisher-Subscriber** (P1) provides a change propagation mechanism that allows publishers in a distributed application to disseminate events that convey information that may be of interest to subscribers.
Applying Publisher-Subscriber 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
Pros & Cons of Publisher-Subscriber

This pattern has two **benefits:**

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

There are also **liabilities:**

- **Must be careful with potential update cascades**
  - Since subscribers have no knowledge of each other’s presence, applications may not recognize the ultimate cost of publishing events through an event channel
  - A seemingly innocuous operation on the subject may therefore cause a cascade of updates to observers & their dependent objects
- **Performance/complexity degradation relative to point-to-point request/response interactions**
Ensuring Platform-neutral & Network-transparent Communication

<table>
<thead>
<tr>
<th>Context</th>
<th>Problems</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Mission computing requires remote IPC • Stringent DRE requirements</td>
<td>• Applications need capabilities to: • Support remote communication • Provide location transparency • Handle faults • Manage end-to-end QoS • Encapsulate low-level system details</td>
<td>• Apply the <strong>Broker</strong> architectural pattern to provide platform-neutral communication between mission computing boards</td>
</tr>
</tbody>
</table>

**Broker** (P1) defines a component-based programming model that clients use to invoke methods on remote services as if they were local.
Ensuring Platform-neutral & Network-transparent Communication

<table>
<thead>
<tr>
<th>Context</th>
<th>Problems</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mission computing requires remote IPC</td>
<td>• Applications need capabilities to:</td>
<td>• Apply the <strong>Broker</strong> architectural pattern to provide platform-neutral communication between mission computing boards</td>
</tr>
<tr>
<td>Stringent DRE requirements</td>
<td>• Support remote communication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Provide location transparency</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Handle faults</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Manage end-to-end QoS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Encapsulate low-level system details</td>
<td></td>
</tr>
</tbody>
</table>

**Structure**

**Broker** (P1) defines a component-based programming model that clients use to invoke methods on remote services as if they were local.
## Ensuring Platform-neutral & Network-transparent Communication

<table>
<thead>
<tr>
<th>Context</th>
<th>Problems</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Mission computing requires remote IPC</td>
<td>• Applications need capabilities to:</td>
<td>• Apply the <strong>Broker</strong> architectural pattern to provide platform-neutral communication between mission computing boards</td>
</tr>
<tr>
<td>• Stringent DRE requirements</td>
<td>• Support remote communication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Provide location transparency</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Handle faults</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Manage end-to-end QoS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Encapsulate low-level system details</td>
<td></td>
</tr>
</tbody>
</table>

### Dynamics

- **: Client**
  - operation (params)
- **: Client Proxy**
  - connect
  - marshal
  - receive_reply
  - unmarshal
- **: Broker**
  - register_service
  - send_request
  - dispatch
  - operation (params)
  - result
- **: Object Adapter**
  - assigned port
- **: Server**
  - start_up

**Broker** (P1) defines a component-based programming model that clients use to invoke methods on remote services as if they were local.
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.

1: Sensors generate data
2: I/O via interrupts
3: Broker handles I/O via upcalls
4: Sensor publishers push events to event channel
5: Event Channel pushes events to subscribers(s)
6: Subscribers perform avionics operations
Pros & Cons of the Broker Pattern

This pattern has five benefits:

- **Portability enhancements**
  - A broker hides OS & network system details from clients & servers by using indirection & abstraction layers, such as APIs, proxies, adapters, & bridges

- **Interoperability with other brokers**
  - Different brokers may interoperate if they understand a common protocol for exchanging messages

- **Reusability of services**
  - When building new applications, brokers enable application functionality to reuse existing services

- **Location transparency**
  - A broker is responsible for locating servers, so clients need not know where servers are located

- **Changeability & extensibility of components**
  - If server implementations change without affecting interfaces clients should not be affected

This pattern also has liabilities:

- **Higher overhead**
  - Applications using brokers may be slower than applications written manually

- **Potentially less reliable**
  - Compared with non-distributed software applications, distributed broker systems may incur lower fault tolerance

- **Testing & debugging may be harder**
  - Testing & debugging of distributed systems is tedious because of all the components involved
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
Key Patterns Used in Real-time CORBA

- **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

## Enhancing Flexibility with Strategy

<table>
<thead>
<tr>
<th>Context</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Multi-domain resuable middleware framework</td>
<td>• Flexible ORBs must support multiple event &amp; request demuxing, scheduling, (de)marshaling, connection mgmt, request transfer, &amp; concurrency policies</td>
<td>• Apply the <em>Strategy</em> pattern to factory out commonality amongst variable ORB algorithms &amp; policies</td>
</tr>
</tbody>
</table>

### Diagram

- **Hook for marshaling strategy**
- **Hook for the request demuxing strategy**
- **Hook for the event demuxing strategy**
- **Hook for the connection management strategy**
- **Hook for the concurrency strategy**
- **Hook for the underlying transport strategy**

---

30
## Consolidating Strategies with Abstract Factory

<table>
<thead>
<tr>
<th>Context</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• A heavily strategized framework or application</td>
<td>• Aggressive use of Strategy pattern creates a configuration nightmare</td>
<td>• Apply the Abstract Factory pattern to consolidate multiple ORB strategies into semantically compatible configurations</td>
</tr>
<tr>
<td></td>
<td>• Managing many individual strategies is hard</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• It’s hard to ensure that groups of semantically compatible strategies are configured</td>
<td></td>
</tr>
</tbody>
</table>

### Concrete factories create groups of strategies

**Diagram:**
- **Thread-per-Connection**: Medical Imaging Concrete Factory
  - **FIFO Dispatching**: ORB Server Abstract Factory
  - **Dispatching Strategy**: Avionics Concrete Factory
  - **Demuxing Strategy**: Rate-based Dispatching
  - **Concurrency Strategy**: Perfect Hashing

**Network Diagram:**
- **ORB CORE**
- **OS KERNEL**
- **OS I/O SUBSYSTEM**
- **NETWORK INTERFACES**
- **GIOP**

**Client Interaction:**
- **Operation**: with Object Reference (OBJ REF)
- **ORB INTERFACE**: with IDL Skeleton

**Object Adapter:**
- **IDL STUBS**
- **ORB INTERFACE**
- **IDL SKELETON**
# Configuring Factories with Component Configurator

<table>
<thead>
<tr>
<th>Context</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
</table>
| • Resource constrained & highly dynamic environments | • Prematurely committing to a particular ORB configuration is inflexible & inefficient  
  • Certain decisions can’t be made until runtime  
  • Forcing users to pay for components they don’t use is undesirable | • Apply the **Component Configurator** pattern to assemble the desired ORB factories (& thus strategies) dynamically |

---

## Context

<table>
<thead>
<tr>
<th>TAO PROCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread-per Rate Concurrency</td>
</tr>
<tr>
<td>Service Repository</td>
</tr>
</tbody>
</table>

## Problem

<table>
<thead>
<tr>
<th>Priority-based Dispatching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect Hashing</td>
</tr>
</tbody>
</table>

## Solution

| Component Configurator (P2) decouples component interfaces from their implementations & provides a mechanism to (re)configure components in an application without having to shutdown & restart |

---

```c
svc.conf
 FILE
dynamic ORB Service_Object *
avionics_orb:make_orb() "-ORBport 2001"
```
## Component Configurator Pattern Dynamics

<table>
<thead>
<tr>
<th>Context</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
</table>
| • Resource constrained & highly dynamic environments | • Prematurely committing to a particular ORB configuration is inflexible & inefficient  
  • Certain decisions can’t be made until runtime  
  • Forcing users to pay for components they don’t use is undesirable | • Apply the **Component Configurator** pattern to assemble the desired ORB factories (& thus strategies) dynamically |

```java
void configure (Script script) {
    // Load DLL and create component.
    dll = load_dll (script.dll_name);
    comp = dll.make_component
        (script.comp_name);
    //Insert component into repository
    repository.insert (comp);
    //Start component.
    comp.service ();
}
```

### Diagram:

- **Component Configurator**
- **DLL**
  - **Service**
- **Component Repository**
  - **Insert**
  - **Remove**
  - **Service**
Pros & Cons of Component Configurator

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
## Minimizing Resource Utilization (1/2)

<table>
<thead>
<tr>
<th>Context</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource constrained &amp; highly dynamic environments</td>
<td>It may not feasible to have all application server implementations running all the time since this ties up end-system resources unnecessarily</td>
<td>Apply the Activator pattern to activate &amp; deactivate application servers automatically</td>
</tr>
</tbody>
</table>

---

**Activator (PLoP12/P4)**

Activator automates scalable on-demand activation & deactivation of service execution contexts to run services accessed by many clients without consuming resources unnecessarily.
An Activator can be used to activate & passivate a server
- e.g., after each method call, after each transaction, etc.
- A container/component in a server can also passivate the server itself
Applying Activator to Bold Stroke

- The Activator pattern can launch servers on-demand
- The Activator pattern is available in various COTS technologies:
  - UNIX Inetd “super server”
  - CORBA Implementation Repository

1. some_request
2. ping
3. is_running
4. LOCATION_FORWARD
5. some_request
6. some_response

Client

| iiop://ringil:5000/poa_name/object_name |
| iiop://ringil:5500/poa_name/object_name |

ImR (ringil:5000)

| poa_name | server.exe | ringil:5500 |
| airplane_poa | plane.exe | ringil:4500 |

Server (ringil:5500)

• The Activator pattern can launch servers on-demand
• The Activator pattern is available in various COTS technologies:
  - UNIX Inetd “super server”
  - CORBA Implementation Repository

Applying Activator to Bold Stroke
Pros & Cons of the Activator Pattern

This pattern has three **benefits**:  

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

This pattern also has **liabilities**:  

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

These technologies codify expertise of skilled 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
Example: High-performance Content Delivery Servers

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

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

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

Graphics Adapter
GUI
Requester
HTML Parser

Transfer Protocol
e.g., HTTP 1.0

OS Kernel
& Protocols

TCP/IP Network

HTTP Client
GET /index.html HTTP/1.0

HTTP Server
www.posa.uci.edu

File Cache
Protocol Handlers
Event Dispatcher
JAWS Content Server Framework

Key Sources of Variation
- Concurrency models
  - e.g., thread pool vs. thread-per-connection
- 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 vs. IIOP
- Operating system APIs
  - e.g., Windows, UNIX, RTOS

Event Dispatcher
- Accepts client connection request events, receives HTTP GET requests, & coordinates JAWS’s event demultiplexing strategy with its concurrency strategy

Protocol Handler
- Performs parsing & protocol processing of HTTP request events.

Cached Virtual Filesystem
- Improves Web server performance by reducing the overhead of file system accesses when processing HTTP GET requests
Applying Patterns to Resolve Key JAWS Design Challenges

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 & efficiently.

This pattern encapsulates data & functions provided by existing C APIs within more concise, robust, portable, maintainable, & cohesive functions and/or classes.
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

```c
void acquire() {
    mutex_lock(mutex);
}
void release() {
    mutex_unlock(mutex);
}
```

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

download.dre.vanderbilt.edu

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

```c
select (width, &read_handles, 0, 0, 0);
if (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()`
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 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** 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.
1. Passive-mode endpoint initialize phase

2. Service handler initialize phase

3. Service processing phase

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

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

Motivation for Synchrony

- If connection latency is negligible
  - e.g., connecting with a server on the same host via a ‘loopback’ device
- If multiple threads of control are available & it is efficient to use a thread-per-connection to connect each service handler synchronously
- If the services must be initialized in a fixed order & the client can’t perform useful work until all connections are established

1. Sync connection initiation phase
2. Service handler initialize phase
3. Service processing phase
Asynchronous Connector Dynamics

Motivation for Asynchrony

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

1. If client is establishing connections over high latency links
2. If client is a single-threaded application
3. If client is initializing many peers that can be connected in an arbitrary order

[Diagram showing the interactions between Application, Connector, Service Handler, and Dispatcher, with events like connect(), complete(), open(), register_handler(), handle_event(), and service().]
Applying the Reactor & 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

Connection Management Phase

Data Transfer Phase
Pros & 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

• 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, & 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-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 & 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 & starting services.
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:

(1) Half-Sync/Half-Async Strategy

(2) Leader/Followers Strategy

• These strategies based on Half-Sync/Half-Async & Leader/Followers patterns
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
- *Process contention scope* (aka “user threading”) where threads in the same process compete with each other (but not directly with threads in other processes)

(2) 1:1 KERNEL THREADING MODEL
- *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.
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 & 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

• 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

• 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
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.

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

This solution yields two benefits:

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

1. Synchronized method invocation & serialization
2. Synchronized method thread suspension
3. Monitor condition notification
4. Synchronized method thread resumption
Applying Monitor Object Pattern in JAWS

The JAWS synchronized request queue implements the queue’s
*not-empty* & *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

www.cs.wustl.edu/~schmidt/C++2java.html
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.

![Diagram illustrating the `accept()` optimization process.](image-url)
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 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—a separate Reactor thread & synchronized request queue used in the Half-Sync/Half-Async pattern.
Leader/Followers Pattern Dynamics

1. Leader thread demuxing
   - Leader thread sleeps until it becomes the leader
   - Follower thread promotes itself

2. Follower thread promotion
   - Follower thread waits for a new event
   - Leader thread processes the current event
   - New leader is promoted

3. Event handler demuxing & event processing
   - Leader thread demuxes events
   - Leader thread processes events
   - Follower thread reacts to new events

4. Rejoining the thread pool
   - Leader thread rejoins the thread pool
   - Follower thread rejoins the thread pool
Two options:
1. If platform supports `accept()` optimization then the Leader/Followers pattern can be implemented by the OS
2. Otherwise, this pattern can be implemented as a reusable framework

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

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

This pattern provides two **benefits**:

- **Performance enhancements**
  - This can improve performance as follows:
    - It enhances CPU cache affinity & 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 & 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
The Proactor pattern structures the JAWS concurrent server to receive & process requests from multiple clients asynchronously.

JAWS HTTP components are split into two parts:
1. Operations that execute asynchronously
   • e.g., to accept connections & receive client HTTP GET requests
2. The corresponding completion handlers that process the async operation results
   • e.g., to transmit a file back to a client after an async connection operation completes
Proactive Connection Management & Data Transfer in JAWS

Connection Management Phase

1. accept()
2. AcceptEx
3. handle_events()
4. connect()
5. accept complete
6. handle_event (AcceptorHandle, ACT)
7. create
8. ReadFile (SockHandle, data, ACT)
9. handle_events()

Data Transfer Phase

1. GET /~index.html
2. read complete
3. handle_event (READ_EVENT)
4. parse request()
5. memory map (File)
6. WriteFile (SockHandle, data, ACT)
7. write complete
8. handle_event (WRITE_EVENT)
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 & 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
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 dynamic linking & initialization

2. Component processing

3. Component termination & dynamic unlinking
Applying the Component Configurator Pattern to Content 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 content server can apply the Component Configurator pattern to configure variousCached Virtual Filesystemstrategies
  • 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 content server.
Reconfiguring JAWS

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

# Configure a image server.
set dynamic File_Cache Component *
web_server.dll:make_File_Cache()
"-t LRU"

# Reconfigure a image server. Remove File_Cache
set dynamic File_Cache Component *
web_server.dll:make_File_Cache()
"-t LFU"
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
Additional Information

- Patterns & frameworks for concurrent & networked objects
  - www.cs.wustl.edu/~schmidt/POSA/

- 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

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