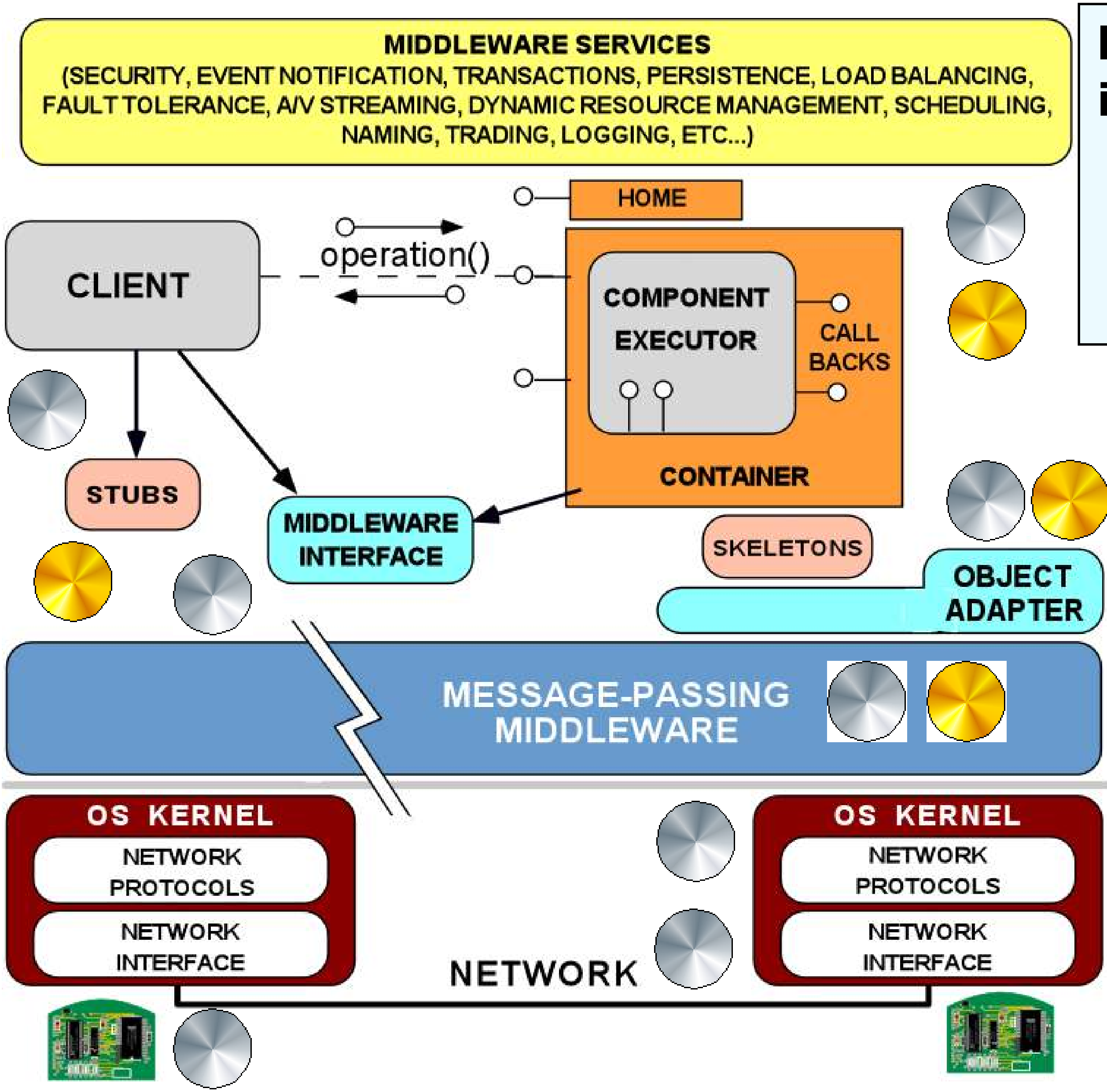
Patterns, Frameworks, & Middleware: Their Synergistic Relationships

Frontiers of Software Practice 2003

Douglas C. Schmidt
d.schmidt@vanderbilt.edu

Professor of EECS
Vanderbilt University
Nashville, Tennessee

Technology Trends (1/3)



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.:*

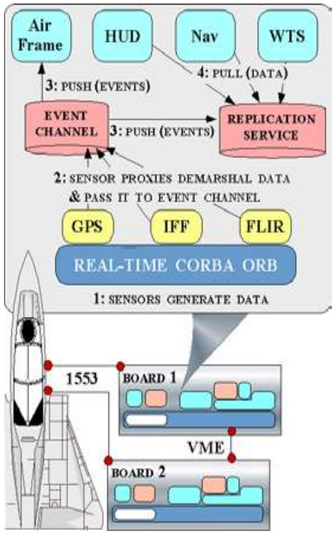
- Intel x86 & Power PC chipsets
- TCP/IP, GSM, Link16
- POSIX, Windows, & VMs
- Middleware & component models
- Quality of service (QoS) aspects

Technology Trends (2/3)

Growing acceptance of a network-centric component paradigm

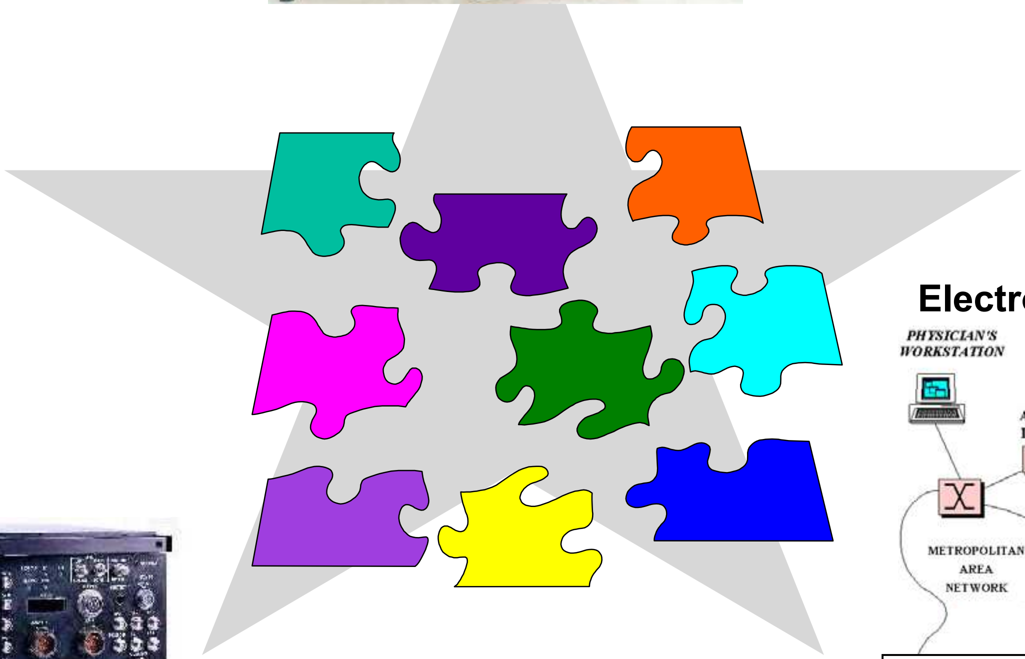
- i.e., distributed applications with a range of QoS needs are constructed by integrating components & frameworks via various communication mechanisms

Avionics Mission Computing

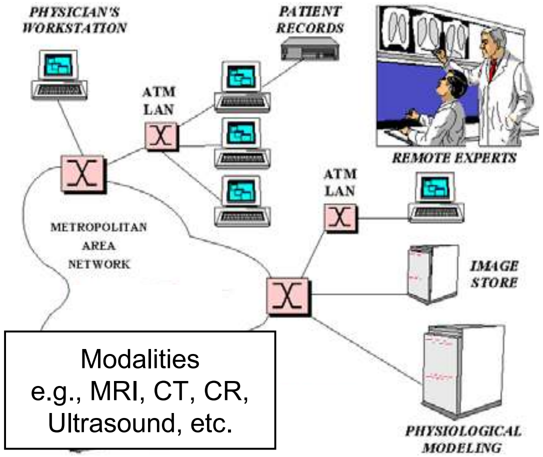


Process Automation Quality Control

Hot Rolling Mills

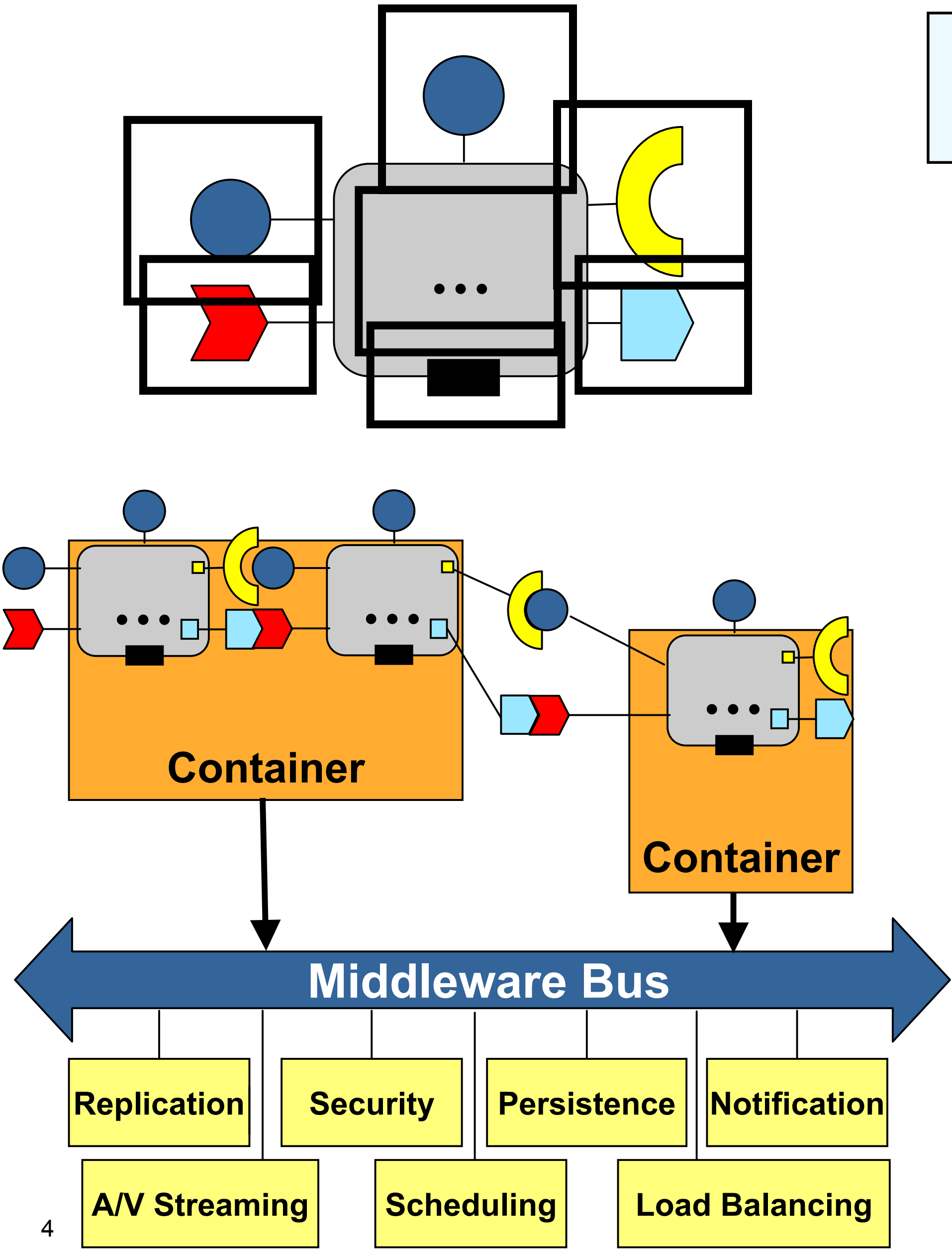


Electronic Medical Imaging



Software Defined Radio

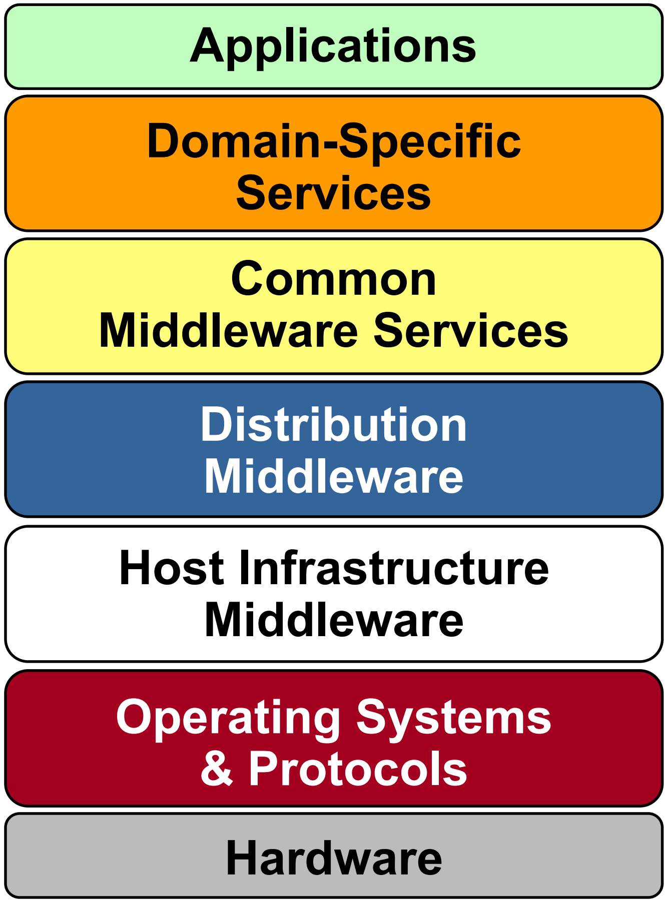
Technology Trends (3/3)



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***

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

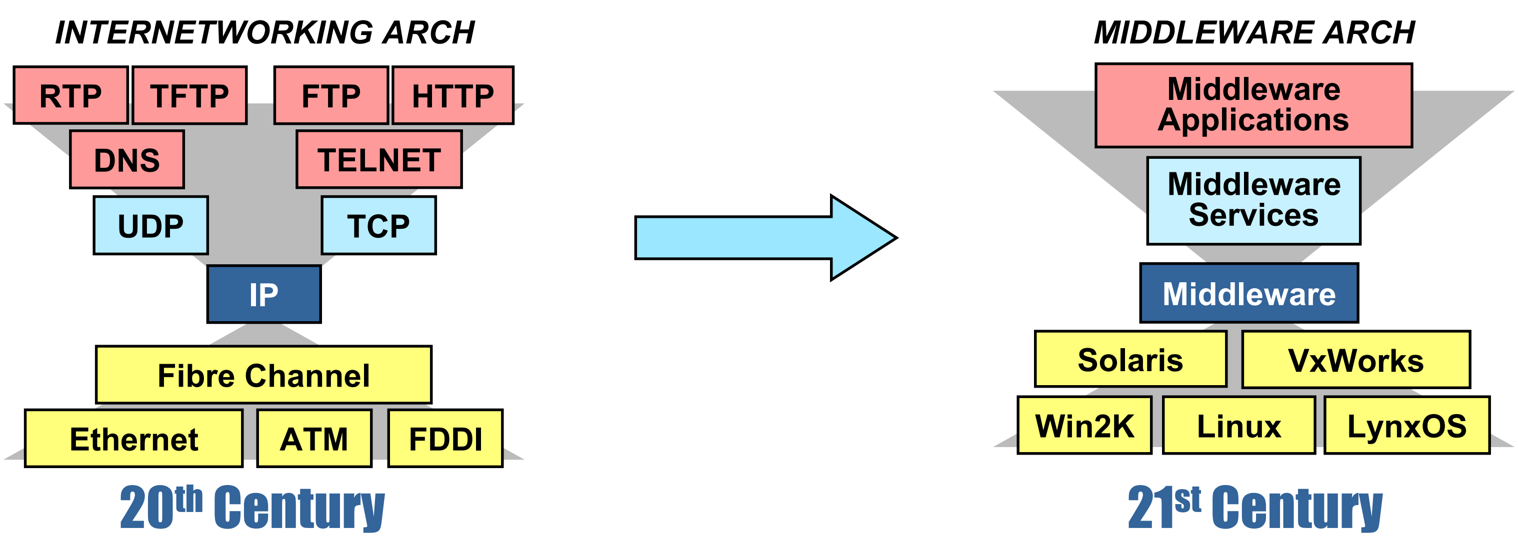
There are layers of middleware, just like there are layers of networking protocols

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 reuseable, off-the-shelf developer-oriented services

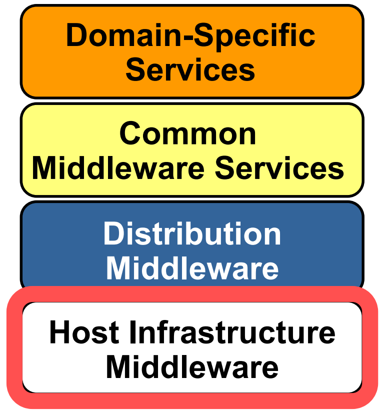
Operating System & Protocols

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

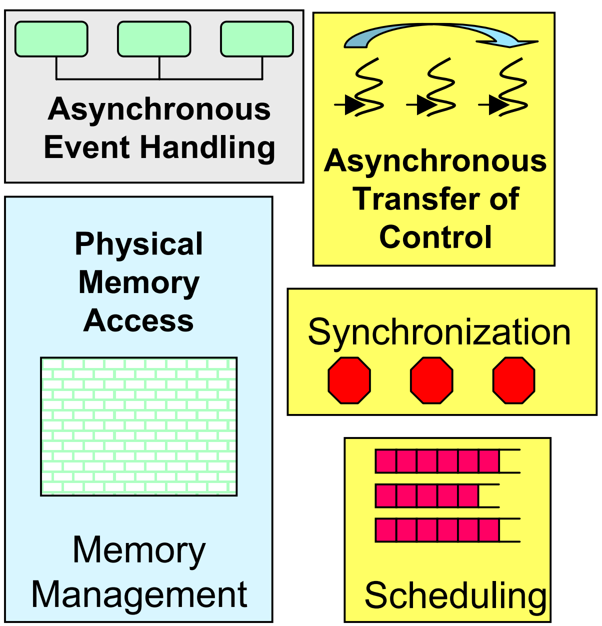


Host Infrastructure Middleware

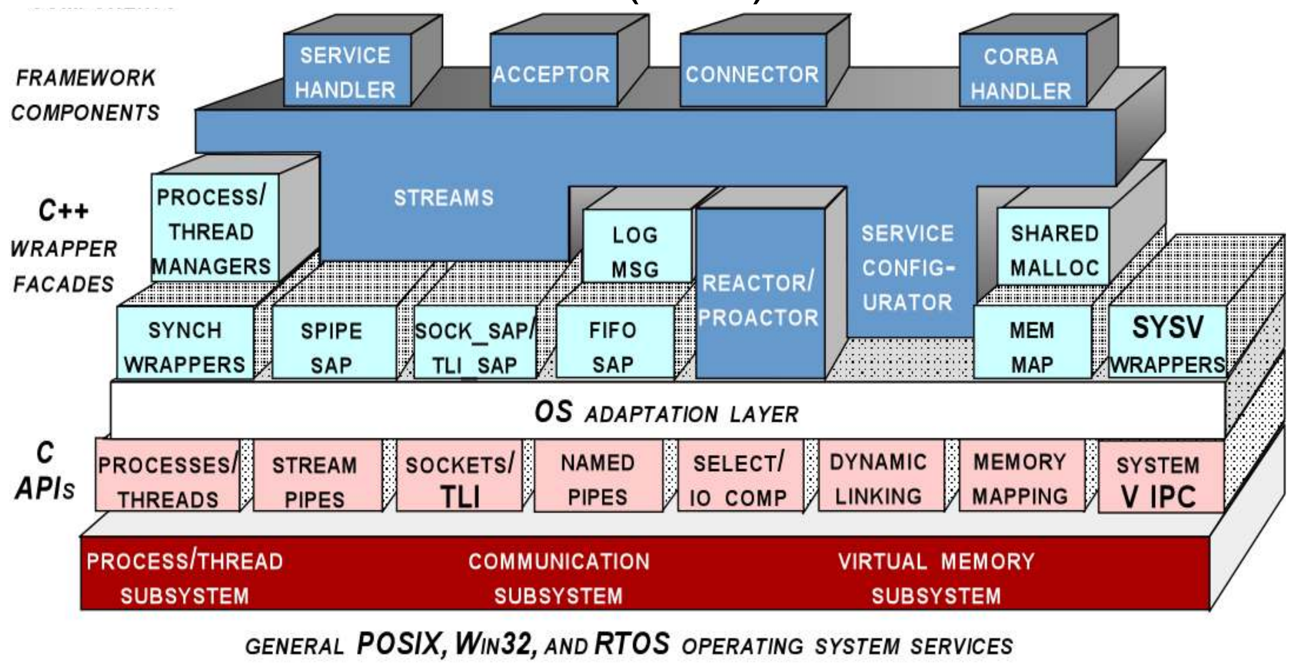
- **Host infrastructure middleware** encapsulates & enhances native OS mechanisms to create reusable network programming components
  - These components abstract away many tedious & error-prone aspects of low-level OS APIs



- **Examples**
  - Java Virtual Machine (JVM), Common Language Runtime (CLR), ADAPTIVE Communication Environment (ACE)



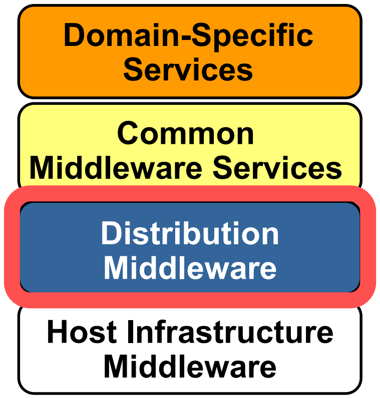
www.rtj.org

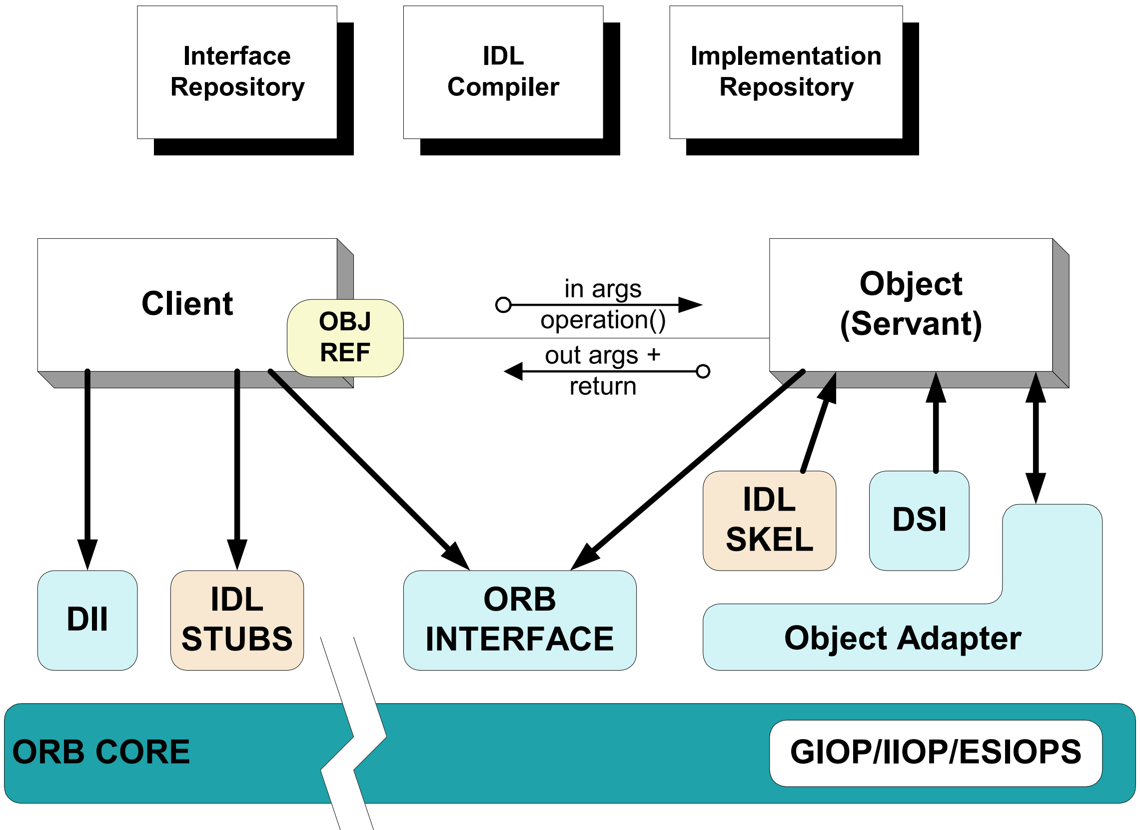


www.cs.wustl.edu/~schmidt/ACE.html

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)

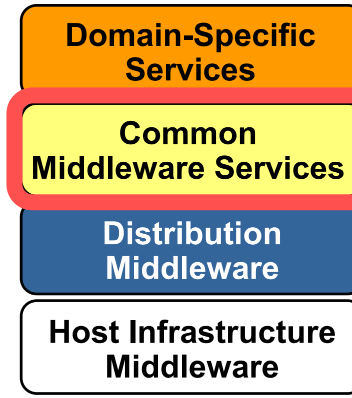


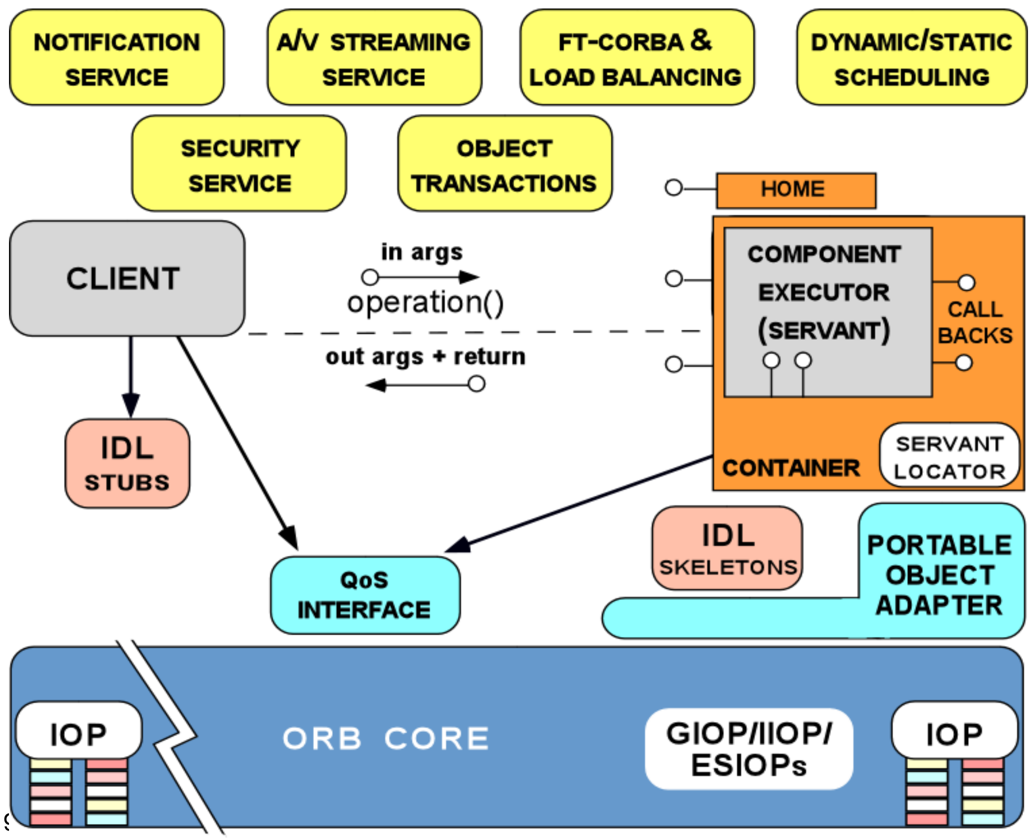


- Distribution middleware avoids hard-coding client & server application dependencies on object location, language, OS, protocols, & hardware

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

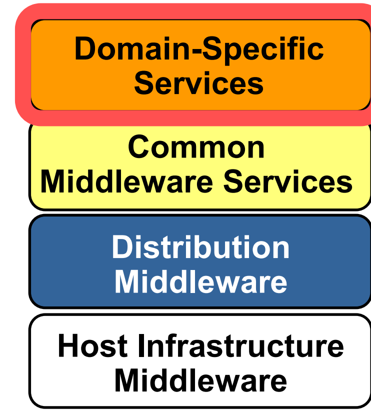




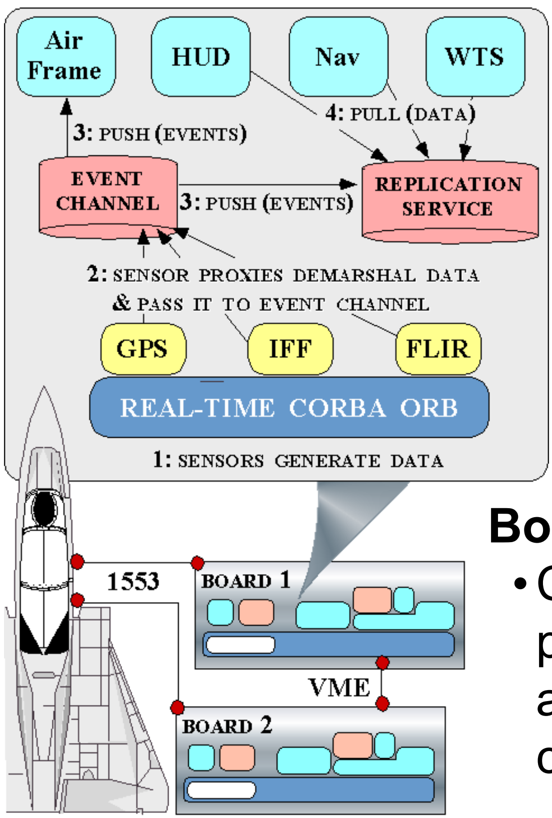
- Common middleware services support many recurring distributed system capabilities, e.g.,
  - Transactional behavior
  - Authentication & authorization,
  - Database connection pooling & concurrency control
  - Active replication
  - Dynamic resource management

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**

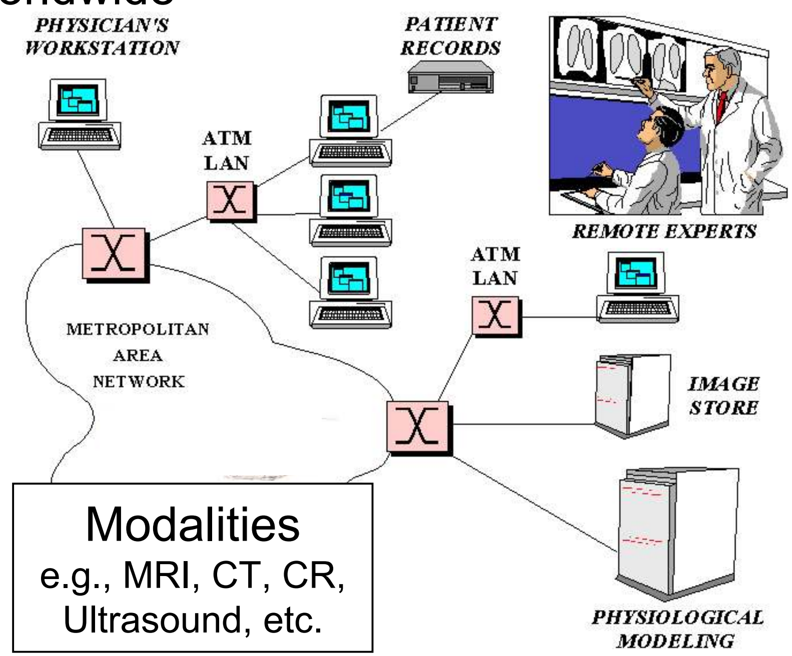


Siemens MED Syngo

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

Boeing Bold Stroke

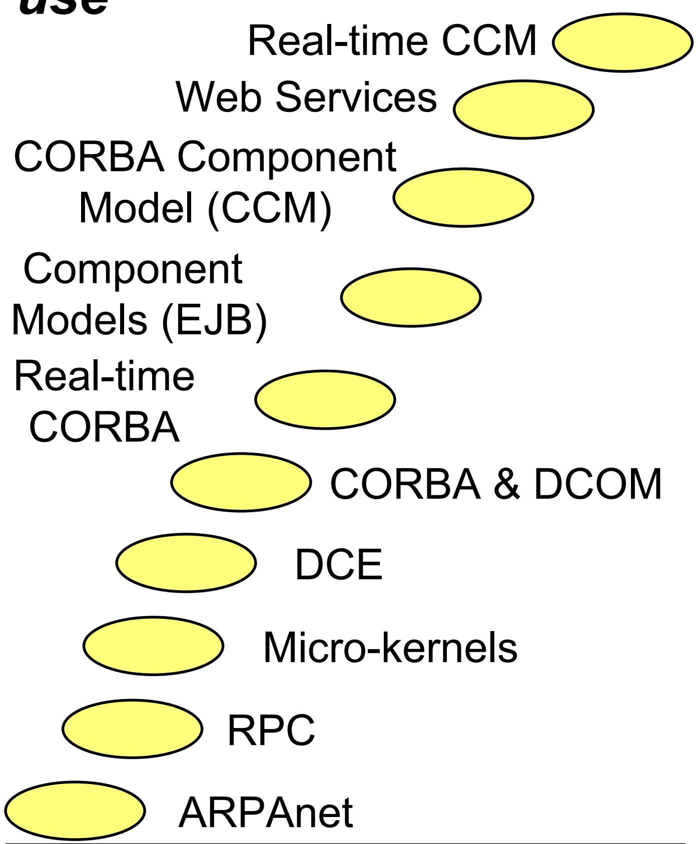
- Common software platform for Boeing avionics mission computing systems



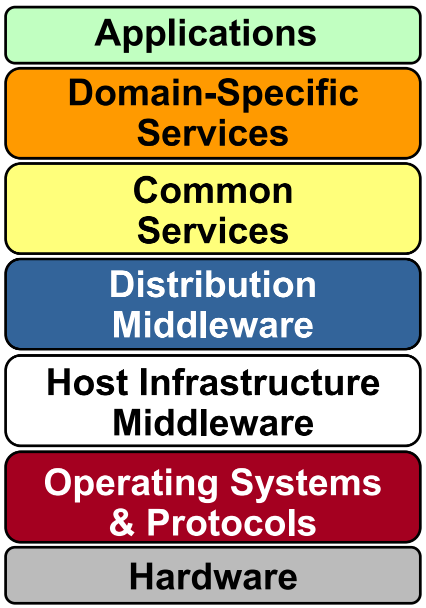
Why We are Succeeding

The past decade has yielded significant progress in QoS-enabled middleware, stemming in large part from the following trends:

- ***Years of iteration, refinement, & successful use***

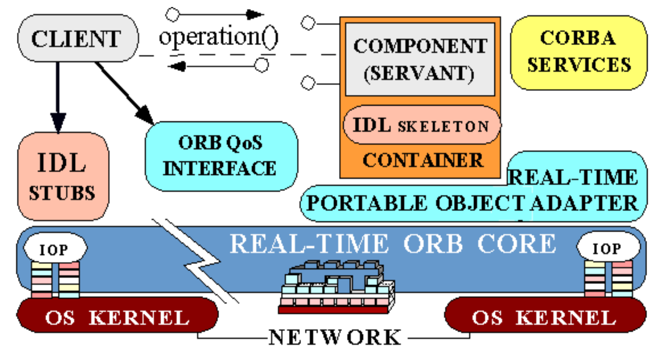


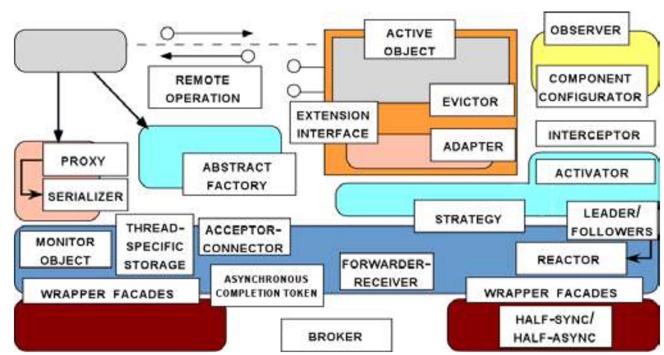
- ***The maturation of middleware standards***



- .NET, J2EE, CCM
- Real-time CORBA
- Real-time Java
- SOAP & Web Services

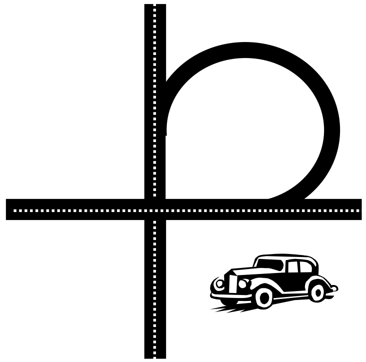
- ***The maturation of component middleware frameworks & patterns***





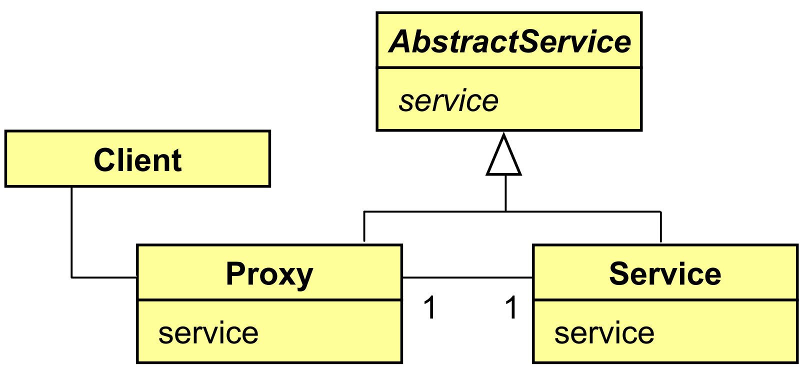
Overview of Patterns

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



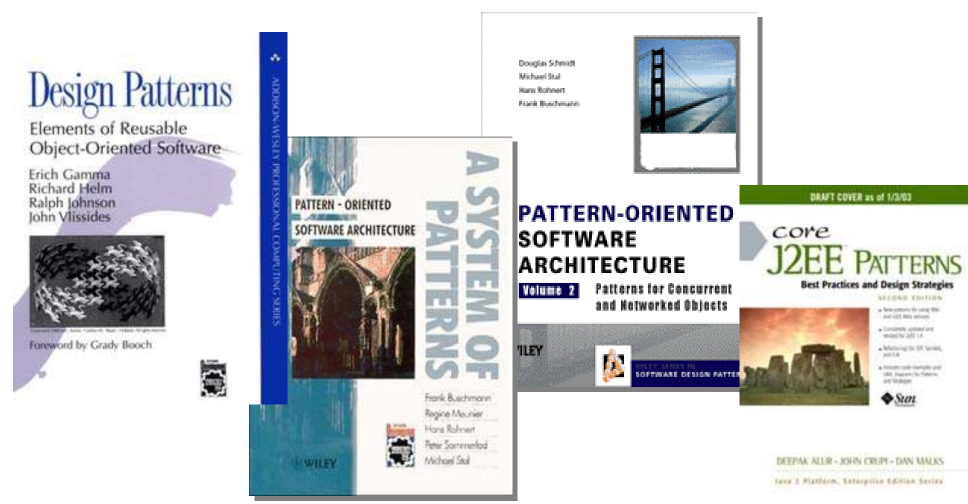
- Help resolve key software design forces
  - ***Flexibility***
  - ***Extensibility***
  - ***Dependability***
  - ***Predictability***
  - ***Scalability***
  - ***Efficiency***

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



The Proxy Pattern

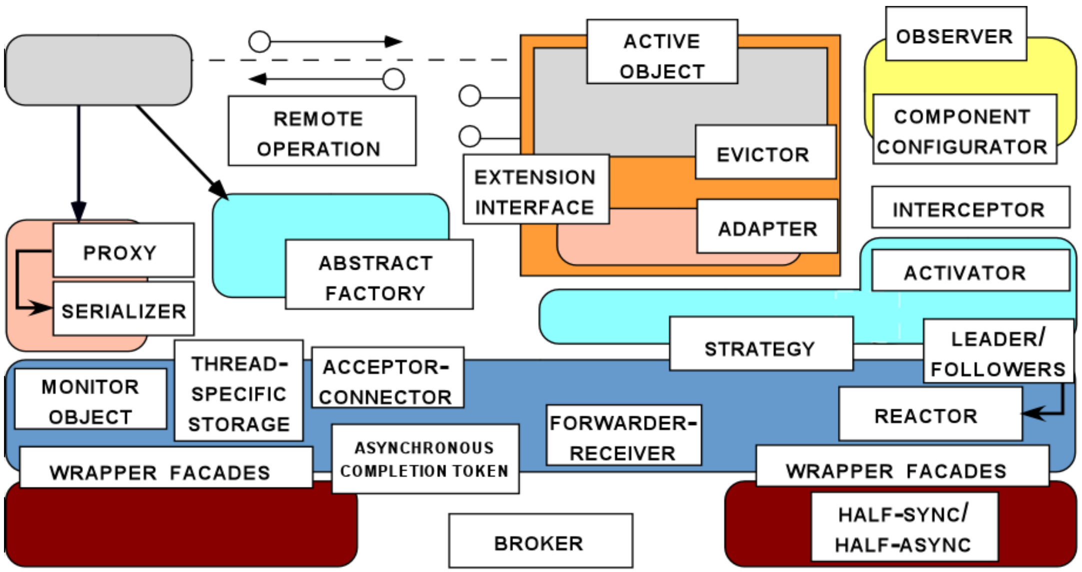
- Generally codify expert knowledge of design strategies, constraints & "best practices"



Overview of Pattern Languages

Motivation

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



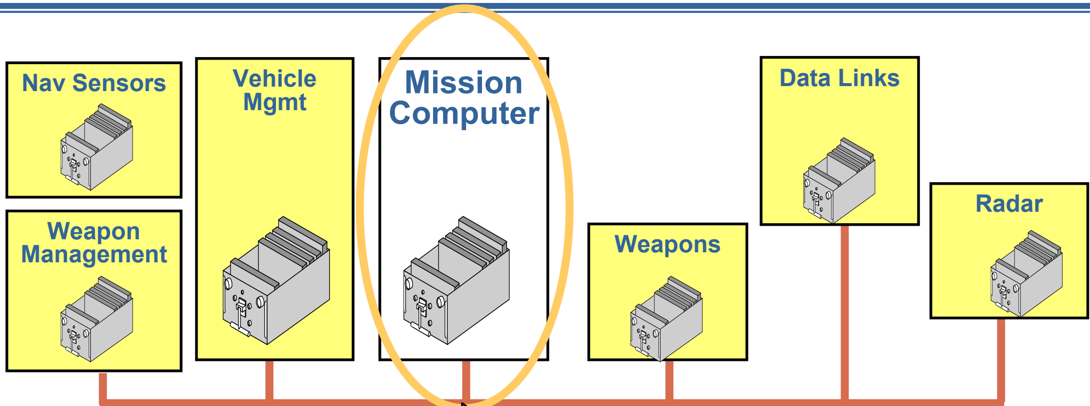
Benefits of Pattern Languages

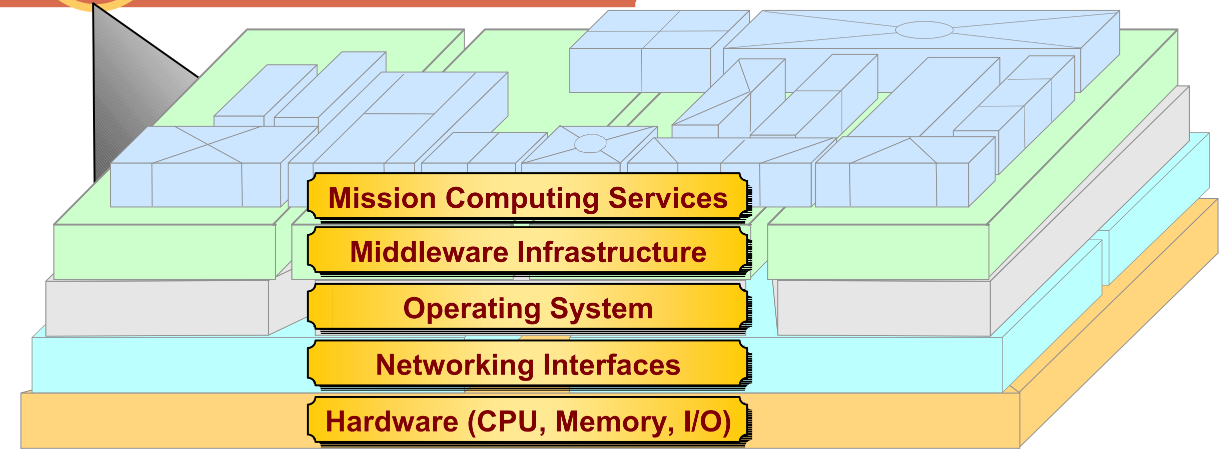
- Define a *vocabulary* for talking about software development problems
- Provide a *process* for the orderly resolution of these problems
- Help to generate & reuse software *architectures*

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

Example: Boeing Bold Stroke

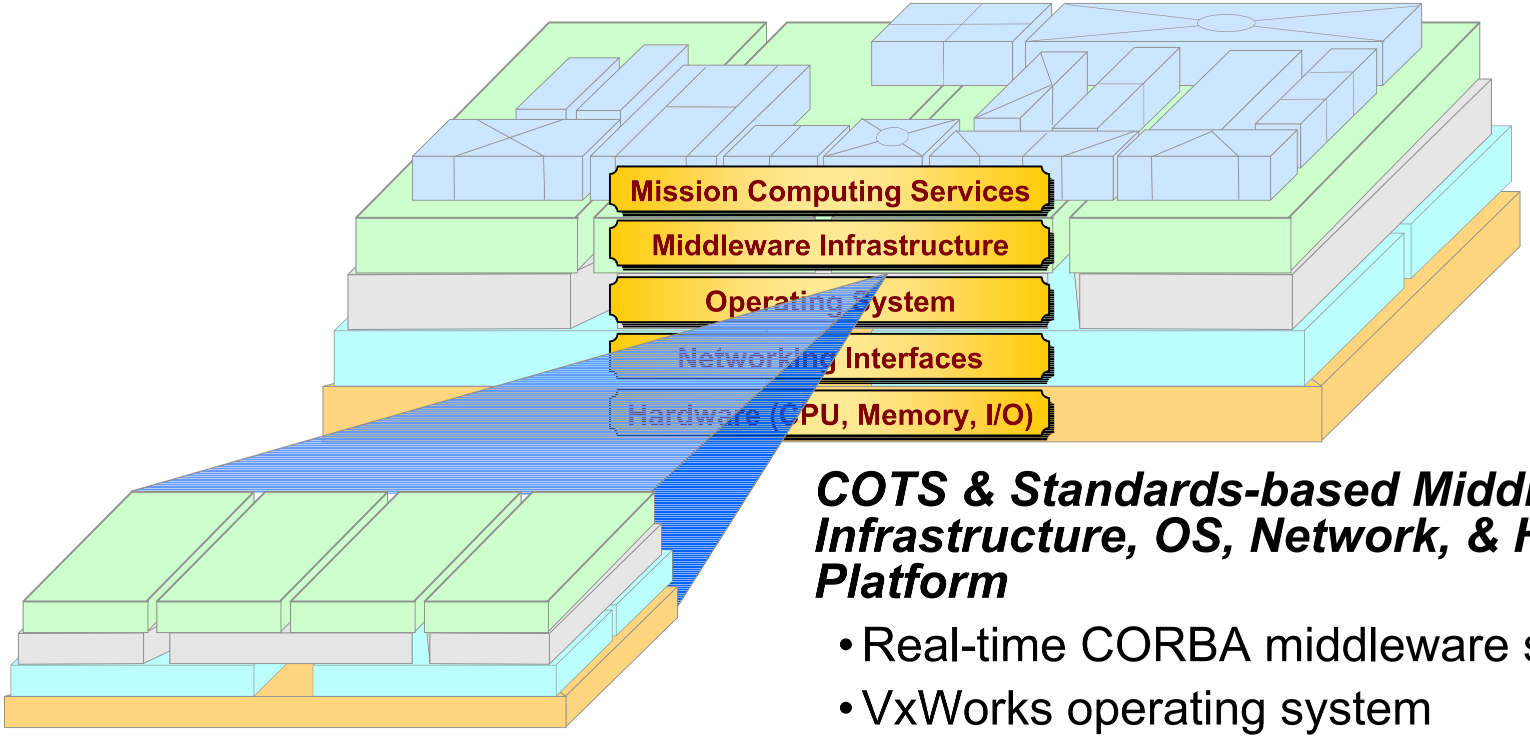




Bold Stroke Architecture

- 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

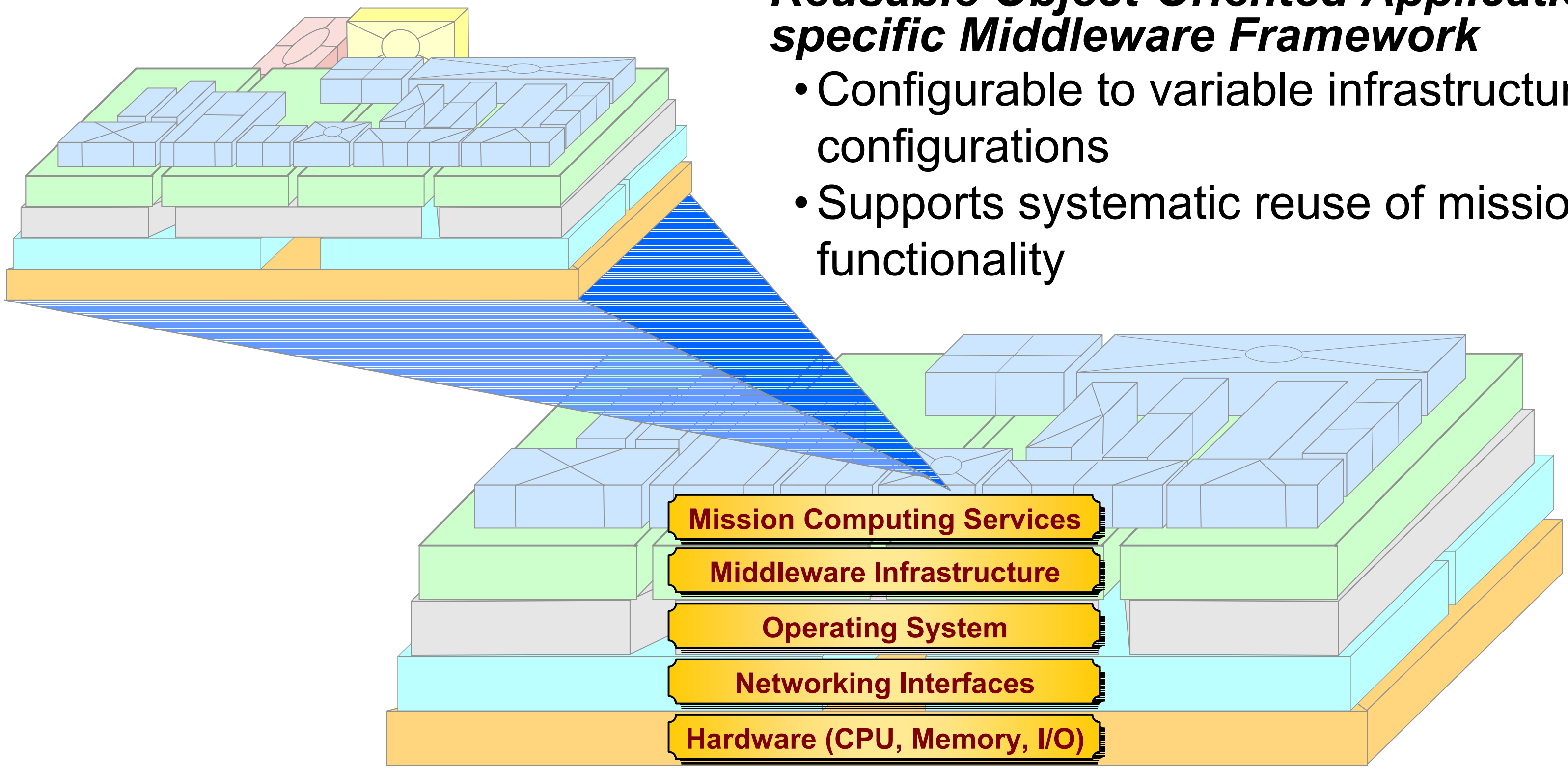
Example: Boeing Bold Stroke



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

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

Example: Boeing Bold Stroke



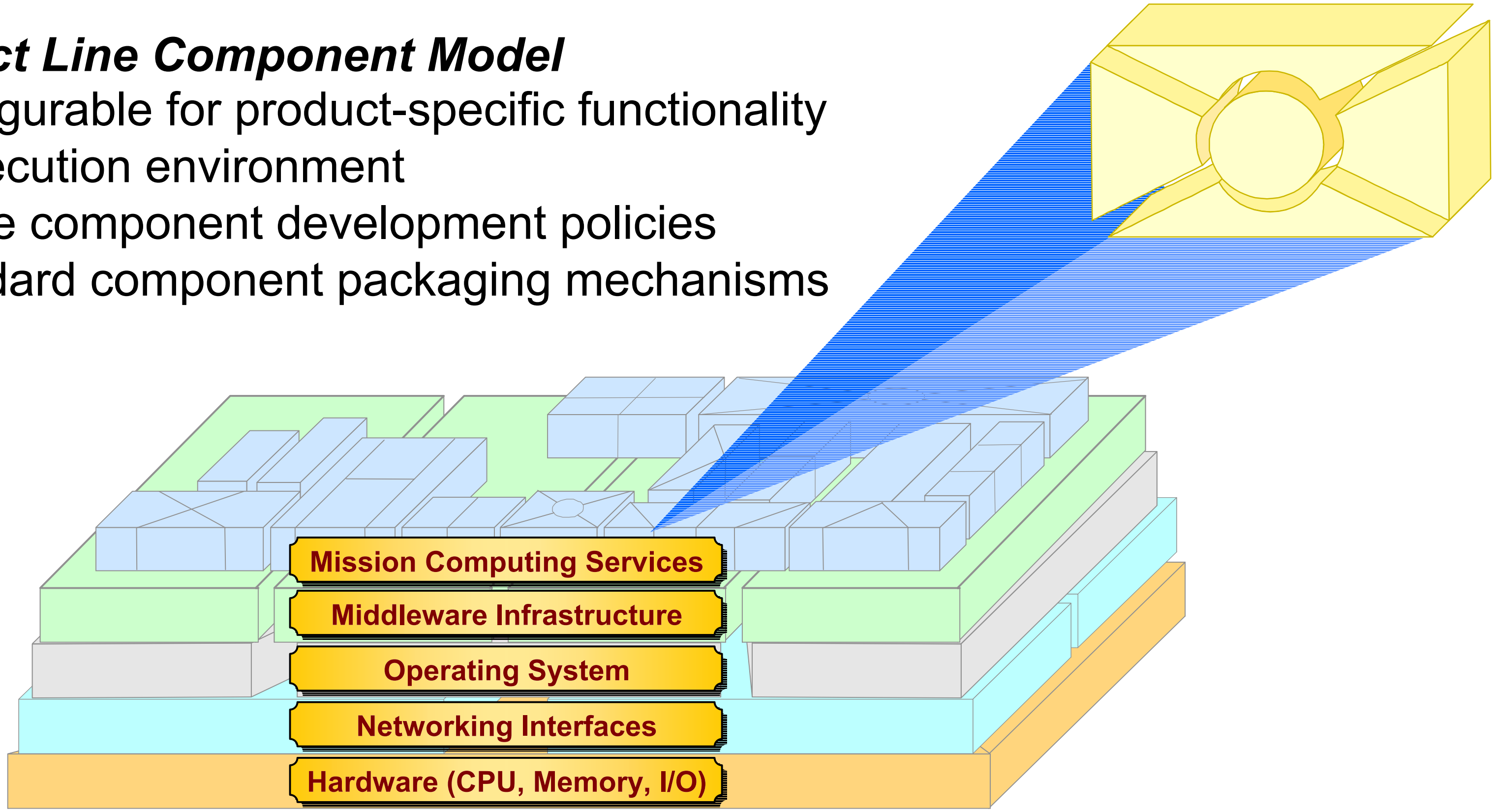
Reusable Object-Oriented Application Domain-specific Middleware Framework

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

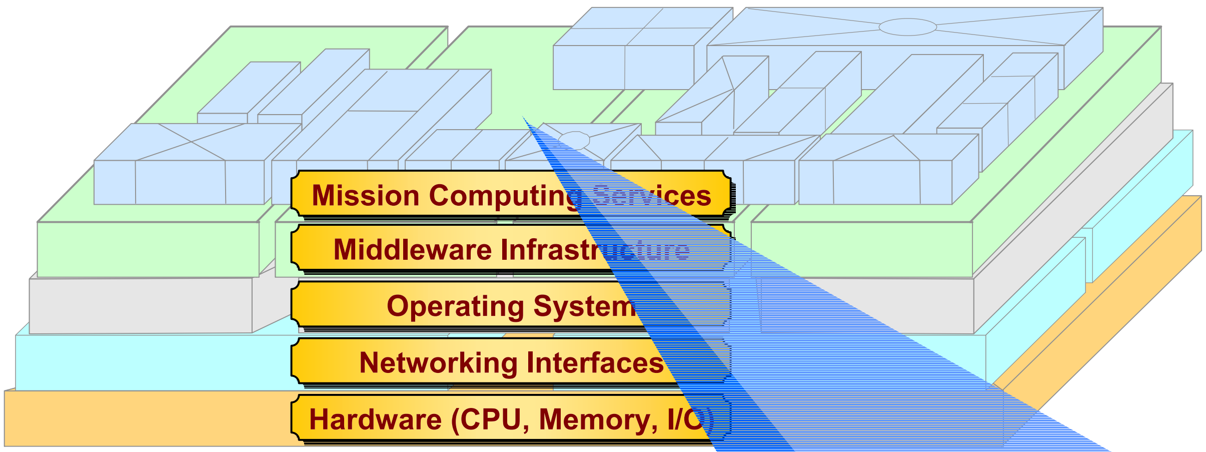
Example: Boeing Bold Stroke

Product Line Component Model

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

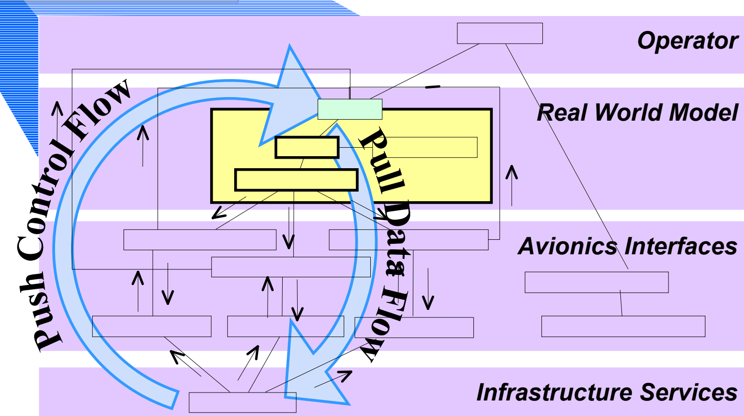


Example: Boeing Bold Stroke



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

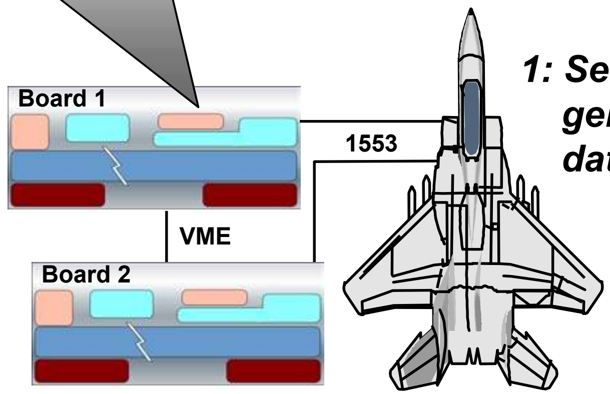
Avionics Mission Computing Functions

- 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

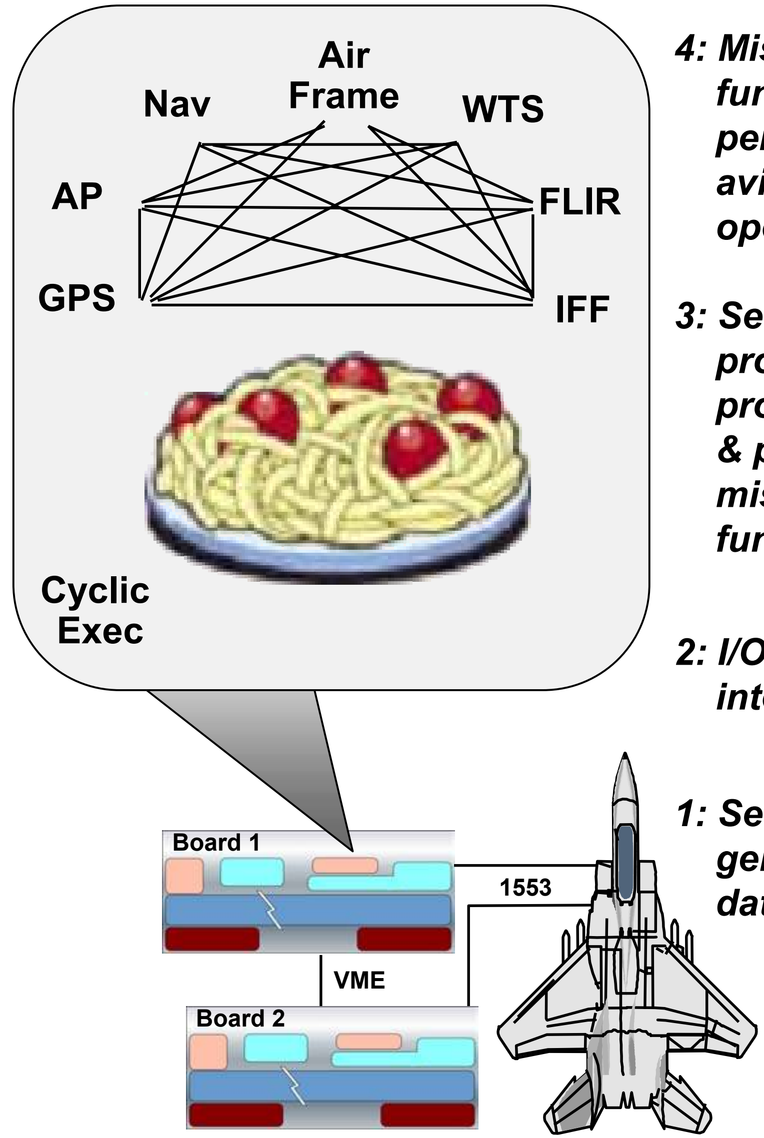
Legacy Avionics Architectures

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***



4: Mission functions perform avionics operations

3: Sensor proxies process data & pass to missions functions

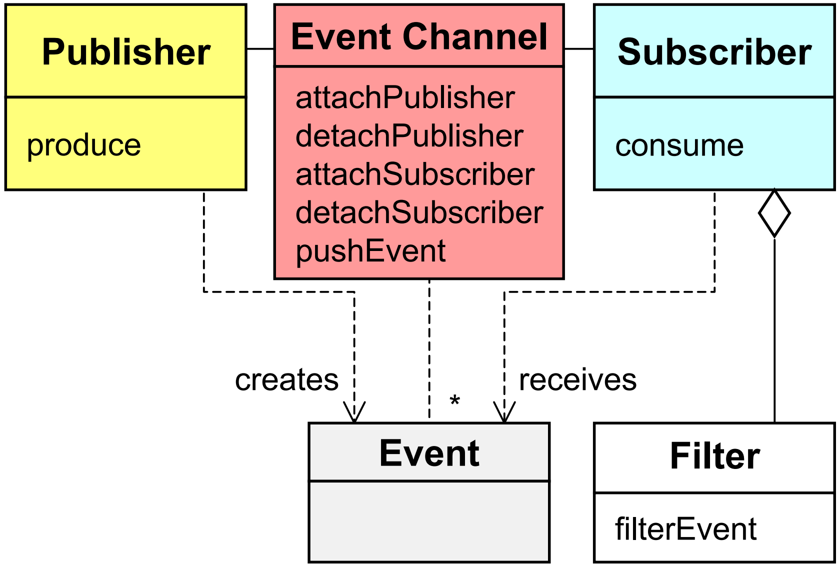
2: I/O via interrupts

1: Sensors generate data

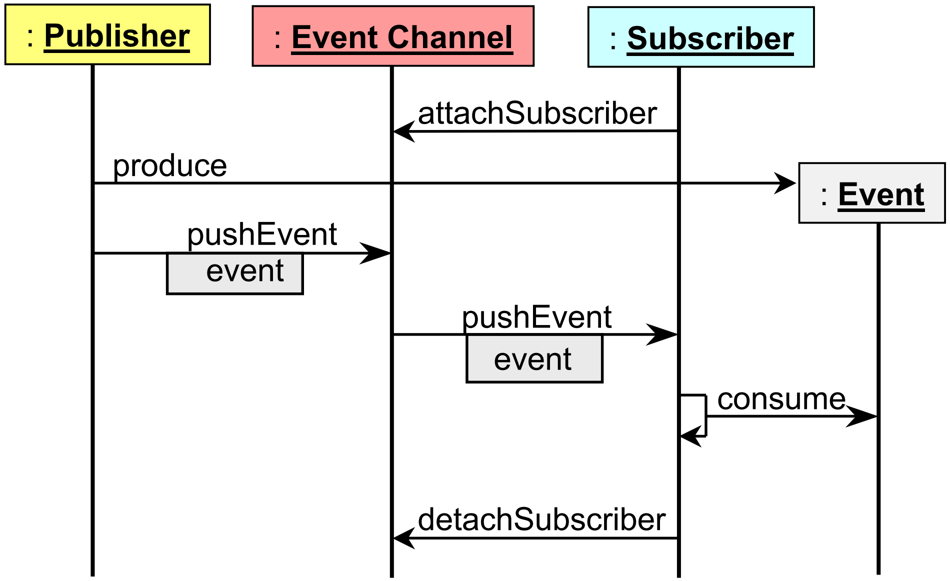
Decoupling Avionics Components

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



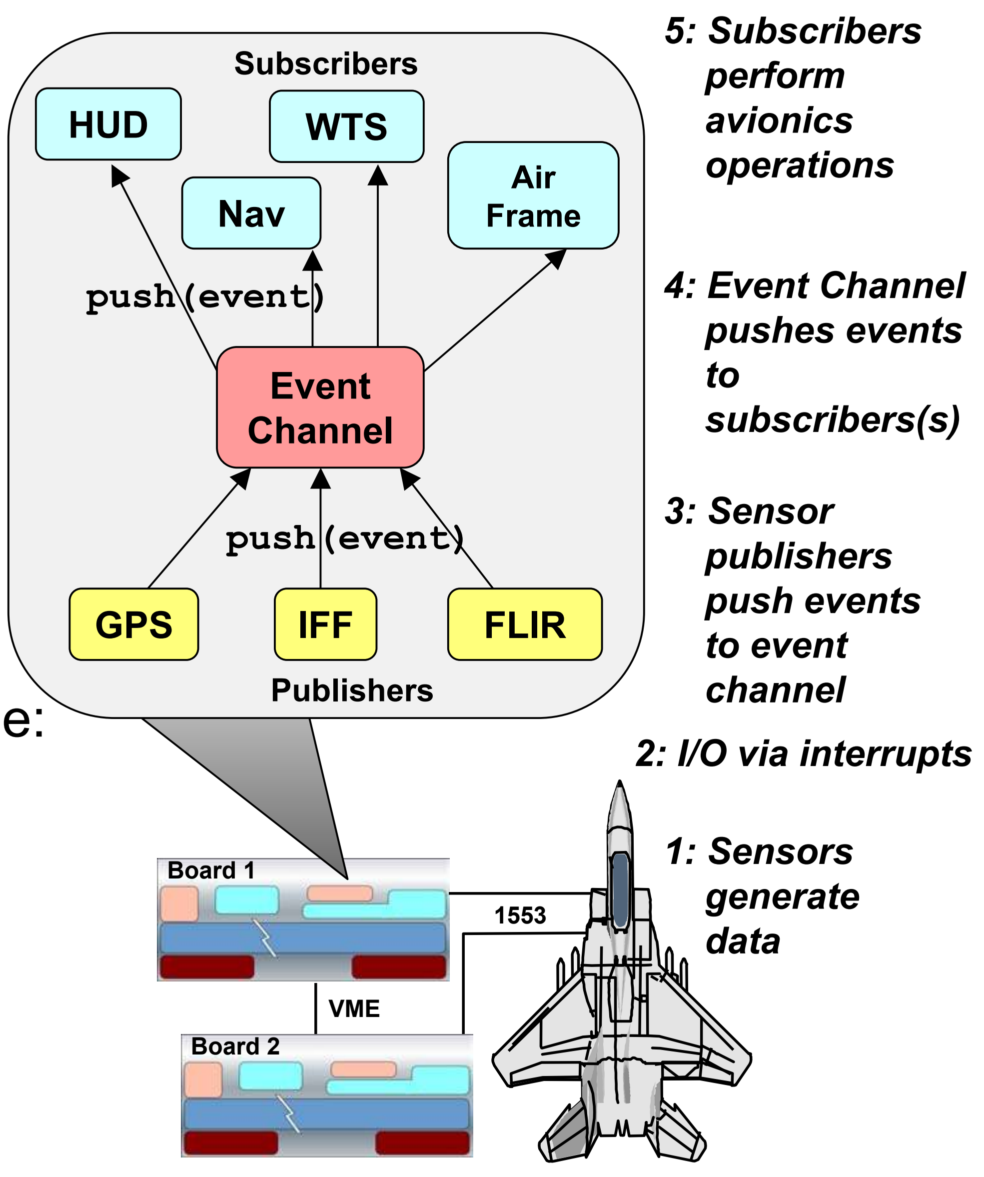
Applying the Publisher-Subscriber Pattern to Bold Stroke

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:

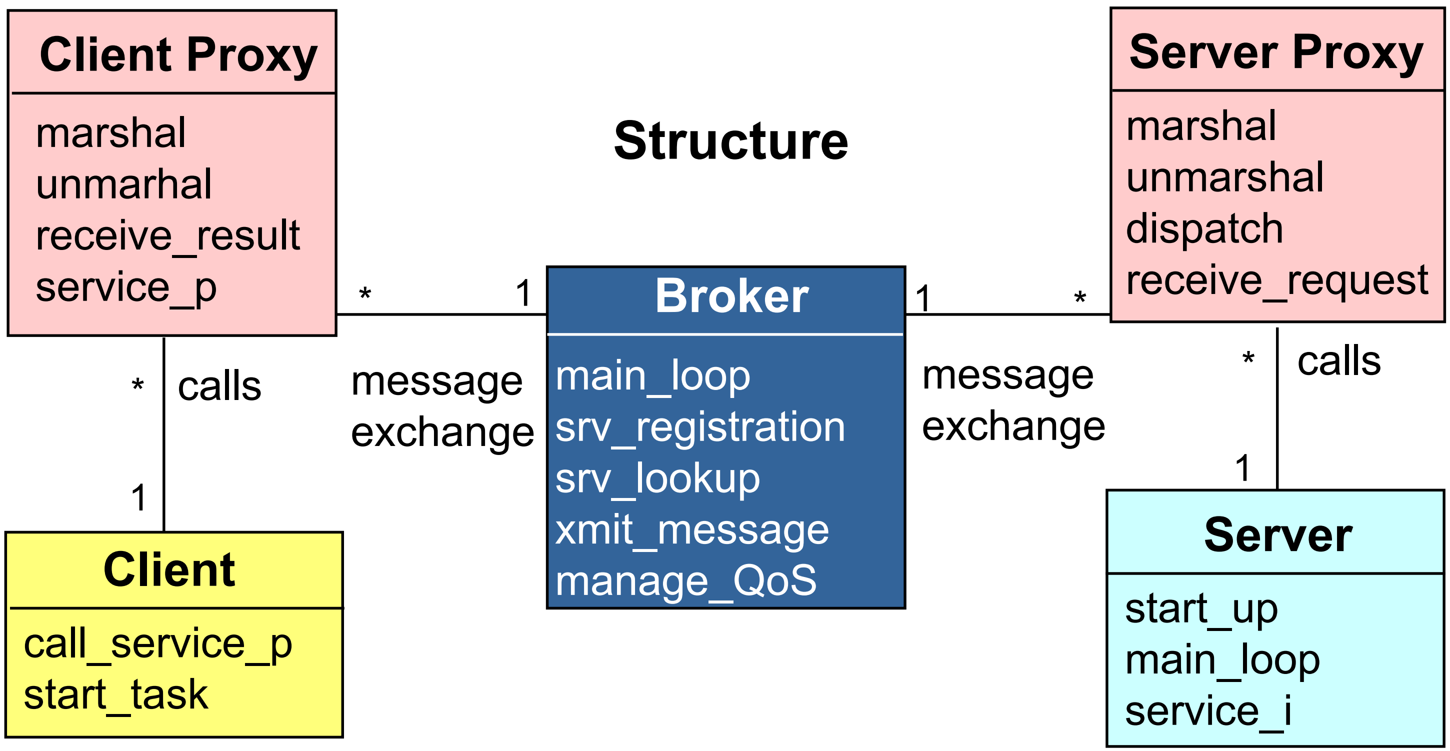
- ***Event notification model***
  - Push control vs. pull data interactions
- ***Scheduling & synchronization strategies***
  - e.g., priority-based dispatching & preemption
- ***Event dependency management***
  - e.g.,filtering & correlation mechanisms



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

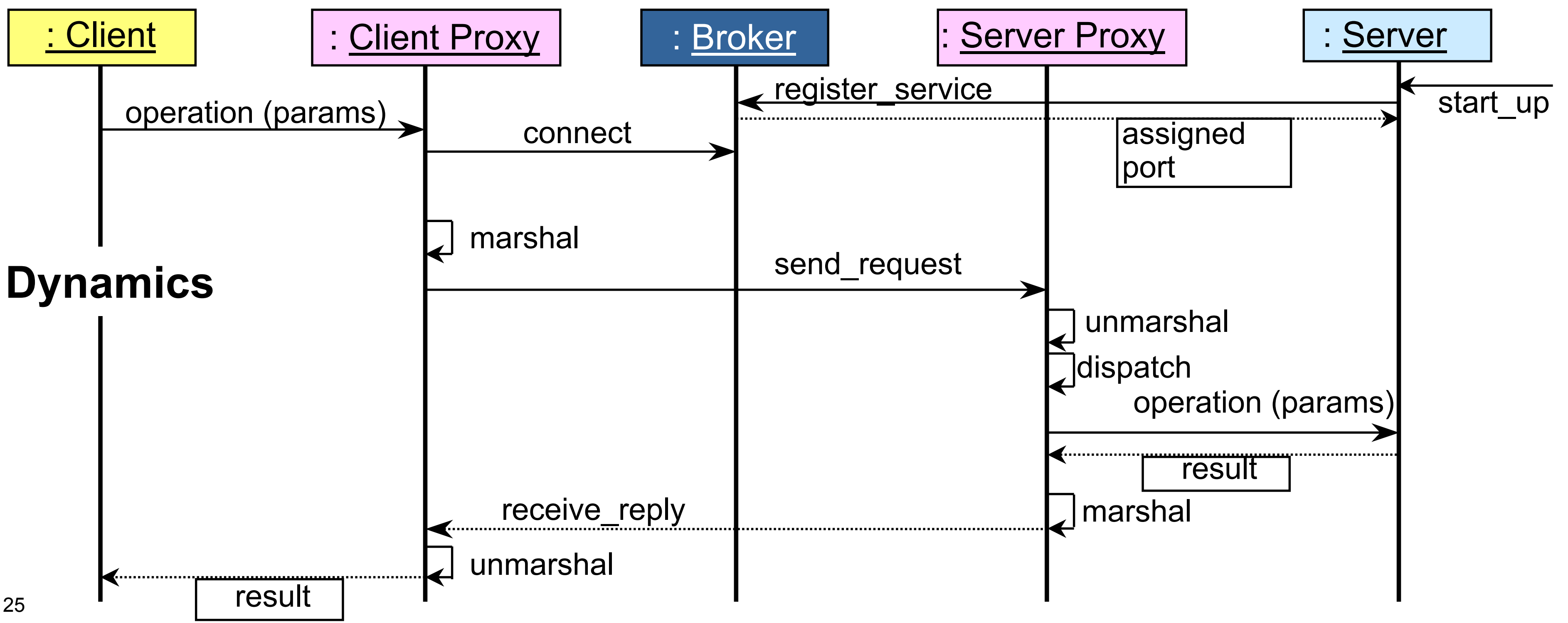


Ensuring Platform-neutral & Network-transparent Communication

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



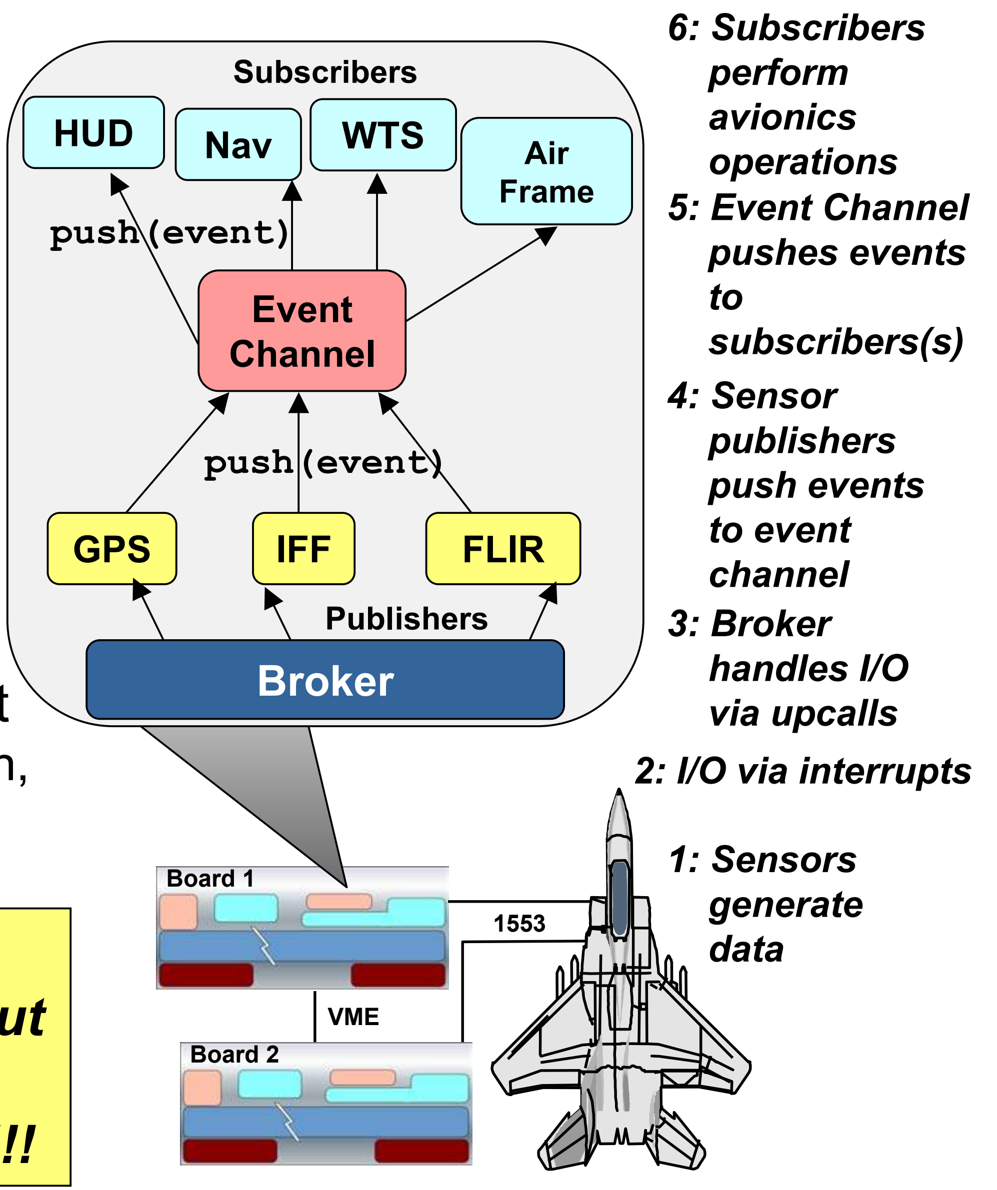
Applying the Broker Pattern to Bold Stroke

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.



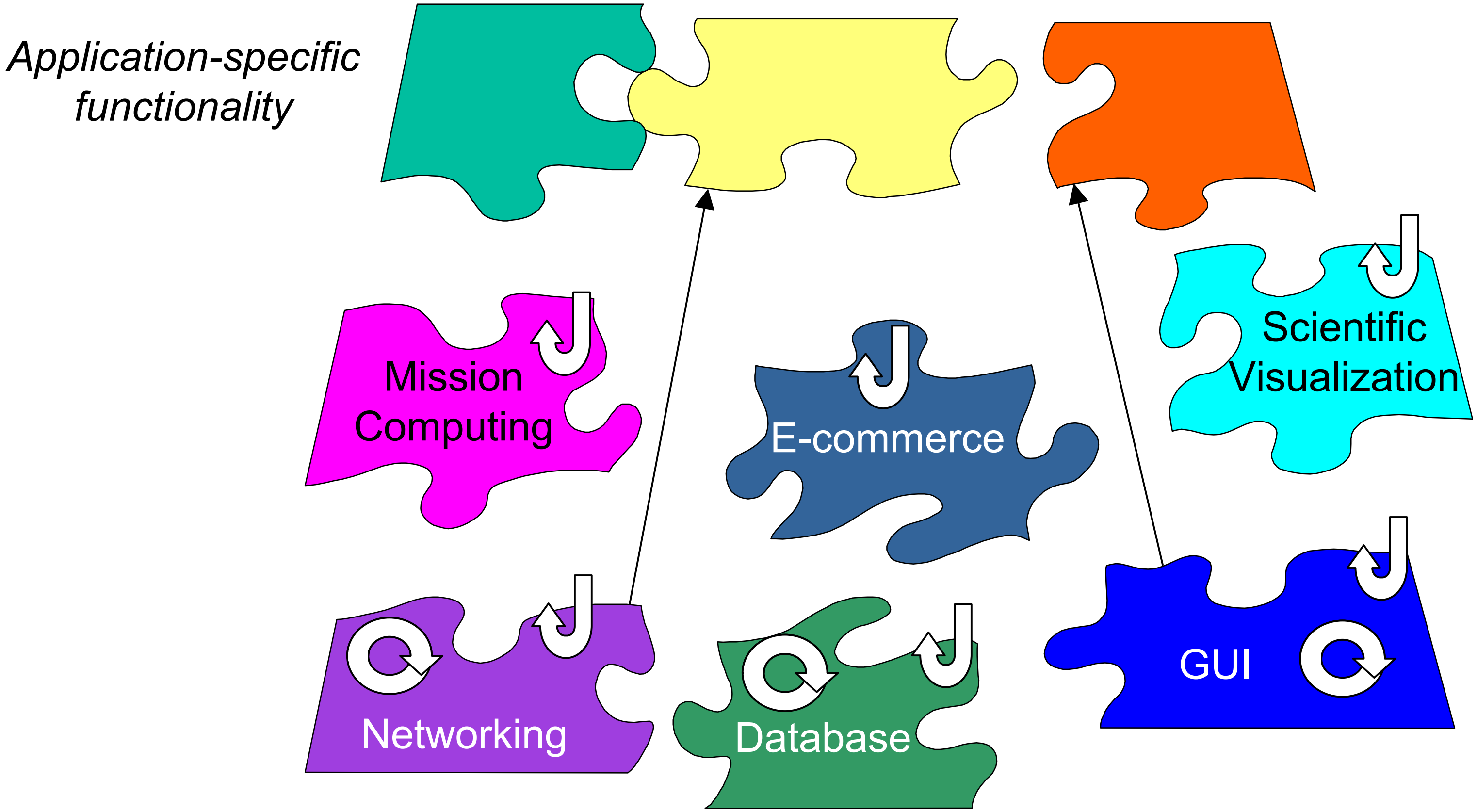
Caveat

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

Overview of Frameworks

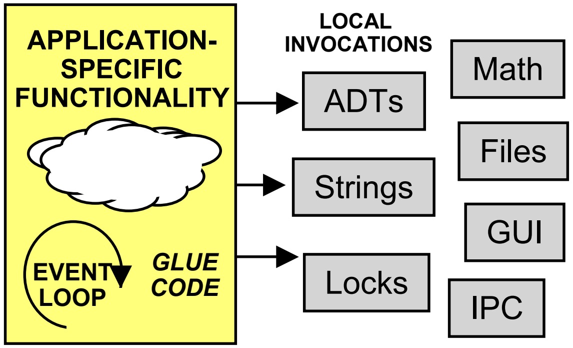
Framework Characteristics

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



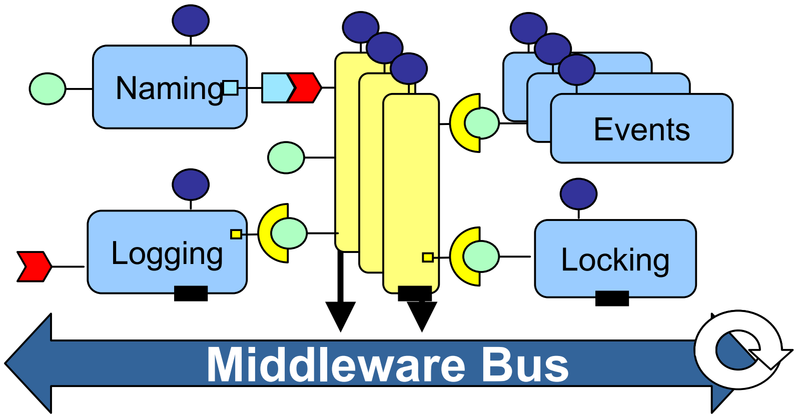
Comparing Class Libraries, Frameworks, & Components

Class Library Architecture



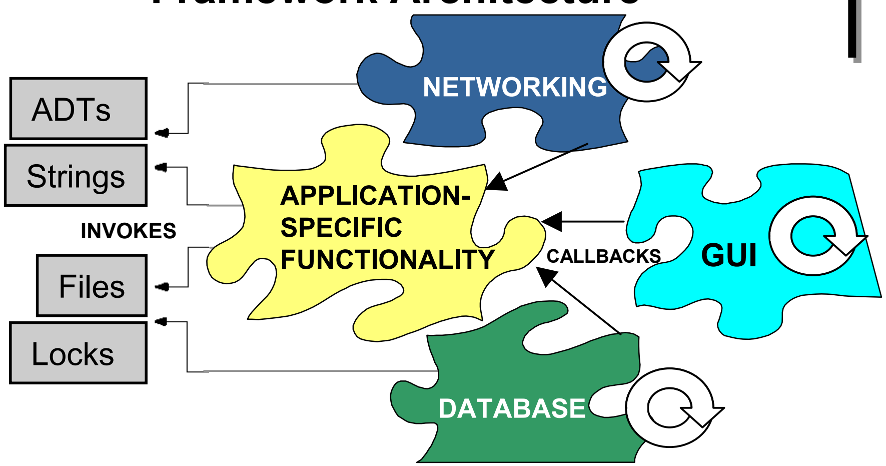
A ***class*** is a unit of abstraction & implementation in an OO programming language

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

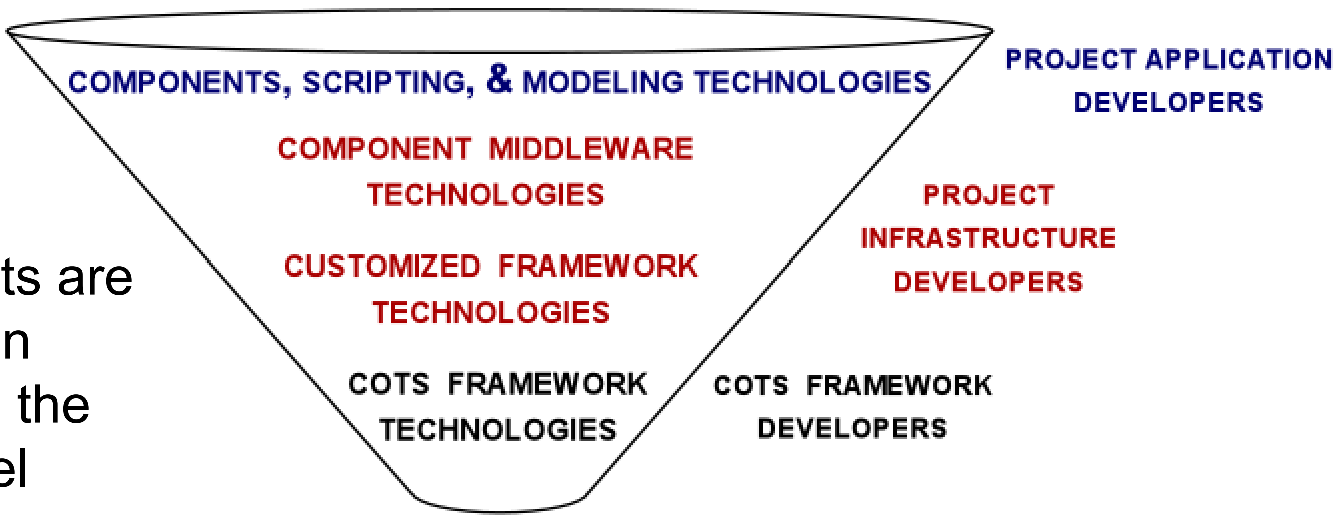
Class Libraries	Frameworks	Components
Micro-level	Meso-level	Macro-level
Stand-alone language entities	"Semi-complete" applications	Stand-alone composition entities
Domain-independent	Domain-specific	**Domain-specific or Domain-independent**
Borrow caller's thread	Inversion of control	Borrow caller's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

Using Frameworks Effectively

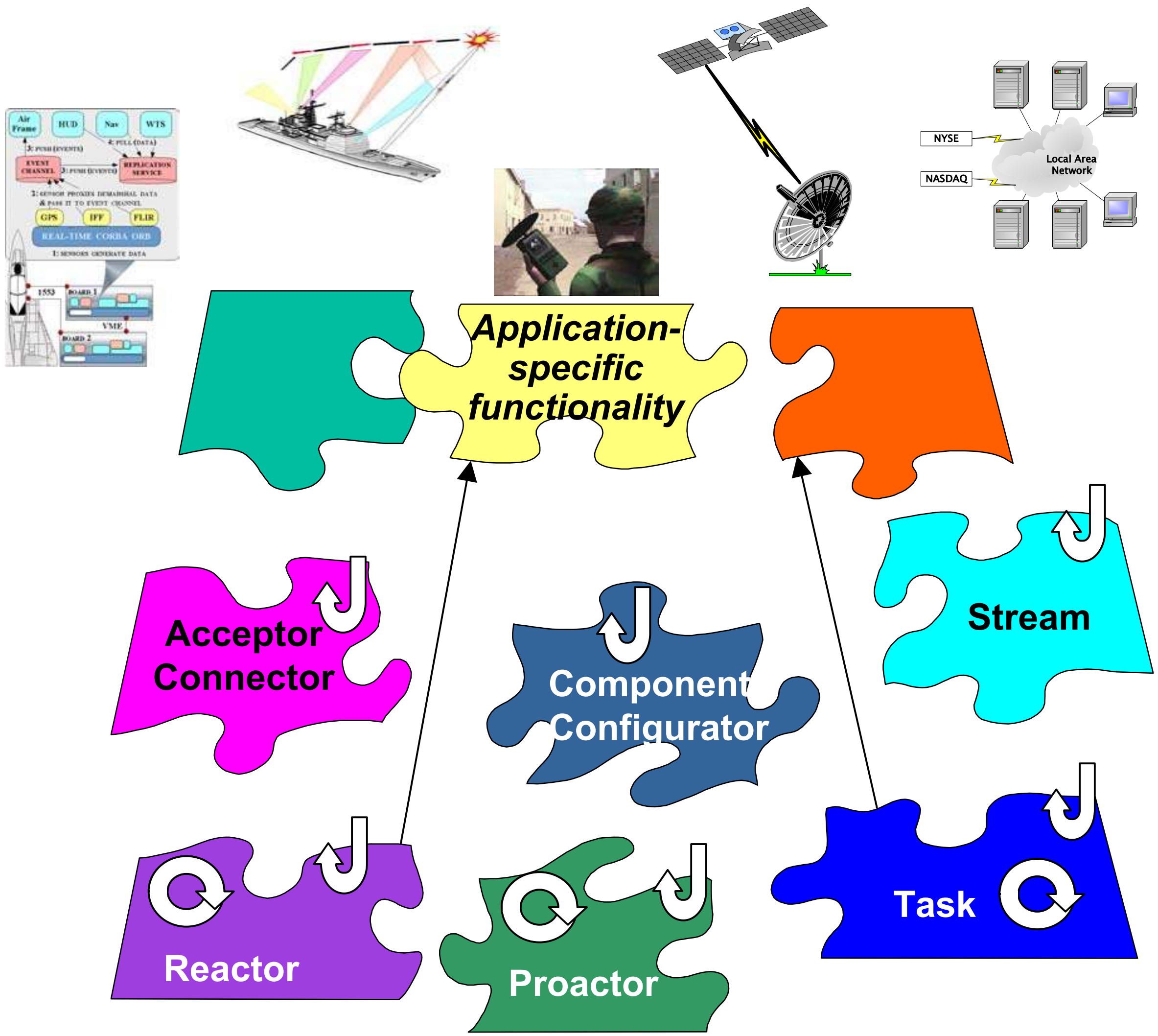
Observations

- 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



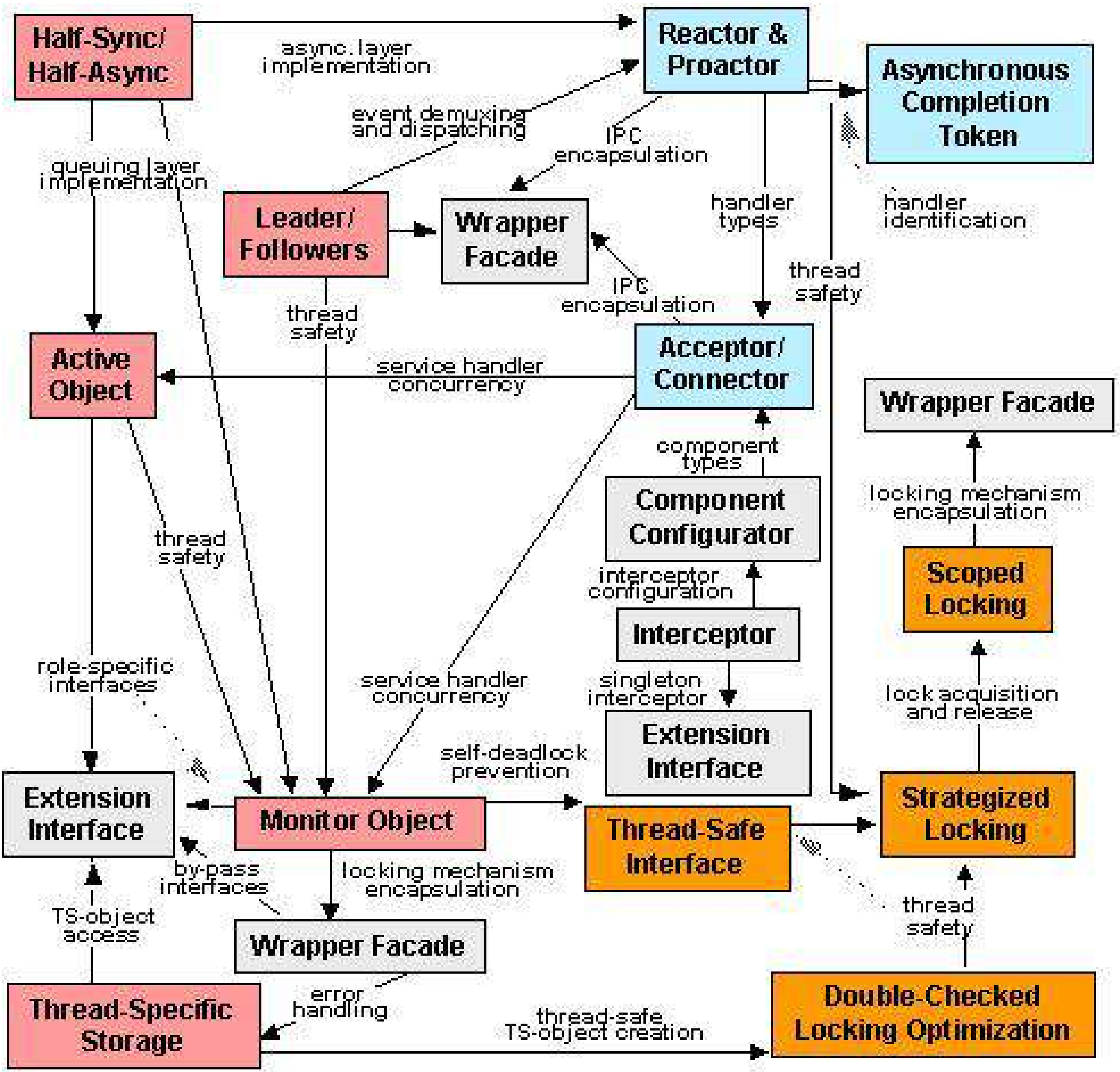
www.cs.wustl.edu/~schmidt/ACE.html

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

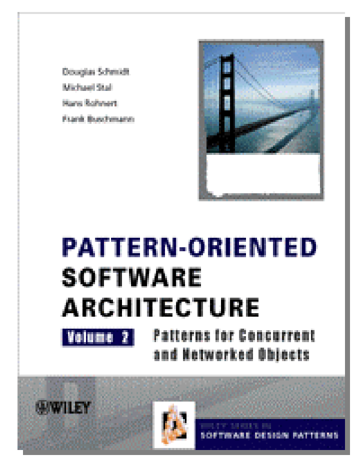
©

The POSA2 Pattern Language

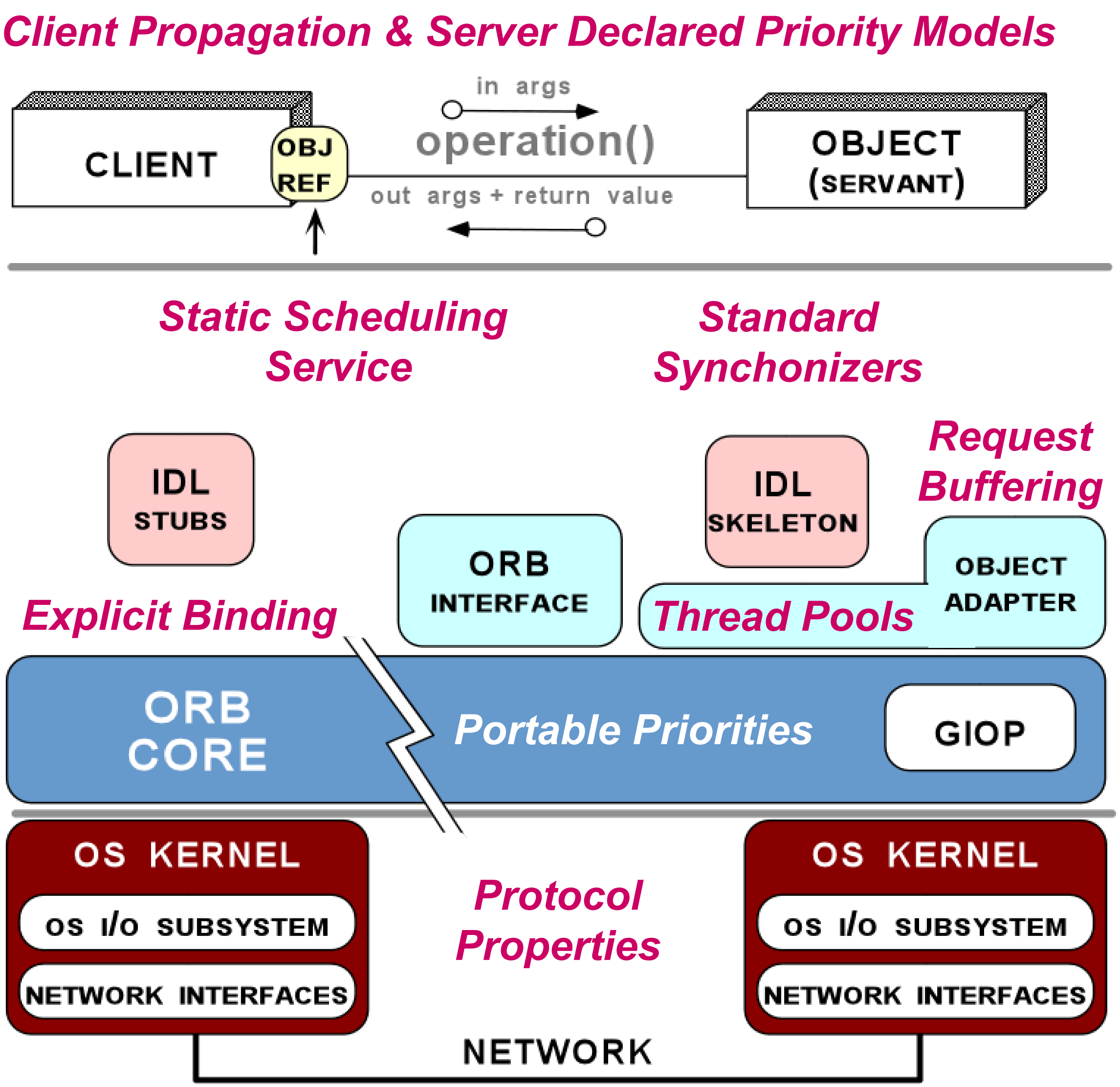


Pattern Benefits

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



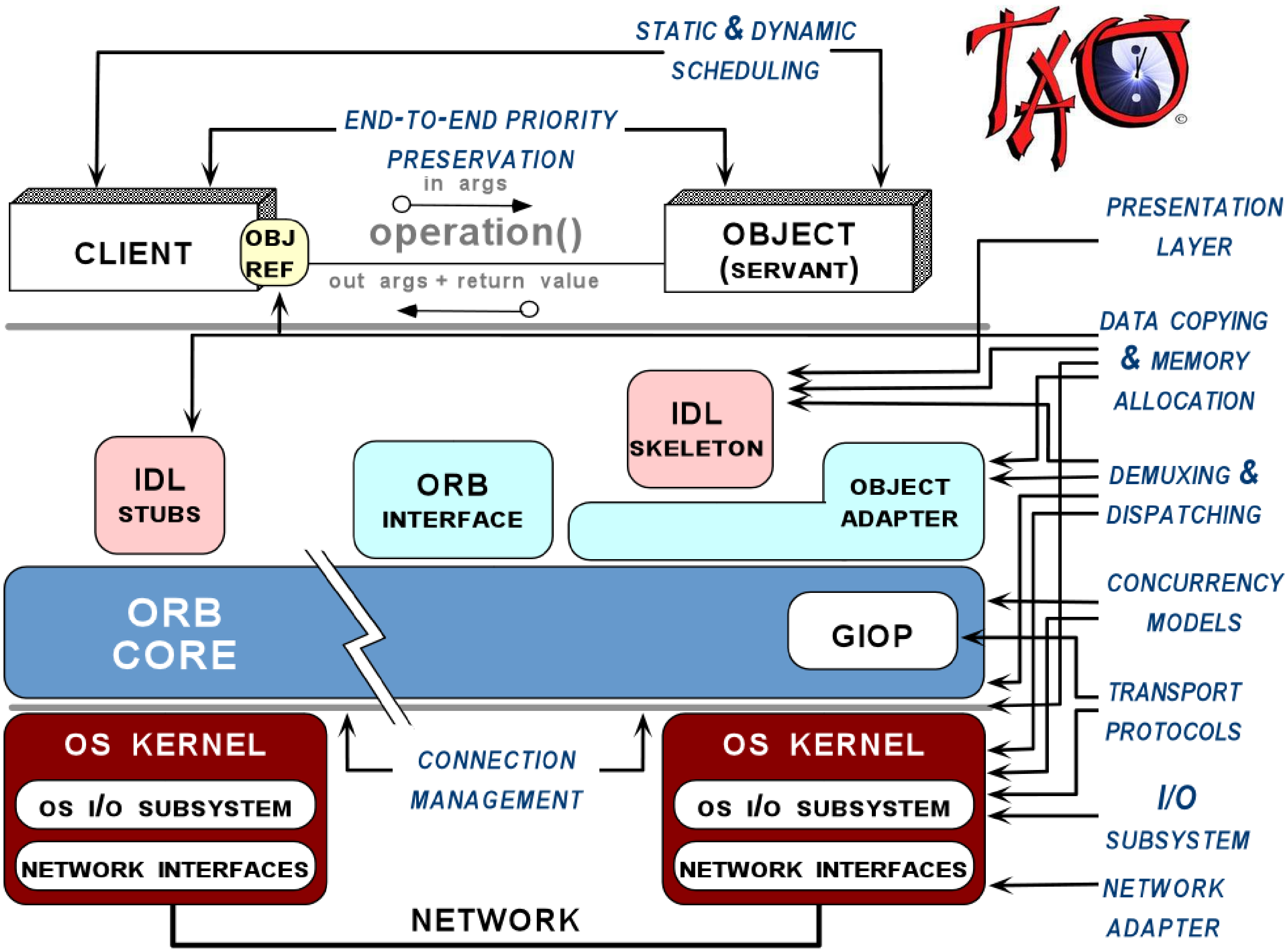
Implementing the Broker Pattern for Bold Stroke Avionics



www.omg.org

- **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

Applying Patterns & Framworks to Middleware: The ACE ORB (TAO)

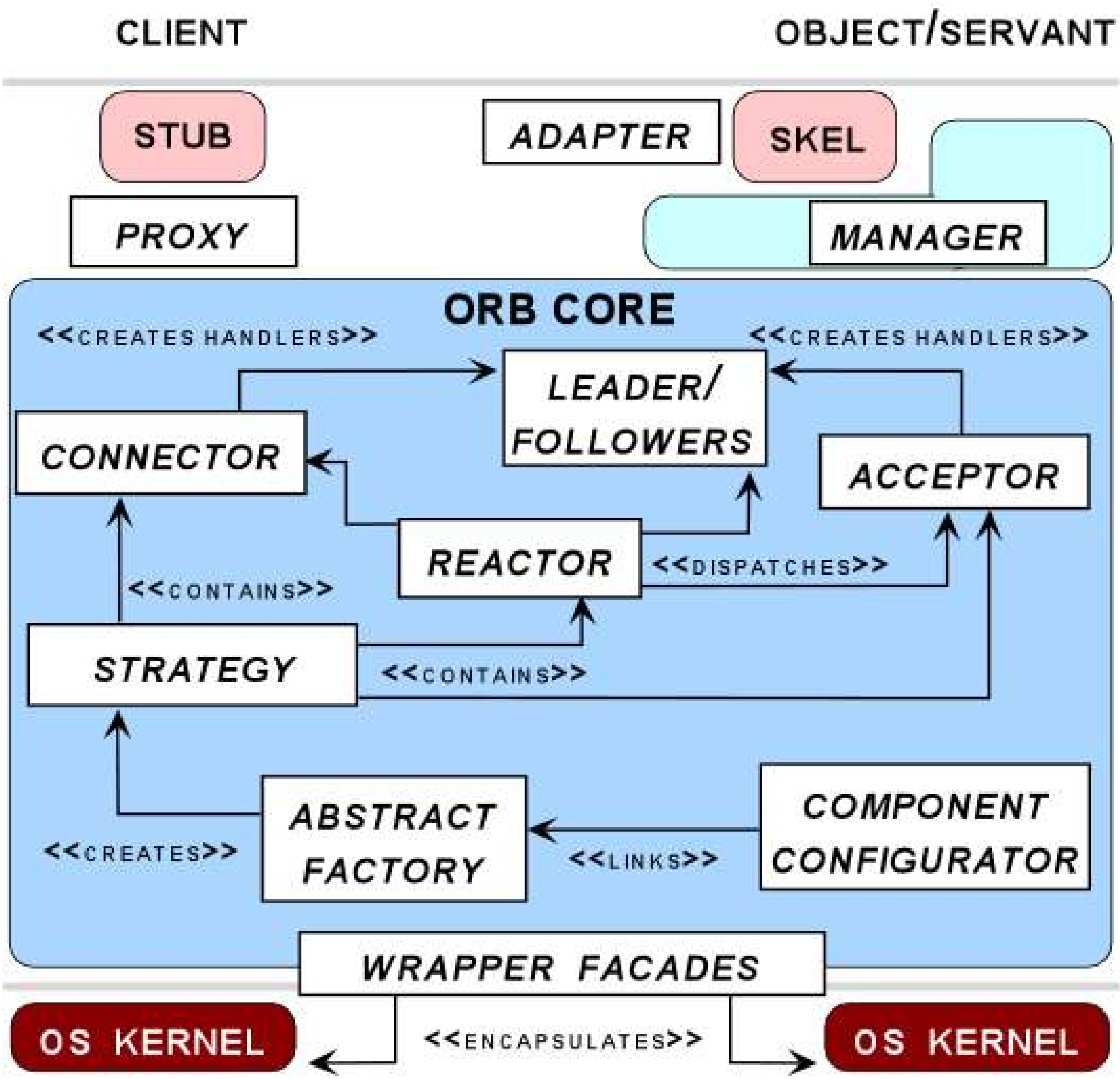


- TAO is an open-source version of Real-time CORBA
- TAO Synopsis
  - > 1,000,000 SLOC
  - 80+ person years of effort
- Pioneered R&D on DRE middleware design, patterns, frameworks, & optimizations
- TAO is basis for many middleware R&D efforts
- Example of good synergy between researchers & practitioners

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

Key Patterns Used in TAO

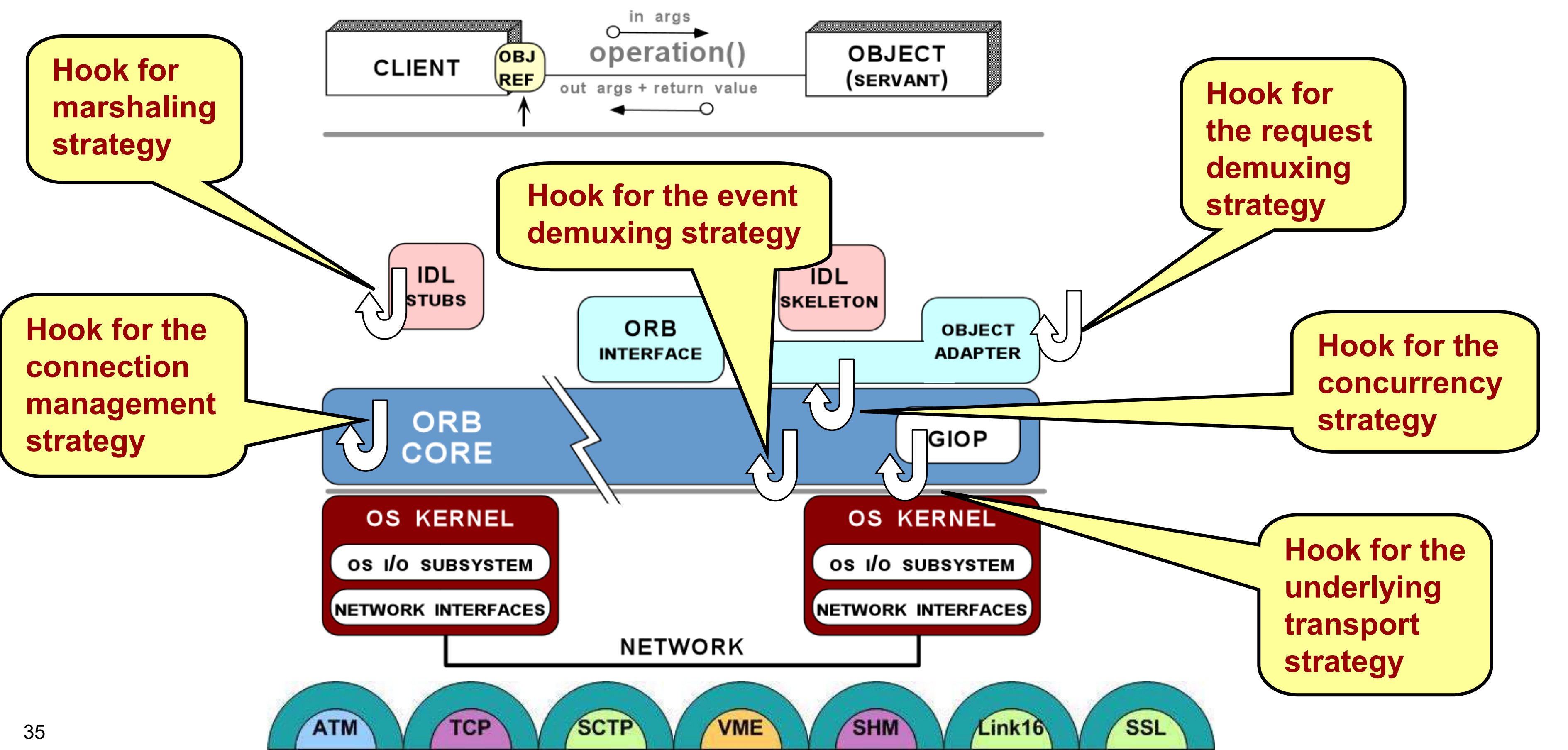
www.cs.wustl.edu/~schmidt/PDF/ORB-patterns.pdf



- ***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 ORB Flexibility w/the Strategy Pattern

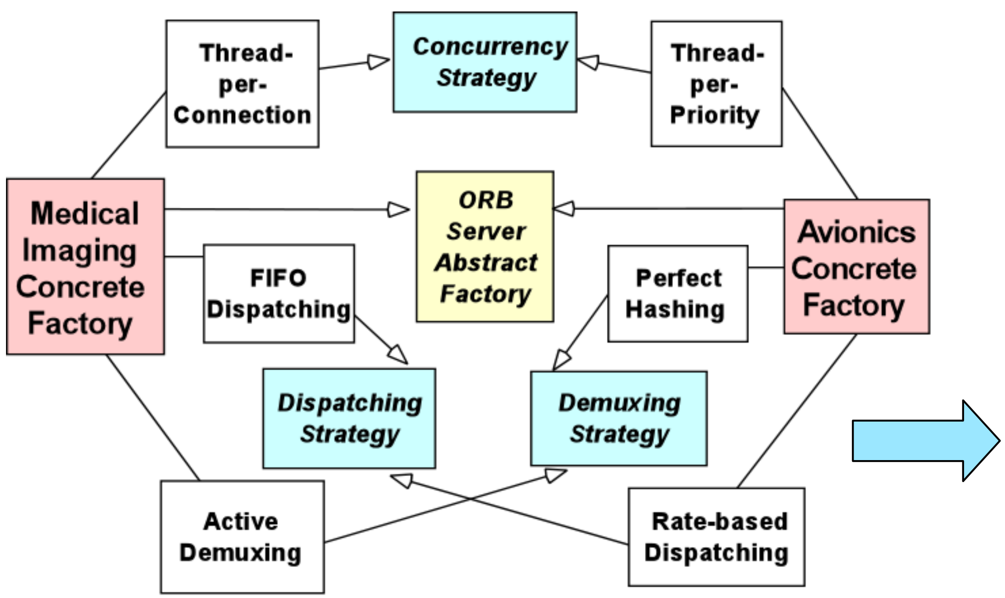
Context	Problem	Solution
• Multi-domain resuable middleware framework	• Flexible ORBs must support multiple event & request demuxing, scheduling, (de)marshaling, connection mgmt, request transfer, & concurrency policies	• Apply the ***Strategy*** pattern to factory out similarity amongst alternative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

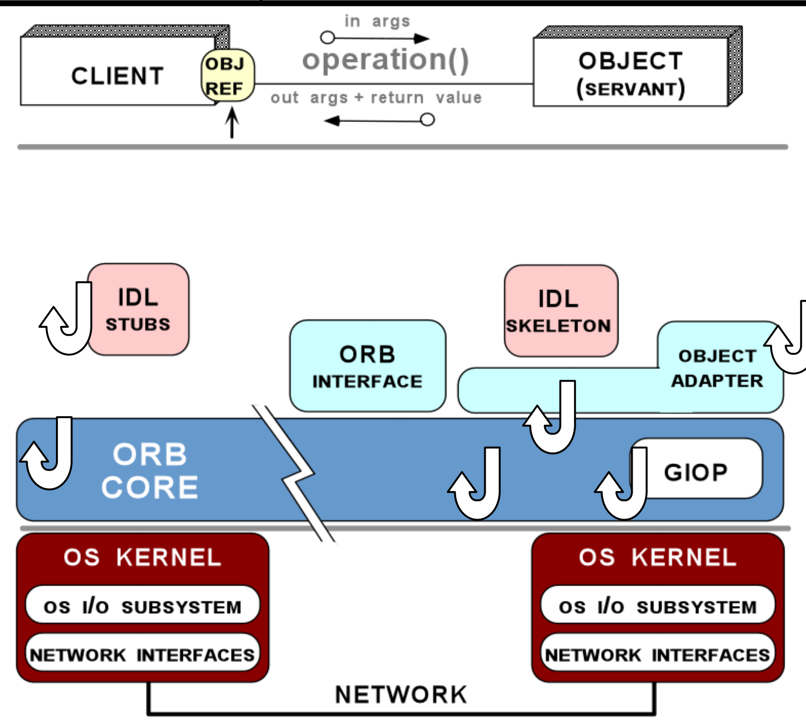


Consolidating Strategies with the Abstract Factory Pattern

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



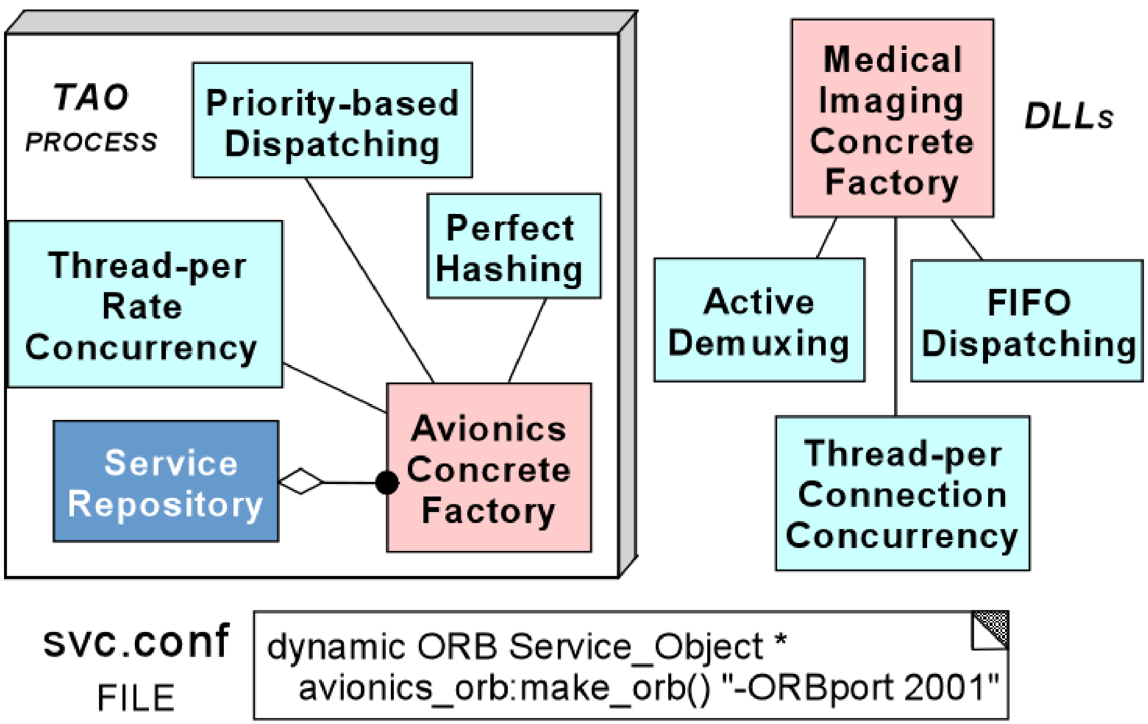


Concrete factories create groups of strategies

Dynamically Configuring Factories w/the Component Configurator Pattern

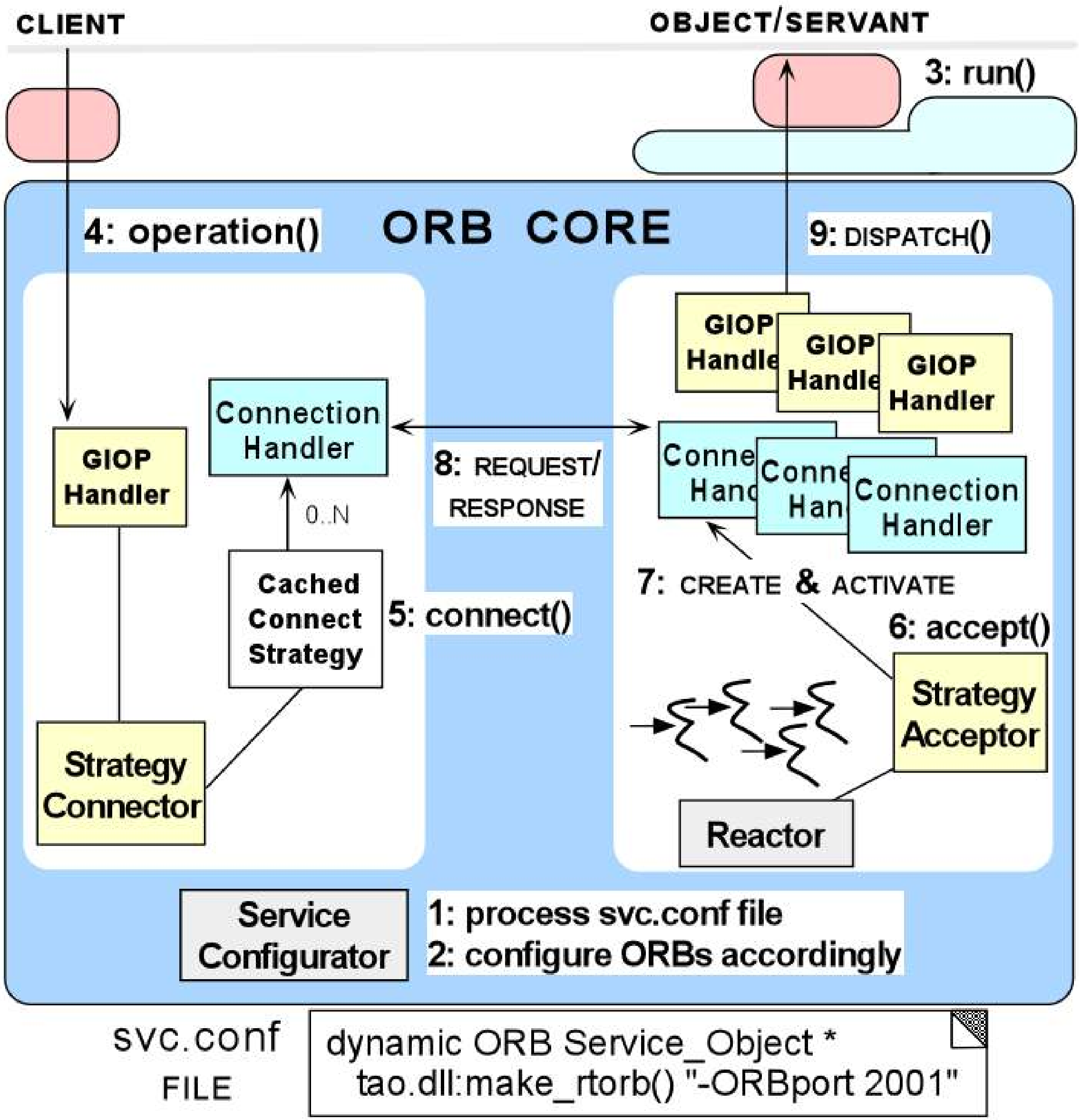
Context	Problem	Solution

• 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

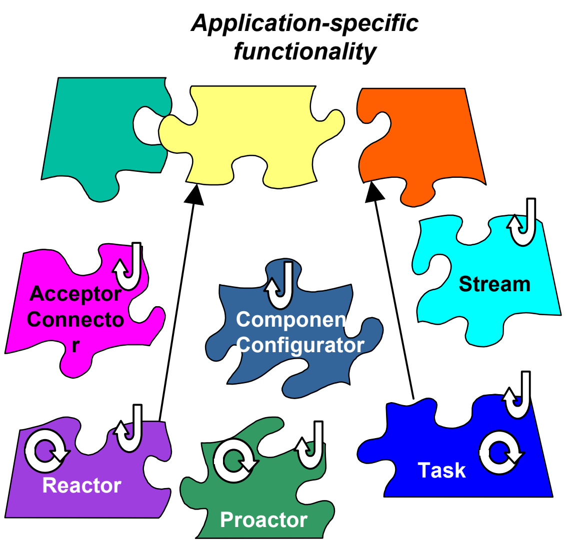


- ***Reactor*** drives the ORB event loop
  - Implements the ***Reactor*** & ***Leader/Followers*** patterns
- ***Acceptor-Connector*** decouples passive/active connection roles from GIOP request processing
  - Implements the ***Acceptor-Connector*** & ***Strategy*** patterns
- ***Service Configurator***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

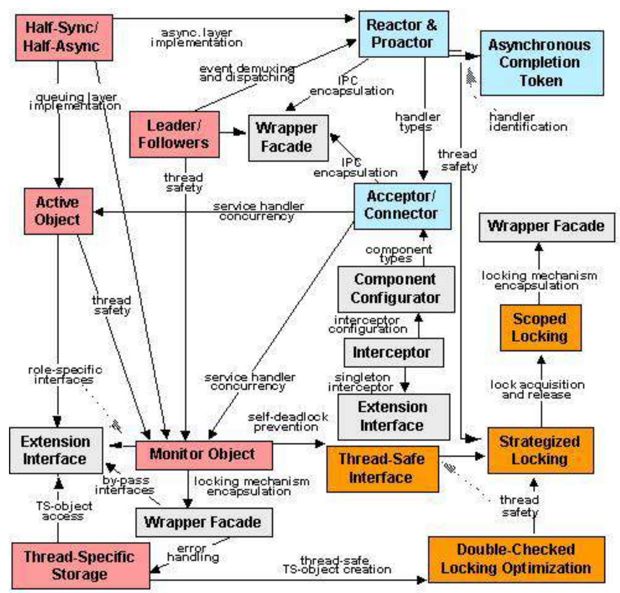
Summary of Pattern, Framework, & Middleware Synergies

The technologies codify expertise of experienced researchers & developers

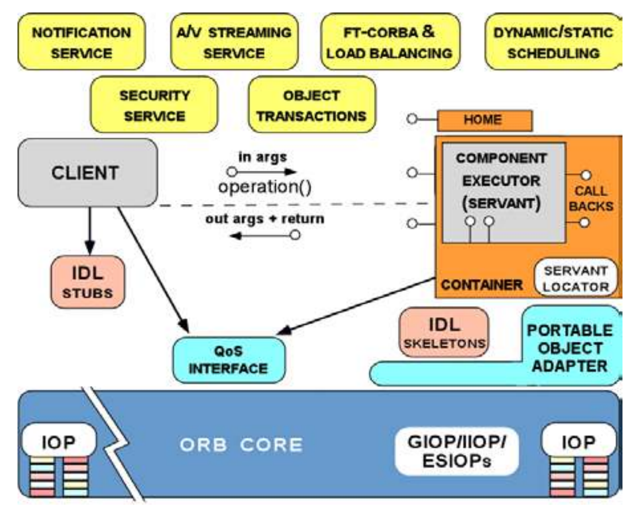
- Frameworks codify expertise in the form of reusable algorithms, component implementations, & extensible architectures



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

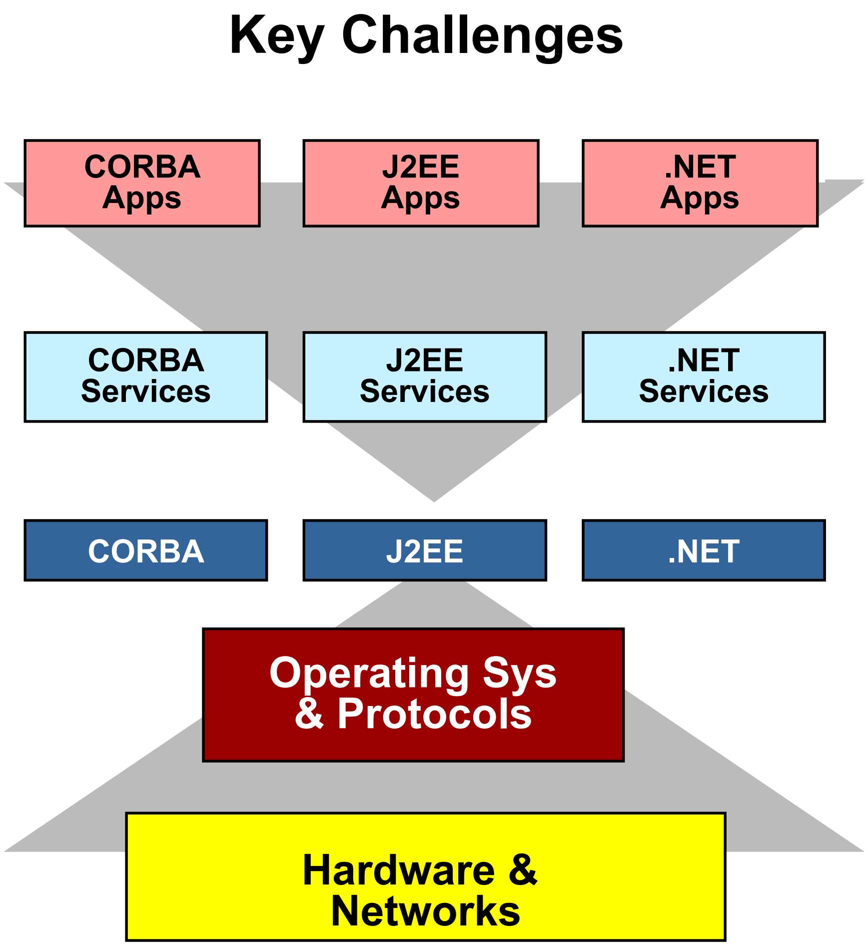


- Middleware codifies expertise in the form of standard interfaces & components that provide applications with a simpler façade to access the powerful (& complex) capabilities of frameworks

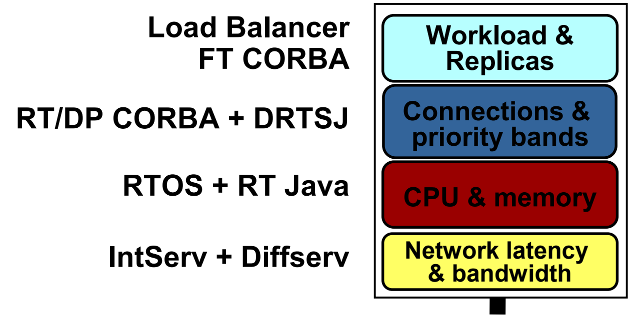


There are now powerful feedback loops advancing these technologies

The Road Ahead (1/2)



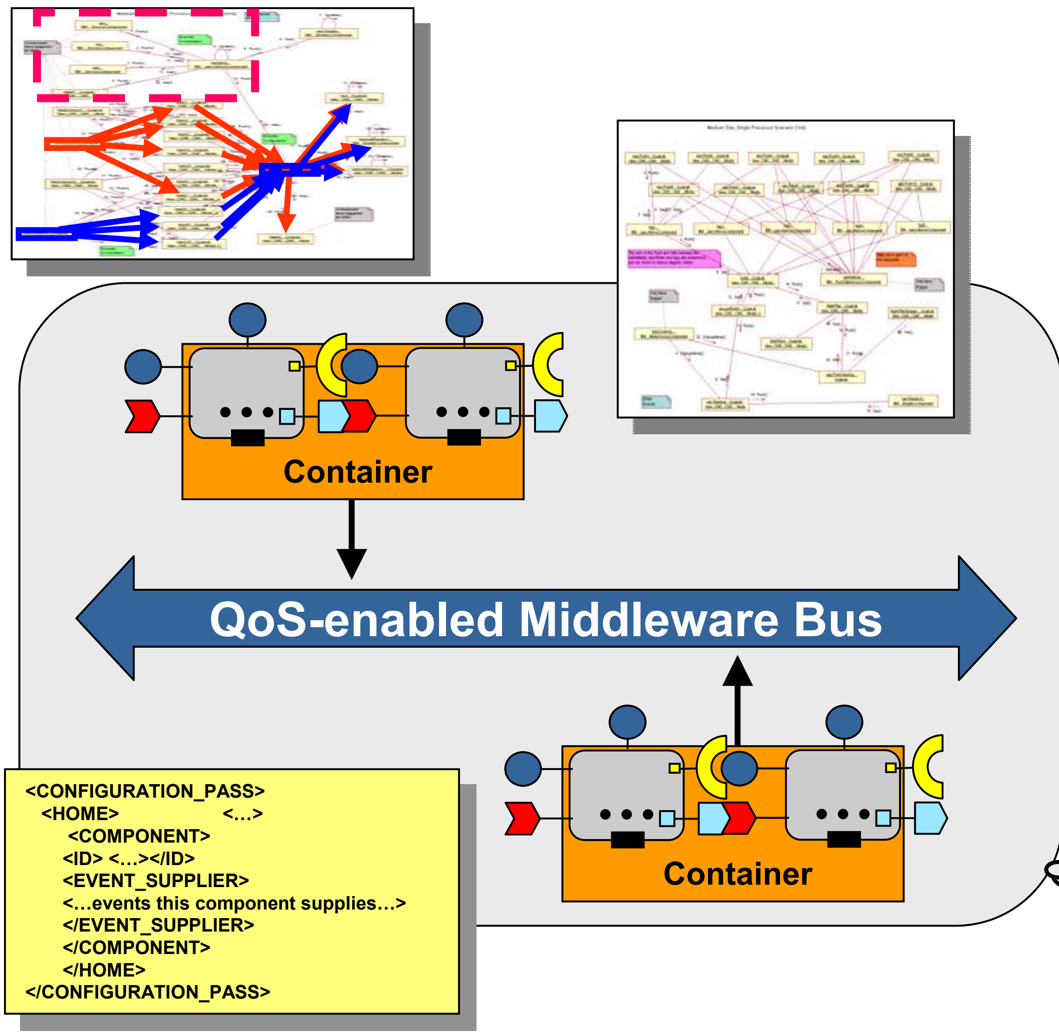
- There is a limit to how much application functionality can be factored into broadly reusable standard COTS middleware
- Middleware has become extremely complicated to use, configure, & provision statically & dynamically



- There are now (& will always be) multiple middleware technologies to choose from

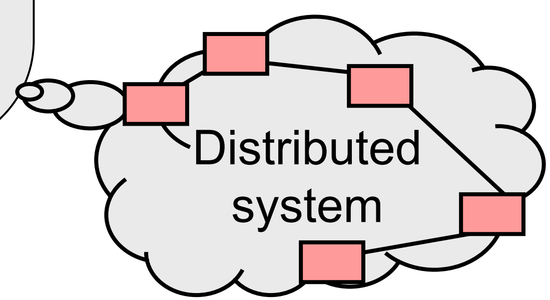
The Road Ahead (2/2)

Solution approach: Integrate model-based software technologies with QoS-enabled component middleware



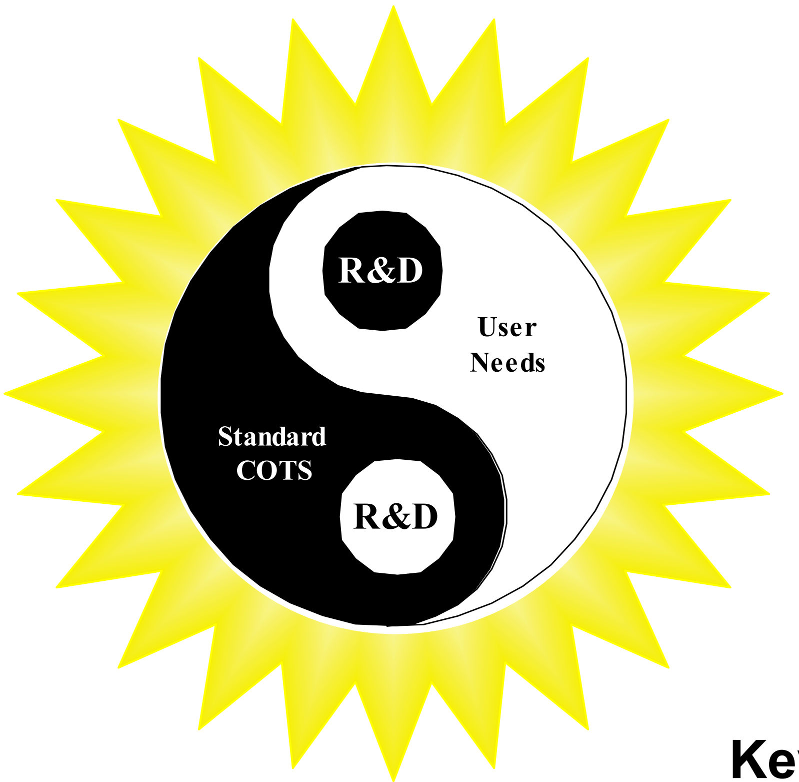
- e.g., standard technologies are emerging that can:
  1. ***Model***
  2. ***Analyze***
  3. ***Synthesize &***
  4. ***Provision***

  multiple layers of QoS-enabled middleware
- These technologies are guided by patterns & implemented by component frameworks



Concluding Remarks

R&D Synergies



- Prior R&D efforts have address some, *but by no means all*, of the challenging DOC middleware research topics

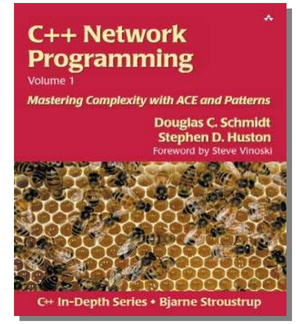
- Researchers & developers of distributed systems face common challenges, e.g.:
  - *connection management*
  - *service initialization*
  - *error handling*
  - *flow & congestion control*
  - *event demultiplexing*
  - *distribution*
  - *concurrency & synchronization*
  - *fault tolerance*
  - *scheduling &*
  - *persistence*
- ***Pattern languages***, ***frameworks***, & ***component middleware*** work together to help resolve these challenges

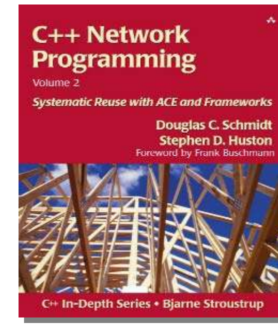
Key open R&D challenges include:

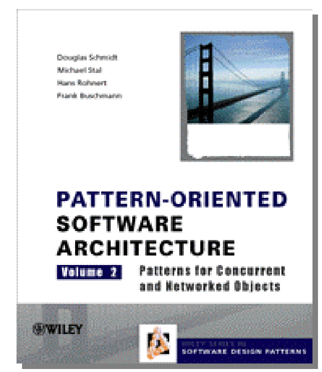
- Layered QoS specification & enforcement
- Separating policies & mechanisms across layers
- Time/space optimizations for middleware & apps
- Multi-level global resource mgmt. & optimization
- High confidence
- Stable & robust adaptive systems

Additional Information

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







- Research papers
  - `www.cs.wustl.edu/~schmidt/research.html`
- Tutorial on patterns, frameworks, & middleware
  - UCLA extension, July 9-11, 2003
  - `www.cs.wustl.edu/~schmidt/UCLA.html`
- Conferences on patterns, frameworks, & middleware
  - DOA, ECOOP, ICDCS, ICSE, Middleware, OOPSLA, PLoP(s), RTAS,