

Evaluating OO Applications

Douglas C. Schmidt

Washington University, St. Louis

**<http://www.cs.wustl.edu/~schmidt/>
schmidt@cs.wustl.edu**

1

Introduction

- Concurrent and distributed systems are increasingly important
- However, developing *efficient*, *robust*, and *extensible* concurrent and distributed applications is hard
 - e.g., must address complex topics that are less problematic or not relevant for non-concurrent, stand-alone applications
- Object-oriented (OO) techniques and OO language features help to enhance concurrent software quality factors
 - Key OO techniques include *design patterns* and *frameworks*
 - Key OO language features include *classes*, *inheritance*, *dynamic binding*, and *parameterized types*
 - Key software quality factors include *modularity*, *extensibility*, *portability*, *reusability*, and *correctness*

2

Motivation for Concurrency

- Concurrent programming is increasing relevant to:
 - *Leverage hardware/software advances*
 - ▷ e.g., multi-processors and OS thread support
 - *Increase performance*
 - ▷ e.g., overlap computation and communication
 - *Improve response-time*
 - ▷ e.g., GUIs and network servers
 - *Simplify program structure*
 - ▷ e.g., synchronous vs. asynchronous network IPC

3

Motivation for Distribution

- Benefits of distributed computing:
 - Collaboration → *connectivity* and *interworking*
 - Performance → *multi-processing* and *locality*
 - Reliability and availability → *replication*
 - Scalability and portability → *modularity*
 - Extensibility → *dynamic configuration* and *reconfiguration*
 - Cost effectiveness → *open systems* and *resource sharing*

4

Caveats

- OO is *not* a panacea
 - However, when used properly it helps minimize “accidental” complexity and improve software quality factors
- Advanced OS features provide additional functionality and performance, e.g.,
 - *Multi-threading*
 - *Multi-processing*
 - *Synchronization*
 - *Shared memory*
 - *Explicit dynamic linking*
 - *Interprocess communication (IPC)*

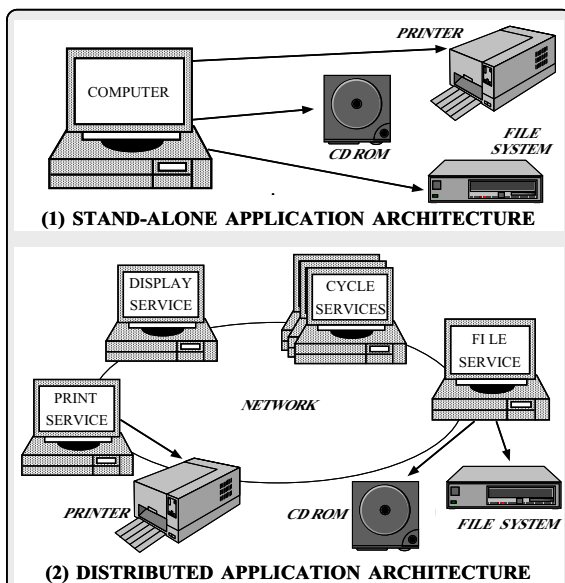
5

Definitions

- *Concurrency*
 - “Logically” simultaneous processing
 - Does *not* imply multiple processing elements
- *Parallelism*
 - “Physically” simultaneous processing
 - Involves multiple processing elements and/or independent device operations
- *Distribution*
 - Partition system/application into multiple components that can reside on different hosts
 - Implies message passing as primary IPC mechanism

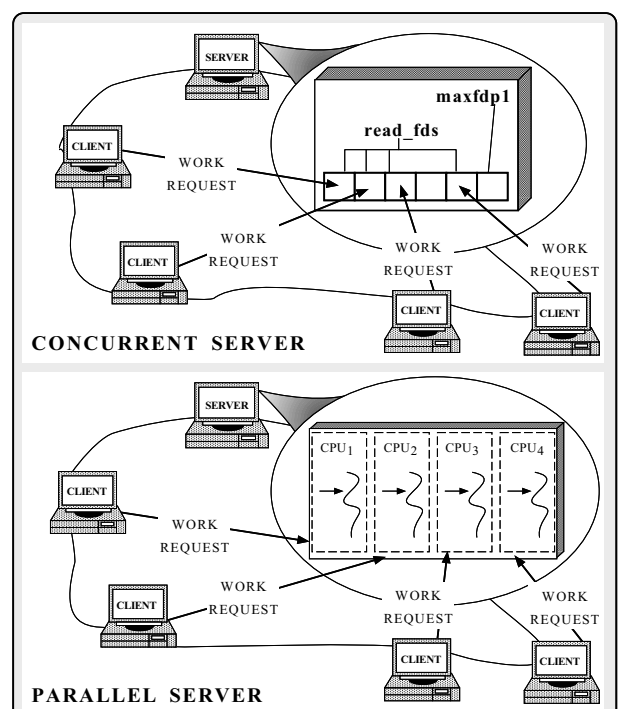
6

Stand-alone vs. Distributed Application Architectures



7

Concurrency vs. Parallelism



8

Sources of Complexity

- Concurrent network application development exhibits both *inherent* and *accidental* complexity
- *Inherent complexity* results from fundamental challenges
 - *Concurrent programming*
 - * Eliminating “race conditions”
 - * Deadlock avoidance
 - * Fair scheduling
 - * Performance optimization and tuning
 - *Distributed programming*
 - * Addressing the impact of latency
 - * Fault tolerance and high availability
 - * Load balancing and service partitioning
 - * Consistent ordering of distributed events

9

Sources of Complexity (cont'd)

- *Accidental complexity* results from limitations with tools and techniques used to develop concurrent applications, *e.g.*,
 - Lack of portable, reentrant, type-safe and extensible system call interfaces and component libraries
 - Inadequate debugging support and lack of concurrent and distributed program analysis tools
 - Widespread use of *algorithmic* decomposition
 - ▷ Fine for *explaining* concurrent programming concepts and algorithms but inadequate for *developing* large-scale concurrent network applications
 - Continuous rediscovery and reinvention of core concepts and components

10

OO Contributions to Concurrent Applications

- UNIX concurrent network programming has traditionally been performed using low-level OS mechanisms, *e.g.*,
 - * *fork/exec*
 - * *Shared memory, mmap, and SysV semaphores*
 - * *Signals*
 - * *sockets/select*
 - * *POSIX pthreads and Solaris threads*
- OO *design patterns* and *frameworks* elevate development to focus on application concerns, *e.g.*,
 - *Service functionality and policies*
 - *Service configuration*
 - *Concurrent event demultiplexing and event handler dispatching*
 - *Service concurrency and synchronization*

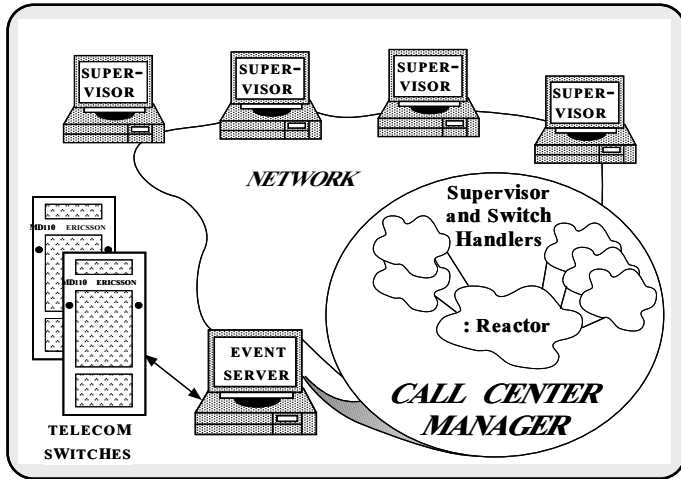
11

Case Study: Reusable Communication Software Frameworks

- Developing portable, reusable, and efficient communication software is hard
- OS platforms are often fundamentally incompatible
 - *e.g.*, different concurrency and I/O models
- Thus, it may be impractical to directly reuse:
 - *Algorithms*
 - *Detailed designs*
 - *Interfaces*
 - *Implementations*

12

System Overview



- OO framework for call center management developed for Ericsson

13

Problem: Cross-platform Reuse

- Original OO framework was developed for UNIX and later ported to Windows NT
- UNIX and Windows NT have fundamentally different I/O models
 - *i.e.*, synchronous vs. asynchronous
- Thus, direct reuse of original framework was infeasible...

14

Solution: Reuse Design Patterns

- Design patterns support reuse of *software architecture*
- Patterns embody successful *solutions* to *problems* that arise when developing software in a particular *context*
- Design patterns greatly reduced project risk at Ericsson by leveraging proven design expertise

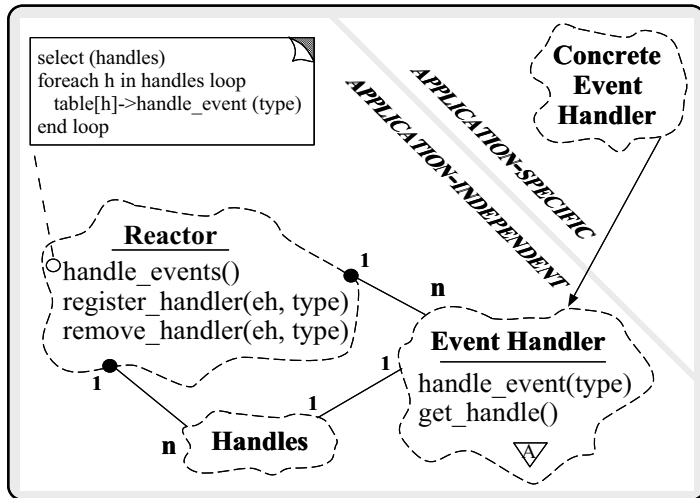
15

Example Pattern: Reactor

- *Intent*
 - Decouple event demultiplexing and event handler dispatching from the services performed in response to events
- This pattern solves several problems for single-threaded communication software:
 1. *How to efficiently demultiplex multiple types of events from multiple sources of events within a single thread of control*
 2. *How to extend application behavior without requiring changes to the event demultiplexing/dispatching framework*
- A pattern description captures the static and dynamic *structure* and *collaboration* among key *participants* in a micro-architecture

16

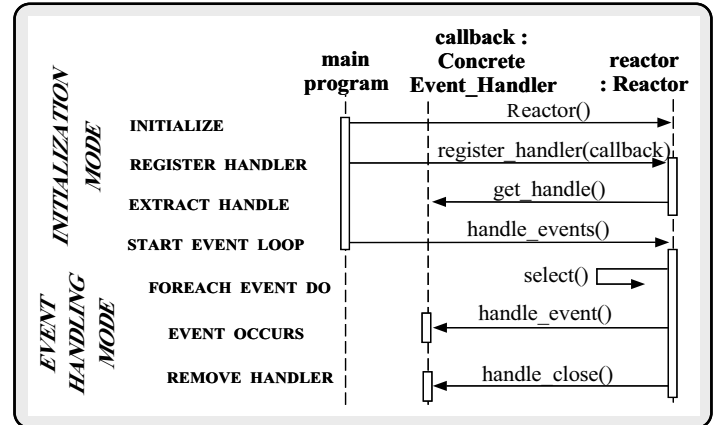
Structure



- Participants in the Reactor pattern

17

Collaborations



- Dynamic interaction among participants in the Reactor pattern

18

Implementing the Reactor on UNIX and Windows NT

- Major difference is UNIX *reactive* I/O vs. Windows NT *proactive* I/O
 - Reactive I/O is synchronous
 - Proactive I/O can be asynchronous
 - Requires additional interfaces to “arm” the I/O mechanism
- Other differences include
 - Resource limitations*
 - e.g., Windows NT limits number of HANDLES per-thread
 - Demultiplexing fairness*
 - e.g., `WaitForMultipleObjects` always returns the lowest active HANDLE

19

Summary of Case Study

- Real-world constraints of OS platforms can preclude direct reuse of communication software
 - e.g., must often use non-portable features for performance
- Reuse of design patterns may be the only viable means to leverage previous development expertise
- Design patterns are useful, but are no panacea
 - Managing expectations is crucial...

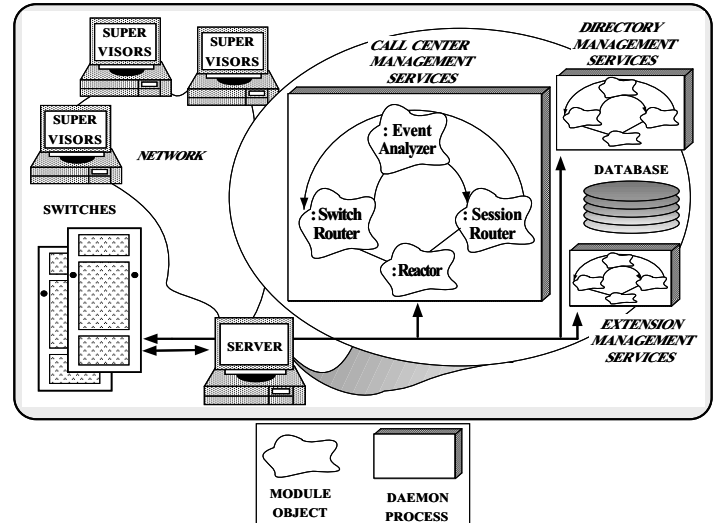
20

Example Uses of OO in Industry

- OO frameworks and patterns have been used in many products
 - PBX network management at Ericsson
 - Global personal communications at Motorola Iridium
 - Distributed electronic medical imaging for Kodak and Siemens
- The following outline several representative use cases

21

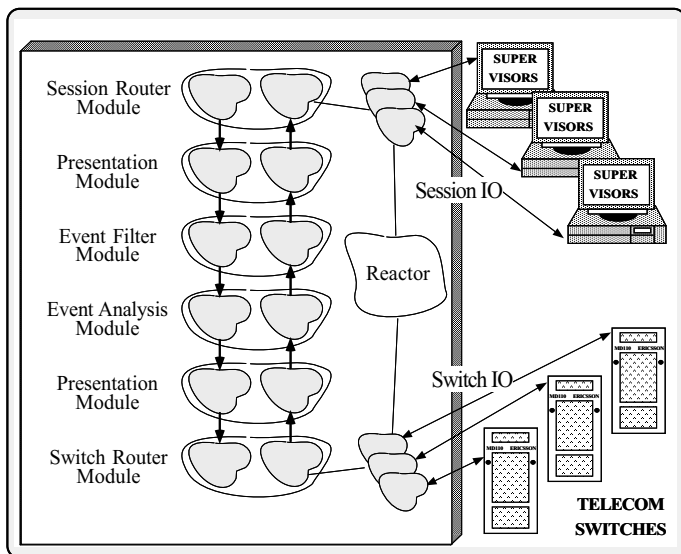
ACE-based Client/Server EOS Architecture



- Based upon the ACE Stream class category

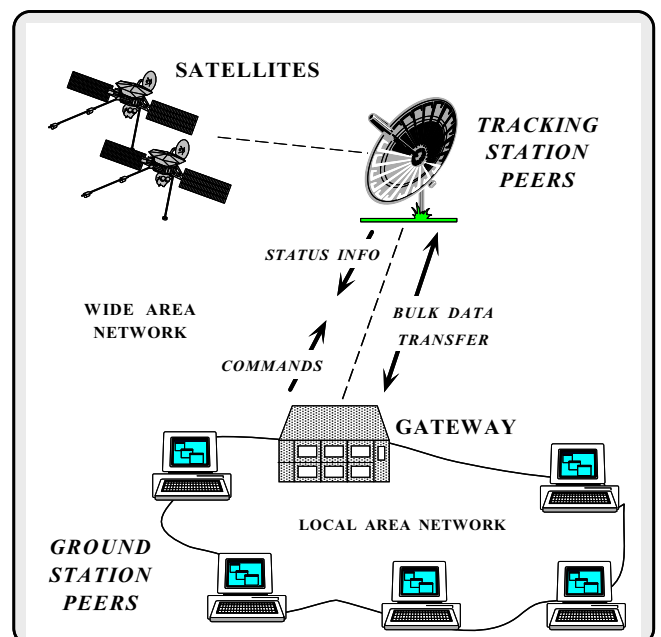
22

Detailed Structure of a CCM



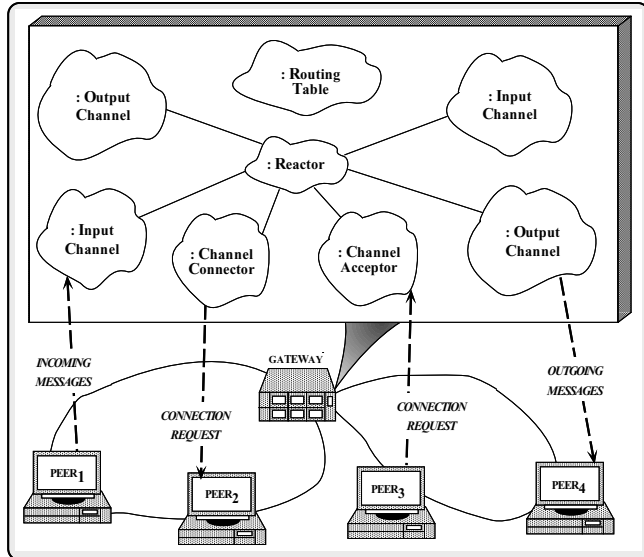
23

Physical Architecture of the Gateway



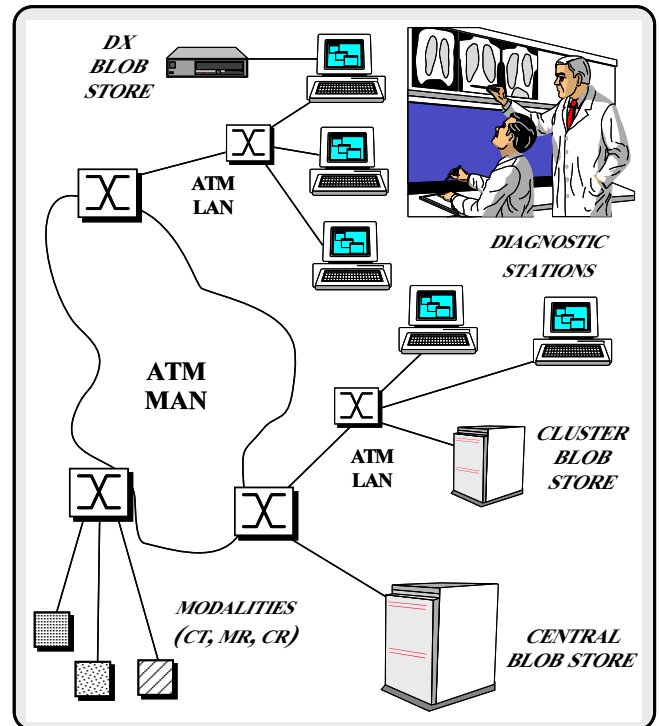
24

OO Software Architecture of the Gateway



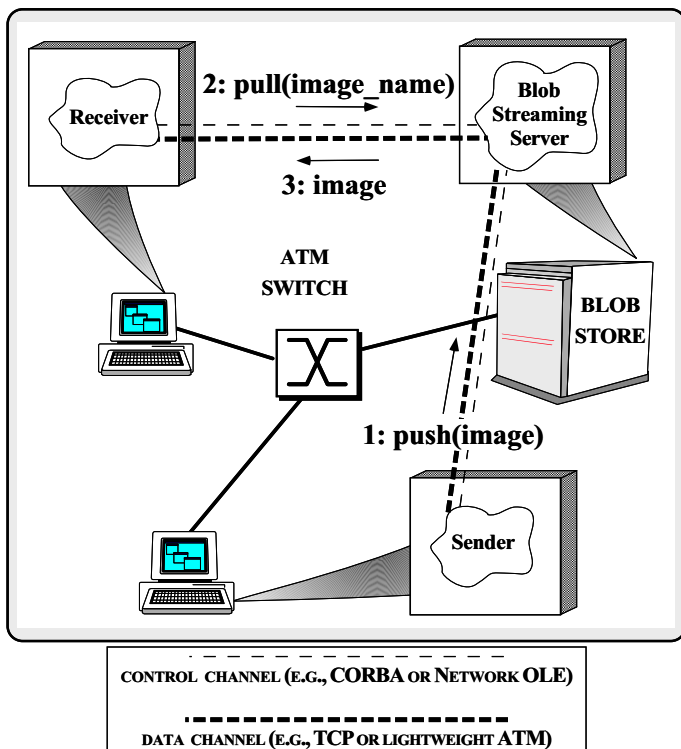
25

Distributed Medical Imaging



26

Push and Pull Models



27

Lessons Learned using OO Design Patterns

- *Benefits of patterns*
 - Enable large-scale reuse of software architectures
 - Improve development team communication
 - Help transcend language-centric viewpoints
- *Drawbacks of patterns*
 - Do not lead to direct code reuse
 - Can be deceptively simple
 - Teams may suffer from pattern overload

28

Lessons Learned using OO Frameworks

- *Benefits of frameworks*
 - Enable direct reuse of code (*cf* patterns)
 - Facilitate larger amounts of reuse than stand-alone functions or individual classes
- *Drawbacks of frameworks*
 - High initial learning curve
 - Many classes, many levels of abstraction
 - The flow of control for reactive dispatching is non-intuitive

29

Lessons Learned using C++

- *Benefits of C++*
 - *Classes* and *namespaces* modularize the system architecture
 - *Inheritance* and *dynamic binding* decouple application *policies* from reusable *mechanisms*
 - *Parameterized types* decouple the reliance on particular types of synchronization methods or network IPC interfaces
- *Drawbacks of C++*
 - Many language features are not widely implemented
 - Development environments are primitive
 - Language has many dark corners and sharp edges

30