Architecture of Software Systems – Lecture **4**

Design Patterns for Distributed Systems

Martin Rehák

Overview

- Selected Design & Architectural patterns for Distributed/Multiplatform computing
- Content based on:
  - Douglas C. Schmidt, Pattern Oriented Software Engineering
  - Gang of Four: Gamma, Helm, Johnson, Vlissides; Design Patterns: Elements of Reusable Object-Oriented Software
  - Guerraoui/Rodrigues: Introduction to Reliable Distributed Programming
  - …

Paradigm Shift

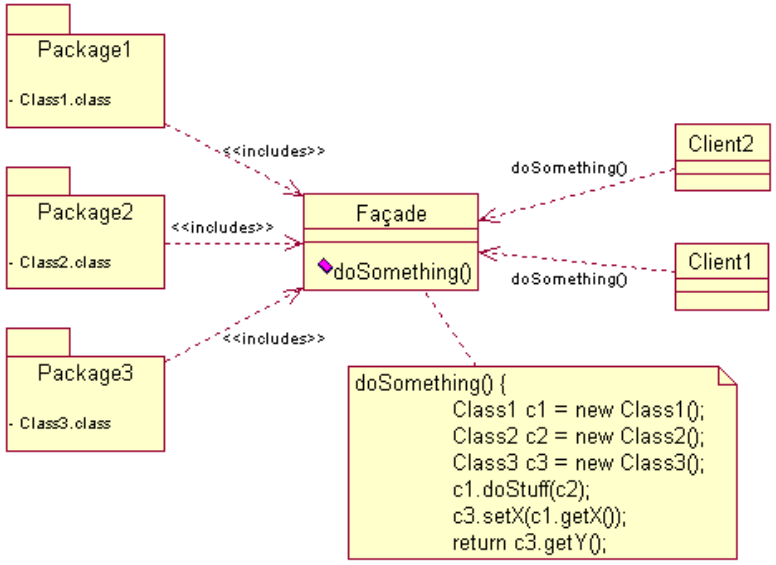
- From local to distributed
- Explicit vs. implicit
- Latency
- Failures
  - **Safety**: a property is a safety property if it can never be restored once it is broken (e.g. *the link will never insert a non-existing message into the media*)
  - **Liveliness**: can be restored anytime in the future (e.g. *subsystem will answer the request*)

Distribution Issues

1. Remote resource localization
2. Remote resource creation and usage
3. State synchronization management
4. Failure detection
5. Failure management, recovery and failover
6. Resource destruction

Facade

- Façade pattern hides the complexity and heterogeneity of system, subsystem or library behind a simple interface
- Typically replaces/hides more than one object
  - Simplifies client access
  - Allows transparent resource management
  - Allows transparent lazy initialization



Facade vs. Interface

- Interface (in Java sense) is far more restricted than Façade
- Only defines a contract between the library/ implementing class and its user – programmer
- Façade pattern allows active handling of more complex issues than actual business logic
  - Most of the code in distributed applications is NOT directly related to business logic

(Wrapper) Façade Example: JDBC

- Java Database Connectivity provides a unified API for most database work in Java
- Unified methods and constants
- Uses the **Adapter** pattern to incorporate third-party DBMS drivers
- Successfully hides most of the connection complexity/decisions from the Application Developer

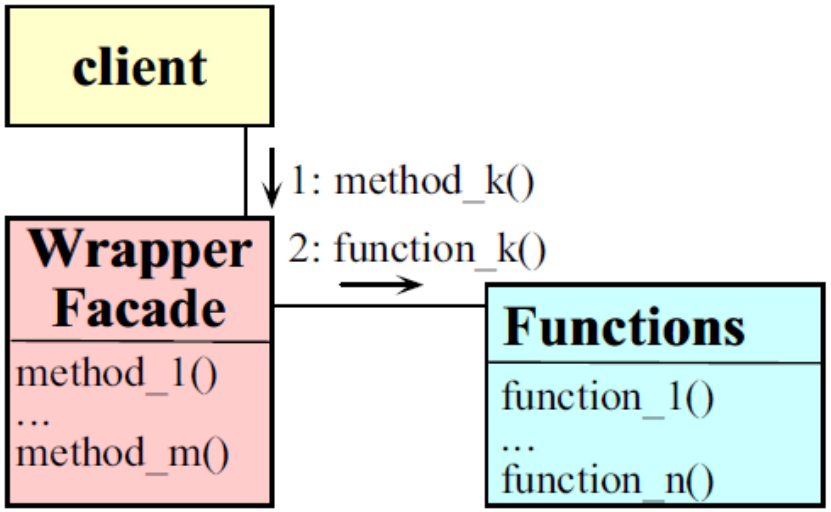
Adapter/Wrapper

- The Adapter pattern provides mapping between two (isofunctional) interfaces
- Ensures syntactic and semantic compatibility of calls

Wrapper Façade

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

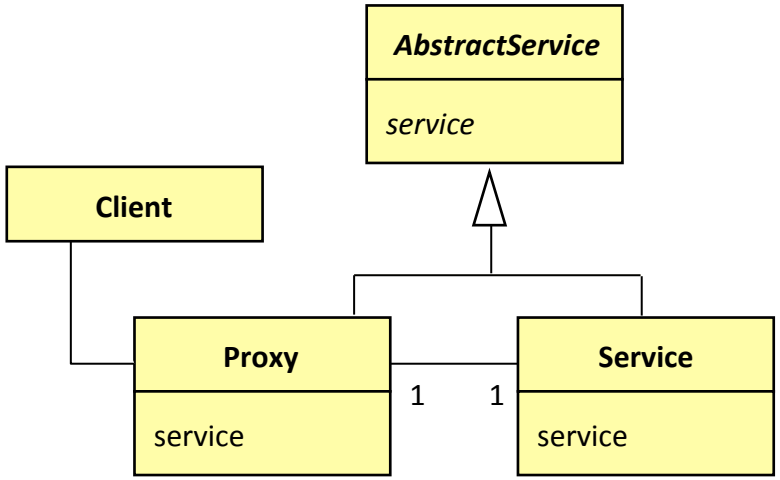
Douglas C. Schmidt, C++ Report 1999



- Usages:
  - Java Swing
  - ACE/ORB
  - Platform independent threading/ synchronization libraries
  - ACE Library

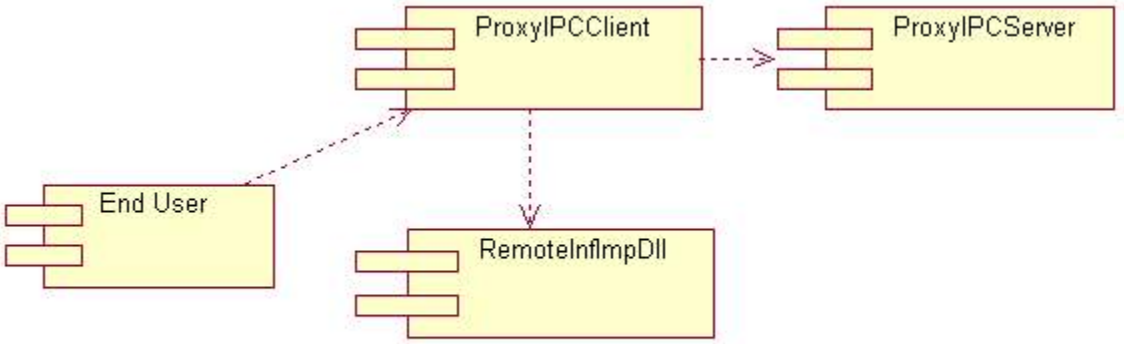
Proxy

- Proxy pattern is a local representation of a remote object, interface or library
- Rarely used alone, frequently combined with Façade, Wrappers and other Patterns



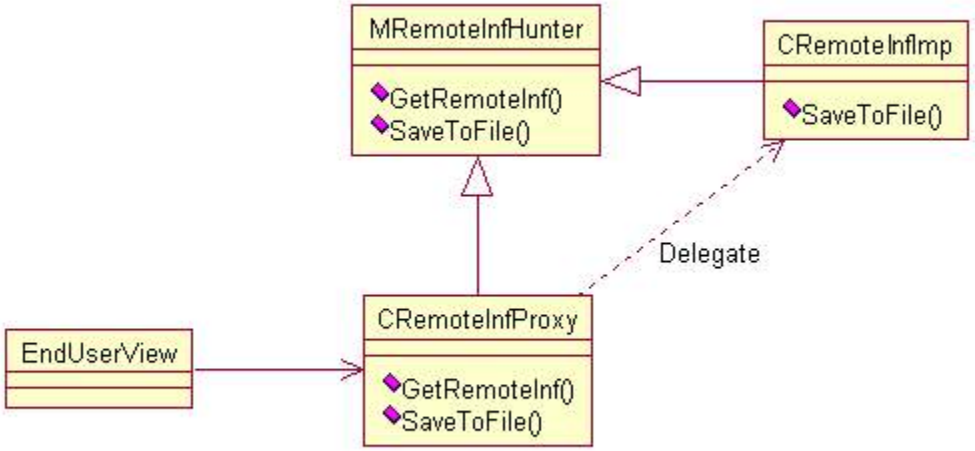
The Proxy Pattern (Douglas Schmidt, POSA)

Proxy example – Symbian (1)

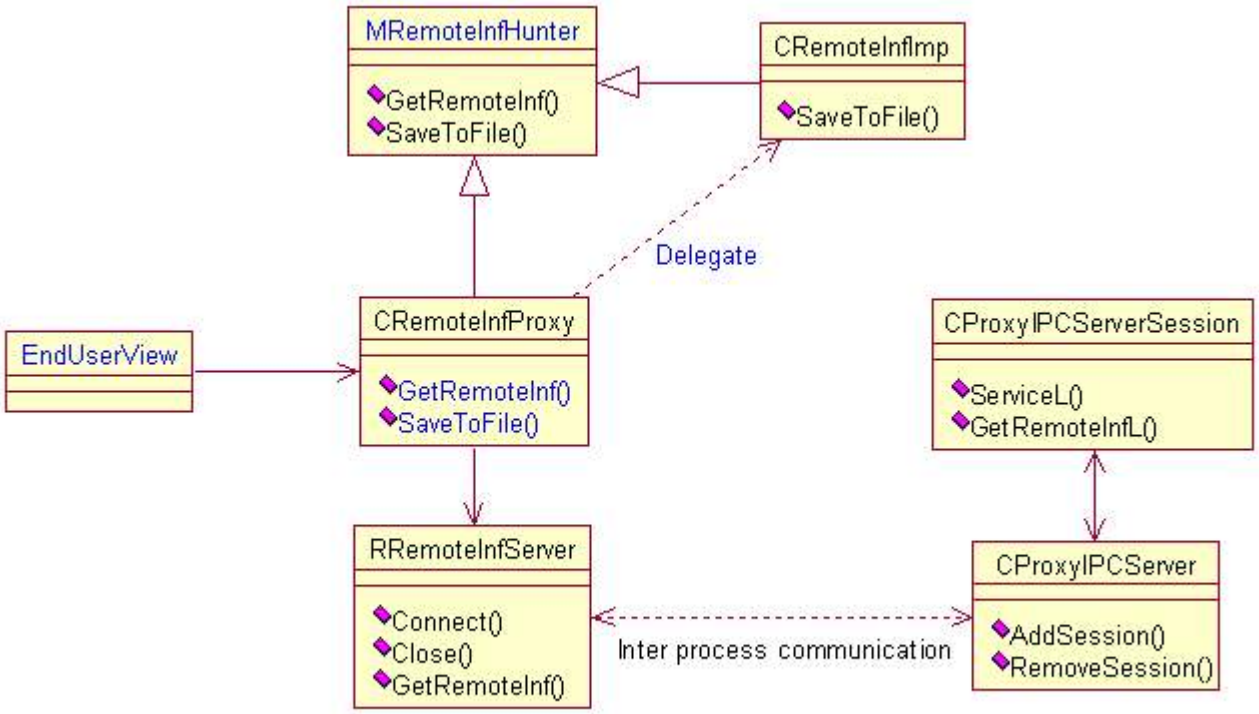


Ericsson. http://www.newlc.com/en/when-symbian-met-design-patterns-2-proxy-pattern

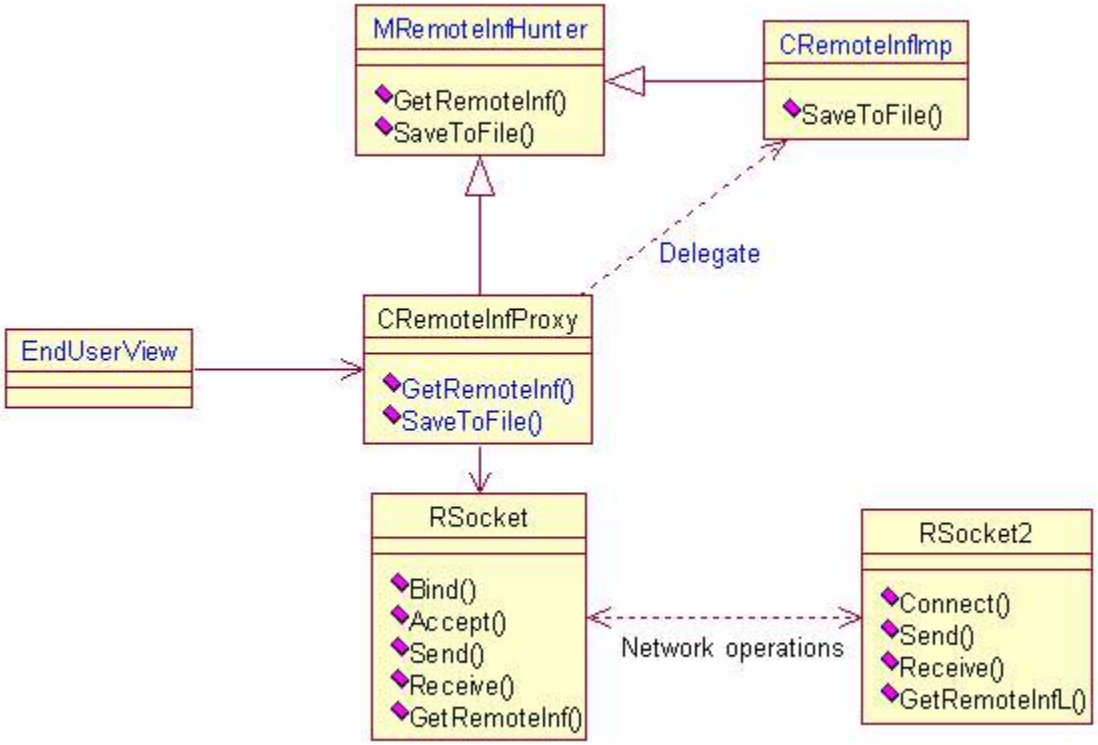
Proxy Example – Symbian (2)



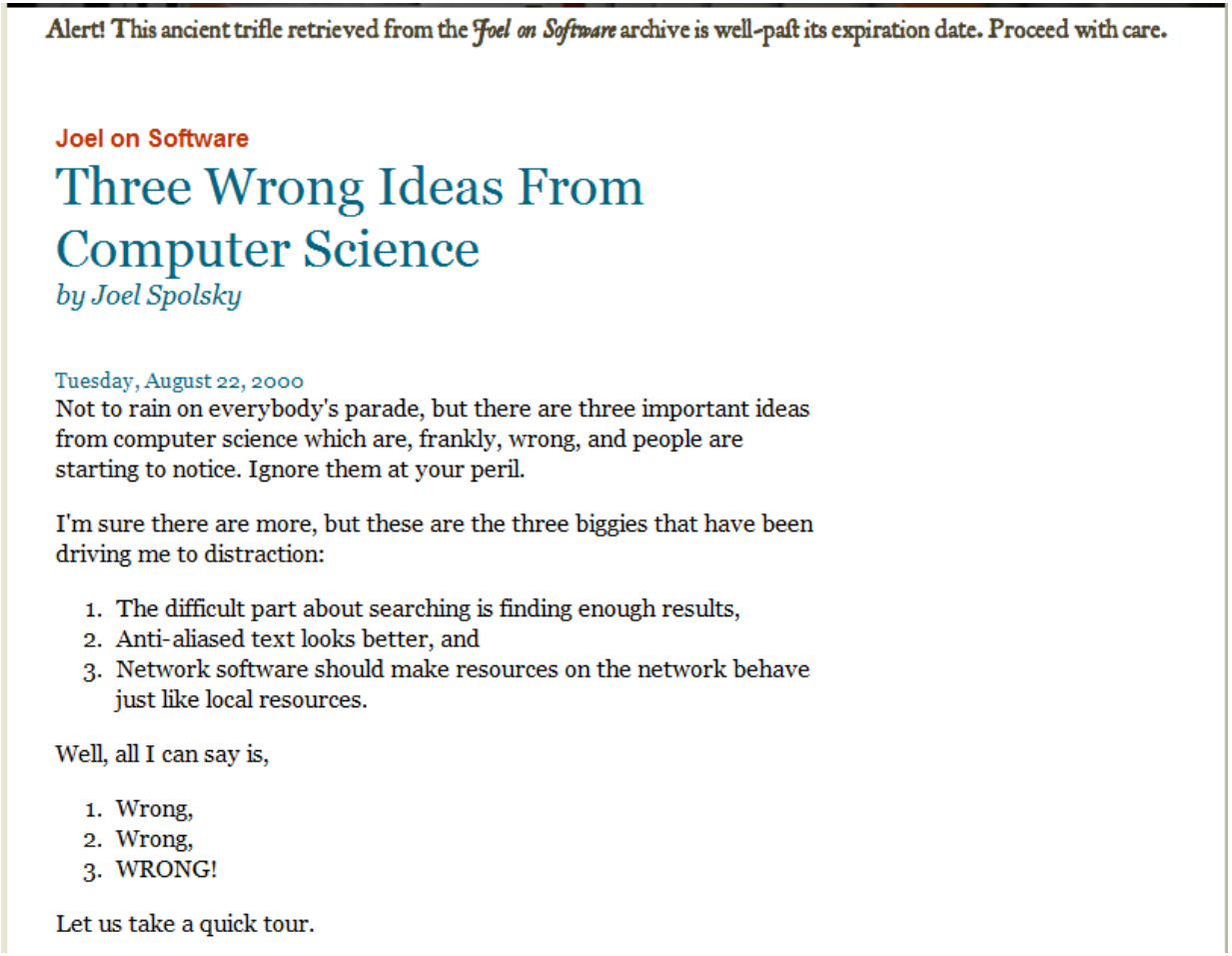
Proxy Example – Symbian (3)



Proxy Example – Symbian (4)



Implicit vs. Explicit Distribution



Alert! This ancient trifle retrieved from the *Joel on Software* archive is well-past its expiration date. Proceed with care.

Joel on Software

Three Wrong Ideas From Computer Science

by Joel Spolsky

Tuesday, August 22, 2000

Not to rain on everybody's parade, but there are three important ideas from computer science which are, frankly, wrong, and people are starting to notice. Ignore them at your peril.

I'm sure there are more, but these are the three biggies that have been driving me to distraction:

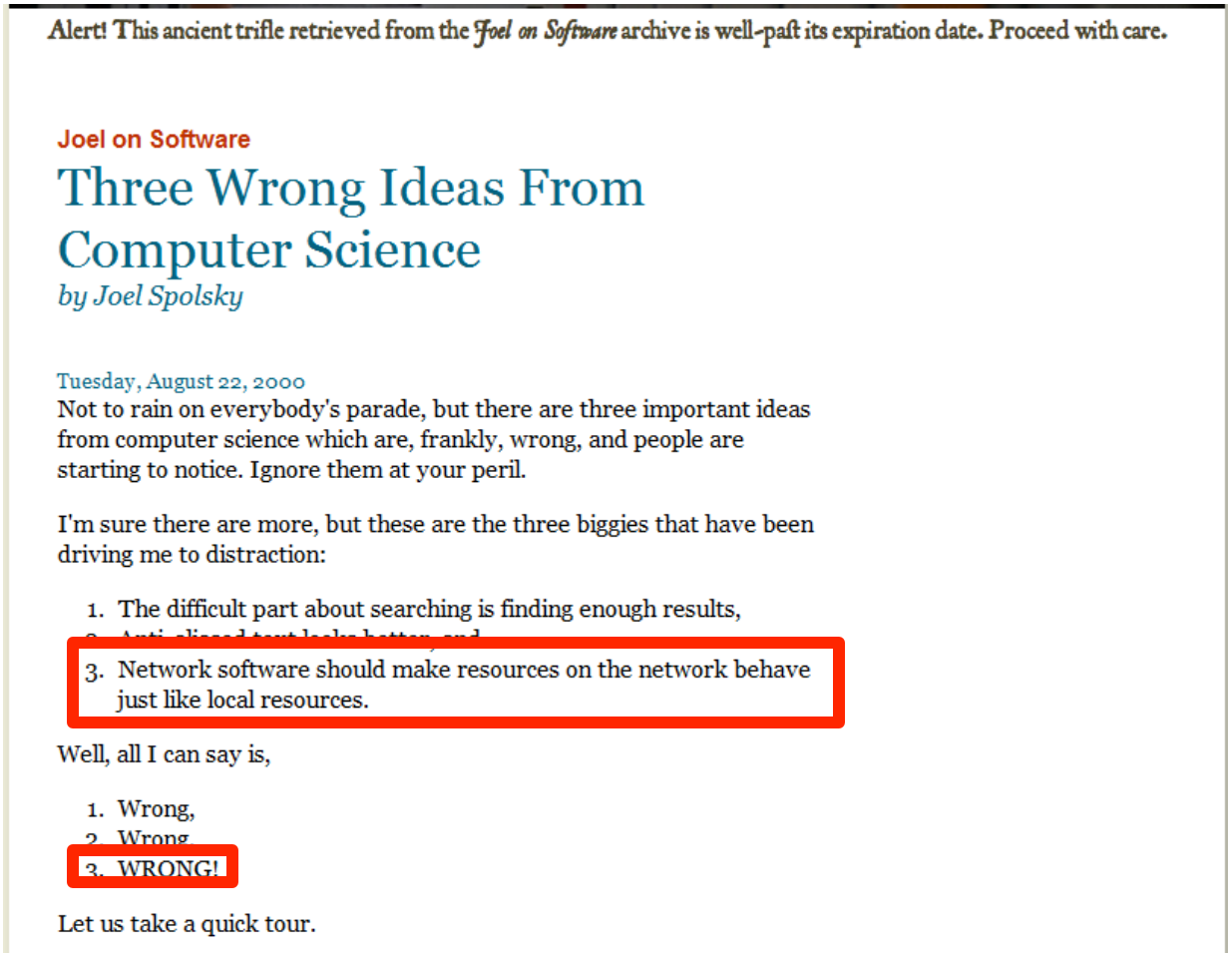
1. The difficult part about searching is finding enough results,
2. Anti-aliased text looks better, and
3. Network software should make resources on the network behave just like local resources.

Well, all I can say is,

1. Wrong,
2. Wrong,
3. WRONG!

Let us take a quick tour.

Implicit vs. Explicit Distribution



Alert! This ancient trifle retrieved from the *Joel on Software* archive is well-past its expiration date. Proceed with care.

Joel on Software

Three Wrong Ideas From Computer Science

by Joel Spolsky

Tuesday, August 22, 2000

Not to rain on everybody's parade, but there are three important ideas from computer science which are, frankly, wrong, and people are starting to notice. Ignore them at your peril.

I'm sure there are more, but these are the three biggies that have been driving me to distraction:

1. The difficult part about searching is finding enough results,
3. Network software should make resources on the network behave just like local resources.

Well, all I can say is,

1. Wrong,
2. Wrong,
3. WRONG!

Let us take a quick tour.

Stateful vs. Stateless design

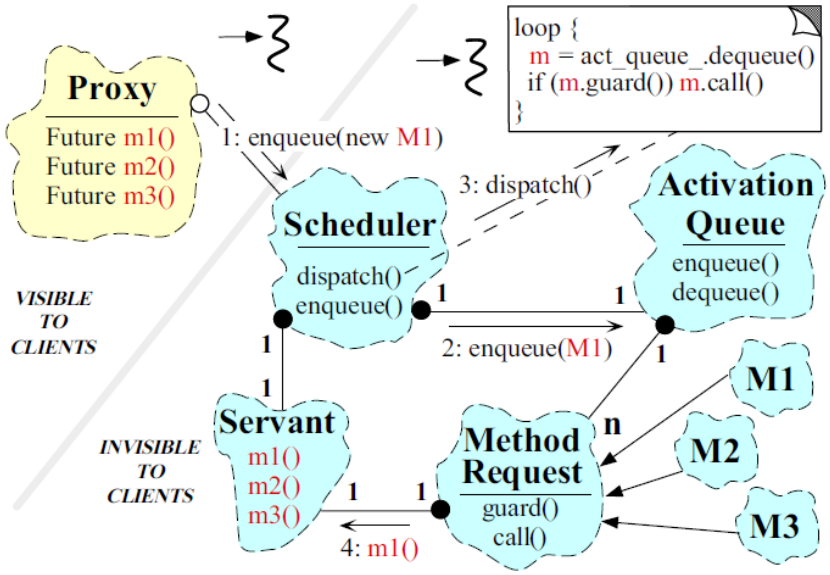
- State consistency and resource management are the points where most **abstractions break** in real life:
  - CORBA
  - COM/DCOM/OLE
- Alternative approaches make the **distribution EXPLICIT** and incorporate it into the design from the beginning
  - Messaging
  - HTTP-based communication

Active Object

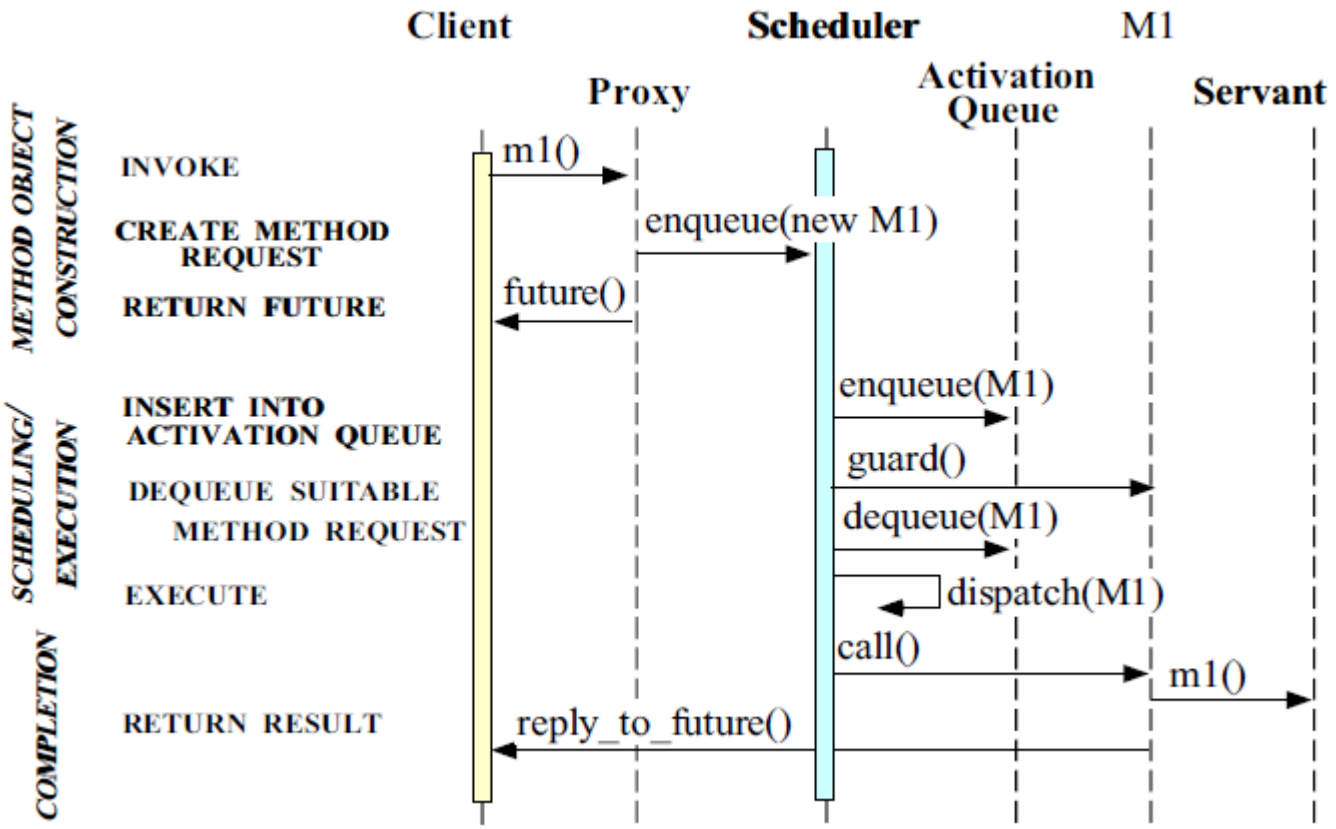
- *The Active Object design pattern decouples method execution from method invocation to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control.*

  D. Schmidt, et al. 2007

- Agents/Multi-Agent Systems and OLTP systems are most frequently based on Active Objects (Messaging)



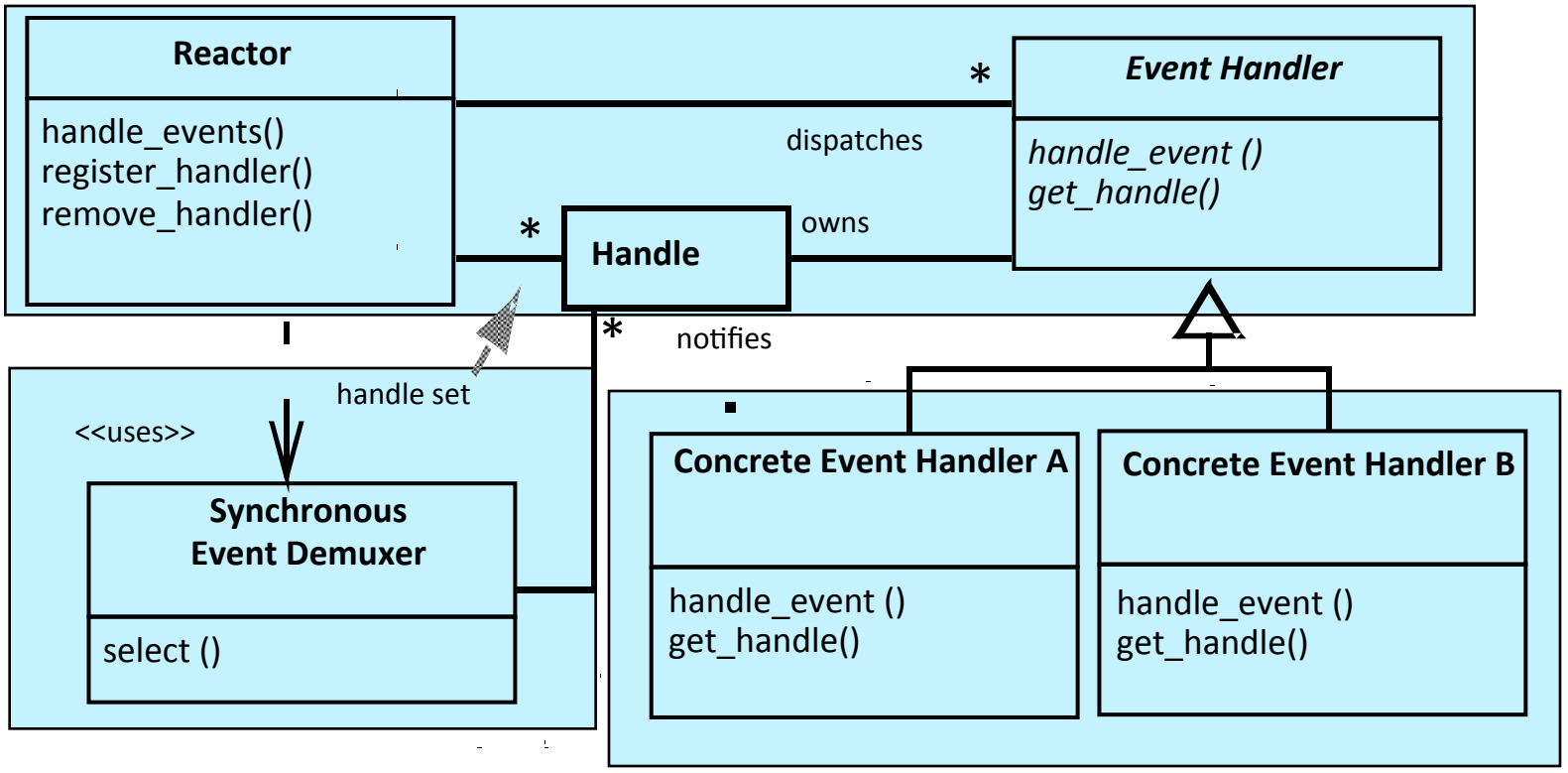
Active Object



Reactor

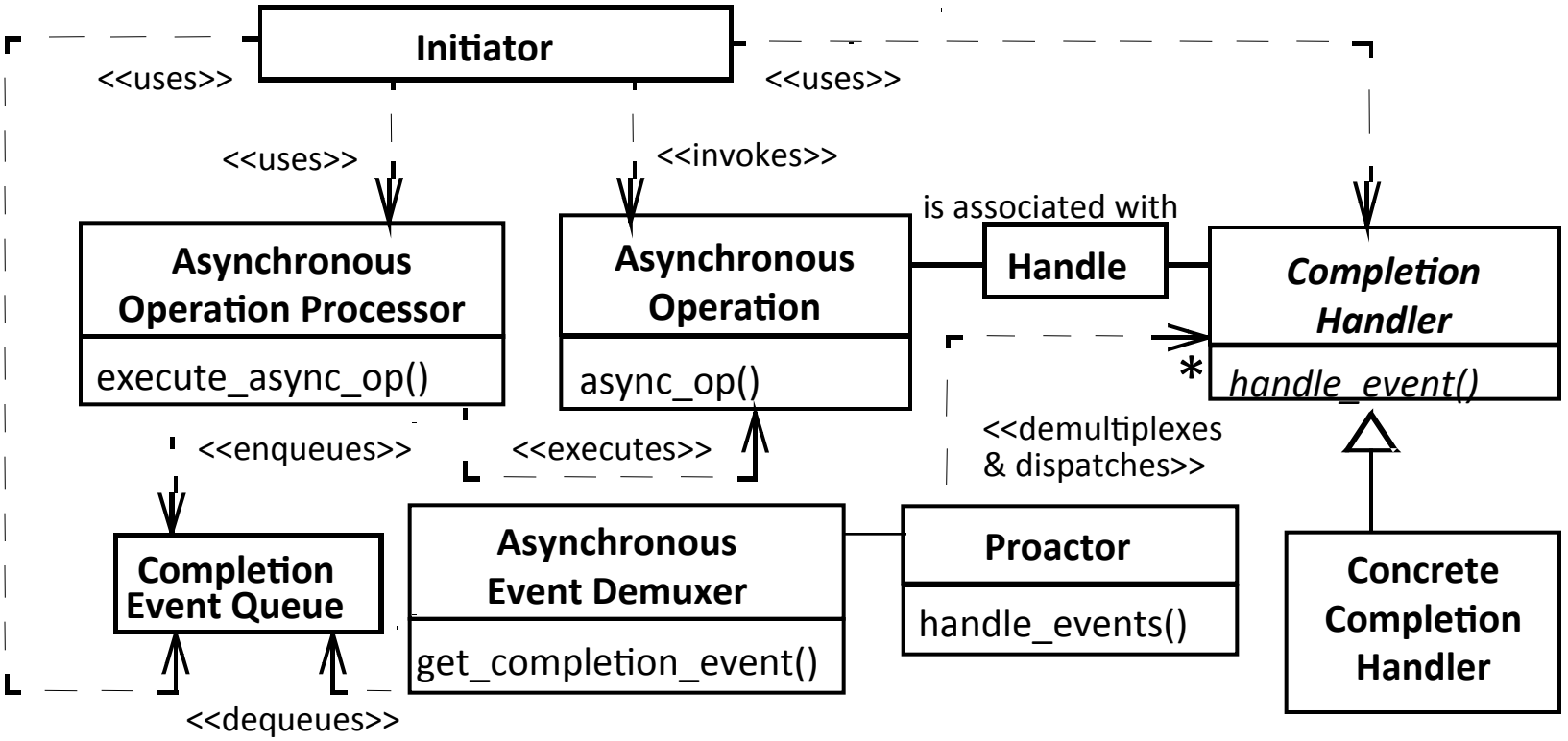
- *The **Reactor** architectural pattern allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients.*

(Schmidt)



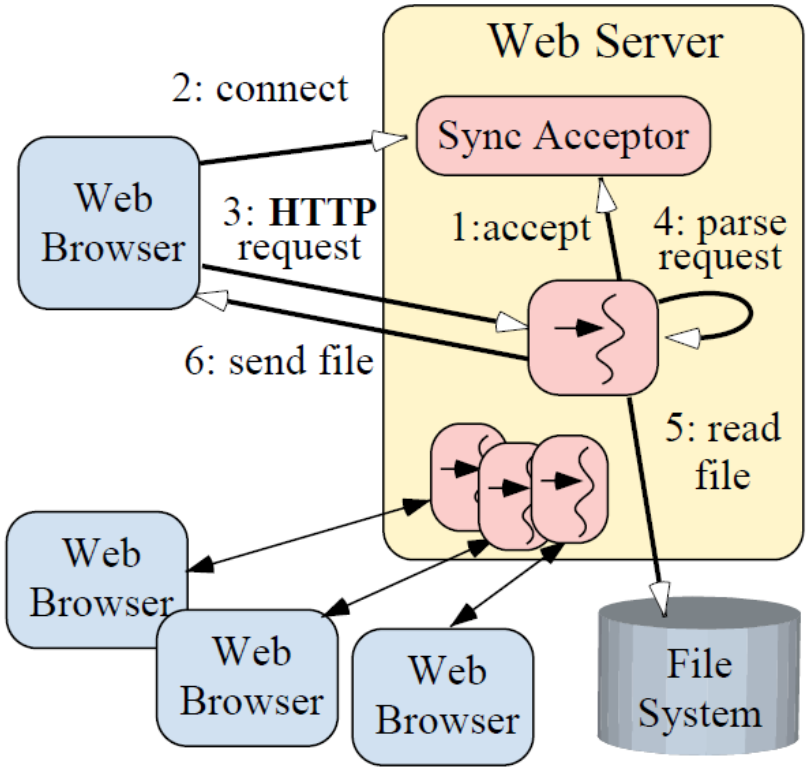
Proactor

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



Reactor/Proactor – Web server

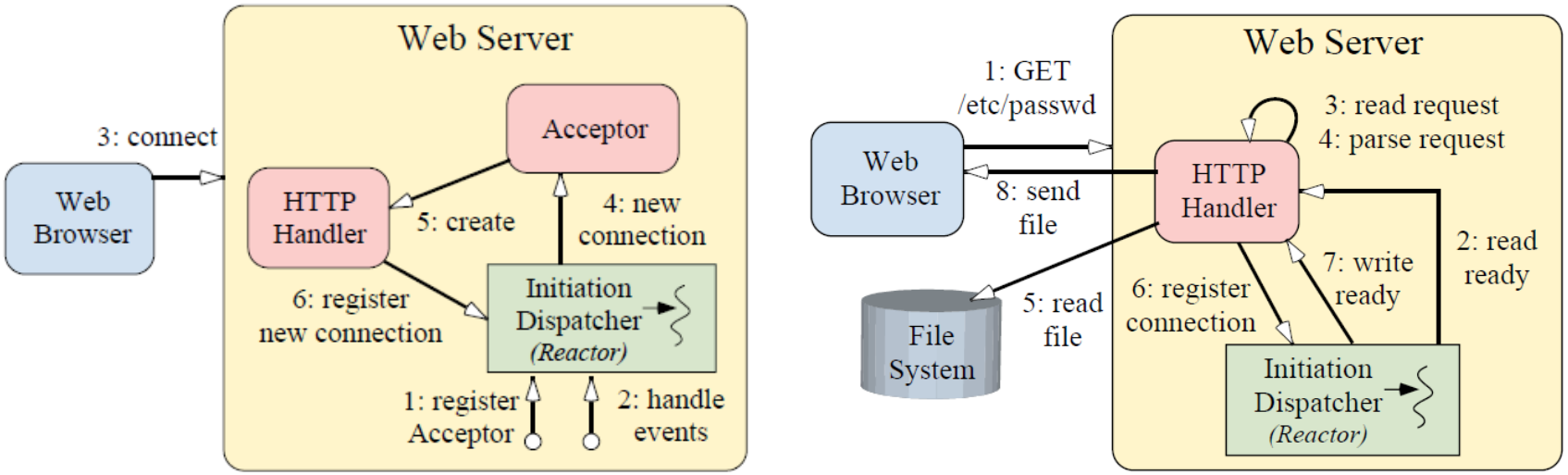
- Concurrency – The server must perform multiple client requests simultaneously;
- Efficiency – The server must minimize latency, maximize throughput, and avoid utilizing the CPU(s) unnecessarily.
- Programming simplicity – The design of the server should simplify the use of efficient concurrency strategies;
- Adaptability – Integrating new or improved transport protocols (such as HTTP 1.1 [3]) should incur minimal maintenance costs.



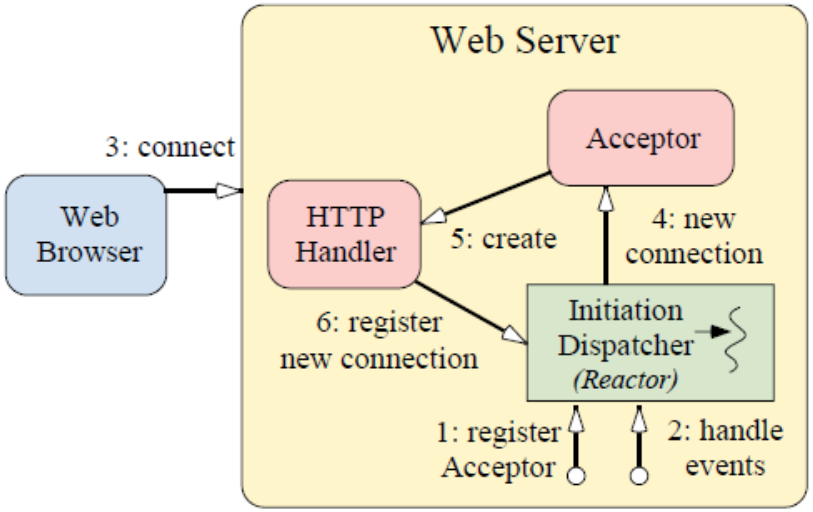
Schmidt, 1997

(NOT a Proactor/Reactor)

Reactor


Web Server
3: connect
Web Browser
HTTP Handler
Acceptor
5: create
4: new connection
6: register new connection
Initiation Dispatcher
(Reactor)
1: register Acceptor
2: handle events
Web Server
1: GET /etc/passwd
Web Browser
8: send file
HTTP Handler
3: read request
4: parse request
2: read ready
7: write ready
6: register connection
5: read file
File System
Initiation Dispatcher
(Reactor)

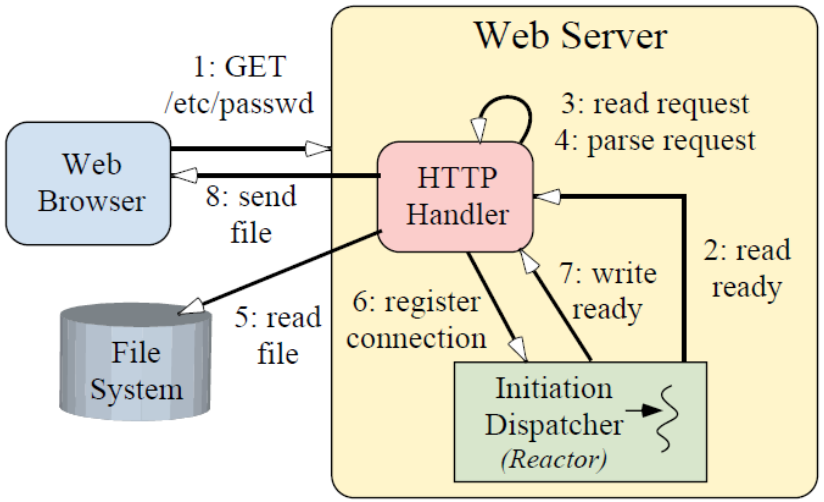
Reactor – Connection



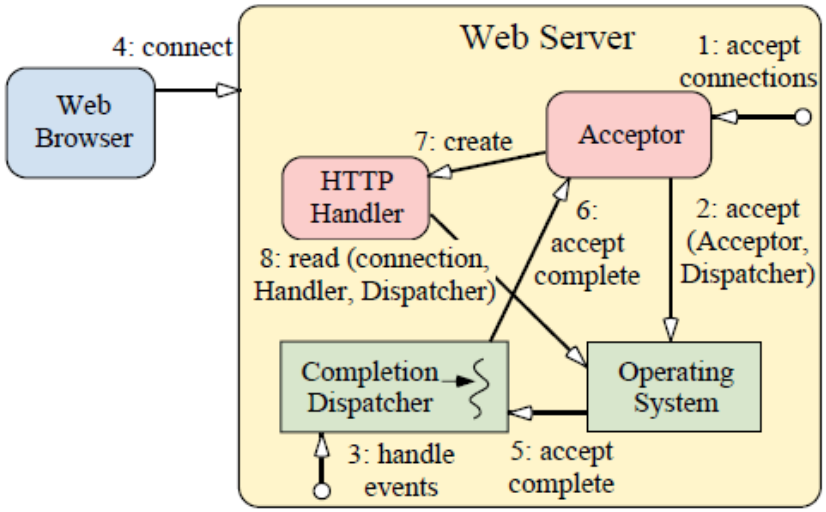
1. The Web Server registers an Acceptor with the Initiation Dispatcher to accept new connections;
2. The Web Server invokes event loop of the Initiation Dispatcher;
3. A client connects to the Web Server;
4. The Acceptor is notified by the Initiation
5. Dispatcher of the new connection request and the Acceptor accepts the new connection;
6. The Acceptor creates an HTTP Handler to service the new client;
7. HTTP Handler registers the connection with the Initiation Dispatcher for reading client request data (that is, when the connection becomes “ready for reading”);
8. The HTTP Handler services the request from the new client.

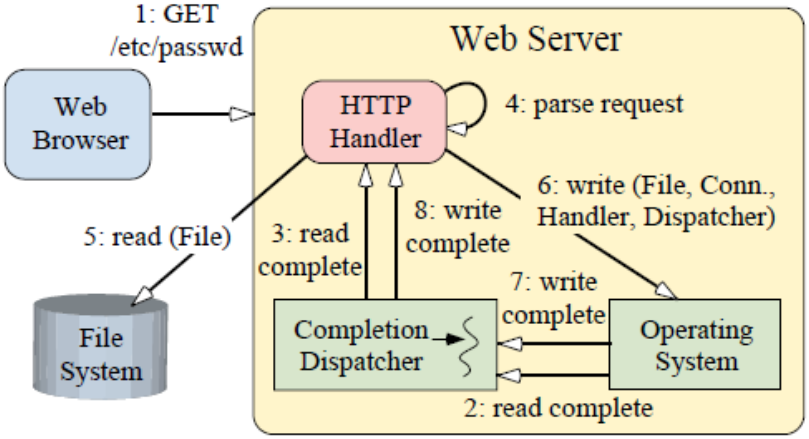
Reactor – Request Processing

1. The client sends an HTTP GET request;
2. The Initiation Dispatcher notifies the HTTP Handler when client request data arrives at the server;
3. The request is read in a non-blocking manner such that the read operation returns EWOULDBLOCK if the operation would cause the calling thread to block (steps 2 and 3 repeat until the request has been completely read);
4. The HTTP Handler parses the HTTP request;
5. The requested file is synchronously read from the file system;
6. The HTTP Handler registers the connection with the Initiation Dispatcher for sending file data (that is, when the connection becomes “ready for writing”);
7. The Initiation Dispatcher notifies the HTTP Handler when the TCP connection is ready for writing;
8. The HTTP Handler sends the requested file to the client in a non-blocking manner such that the write operation returns EWOULDBLOCK if the operation would cause the calling thread to block (steps 7 and 8 will repeat until the data has been delivered completely).

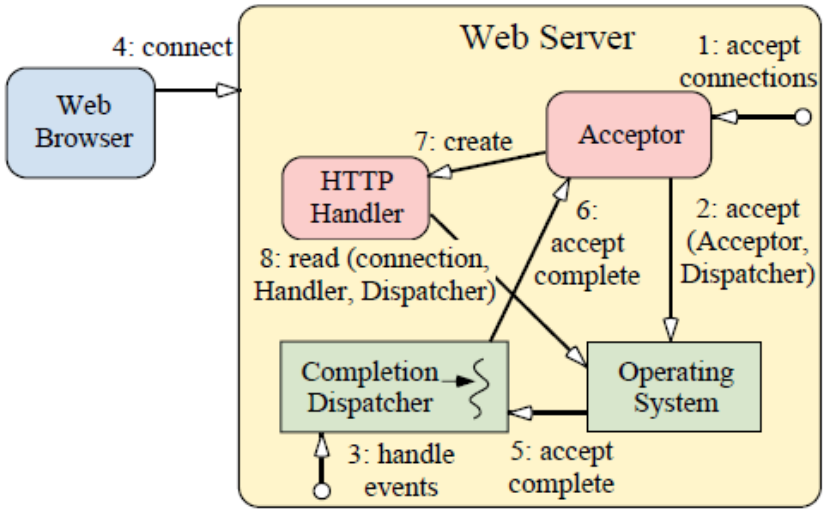


Proactor


4: connect
Web Browser
Web Server
1: accept connections
Acceptor
7: create
HTTP Handler
8: read (connection, Handler, Dispatcher)
6: accept complete
2: accept (Acceptor, Dispatcher)
Completion Dispatcher
Operating System
3: handle events
5: accept complete


1: GET /etc/passwd
Web Browser
Web Server
HTTP Handler
4: parse request
6: write (File, Conn., Handler, Dispatcher)
5: read (File)
3: read complete
8: write complete
7: write complete
File System
Completion Dispatcher
Operating System
2: read complete

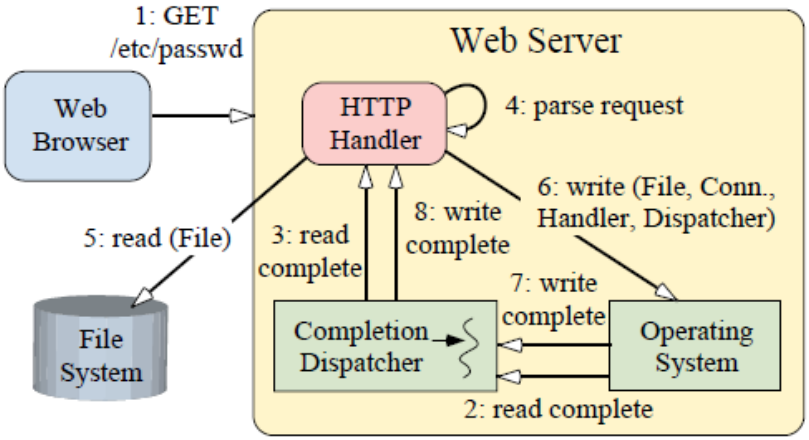
Proactor - Connection



1. The Web Server instructs the Acceptor to initiate an asynchronous accept;
2. The Acceptor initiates an asynchronous accept with the OS and passes itself as a Completion Handler and a reference to the Completion Dispatcher that will be used to notify the Acceptor upon completion of the asynchronous accept;
3. The Web Server invokes the event loop of the Completion Dispatcher;
4. The client connects to the Web Server;
5. When the asynchronous accept operation completes, the Operating System notifies the Completion Dispatcher;
6. The Completion Dispatcher notifies the Acceptor;
7. The Acceptor creates an HTTP Handler;
8. The HTTP Handler initiates an asynchronous operation to read the request data from the client and passes itself as a Completion Handler and a reference to the Completion Dispatcher that will be used to notify the HTTP Handler upon completion of the asynchronous read.

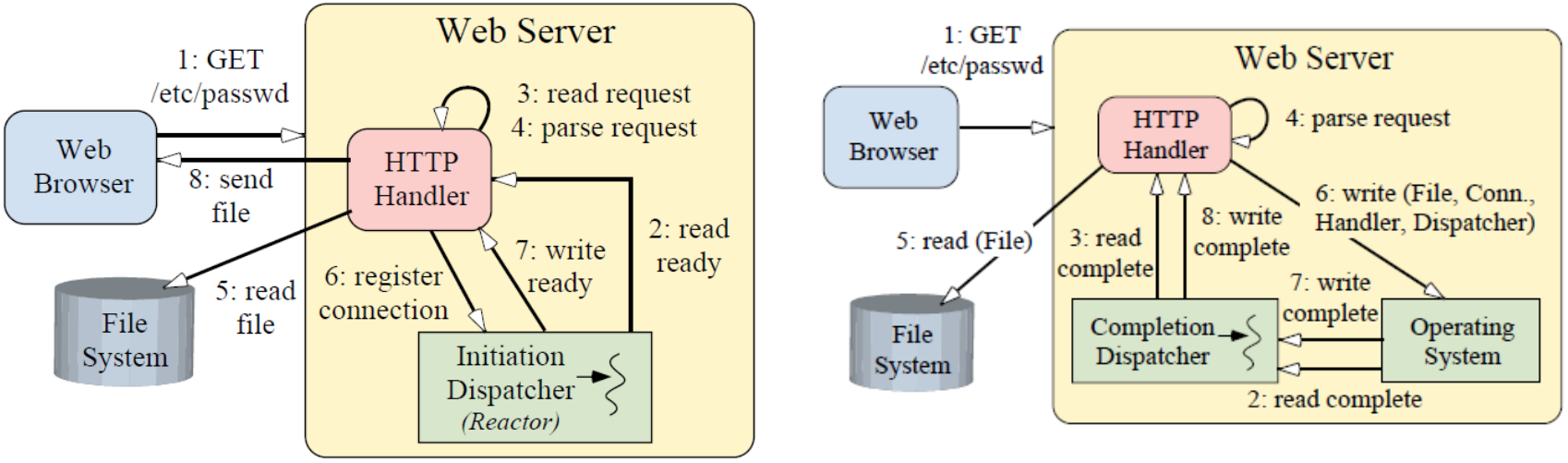
Proactor - Processing

1. The client sends an HTTP GET request;
2. The read operation completes and the Operating System notifies the Completion Dispatcher;
3. The Completion Dispatcher notifies the HTTP Handler (steps 2 and 3 will repeat until the entire request has been received);
4. The HTTP Handler parses the request;
5. The HTTP Handler synchronously reads the requested file;
6. The HTTP Handler initiates an asynchronous operation to write the file data to the client connection and passes itself as a Completion Handler and a reference to the Completion Dispatcher that will be used to notify the HTTP Handler upon completion of the asynchronous write;
7. When the write operation completes, the Operating System notifies the Completion Dispatcher;
8. The Completion Dispatcher then notifies the Completion Handler (steps 6-8 continue until the file has been delivered completely).

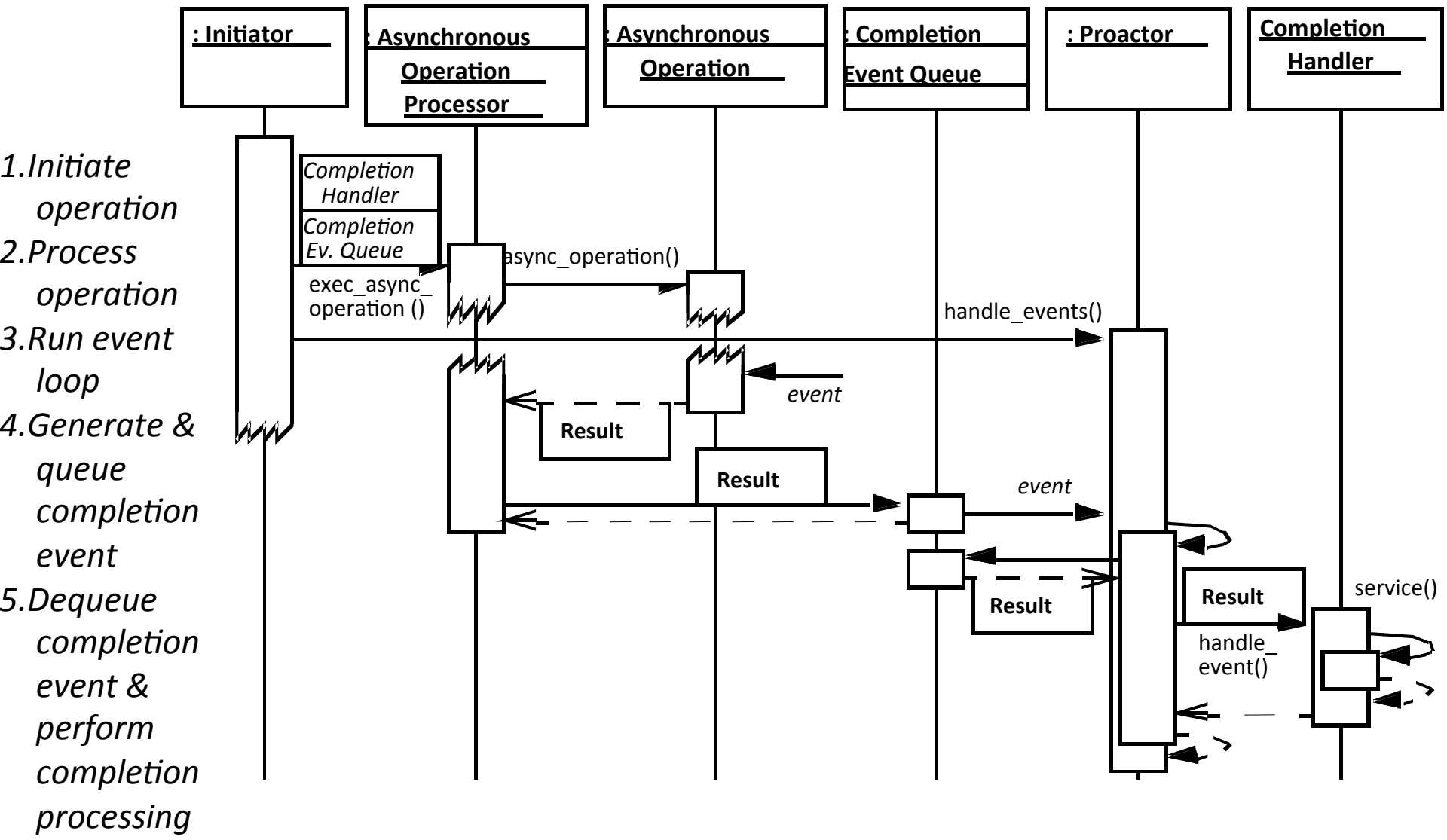


Reactor vs. Proactor

Processing connections in web server. Reactor (left) vs. Proactor (right)



Proactor (Details)



Distributing Workload

- Single server – baseline, obviously prone to failures
- Single server with standby backup
  - Activity/liveliness monitoring
  - Consistency Update
- Quality tiers:
  - **Cold**: HW ready and wired, no SW stack
  - **Warm**: Wired, installed, powered, with no state
  - **Hot**: State maintained, system operational, but not servicing request
- **Active-Active**: Load Balanced, parallel processing of requests

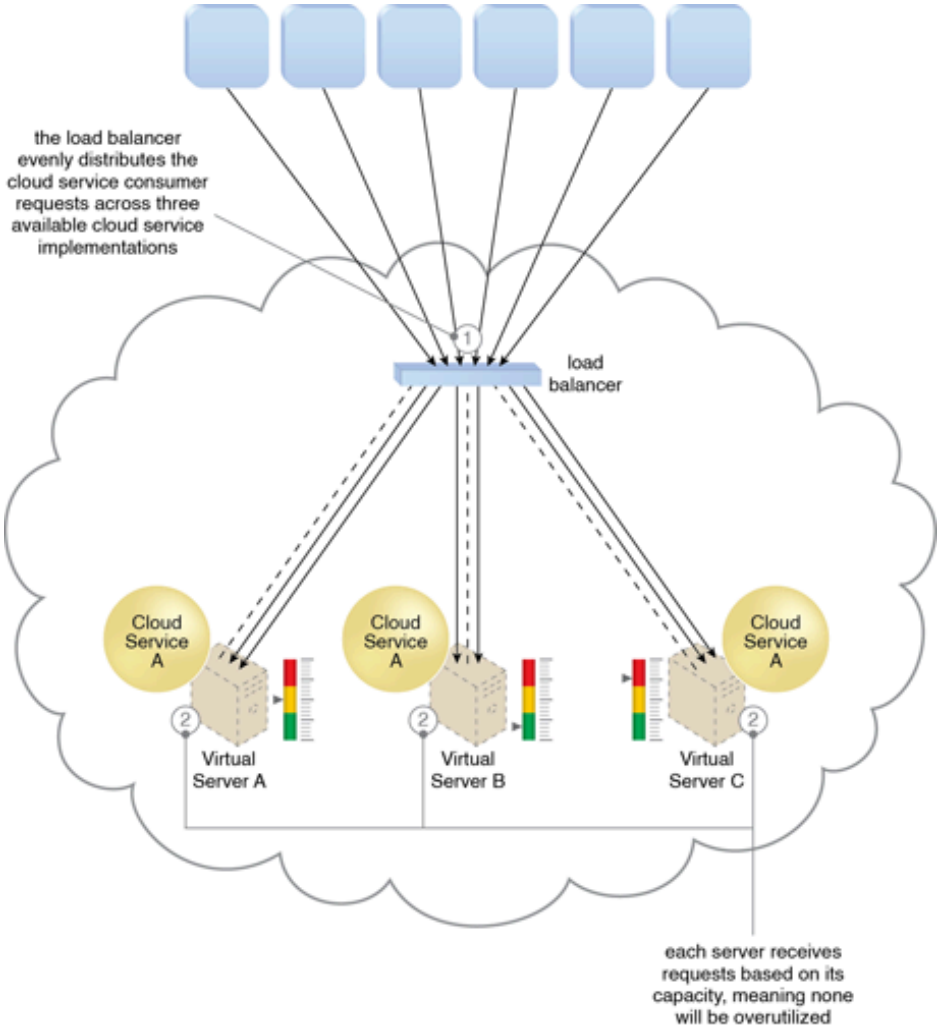
HW Cost Implications

Configuration	HW Config/requirements
Cold	N+1 (or N+x). Resources can be pooled. Failover to AWS/cloud possible.
Warm	N + 1 (or x) per each system tier, (web server, app server, db server)
Hot	2N (or more) – system duplicated
Active-Active	N + 1 (or N+x)

Why not use Active-Active all over?

- Load-balancing
- Easy for isolated, stateless requests
- More complex for sticky sessions
  - We need to maintain per session state

- Consistent distribution for long-term-stage
  - Individual duplication necessary for each **unique** resource



Corollaly

- Uniqueness is expensive
- Pooling is efficient and scalable
- State persistence/management incurs non-trivial costs and constraints

Conclusion

1. Remote resource localization
   1. Yellow pages, **Singleton**
2. Remote resource creation and usage
   1. **Factory**, **Façade, Reactor, Proactor, Half-Sync/Async**
3. State synchronization management
   1. **Façade**, **Proxy, Active Object**
4. Failure detection
5. Failure management, recovery and failover
6. Resource destruction
   1. **Proxy**, **Façade**, Garbage collection, …