Design Patterns [TDP]
A.A. 2017/2018

Reactor and Proactor

Examples of event handling patterns

Sara Vitellaro

Context

Focus on: **event handling programming**

Context: develop efficient and robust concurrent and networked applications

Main traditional and competitive *Applications designs*:

- **thread**-based
- **event**-driven

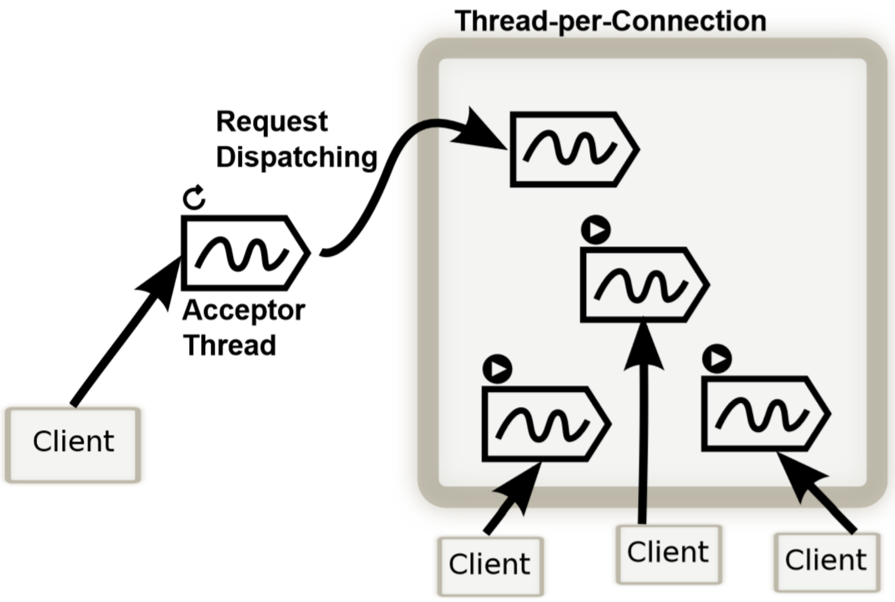
Concurrency strategies:

- blocking I/O
- non blocking synchronous
- non blocking asynchronous

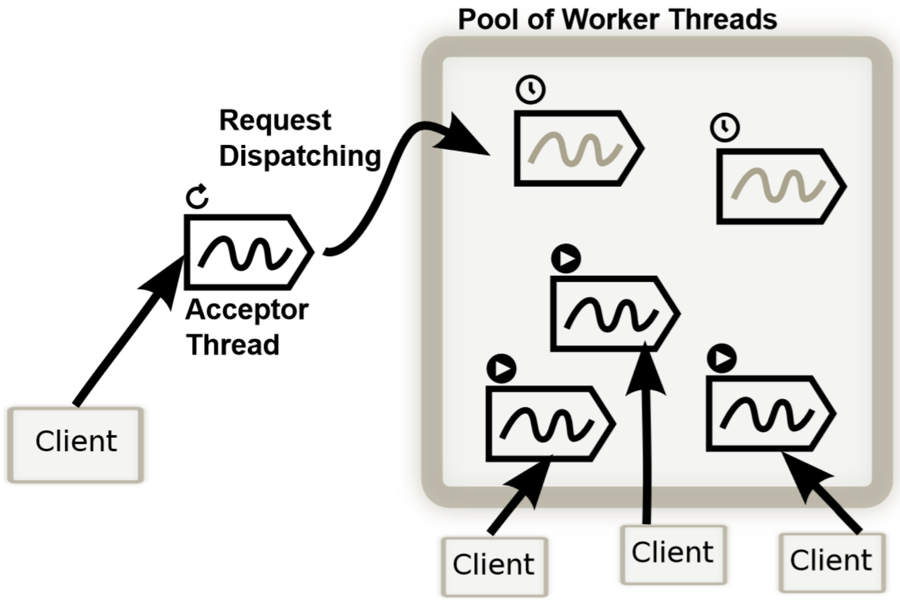
Context
Multi-threaded Architectures

Multiple threads can *synchronously* process multiple concurrent service requests: the application spawns *a dedicated thread of control* to *handle each* specific request.

- **Thread-per-connection**



- **Thread Pool**



Multi-threaded Architectures

Common drawbacks

– *increased performance overhead*:
context switching, synchronization, data movements

– *increased synchronization complexity*:
control schemes for the access to shared resources

– *threading and concurrency policy correlation*:
better to correlate threading strategy to available CPU resources

– *portability*:
different threading semantics in different operating systems

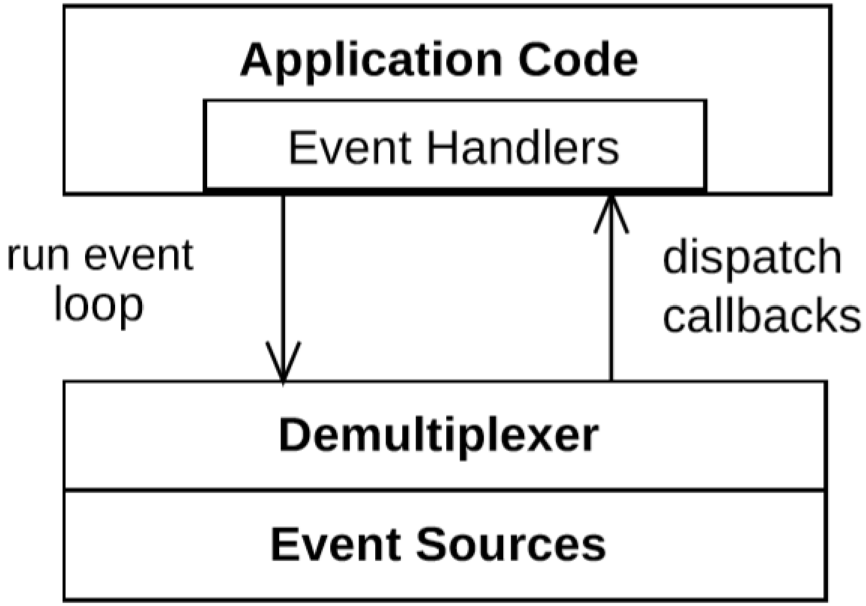
Context

Event-driven Architectures

- Designed as **Layered architecture**

- **Inversion of control**:
  event handlers perform
  application-specific operations
  in response to callbacks

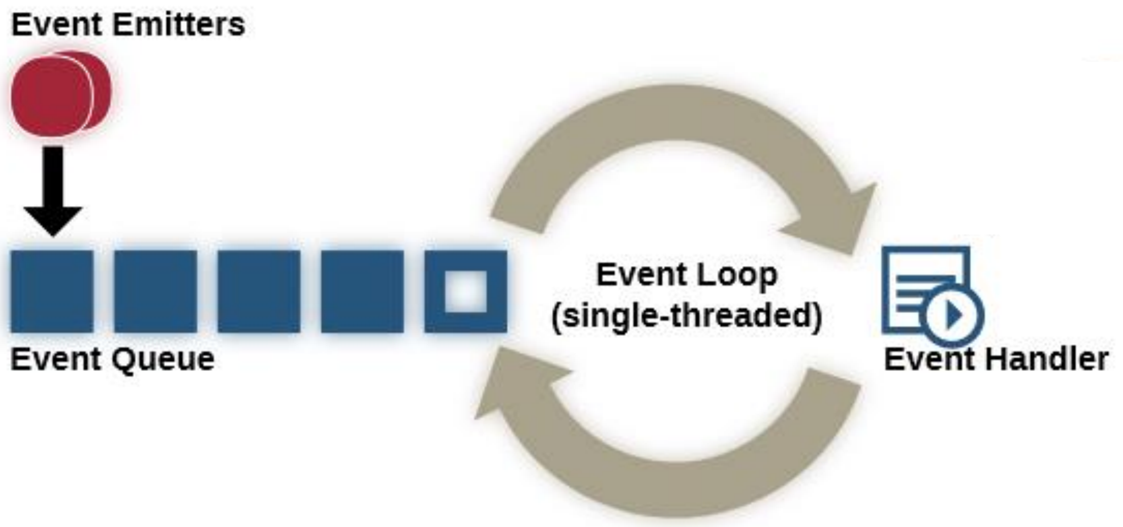
- **Separation of concerns**:
  between application functionalities
  and event-handling subsystem



Event-driven Architectures

Components of the event-driven Layered architecture:

- **Event sources**: detect and retrieve events
- **Demultiplexer**: waits for events to occur on the event sources set and dispatches them to their related event handlers callbacks
- **Event handlers**: executes application-specific operations in response to callbacks



Event-driven Architectures

Main **differences** from traditional ‘*self-direct*’ flow of control:

- the behaviour is **caused** by asynchronous events;
- most events have to be **handled promptly**;
- **finite state machines** to control event processing;
- no control over the **order** in which events arrive.

Reactor and Proactor patterns

Reactor and Proactor are two **event handling patterns** that propose *two different solutions* for concurrent applications.

They indicate how to effectively initiate, receive, demultiplex, dispatch and perform various types of events in networked software frameworks.

The *design* of these patterns provides **reusable** and **configurable** solutions and application components.

Reactor pattern

The Reactor pattern allows event-driven applications to demultiplex and dispatch synchronously and serially service requests that are received simultaneously from one or more clients.

- It waits for ***indication events*** to occur on some event sources.

  *Indication event: e*vent that identify the arrival of a request.

- *Non-blocking Synchronous I/O* strategy

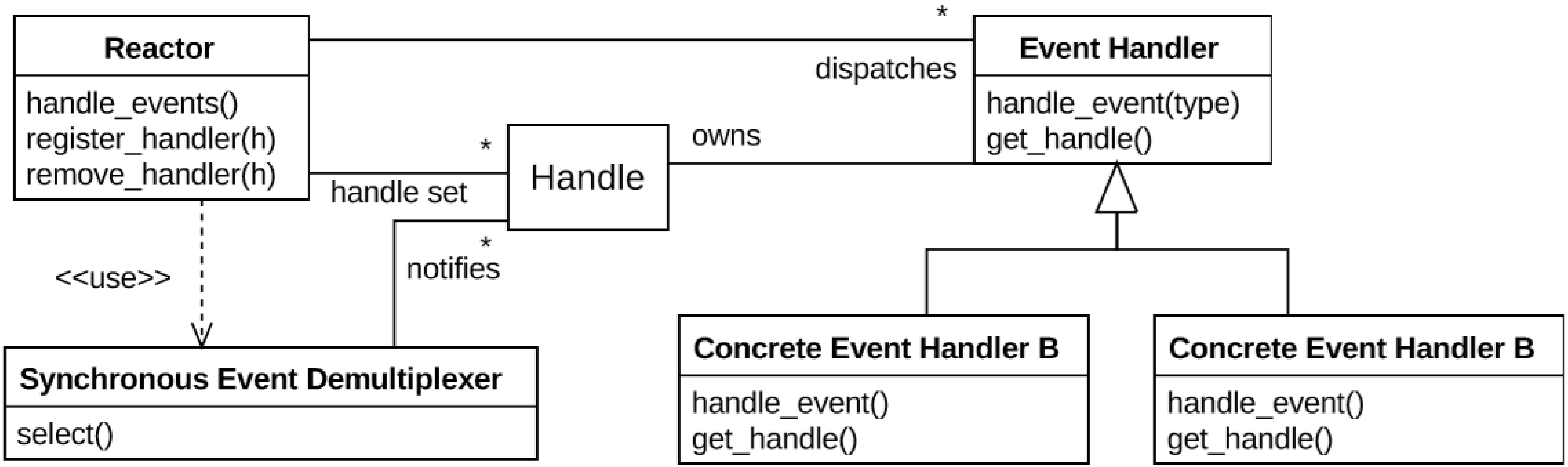
  the control is returned to the caller immediately: the call was performed and the results are ready, or the system has no resources to execute the requested operation.

Participants

- **Handle**: identifies event sources that can produce and queue indication events, from either external or internal requests
- **Event Handler**: defines an interface with a set of hook methods that represents the dispatching operations available to process events
- **Concrete Event Handler**: specializes the event handler for a particular service, and implements the hook methods
- **Reactor**: specifies an interface to register and remove event handlers and handles; runs the event loop to react to each indication event by demultiplexing it from the handle to the event handler and dispatching the proper hook method
- **Synchronous Event Demultiplexer**: function that blocks awaiting indication events to occur on a set of handles

Reactor

Structure



Dynamics

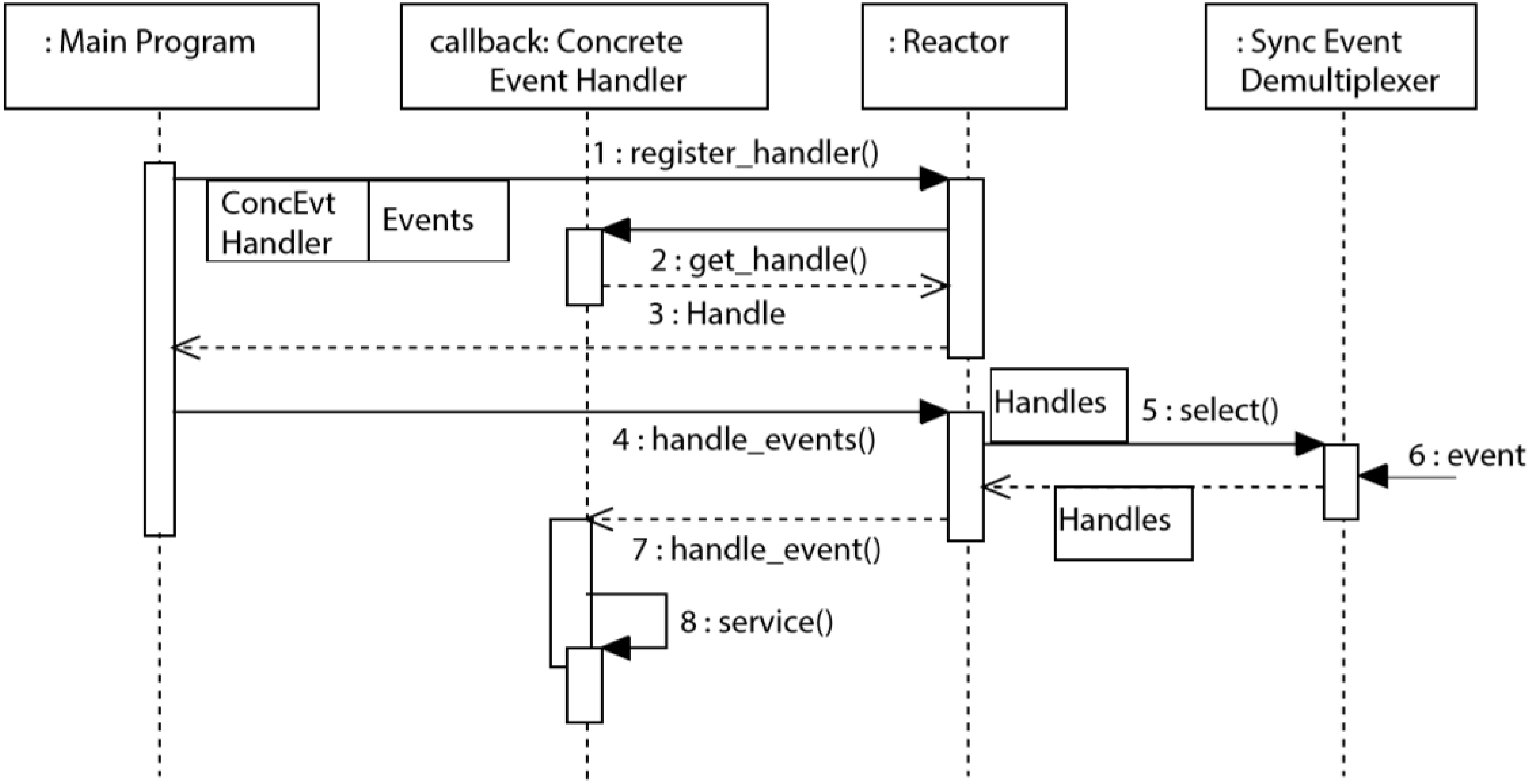
In the Reactor pattern the **flow of control alternates** between the Reactor and the Event Handler components:

- the Reactor is responsible to **wait** for *indication events*, demultiplex and dispatch them;
- the Event Handlers components **react** to the occurrence of a specific event to process it.

The structure introduced by the Reactor pattern '**inverts**' the flow of control within an application, in a way called “***Hollywood principle***”.

Reactor

Dynamics



Reactor

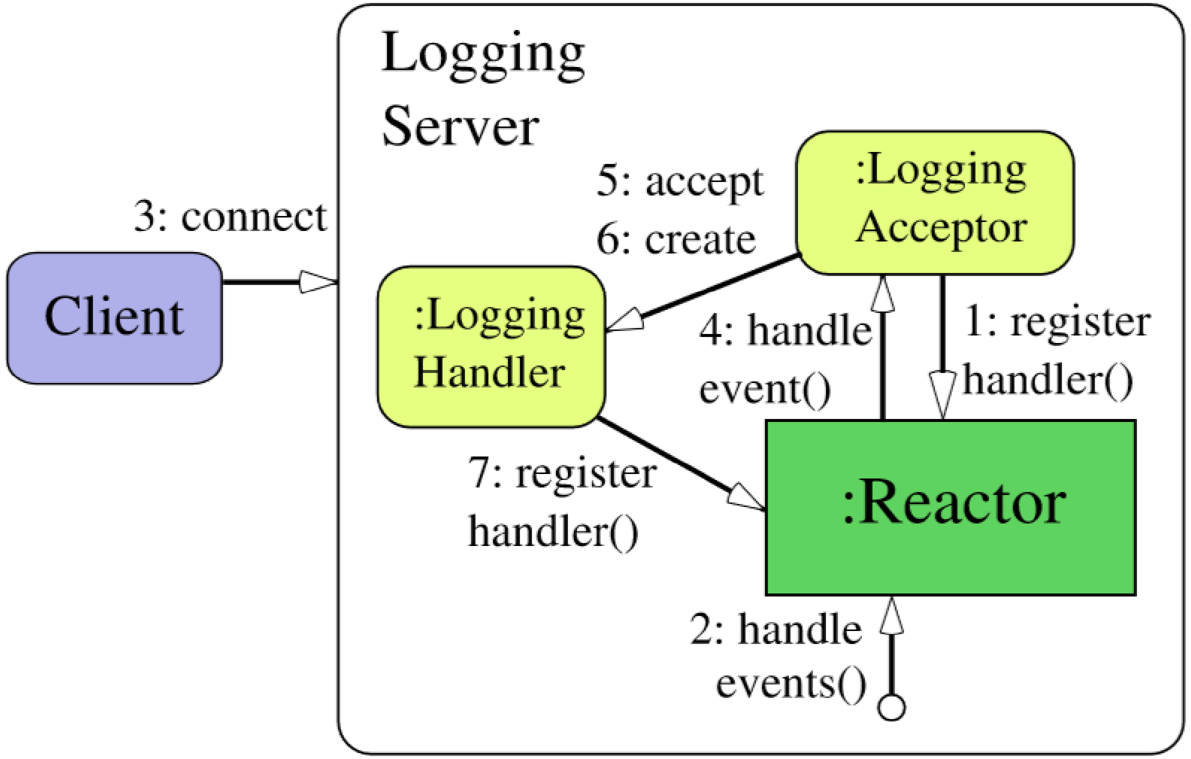
A simple example

Telephony scenario:

- Telecommunication network -> *Reactor*
- Client -> *Event Handler*
  register himself to it to ‘handle’ a call received on his phone number
- Phone number -> *the Handle*
- Somebody calls the number -> *incoming indication event*
- the network *notifies* the client that a request event is pending, making the phone ring -> *demultiplex and dispatch*
- the client *reacts* by picking up the phone and ‘*processes*’ the request answering to the connected part -> *specific handle_event()*

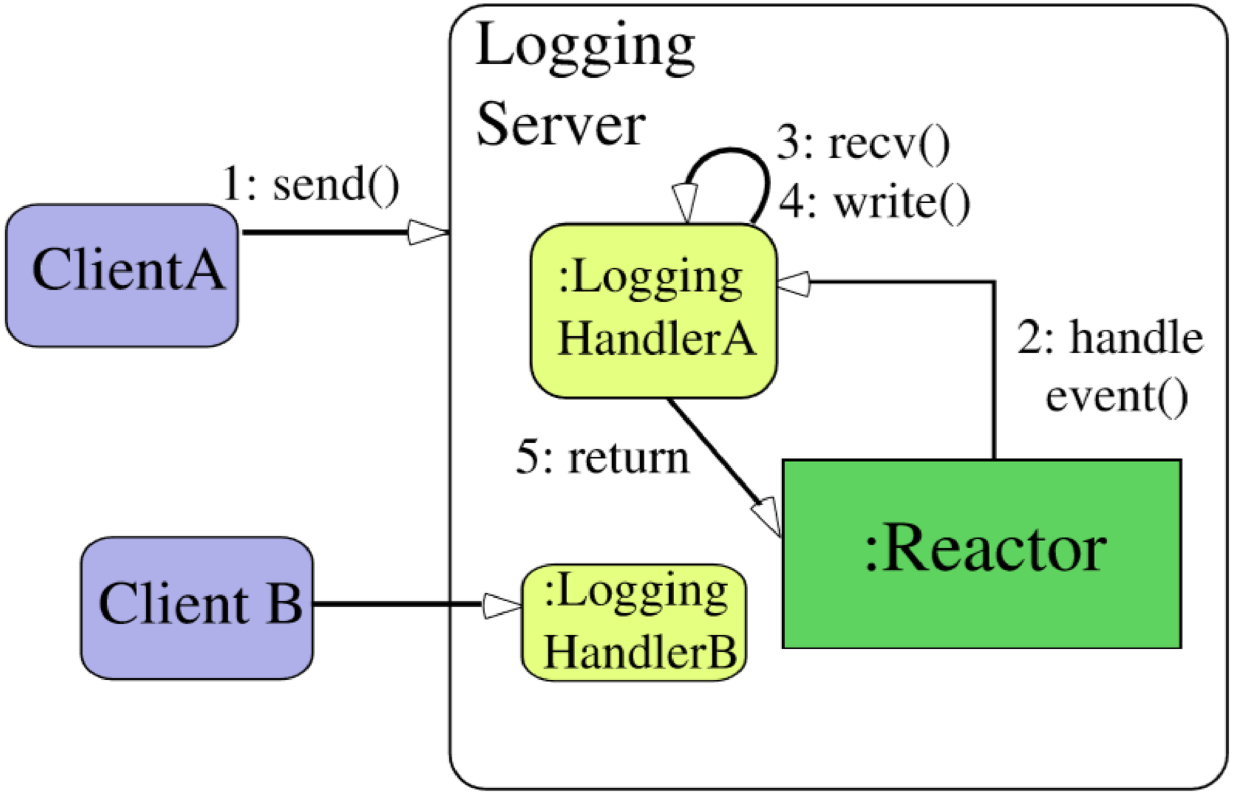
Reactor

Scenario: reactive logging server (1)



1. The Server register the Logging Acceptor Handler to handle client connection requests indication events;
2. the Server calls the method to start the event loop in the Reactor. It calls the Synchronous Event Demultiplexer to wait for connections;
3. a Client tries to connect to the Server;
4. the Reactor notifies the Logging Acceptor,
5. accepts the new connection,
6. and creates a Logging Handler to serve it;
7. the Logging Handler registers its handle with the Reactor to be notified when it is ‘ready for reading’;

Reactor
Scenario: reactive logging server (2)



1. a Client sends a logging record request. The Synchronous Event Demultiplexer notifies the Reactor that an indication event is pending on a handle in its handle set;
2. the Reactor notifies the Logging Handler associated with this handle;
3. the Logging Handler begins to receive the record in a non-blocking manner (loop 2-3);
4. when the reception is completed, the Logging Handler processes the records and writes it to the appropriate output;
5. the Logging Handler returns control to the Reactor events loop to continue to wait for incoming indication events.

Implementation

- Event handler interface
  - create an event handler object
  - or register a pointer to a function

- Dispatch interface strategy
  - single method, with type parameter
  - or multi-method, with several different hooks

- Concrete Event Handlers
  - executes operations on a handle
  - maintains all useful state information associated with the request
  - can be subdivided by functionality, into connection and service ones

Reactor

Implementation

- Reactor interface
  - registers and remove handlers and handles
  - and runs the application reactive event loop
- Demultiplexing table
  - stores tuples that associates handles and event handlers
  - uses handles as a ‘key’
  - various possible search strategies
- Determine the number of Reactors needed
  - centralize the work on a single Reactor instance
  - or multiple Reactor threads: event handlers in parallel, but needs additional synchronization mechanisms
- Choose a Synchronous Event Demultiplexing mechanism
  - often an existing operating system mechanism

Types of demultiplexing mechanism

- **select()**: portable but inefficient with O(n) descriptor selection, limited to 1024 descriptors, stateless
- **poll()**: allows more fine-grained control of events, but still O(n) descriptor selection, stateless
- **epool()**: keeps info, dynamic descriptor set, efficient with O(1) descriptor selection, only on Linux platforms
- **kqueue()**: more general mechanism, O(1) descriptor selection, only on OS X and FreeBSD systems
- **WaitForMultipleObjects()**: works on multiple types of synchronization objects, only on Windows

→ **ACE** and **Boost** libraries supply a common interface to choose the *best* Reactor implementation *depending on the execution platform support*

Variants

- **concurrent Event Handlers**: to improve performance, event handlers can run on their own thread, instead of borrowing the Reactor thread;
- **concurrent synchronous event demultiplexers**, called on the handle set by multiple threads, to improve throughput;
- **re-entrant Reactors**: event loop called by reactive Concrete Event handlers;
- **integrated demultiplexing of Timer and I/O events**: allow applications to register time based event handlers.

Reactor

Benefits and Liabilities

+ increase **separation** of concerns
+ improve **modularity**, **reusability** and **configurability**
+ improve application **portability**
+ low overhead for **concurrency control**

- **non pre-emption** → Proactor
- **scalability** → Proactor
- **complexity** of debugging and testing

Proactor pattern

The Proactor pattern allows event-driven applications to demultiplex and dispatch service requests in an efficient asynchronous way.

- It waits for ***completion events*** to occur on some event sources.

  Completion event: event that identify the end of the execution of an asynchronous operation.

- *Non-blocking Asynchronous I/O* strategy

  the control is returned to the caller immediately, indicating that the requested operation was initiated.
  The called system will notify the caller.

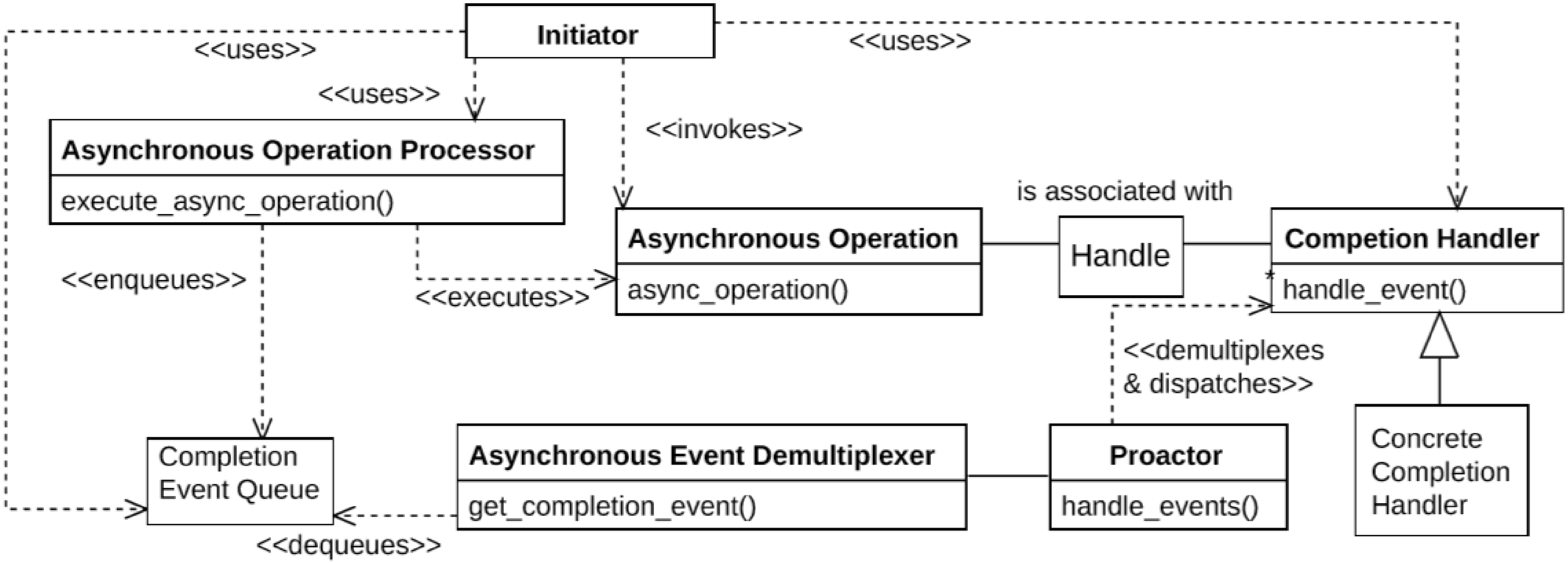
Proactor
Participants

- **Handle**: identifies event sources that can generate completion events, from either external or internal requests
- **Completion Handler**: defines an interface with a set of hook methods for the operations available to process results of asynchronous operations
- **Concrete Completion Handler**: specializes the completion handler for a particular service, and implements the hook methods
- **Proactor**: provides the application’s event loop, demultiplexes completion events to the related completion handers, and dispatches hook methods to process the results
- **Asynchronous Event Demultiplexer**: function that blocks awaiting completion events to be added to a completion queue, and returns them to the caller

Participants

- **Completion Event Queue**: buffers completion events while they are waiting to be demultiplexed to the respective completion handlers
- **Asynchronous operations**: represent potentially long-duration operations that are used to service on behalf of application
- **Asynchronous Operation Processor**: executes asynchronous operations invoked on handles, generates the respective completion event, and queues it
- **Initiator**: entity local to the application, initiates an asynchronous operation, registers a completion handler and a Proactor with an asynchronous operation processor, which notifies it when operations completes

Proactor
Structure



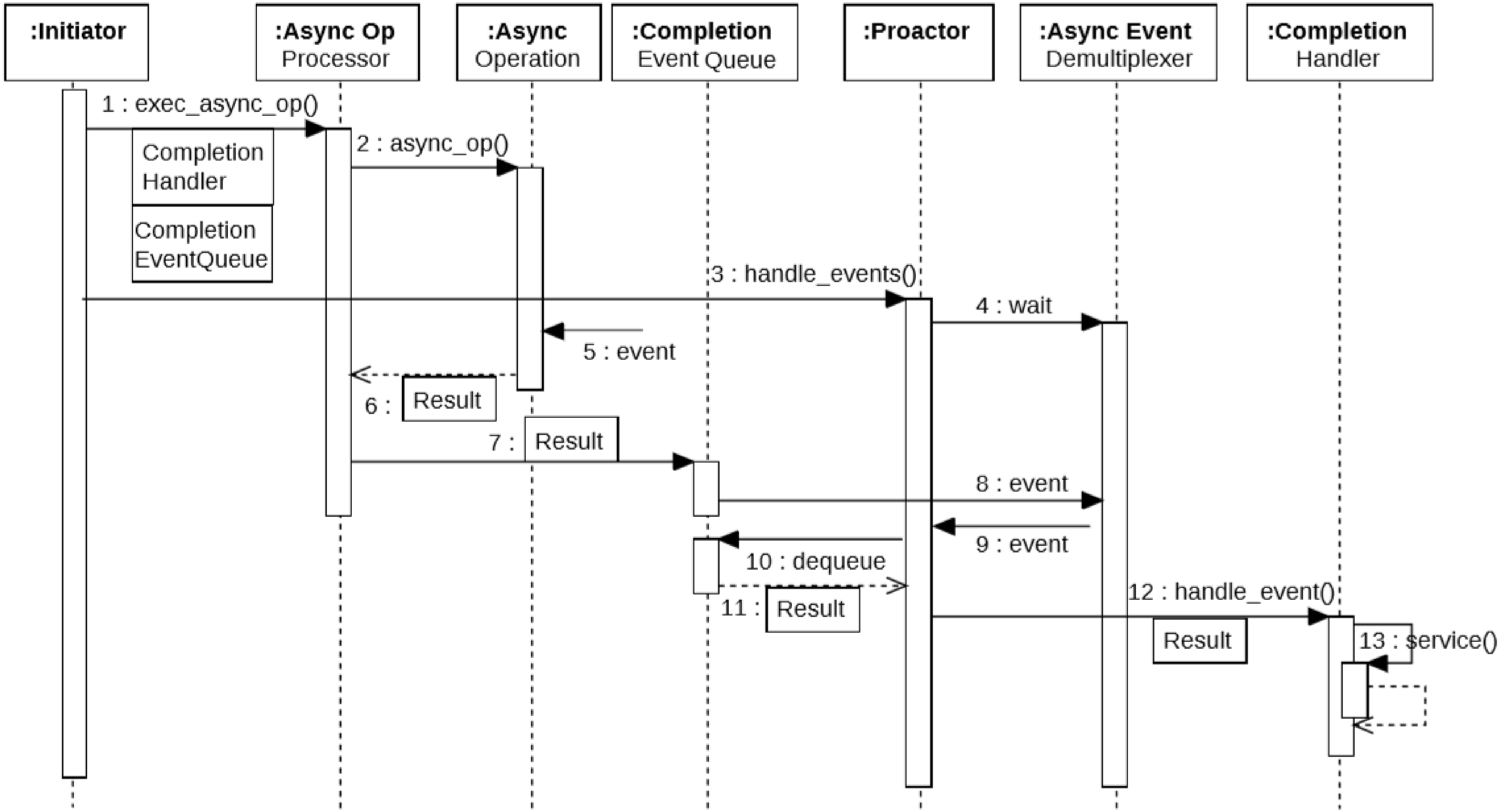
Proactor
Dynamics

In the Proactor pattern, at a high level of abstraction, applications *invoke* operations asynchronously and are *notified* about their completion.

The Proactor solution proposes to split every application service into:

- *long-duration* operations, that execute *asynchronously*;
- *completion handlers*, that *processes* the results of the associated asynchronous operations, potentially invoking additional asynchronous operations.

Proactor
Dynamics

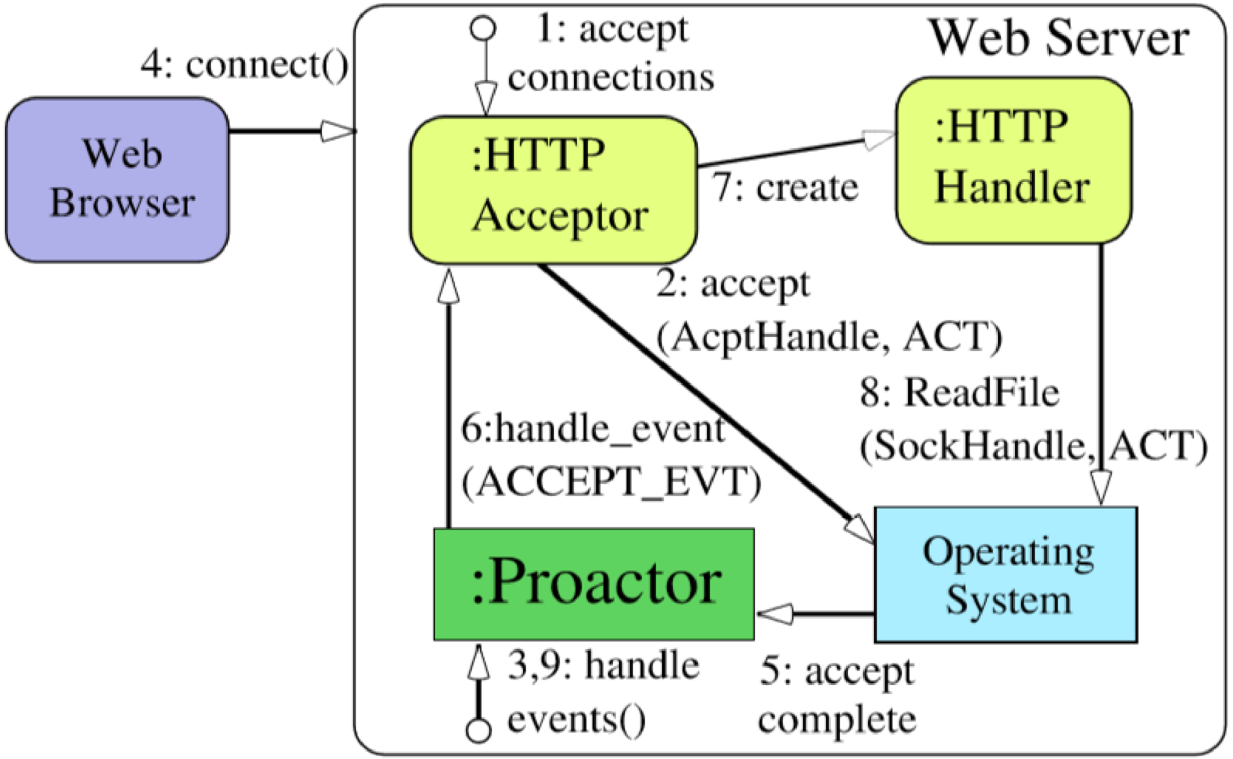


A simple example

Telephony scenario:

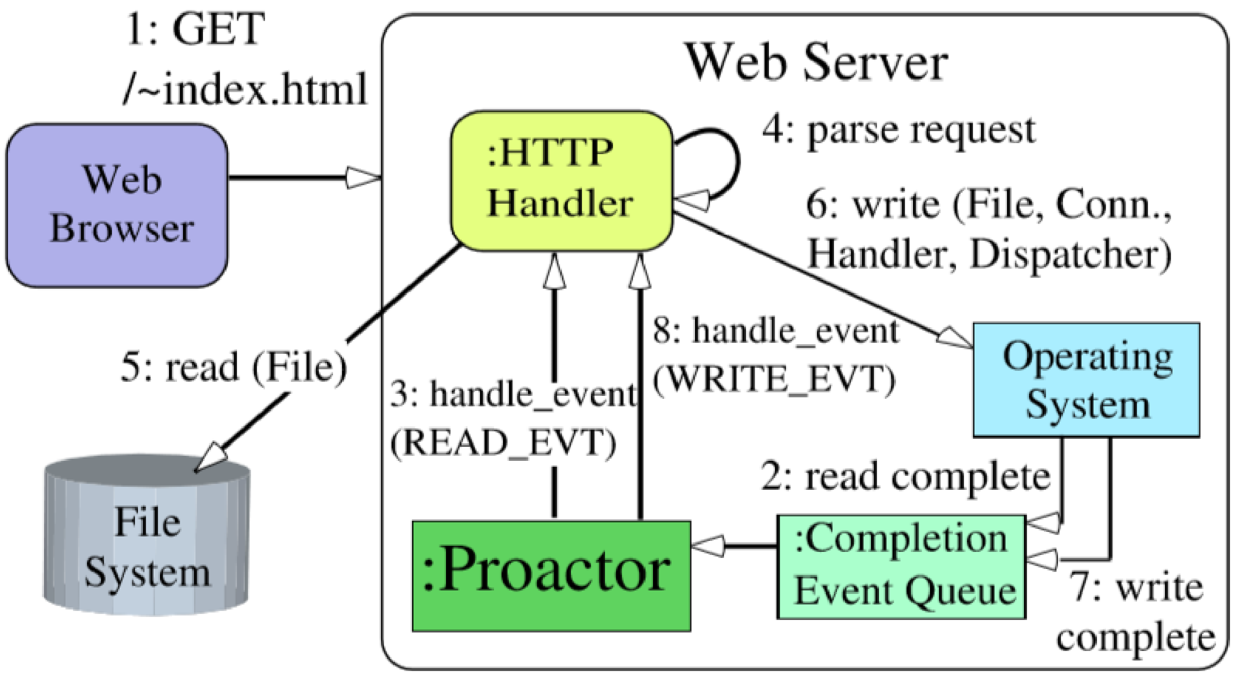
- *You* call a friend -> *Initiator*
- but he cannot answer. You leave a message on his *voice mail*
  -> *asynchronous operation processor*
- While waiting for the call-back, you can do other things.
- Your friend *listen* to the voice mail
  -> *completion event of the asynchronous operation*
- *he* calls you back -> *Proactor*
- *you* talk together -> *Completion Handler, specific handle_event()*

Proactor
Scenario: proactive Web server (1)



1. The Server invokes a method to initiate an asynchronous accept;
2. the Acceptor starts an asynchronous accept with the operating system;
3. the Server invokes the Proactor's event loop;
4. a client tries to connect;
5. the Asynchronous Operation Processor serves the request, and generate and insert the accept completion event in the queue;
6. the Asynchronous Event Demultiplexer dequeues the completion event, and the Proactor dispatches the relative hook method;
7. the Acceptor creates a Handler;
8. this Handler initiates an asynchronous read operation to obtain the request data sent;
9. control returns to the Proactor's event loop;

Proactor
Scenario: proactive Web server (2)



1. a connected client sends a GET request;
2. the (prev.) read asynchronous operation completes and is queued by the OS;
3. the Asynchronous Event Demultiplexer dequeues the completion event, returns it to the Proactor that dispatches the relative hook method;
4. when the entire request has been received, the Handler parses the request,
5. reads the requested file form the memory (can be asynchronous too),
6. and initiates an asynchronous write operation to transfer the file data to the client;
7. the OS queues a write completion event;
8. the Asynchronous Event Demultiplexer dequeues the completion event, returns it to the Proactor that dispatches the relative hook method.

Proactor

Implementation

- Completion handler interface
  - create a completion handler object
  - or register a pointer to a function;
- Dispatch interface strategy
  - single method, with type parameter
  - or multi-method, with several separate hooks
- Concrete Completion Handlers
  - maintains all useful state information associated with the request
  - can be subdivided by functionality, into connection and service ones
  - stores a pointer to a Proactor to invoke asynchronous operations themselves
- Implement the Asynchronous Operation Processor
  - Asynchronous Completion Token pattern to collect all useful information
  - maximize portability and flexibility

Proactor
Implementation

- Proactor interface
  - runs the application event loop to dequeue, demultiplex and dispatch completion events
  - provides a method to associate a handle to a particular event queue

- Implement Proactor interface
  - choose the completion event queue
  - choose the asynchronous event demultiplexing mechanism
  - determine how to demultiplex and dispatch completion events

- Determine the number of Proactors needed
  - centralize the work on a single Proactor instance
  - or multiple Proactor threads for run-time diversification

- Implement the initiator
  - used to initiate asynchronous operations service processing

Proactor

Variants

- **Asynchronous Completion Handlers**: to improve performance, completion handlers could act as initiators and invoke long-duration asynchronous operations;
- **concurrent asynchronous event Demultiplexer**: a pool of threads that share an asynchronous event Demultiplexer, particularly scalable;
- **shared Completion handlers**: multiple asynchronous operations initiated simultaneously can share the same concrete completion handler;
- **Asynchronous operation Processor emulation**: in operating system platforms that do not export asynchronous operations to applications.

Proactor

Benefits and Liabilities

+ increase **separation of concerns**
+ improve application **portability**
+ **encapsulate concurrency** mechanisms
+ concurrency policy **independent** from threading policy
+ **increase** performance
+ **simplify** application **synchronization**

- **no control** over **scheduling** of **operations**
- **efficiency** depends on the platform
- **complexity** of **debugging** and **testing**

Reactor and Proactor
Comparison

Both approaches can be used for ***event-driven programming***.

In the Reactor pattern, with *non-blocking* operations you *wait* until an operation *can complete immediately* before attempting to perform it.

In the Proactor pattern, you *start* operations that are *performed asynchronously*, and then you are *notified* when they are *completed*.

Difference (again):

- using Reactor, a program waits for the *event* of a socket being *readable* and then reads from it;
- using Proactor, the program instead waits for the *event* of a socket *read completing*.

Reactor and Proactor

Known uses

Various libraries and implementations have been developed to **abstract** from the differences among operating systems and provide **alternatives** to satisfy applications performance requirements:

- **ACE framework**: portable ACE Reactor and ACE Proactor

  UniPi project *ASSIST*: a programming environment for parallel and distributed programs that uses ACE Reactor library to perform concurrency and communication handling.

- **Boost.Asio library**: offers side-by-side support for synchronous and asynchronous operations, based on the Proactor pattern

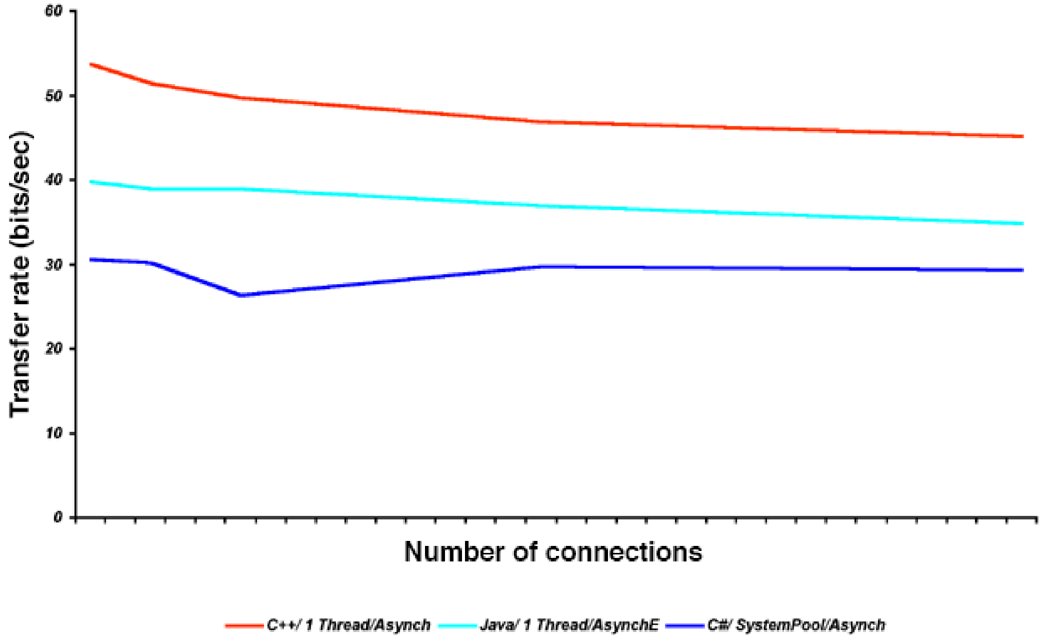
- **TProactor**: emulated Proactor

Known uses: TProactor

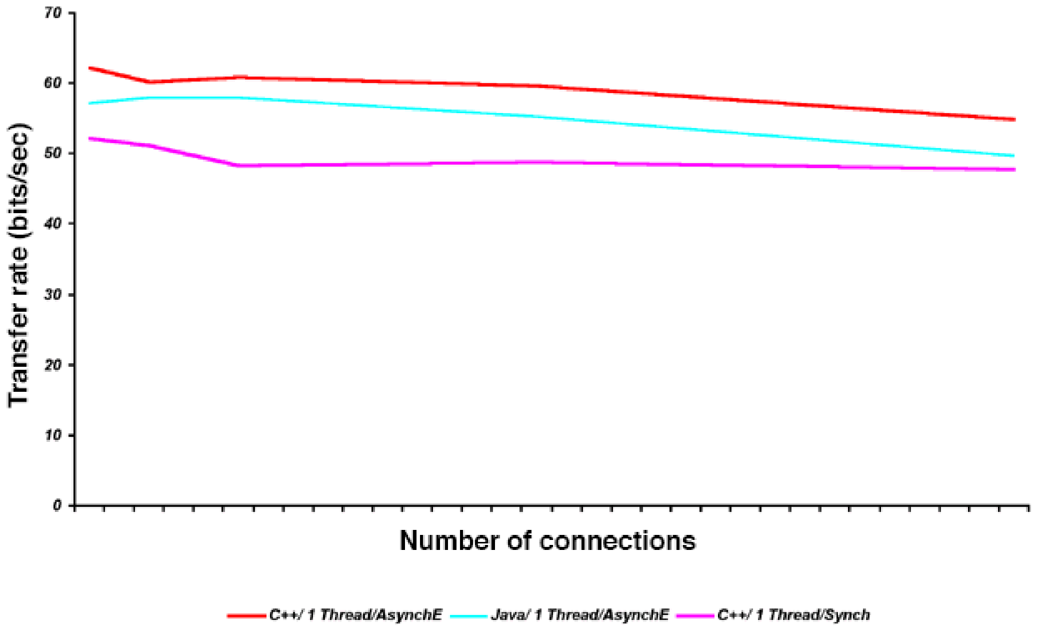
High configurable and full portable platform-independent solution.

Emulated Proactor: hides the reactive nature of available APIs and exposes a common fully proactive asynchronous interface.

Performance comparison



Executions on Windows



Executions on Linux RedHat

Reactor and Proactor

Some code

Java has abstracted out the *differences* between platform specific system call implementations with its **NIO** and **NIO.2** API.

- https://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html
- https://www.ibm.com/developerworks/java/library/j-nio2-1/index.html

Java code references: simple echo server

- https://www.javacodegeeks.com/2012/08/io-demystified.html

Some real world projects and uses:

- Spring Project
- Node.js

Biblio

- **Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, volume 2**; Douglas C Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann; 2000
- *Reactor - an Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Synchronous events*; Douglas C Schmidt; 1995
- *Reactor Pattern explained*, Tech Stuff; 2013
- *Proactor- an Object Behavioral Pattern for Demultiplexing and Dispatching handlers for Asynchronous events*; Irfan Pyarali, Tim Harrison, Douglas C Schmidt, Thomas D Jordan; 1997
- *Proactor Pattern: release the power of asynchronous operations*, Alexey Shmalko; 2014
- *Comparing two high performance I/O design Patterns*;
  Alexander Libman, Vladimir Gilbourd; 2005
- *The implementation of ASSIST, an environment for parallel and distributed programming*;
  Marco Vanneschi, Massimo Torquati, et al.; 2003