Patterns & Frameworks for Asynchronous Event Handling: Part 1

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

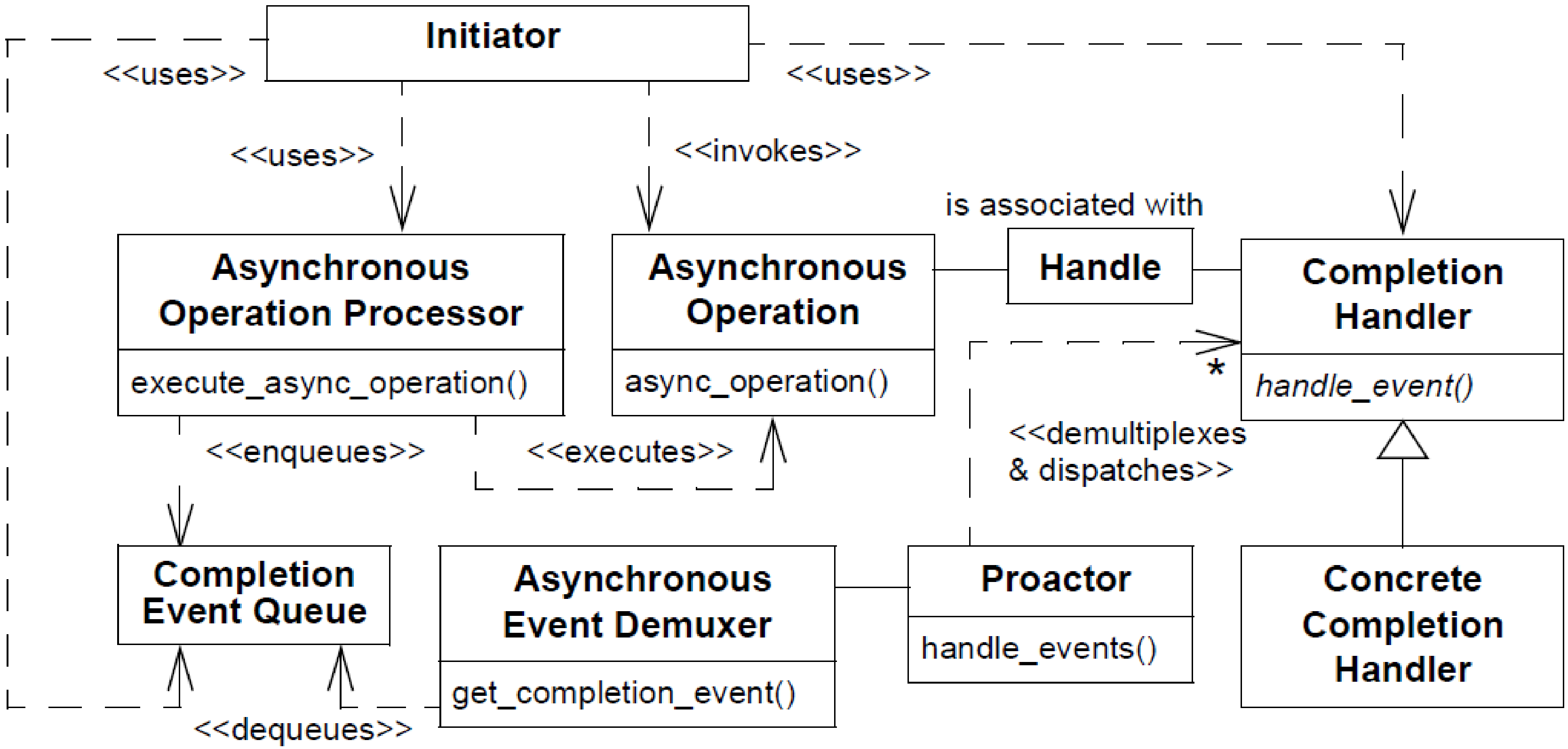
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

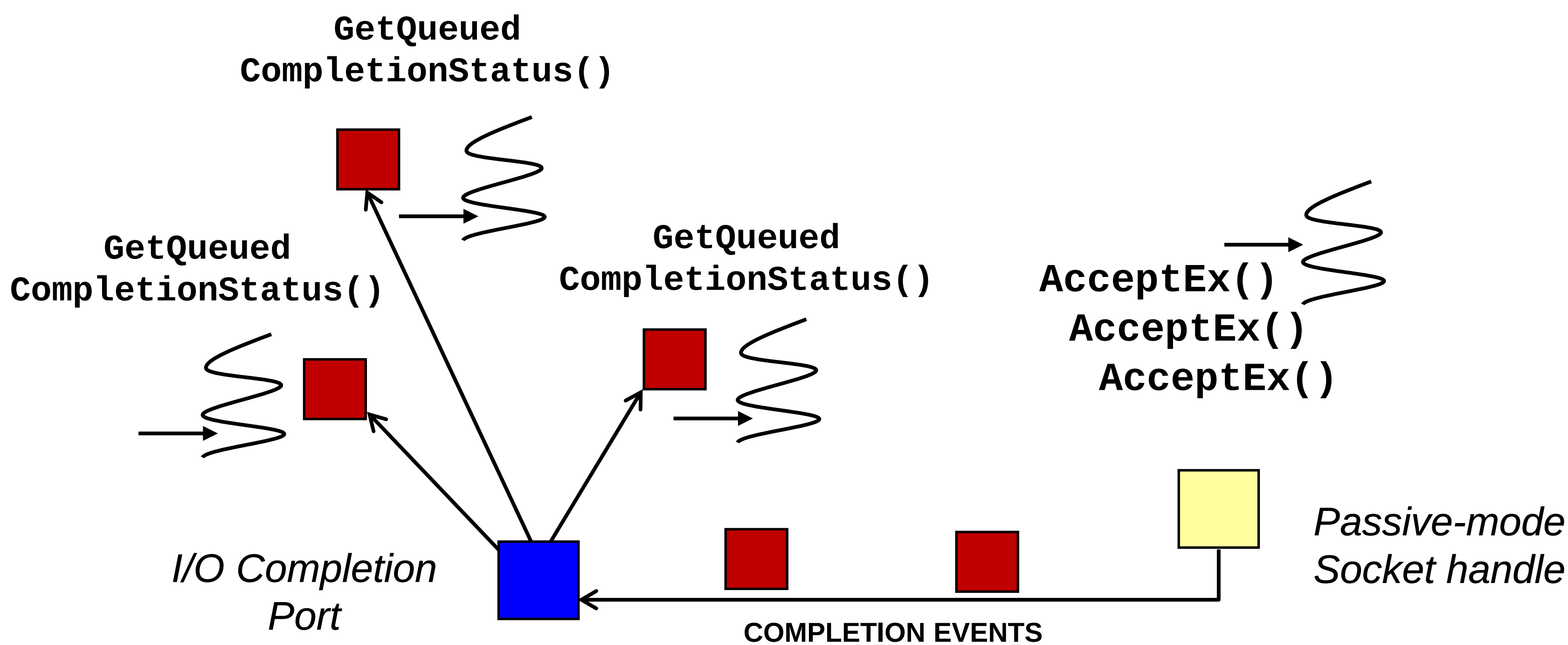
Topics Covered in this Part of the Module

- Describe the *Proactor* pattern



Using Asynchronous I/O Effectively

Context	Problem
• Synchronous event handling & multi-threading may not achieve most scalable web server when OS supports async I/O	• Leveraging efficiency & scalability of async I/O is hard due to time/space separation of async operation invocations & their subsequent completion events



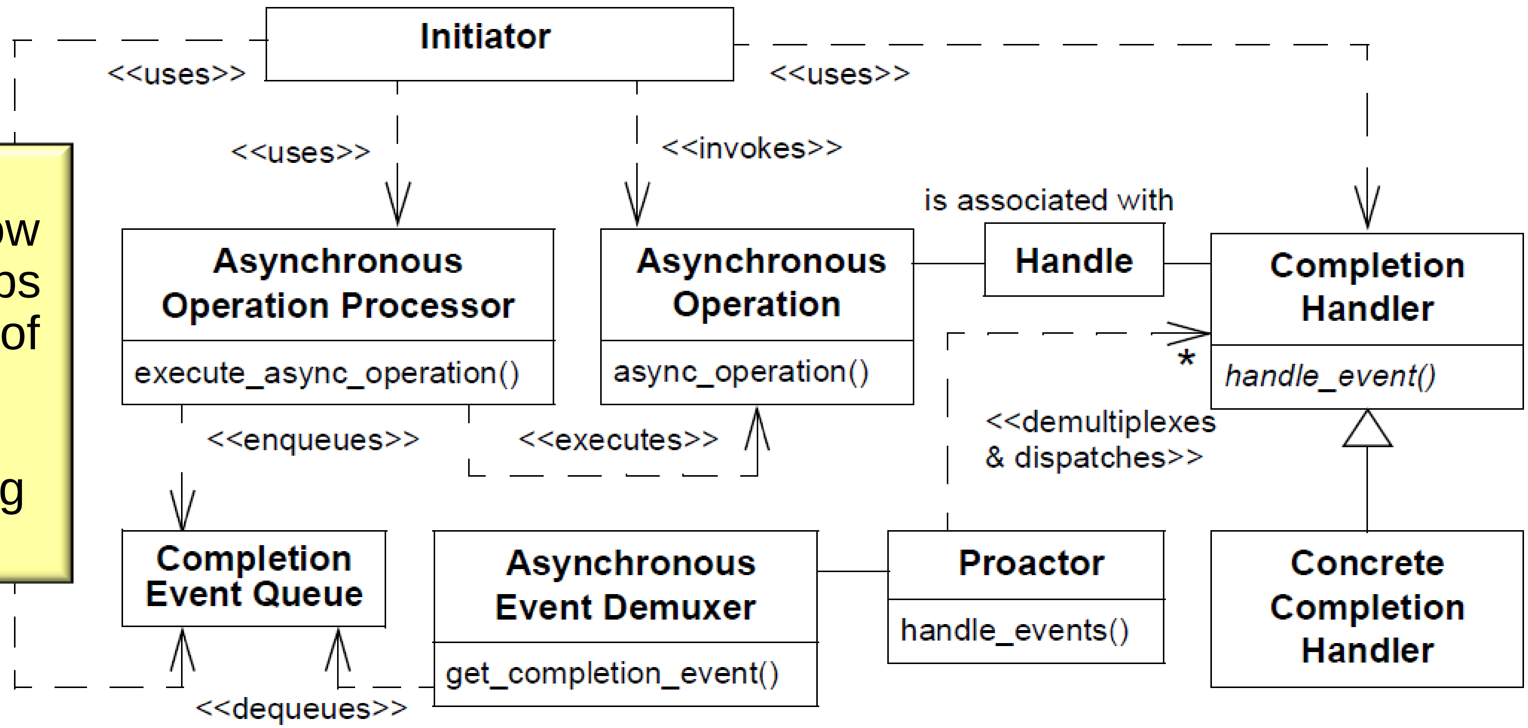
See en.wikipedia.org/wiki/Input/output_completion_port for more info

Using Asynchronous I/O Effectively

Context	Problem	Solution
• Synchronous event handling & multi-threading may not achieve most scalable web server when OS supports async I/O	• Leveraging efficiency & scalability of async I/O is hard due to time/space separation of async operation invocations & their subsequent completion events	• Apply the *Proactor* pattern to efficiently demux async I/O operations

Structure

Proactor uses async I/O to allow event-driven apps to gain benefits of concurrency performance without incurring its liabilities

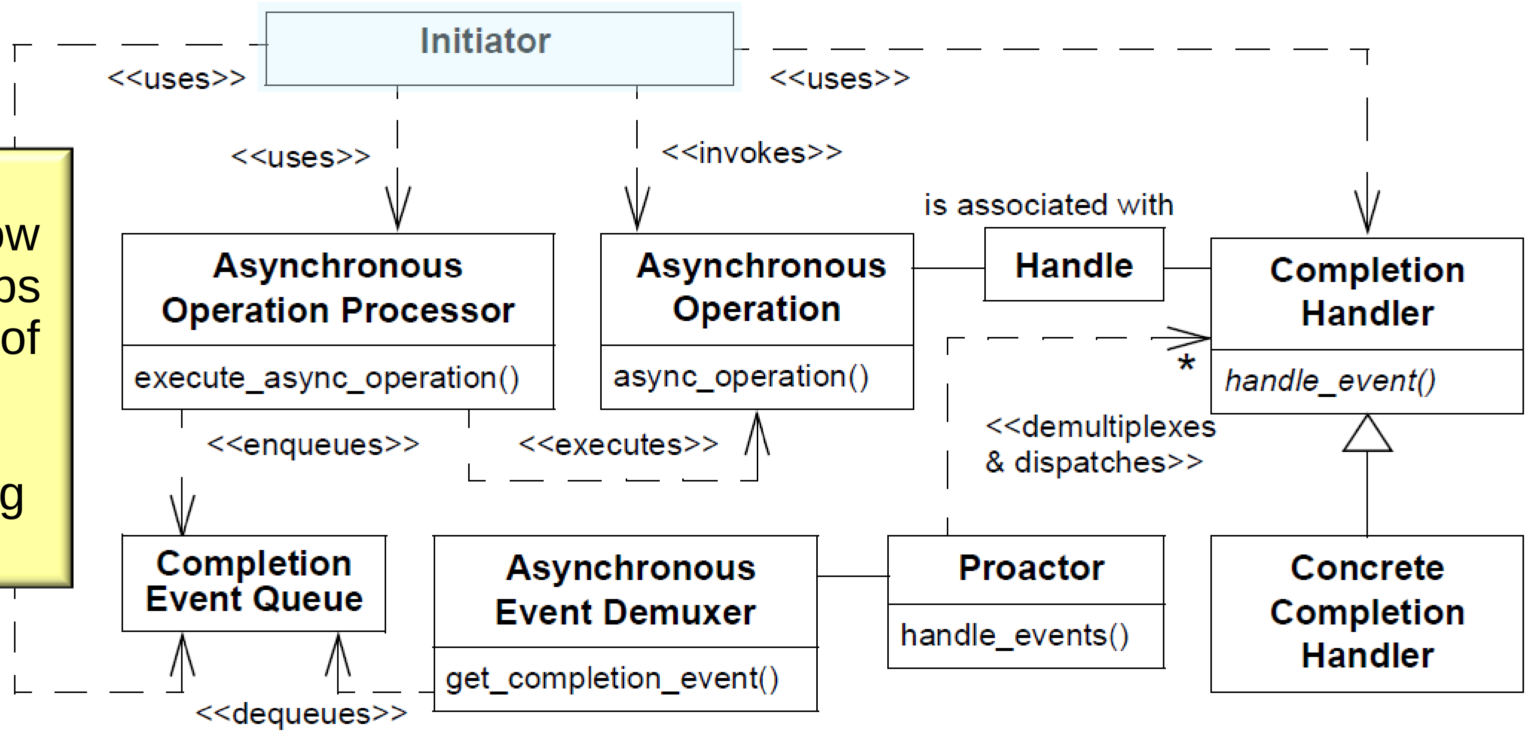


Using Asynchronous I/O Effectively

Context	Problem	Solution
• Synchronous event handling & multi-threading may not achieve most scalable web server when OS supports async I/O	• Leveraging efficiency & scalability of async I/O is hard due to time/space separation of async operation invocations & their subsequent completion events	• Apply the *Proactor* pattern to efficiently demux async I/O operations

Structure

Proactor uses async I/O to allow event-driven apps to gain benefits of concurrency performance without incurring its liabilities

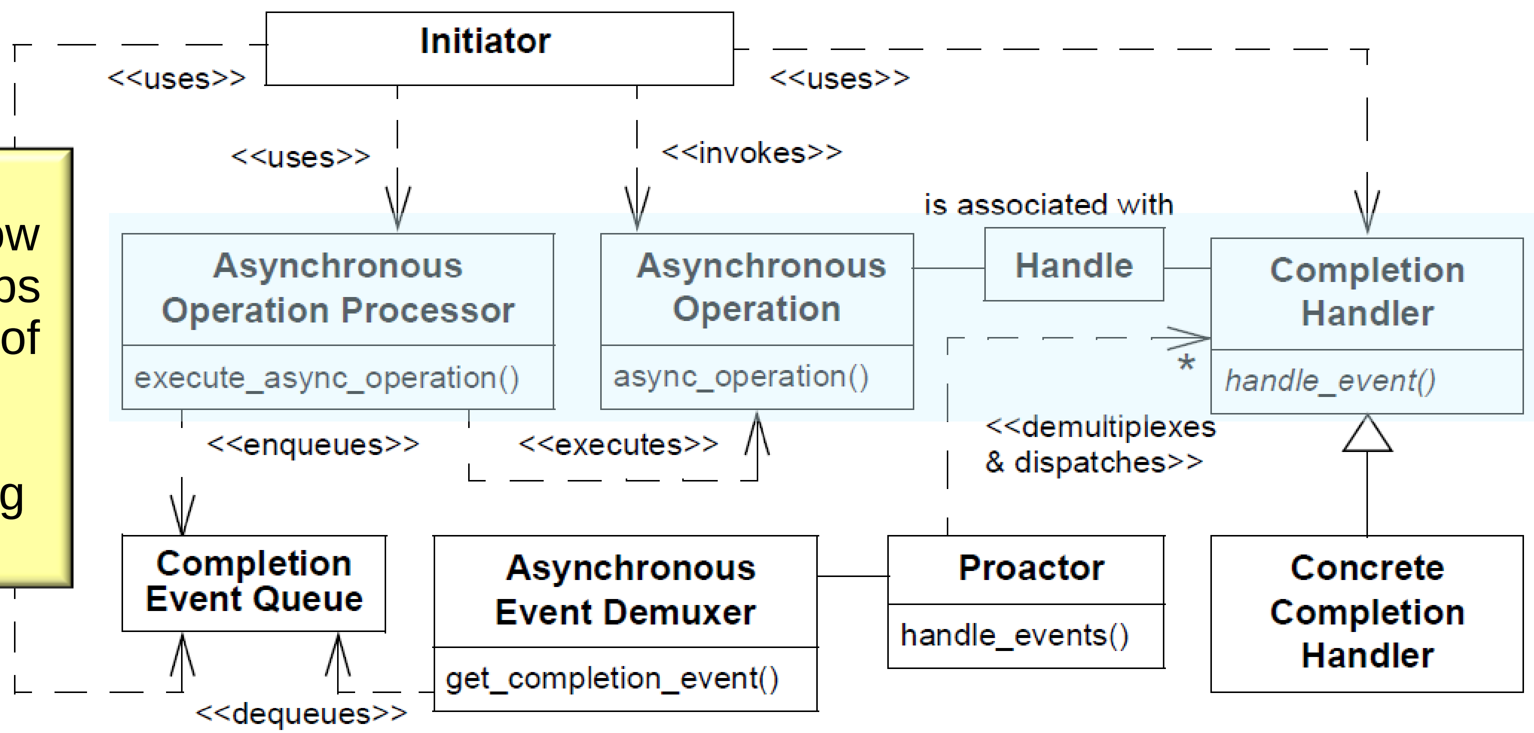


Using Asynchronous I/O Effectively

Context	Problem	Solution
• Synchronous event handling & multi-threading may not achieve most scalable web server when OS supports async I/O	• Leveraging efficiency & scalability of async I/O is hard due to time/space separation of async operation invocations & their subsequent completion events	• Apply the *Proactor* pattern to efficiently demux async I/O operations

Structure

Proactor uses async I/O to allow event-driven apps to gain benefits of concurrency performance without incurring its liabilities

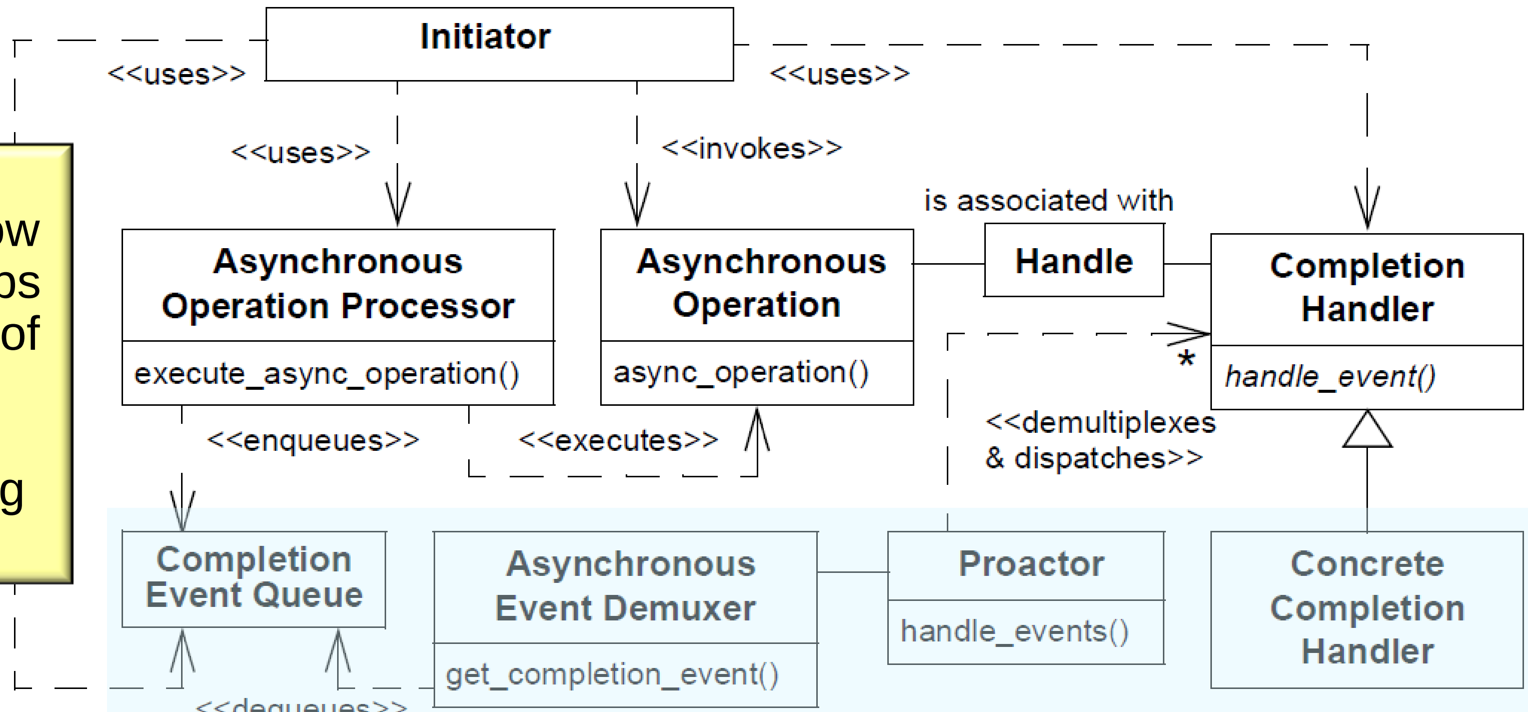


Using Asynchronous I/O Effectively

Context	Problem	Solution
• Synchronous event handling & multi-threading may not achieve most scalable web server when OS supports async I/O	• Leveraging efficiency & scalability of async I/O is hard due to time/space separation of async operation invocations & their subsequent completion events	• Apply the *Proactor* pattern to efficiently demux async I/O operations

Structure

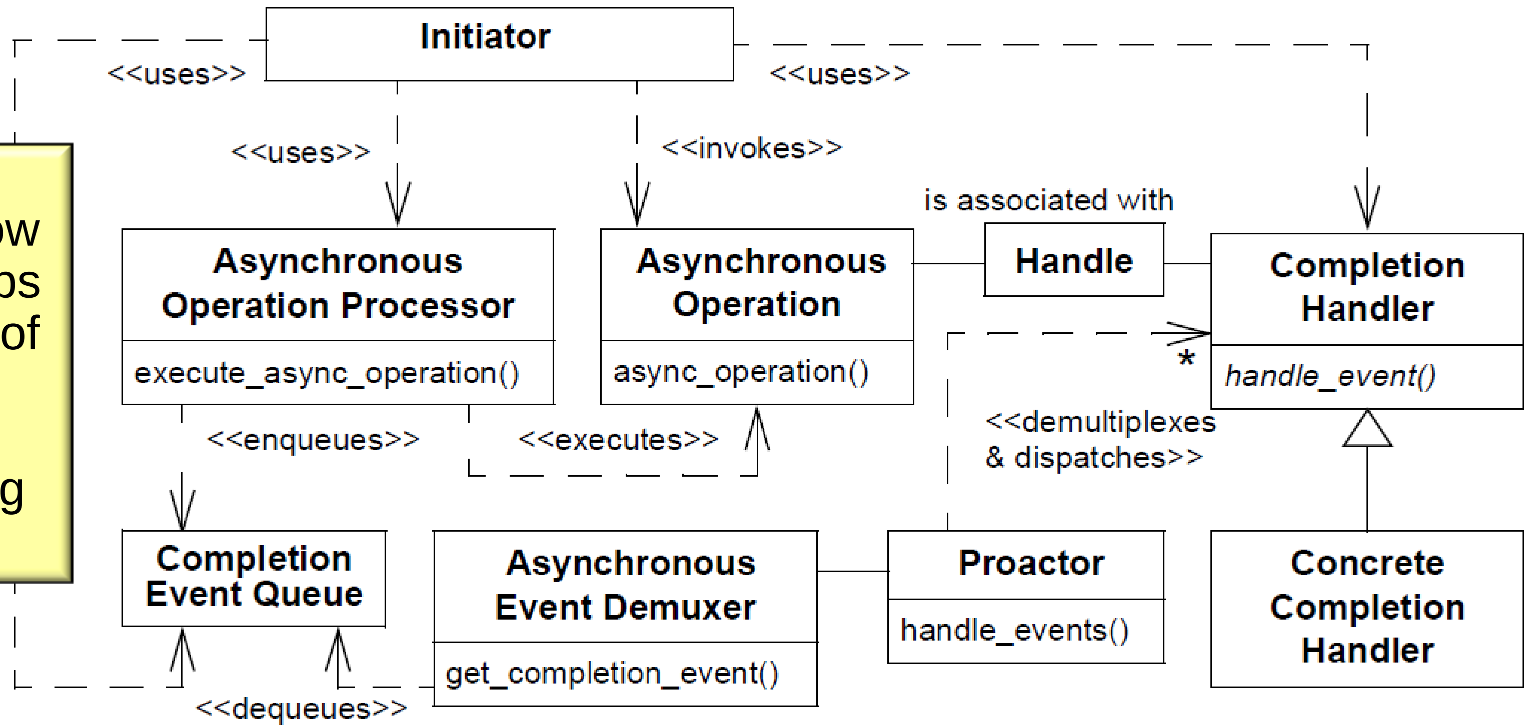
Proactor uses async I/O to allow event-driven apps to gain benefits of concurrency performance without incurring its liabilities



Using Asynchronous I/O Effectively

Context	Problem	Solution
• Synchronous event handling & multi-threading may not achieve most scalable web server when OS supports async I/O	• Leveraging efficiency & scalability of async I/O is hard due to time/space separation of async operation invocations & their subsequent completion events	• Apply the *Proactor* pattern to efficiently demux async I/O operations

Structure



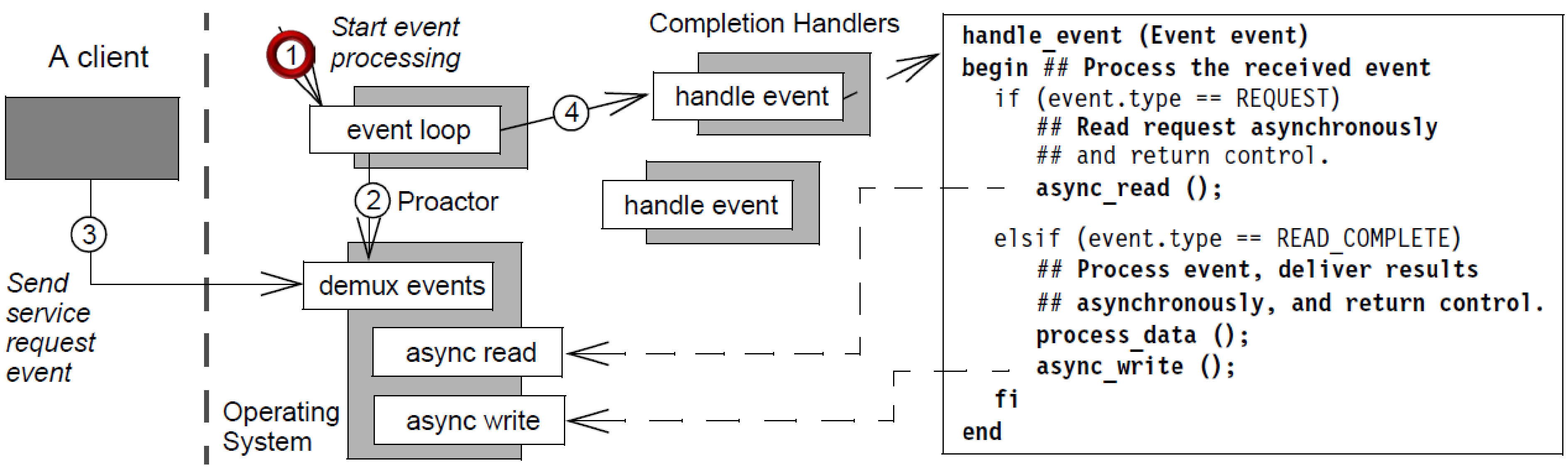
Proactor uses async I/O to allow event-driven apps to gain benefits of concurrency performance without incurring its liabilities

See www.dre.vanderbilt.edu/~schmidt/PDF/proactor.pdf for more info

Using Asynchronous I/O Effectively

Context	Problem	Solution
• Synchronous event handling & multi-threading may not achieve most scalable web server when OS supports async I/O	• Leveraging efficiency & scalability of async I/O is hard due to time/space separation of async operation invocations & their subsequent completion events	• Apply the *Proactor* pattern to efficiently demux async I/O operations

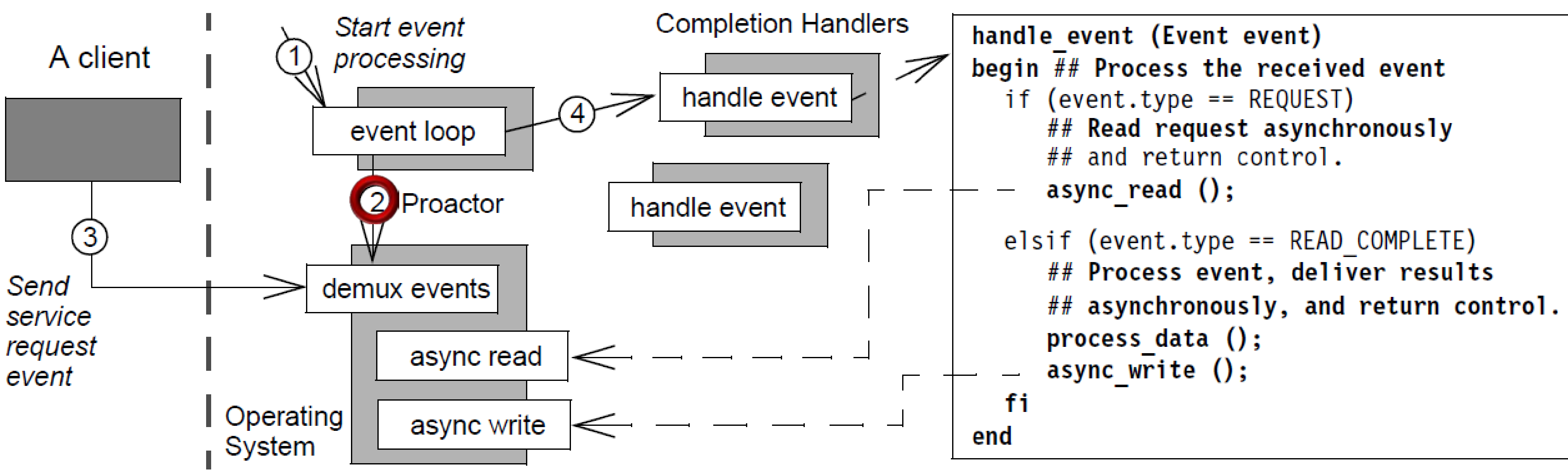
Dynamics



Using Asynchronous I/O Effectively

Context	Problem	Solution
• Synchronous event handling & multi-threading may not achieve most scalable web server when OS supports async I/O	• Leveraging efficiency & scalability of async I/O is hard due to time/space separation of async operation invocations & their subsequent completion events	• Apply the *Proactor* pattern to efficiently demux async I/O operations

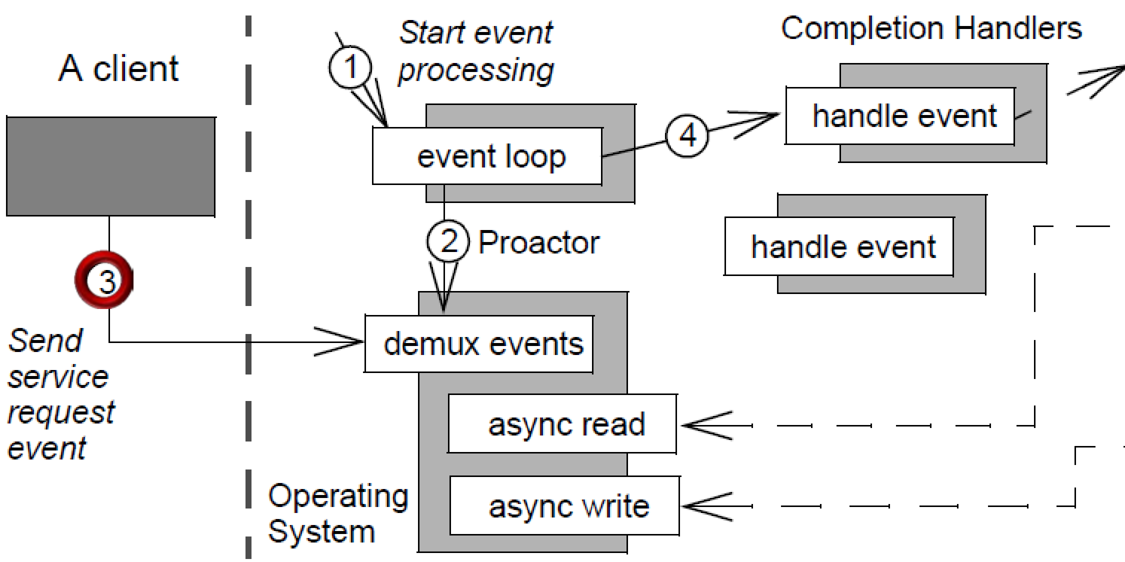
Dynamics



Using Asynchronous I/O Effectively

Context	Problem	Solution
• Synchronous event handling & multi-threading may not achieve most scalable web server when OS supports async I/O	• Leveraging efficiency & scalability of async I/O is hard due to time/space separation of async operation invocations & their subsequent completion events	• Apply the *Proactor* pattern to efficiently demux async I/O operations

Dynamics

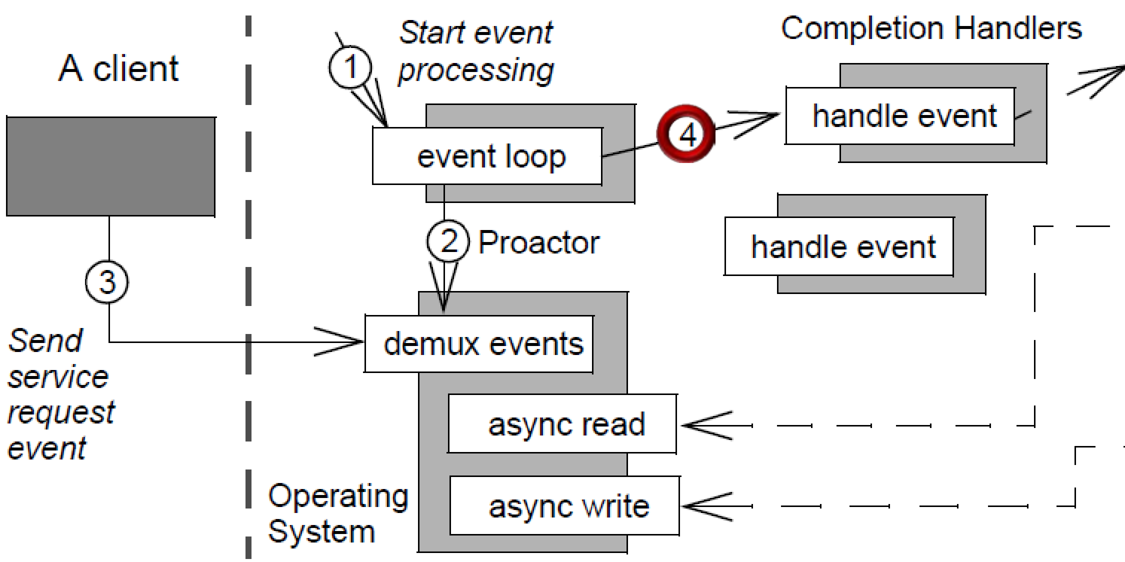



```
handle_event (Event event)
begin ## Process the received event
   if (event.type == REQUEST)
      ## Read request asynchronously
      ## and return control.
      async_read ();
   elsif (event.type == READ_COMPLETE)
      ## Process event, deliver results
      ## asynchronously, and return control.
      process_data ();
      async_write ();
   fi
end
```

Using Asynchronous I/O Effectively

Context	Problem	Solution
• Synchronous event handling & multi-threading may not achieve most scalable web server when OS supports async I/O	• Leveraging efficiency & scalability of async I/O is hard due to time/space separation of async operation invocations & their subsequent completion events	• Apply the *Proactor* pattern to efficiently demux async I/O operations

Dynamics

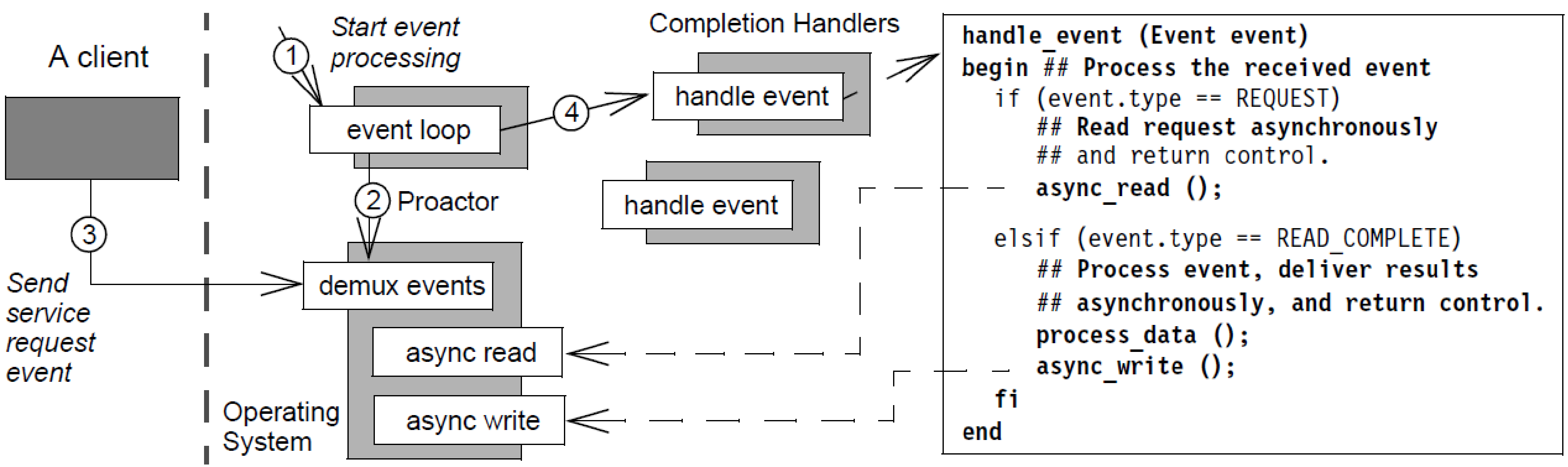



```
handle_event (Event event)
begin ## Process the received event
   if (event.type == REQUEST)
      ## Read request asynchronously
      ## and return control.
      async_read ();
   elsif (event.type == READ_COMPLETE)
      ## Process event, deliver results
      ## asynchronously, and return control.
      process_data ();
      async_write ();
   fi
end
```

Using Asynchronous I/O Effectively

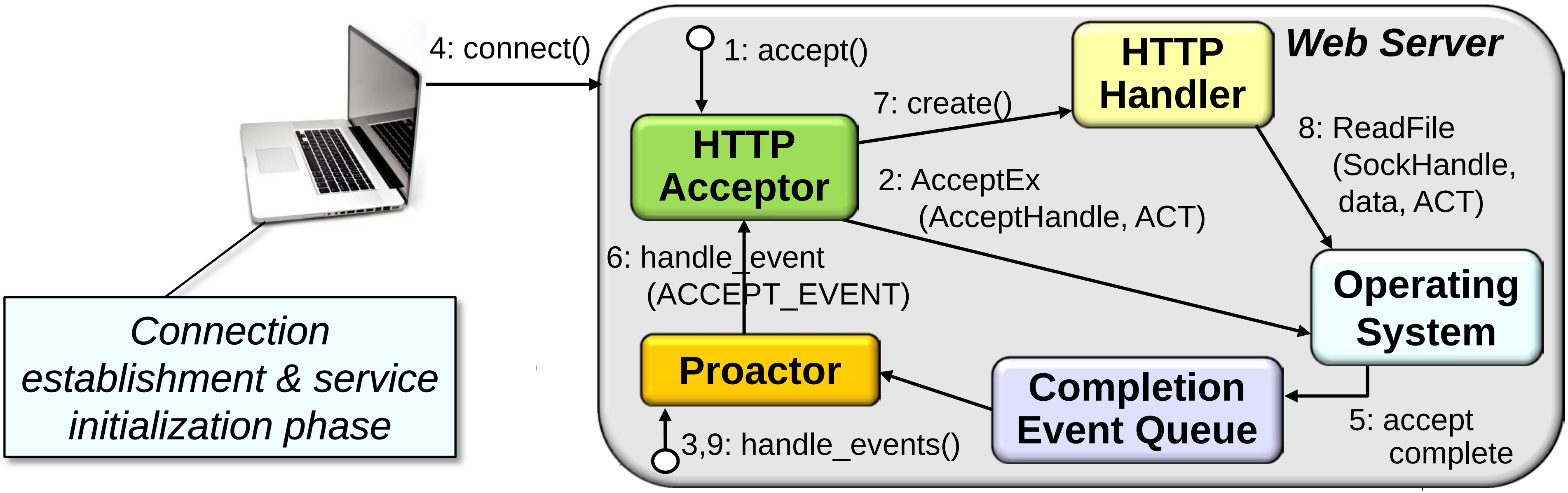
Context	Problem	Solution
• Synchronous event handling & multi-threading may not achieve most scalable web server when OS supports async I/O	• Leveraging efficiency & scalability of async I/O is hard due to time/space separation of async operation invocations & their subsequent completion events	• Apply the *Proactor* pattern to efficiently demux async I/O operations

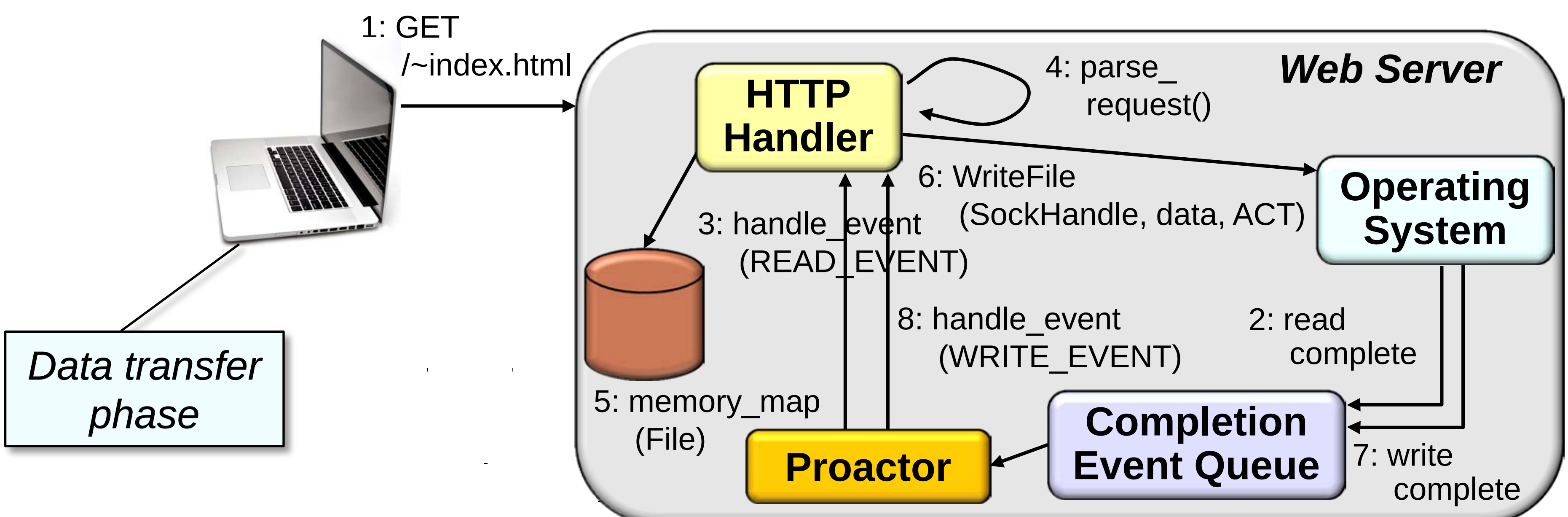
Dynamics



Note *similarities & differences* with the *Reactor* pattern

Applying the Proactor Pattern to JAWS

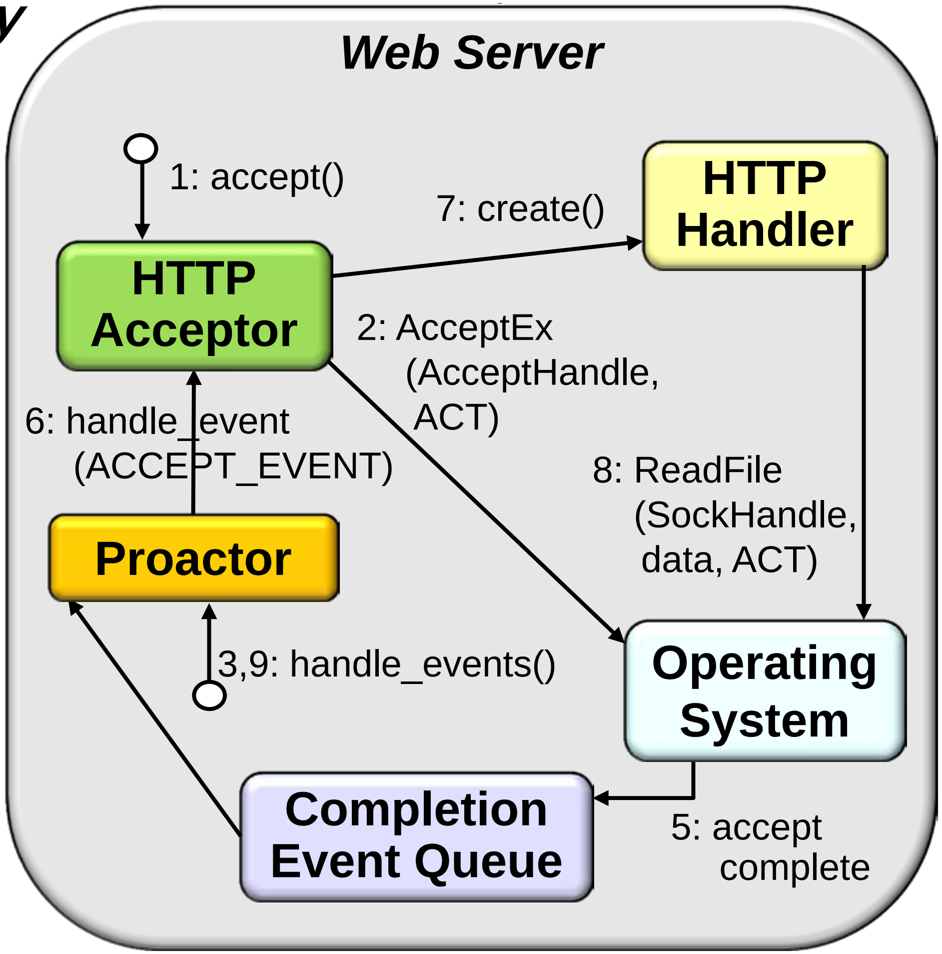




Benefits of Proactor Pattern

Separation of concerns & portability

- Decouples app-independent & app-specific async operations



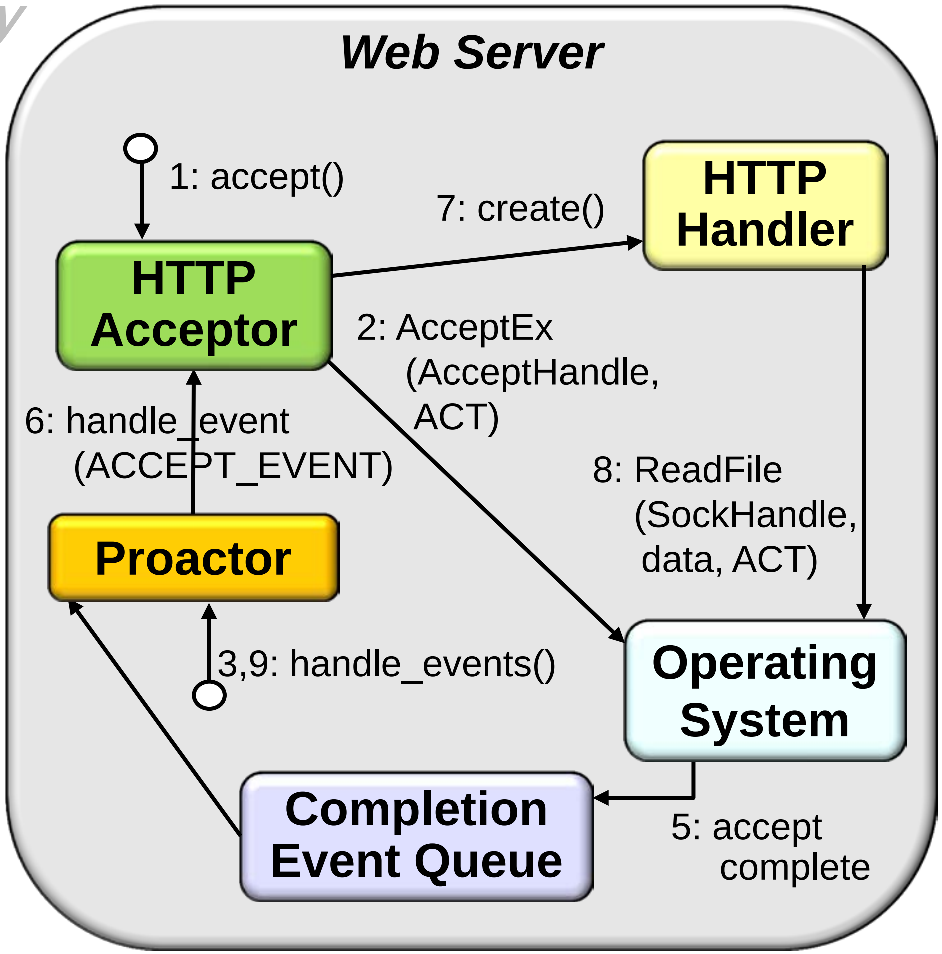
Benefits of Proactor Pattern

Separation of concerns & portability

- Decouples app-independent & app-specific async operations

Decoupling of threading from concurrency

- Async operation processor executes long-duration operations so apps can spawn fewer threads



Benefits of Proactor Pattern

Separation of concerns & portability

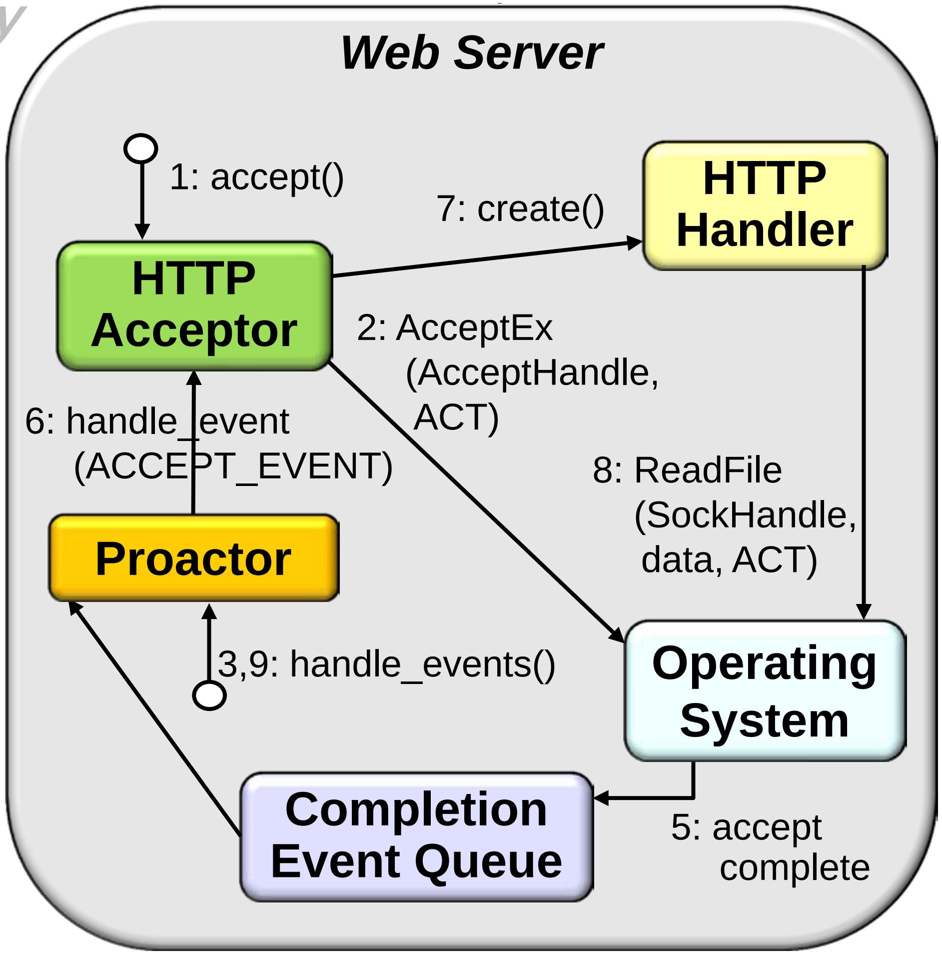
- Decouples app-independent & app-specific async operations

Decoupling of threading from concurrency

- Async operation processor executes long-duration operations so apps can spawn fewer threads

Performance

- Avoids context switching costs by activating only those threads that have events to process



Benefits of Proactor Pattern

Separation of concerns & portability

- Decouples app-independent & app-specific async operations

Decoupling of threading from concurrency

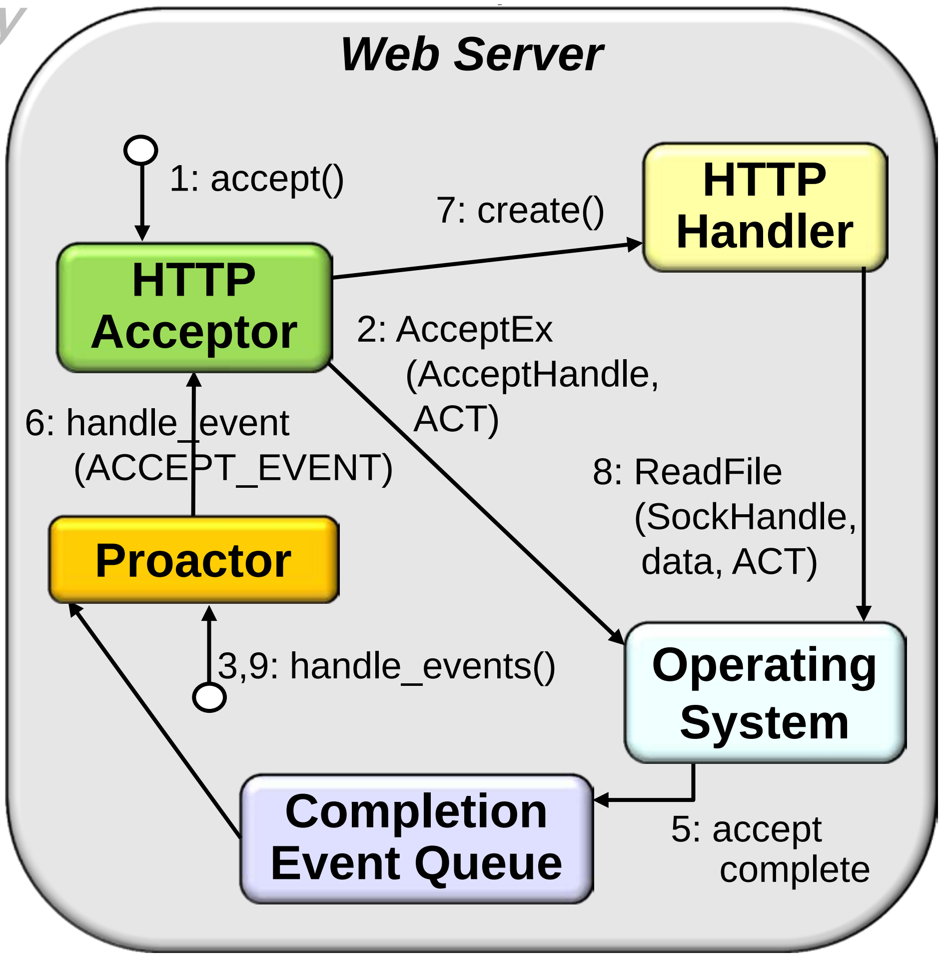
- Async operation processor executes long-duration operations so apps can spawn fewer threads

Performance

- Avoids context switching costs by activating only those threads that have events to process

Simplification of app synchronization

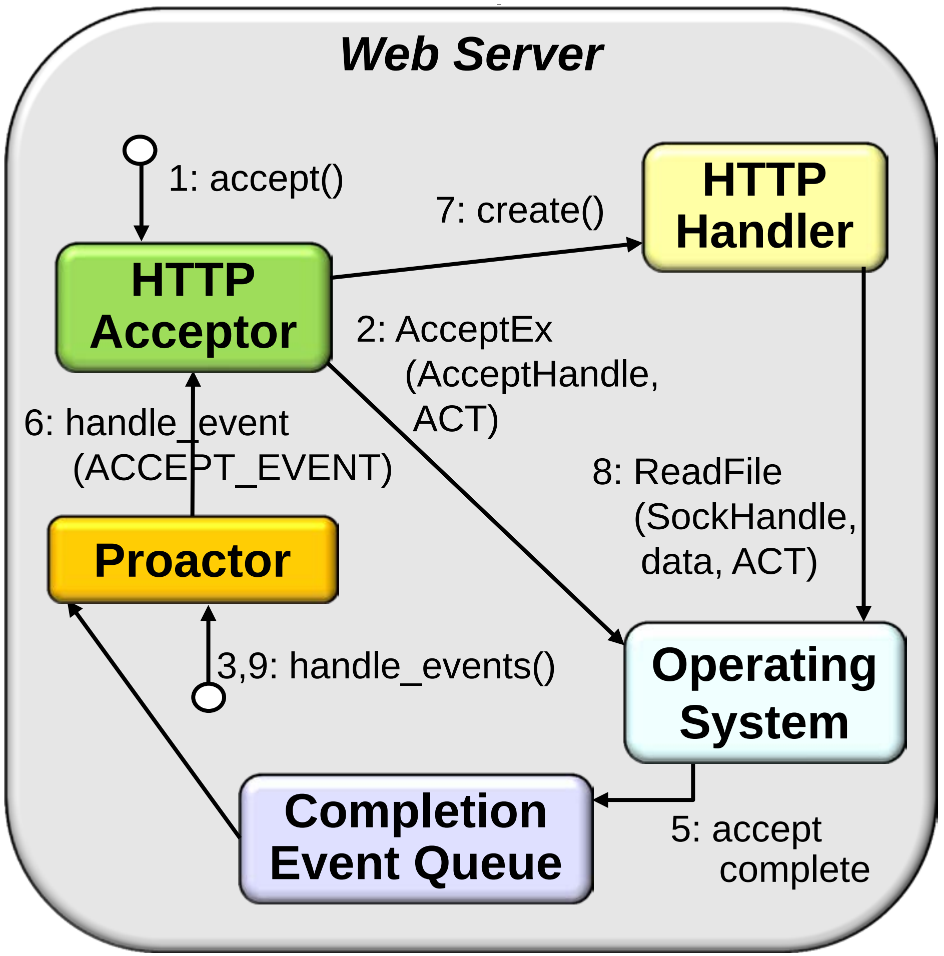
- If completion handlers spawn no threads, apps can be written without synchronization concerns



Limitations of Proactor Pattern

Restricted applicability

- Requires native OS support for asynchronous operations



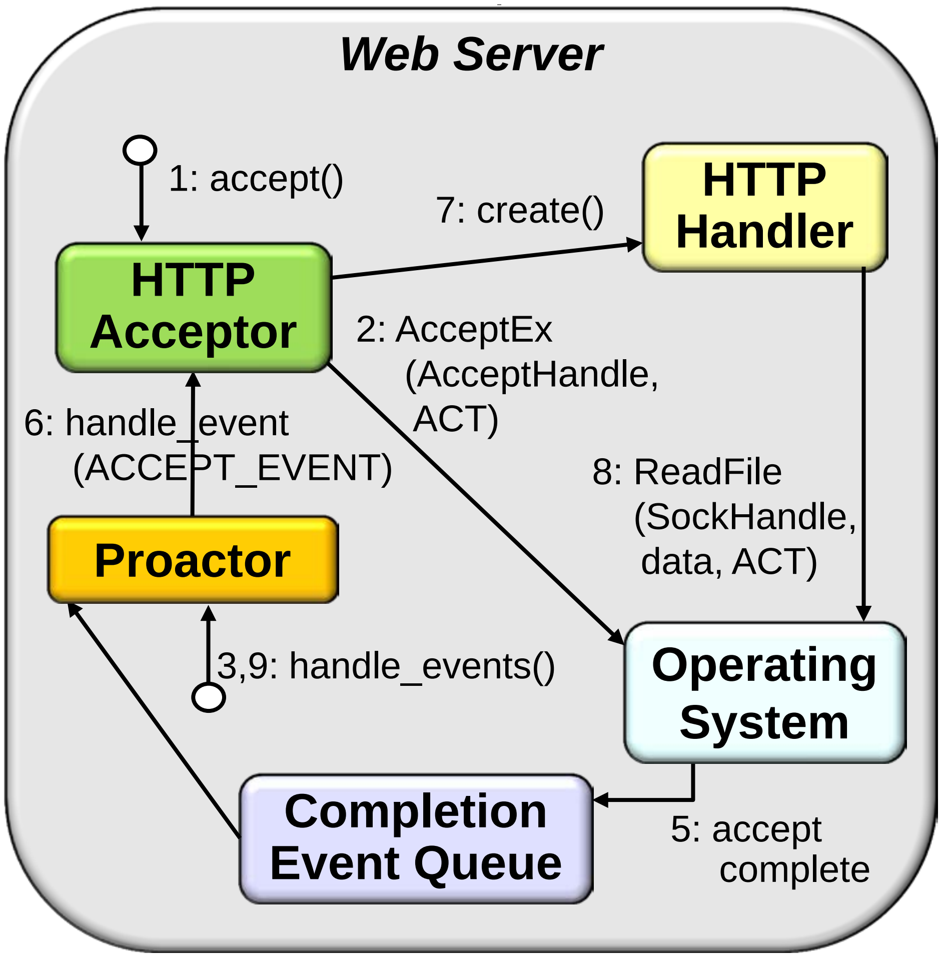
Limitations of Proactor Pattern

Restricted applicability

- Requires native OS support for asynchronous operations

Complexity of programming, debugging, & testing

- It is hard to program apps & services using asynchrony due to separation in time & space between operation invocation & completion



Limitations of Proactor Pattern

Restricted applicability

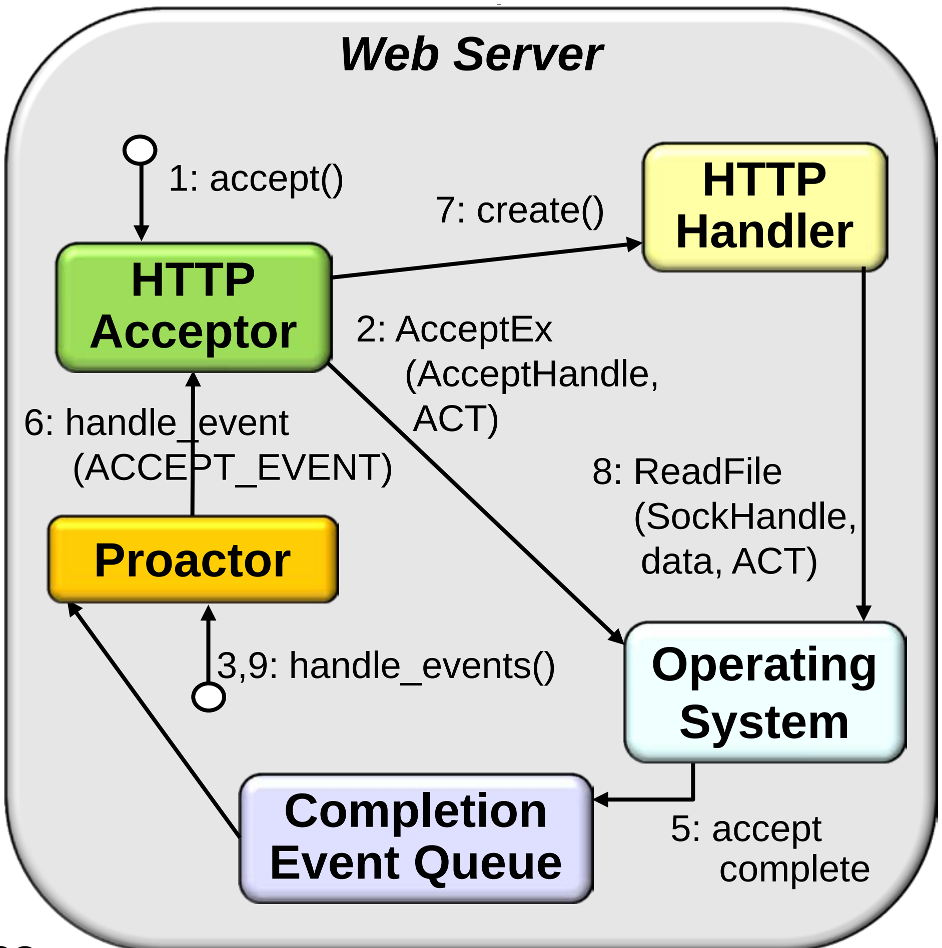
- Requires native OS support for asynchronous operations

Complexity of programming, debugging, & testing

- It is hard to program apps & services using asynchrony due to separation in time & space between operation invocation & completion

Scheduling, controlling, & canceling asynchronously running operations

- Initiators may not be able to control order in which asynchronous operations are executed by asynchronous operation processor
- May also not be able to cancel operations



Patterns & Frameworks for Asynchronous Event Handling: Part 2

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

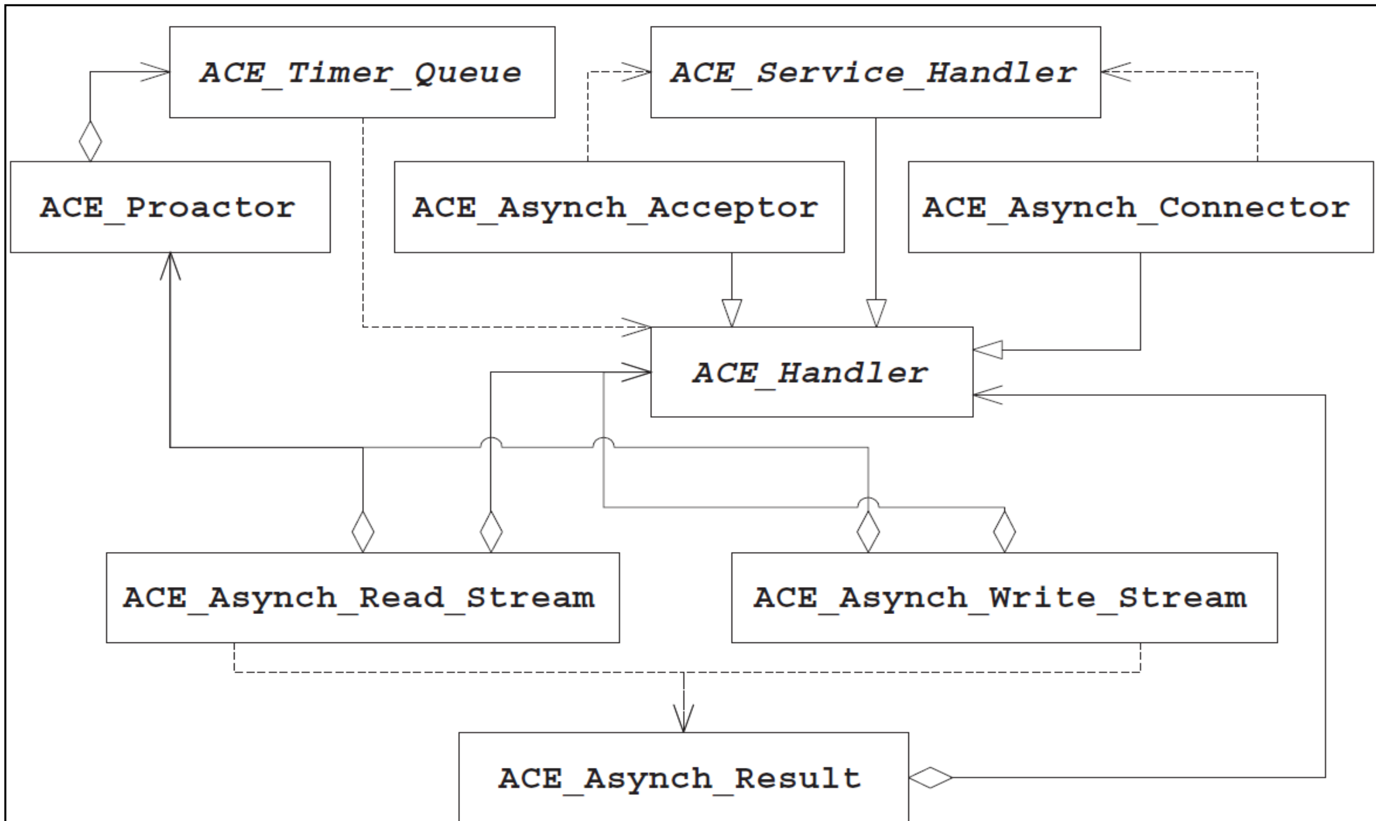
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

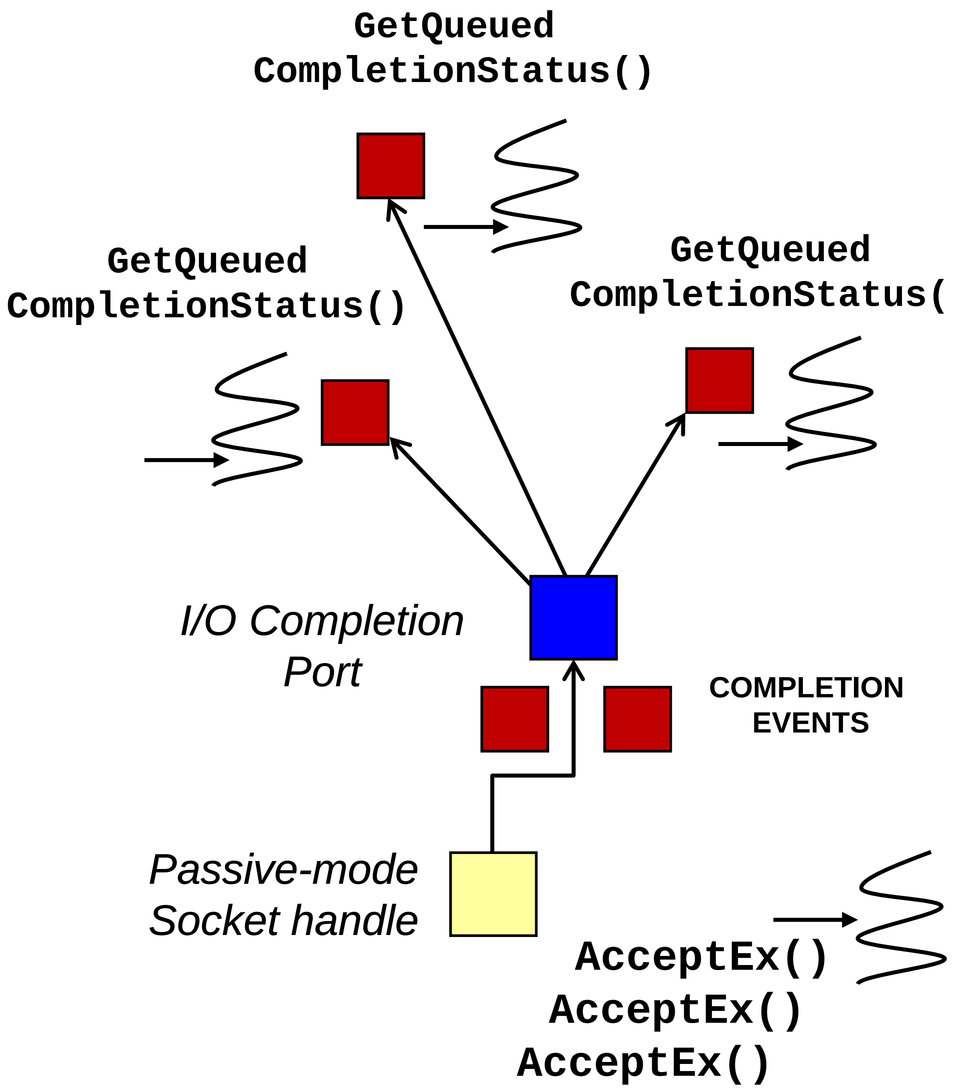
- Describe the *Proactor* pattern
- Describe the ACE *Proactor* framework



The *Proactor* is the most complicated ACE framework

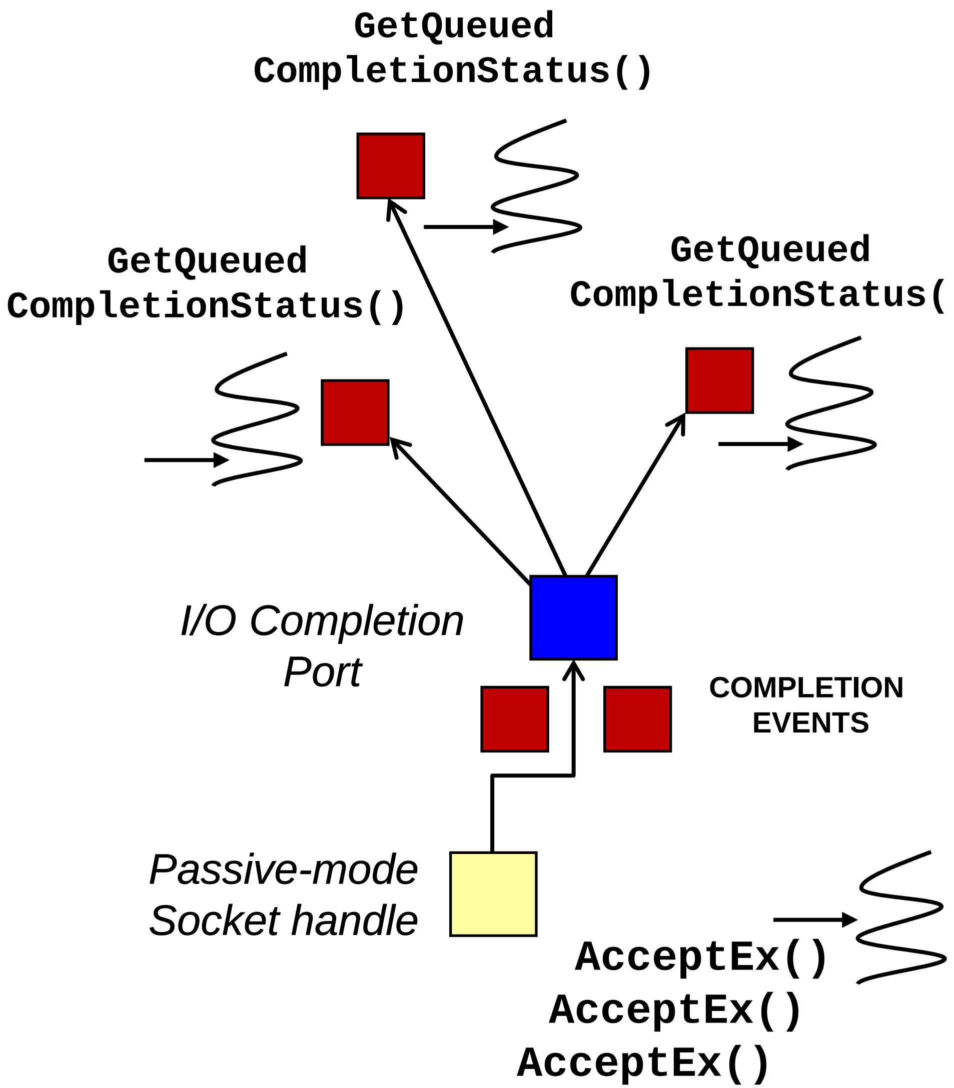
Motivation for the ACE Proactor Framework

- OS support for asynchronous operations has several limitations
  - Tedious & error-prone to program
  - Non-portable and/or inefficient



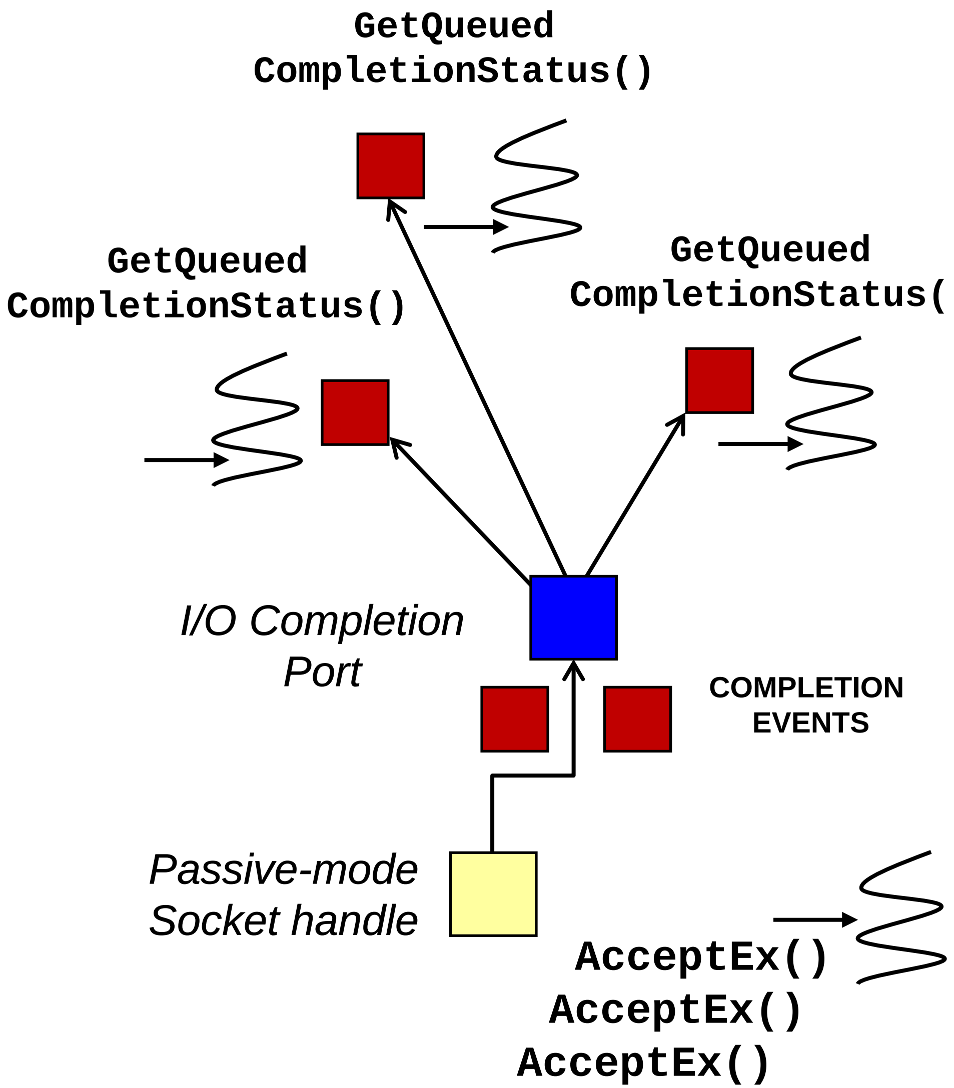
Motivation for the ACE Proactor Framework

- OS support for asynchronous operations has several limitations
  - Tedious & error-prone to program
  - Non-portable and/or inefficient
- Modern Windows platforms support both overlapped I/O & I/O completion ports
  - These mechanisms are effcient, but tricky to program



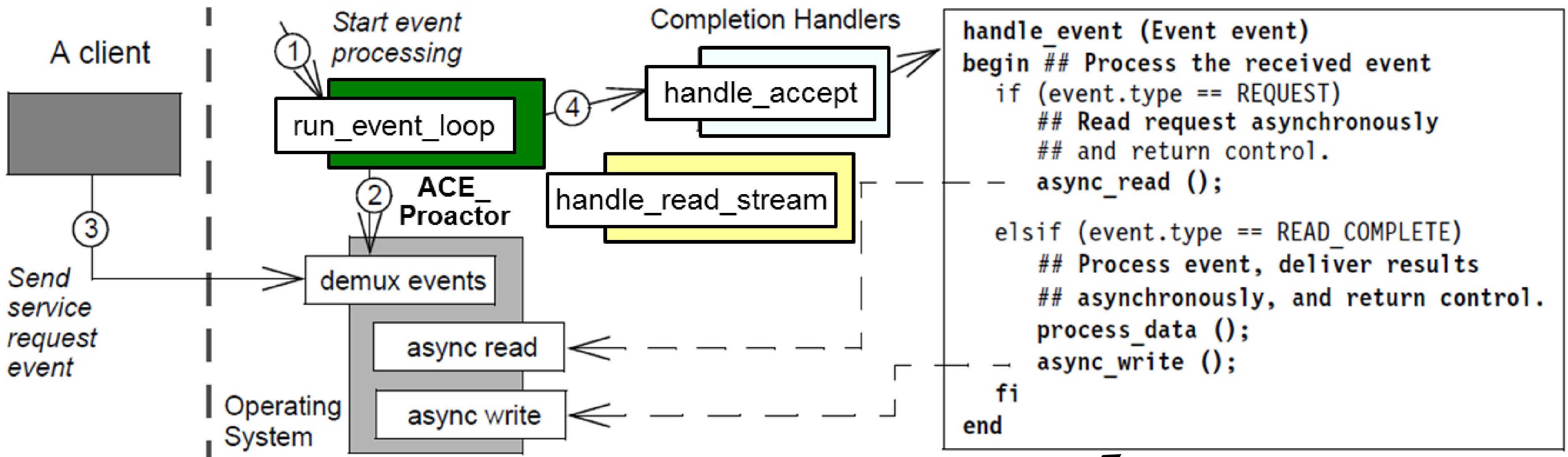
Motivation for the ACE Proactor Framework

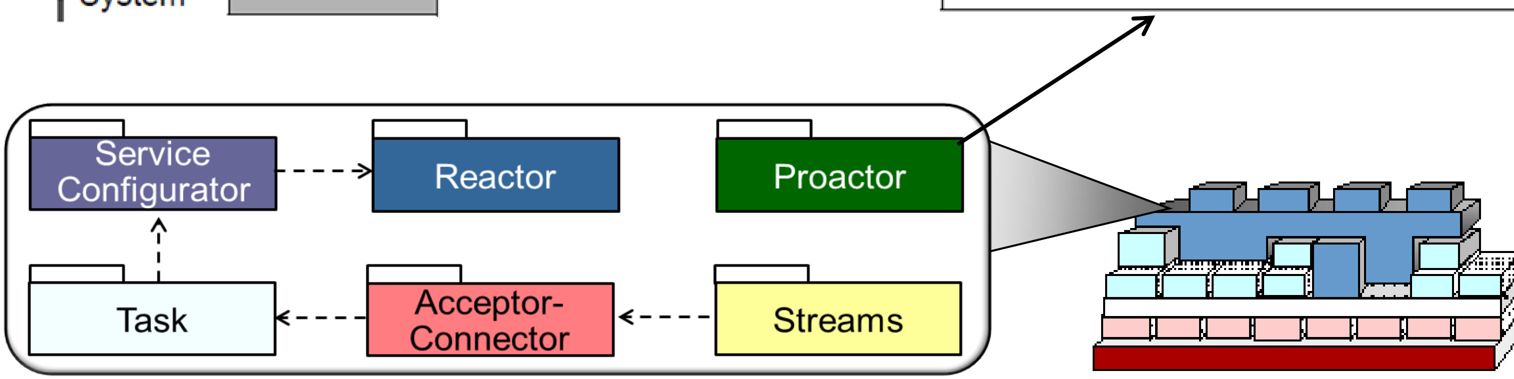
- OS support for asynchronous operations has several limitations
  - Tedious & error-prone to program
  - Non-portable and/or inefficient
- Modern Windows platforms support both overlapped I/O & I/O completion ports
  - These mechanisms are efficient, but tricky to program
- Some operating systems implement the POSIX.4 AIO specification
  - The spec focuses on disk I/O & implementations are often buggy & inefficient



Overview of the ACE Proactor Framework

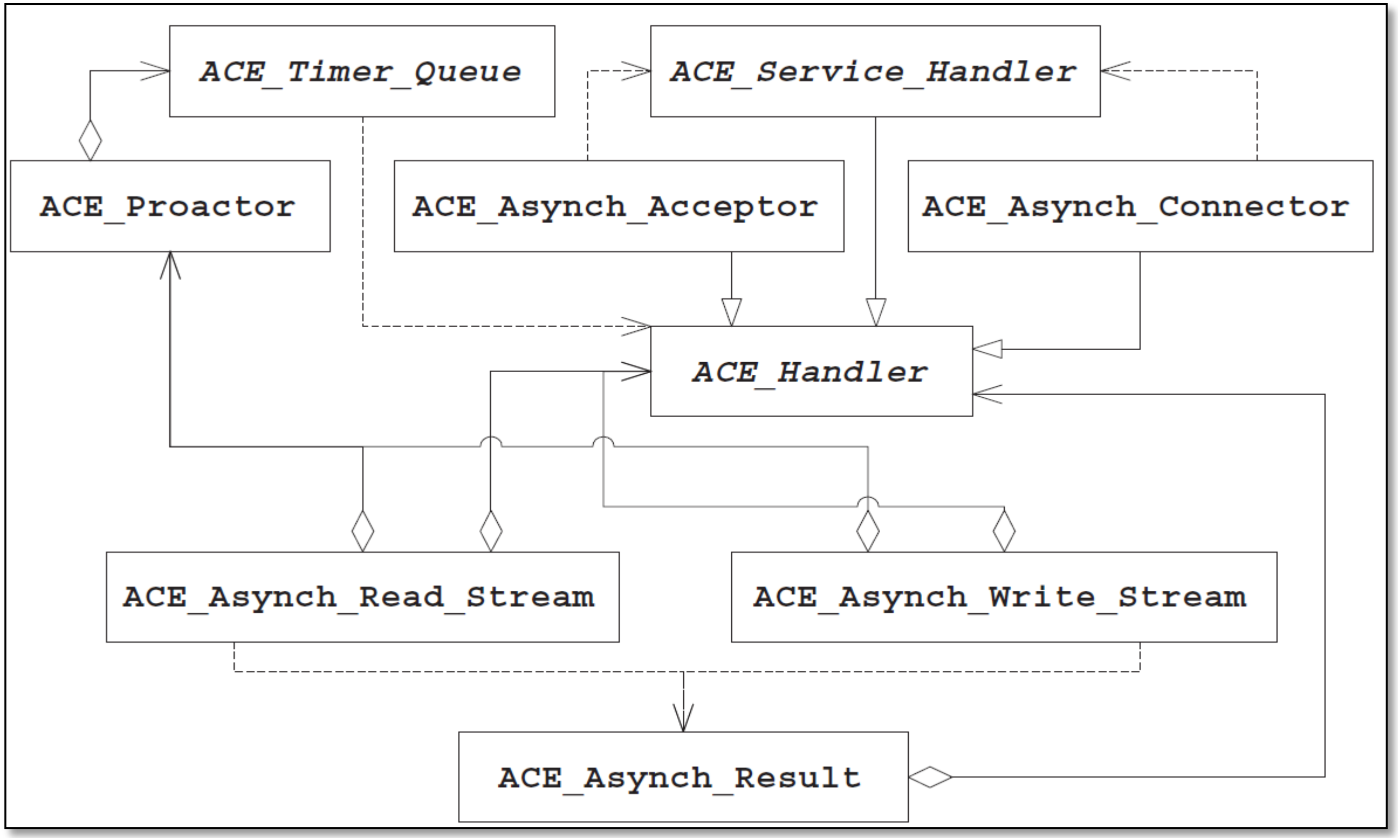
- Classes in this framework allow event-driven apps to process completion events for operations invoked asynchronously





Overview of the ACE Proactor Framework

- Classes in this framework allow event-driven apps to process completion events for operations invoked asynchronously
- Apps inherit from **`ACE_Service_Handler`** & override its hook methods, which ACE *Proactor* framework then dispatches to process completion events



These classes are designed in accordance with the *Proactor* pattern

Overview of the ACE Proactor Framework

- Classes in this framework allow event-driven apps to process completion events for operations invoked asynchronously
- Apps inherit from **ACE_Service_Handler** & override its hook methods, which ACE *Proactor* framework then dispatches to process completion events
- Key classes in the ACE *Proactor* framework include

ACE Class	Description
ACE_Handler **ACE_Asynch_Read_Stream** **ACE_Asynch_Write_Stream**	Initiate asynchronous read & write operations on an I/O stream & associate each with an **ACE_Handler** object that will receive the results of those operations
ACE_Asynch_Acceptor **ACE_Asynch_Connector**	Implementation of *Acceptor-Connector* pattern that establishes new TCP/IP connections asynchronously
ACE_Service_Handler	Defines the target of the **ACE_Asynch_Acceptor** & **ACE_Asynch_Connector** connection factories & provides the hook methods to initialize a TCP/IP connected service
ACE_Proactor	Manages timers & asynchronous I/O completion event demultiplexing

Overview of the ACE Proactor Framework

- Classes in this framework allow event-driven apps to process completion events for operations invoked asynchronously
- Apps inherit from **`ACE_Service_Handler`** & override its hook methods, which ACE *Proactor* framework then dispatches to process completion events
- Key classes in the ACE *Proactor* framework include

ACE Class	Description
`ACE_Handler` **`ACE_Asynch_Read_Stream`** **`ACE_Asynch_Write_Stream`**	Initiate asynchronous read & write operations on an I/O stream & associate each with an **`ACE_Handler`** object that will receive the results of those operations
`ACE_Asynch_Acceptor` **`ACE_Asynch_Connector`**	Implementation of *Acceptor-Connector* pattern that establishes new TCP/IP connections asynchronously
`ACE_Service_Handler`	Defines the target of the **`ACE_Asynch_Acceptor`** & **`ACE_Asynch_Connector`** connection factories & provides the hook methods to initialize a TCP/IP connected service
`ACE_Proactor`	Manages timers & asynchronous I/O completion event demultiplexing

Overview of the ACE Proactor Framework

- Classes in this framework allow event-driven apps to process completion events for operations invoked asynchronously
- Apps inherit from **ACE_Service_Handler** & override its hook methods, which ACE *Proactor* framework then dispatches to process completion events
- Key classes in the ACE *Proactor* framework include

ACE Class	Description
ACE_Handler **ACE_Asynch_Read_Stream** **ACE_Asynch_Write_Stream**	Initiate asynchronous read & write operations on an I/O stream & associate each with an **ACE_Handler** object that will receive the results of those operations
ACE_Asynch_Acceptor **ACE_Asynch_Connector**	Implementation of *Acceptor-Connector* pattern that establishes new TCP/IP connections asynchronously
ACE_Service_Handler	Defines the target of the **ACE_Asynch_Acceptor** & **ACE_Asynch_Connector** connection factories & provides the hook methods to initialize a TCP/IP connected service
ACE_Proactor	Manages timers & asynchronous I/O completion event demultiplexing

Overview of the ACE Proactor Framework

- Classes in this framework allow event-driven apps to process completion events for operations invoked asynchronously
- Apps inherit from **ACE_Service_Handler** & override its hook methods, which ACE *Proactor* framework then dispatches to process completion events
- Key classes in the ACE *Proactor* framework include

ACE Class	Description
ACE_Handler **ACE_Asynch_Read_Stream** **ACE_Asynch_Write_Stream**	Initiate asynchronous read & write operations on an I/O stream & associate each with an **ACE_Handler** object that will receive the results of those operations
ACE_Asynch_Acceptor **ACE_Asynch_Connector**	Implementation of *Acceptor-Connector* pattern that establishes new TCP/IP connections asynchronously
ACE_Service_Handler	Defines the target of the **ACE_Asynch_Acceptor** & **ACE_Asynch_Connector** connection factories & provides the hook methods to initialize a TCP/IP connected service
ACE_Proactor	Manages timers & asynchronous I/O completion event demultiplexing

Overview of the ACE Proactor Framework

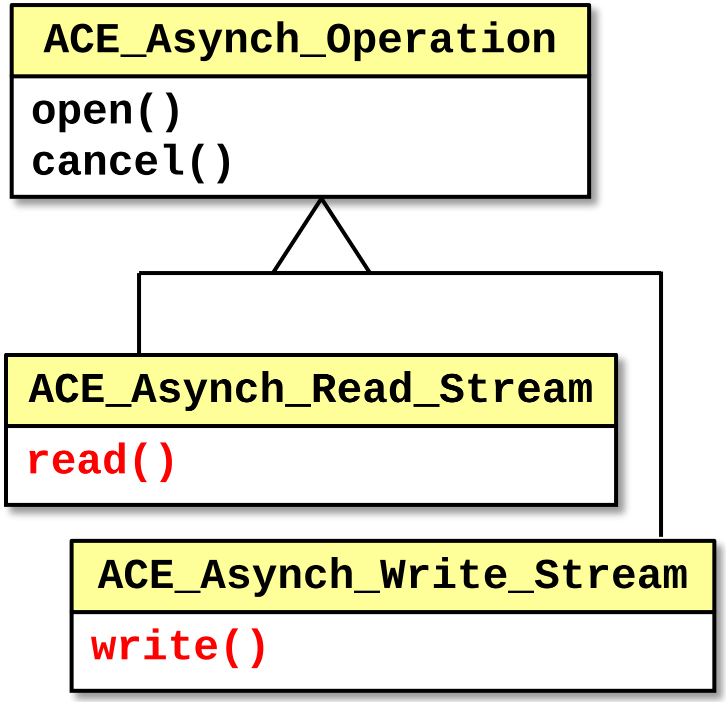
- Classes in this framework allow event-driven apps to process completion events for operations invoked asynchronously
- Apps inherit from **ACE_Service_Handler** & override its hook methods, which ACE *Proactor* framework then dispatches to process completion events
- Key classes in the ACE *Proactor* framework include

ACE Class	Description
ACE_Handler **ACE_Asynch_Read_Stream** **ACE_Asynch_Write_Stream**	Initiate asynchronous read & write operations on an I/O stream & associate each with an **ACE_Handler** object that will receive the results of those operations
ACE_Asynch_Acceptor **ACE_Asynch_Connector**	Implementation of *Acceptor-Connector* pattern that establishes new TCP/IP connections asynchronously
ACE_Service_Handler	Defines the target of the **ACE_Asynch_Acceptor** & **ACE_Asynch_Connector** connection factories & provides the hook methods to initialize a TCP/IP connected service
ACE_Proactor	Manages timers & asynchronous I/O completion event demultiplexing

The ACE_Asynch_[Read|Write]_Stream Classes

These factory classes enable applications to initiate portable asynchronous **`read()`** & **`write()`** operations that provide the following capabilities:

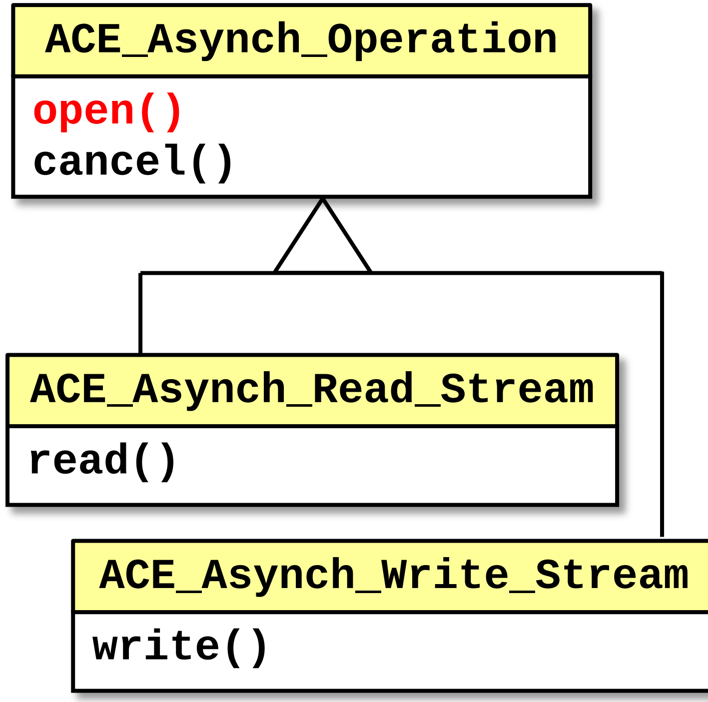
- They can initiate asynchronous I/O operations on a stream-oriented IPC mechanism, such as a TCP socket



The ACE_Asynch_[Read|Write]_Stream Classes

These factory classes enable applications to initiate portable asynchronous **`read()`** & **`write()`** operations that provide the following capabilities:

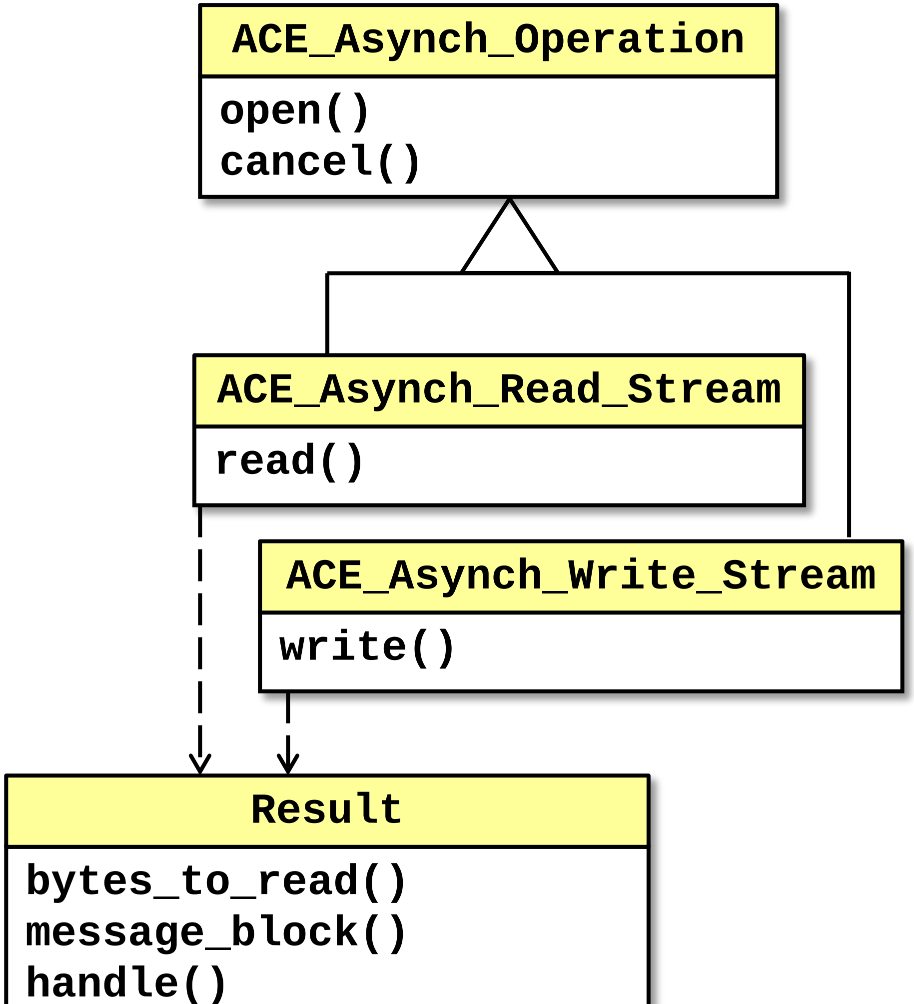
- They can initiate asynchronous I/O operations on a stream-oriented IPC mechanism, such as a TCP socket
- They bind an I/O handle, an **`ACE_Handler`** object, & a **`ACE_Proactor`** to process I/O completion events efficiently



The ACE_Asynch_[Read|Write]_Stream Classes

These factory classes enable applications to initiate portable asynchronous **`read()`** & **`write()`** operations that provide the following capabilities:

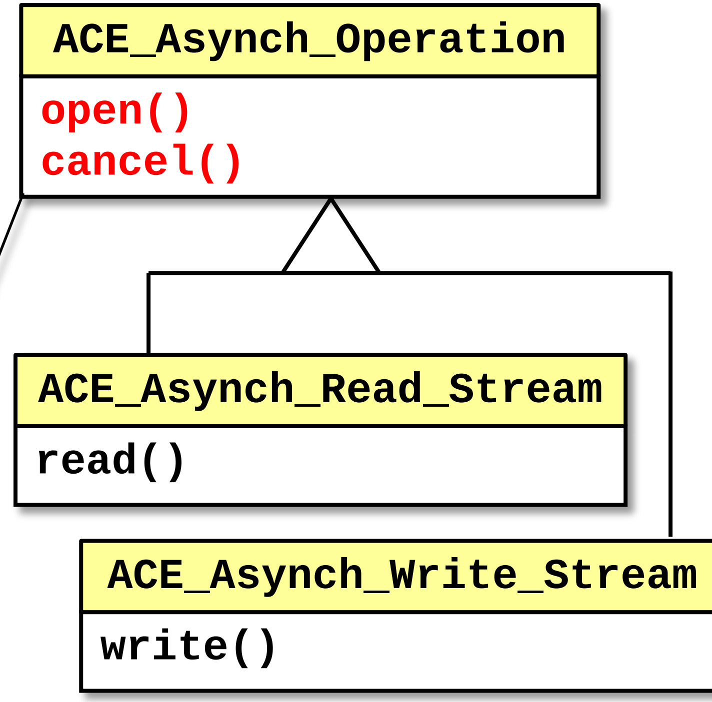
- They can initiate asynchronous I/O operations on a stream-oriented IPC mechanism, such as a TCP socket
- They bind an I/O handle, an **`ACE_Handler`** object, & a **`ACE_Proactor`** to process I/O completion events efficiently
- They create an object that carries an operation's parameters through the ACE *Proactor* framework to its completion handler



The ACE_Asynch_[Read|Write]_Stream Classes

These factory classes enable applications to initiate portable asynchronous **`read()`** & **`write()`** operations that provide the following capabilities:

- They can initiate asynchronous I/O operations on a stream-oriented IPC mechanism, such as a TCP socket
- They bind an I/O handle, an **`ACE_Handler`** object, & a **`ACE_Proactor`** to process I/O completion events efficiently
- They create an object that carries an operation's parameters through the ACE *Proactor* framework to its completion handler



*They derive from **`ACE_Asynch_Operation`**, which provides the interface to initialize the object & cancel outstanding I/O operations*

Handles *variability* of asynchronous I/O operations via a *common* API

The ACE_Handler Class

The base class of all asynchronous completion handlers in the ACE *Proactor* framework

- It provides hook methods to handle completion of all asynchronous I/O operations defined in ACE
  - Including connection establishment & I/O operations on files & IPC streams

ACE_Handler
handle_accept()
handle_connect()
handle_read_stream
handle_write_stream()
handle_transmit_file()
handle_read_file()
handle_write_file()
handle_time_out()
…

The ACE_Handler Class

The base class of all asynchronous completion handlers in the ACE *Proactor* framework

- It provides hook methods to handle completion of all asynchronous I/O operations defined in ACE
  - Including connection establishment & I/O operations on files & IPC streams
- It provides a hook method to handle timer expiration

ACE_Handler
handle_accept()
handle_connect()
handle_read_stream
handle_write_stream()
handle_transmit_file()
handle_read_file()
handle_write_file()
handle_time_out()
…

The ACE_Handler Class

The base class of all asynchronous completion handlers in the ACE *Proactor* framework

- It provides hook methods to handle completion of all asynchronous I/O operations defined in ACE
  - Including connection establishment & I/O operations on files & IPC streams
- It provides a hook method to handle timer expiration

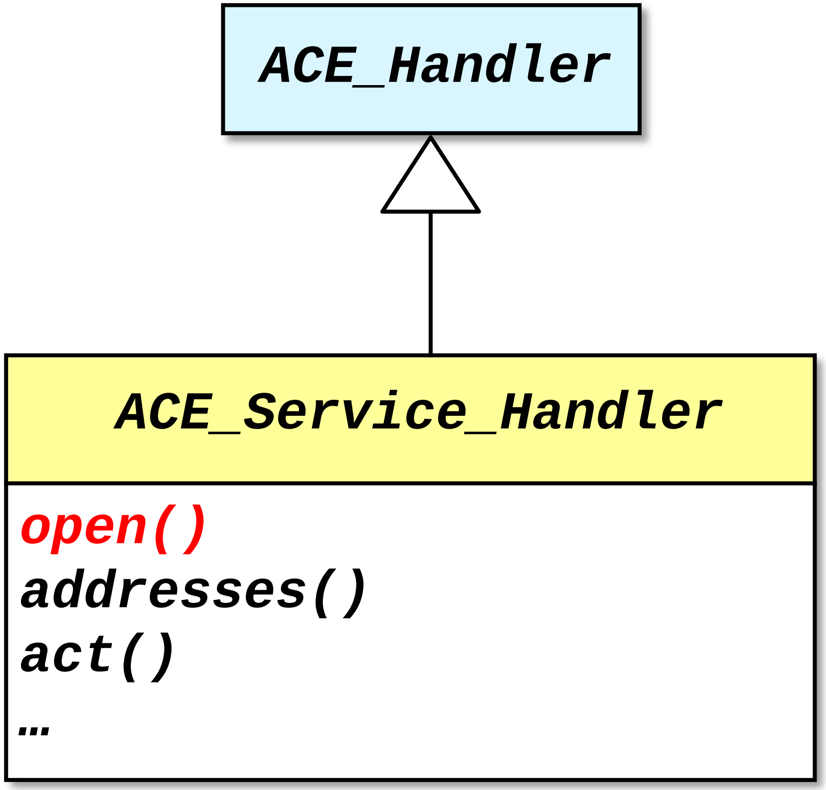
ACE_Handler
handle_accept() ***handle_connect()*** ***handle_read_stream*** ***handle_write_stream()*** ***handle_transmit_file()*** ***handle_read_file()*** ***handle_write_file()*** ***handle_time_out()*** ***…***

Handles *variability* of asynchronous event completion via a *common* API

The ACE_Service_Handler Class

This class defines the interface for the **`ACE_Asynch_Acceptor`** & **`ACE_Asynch_Connector`** to activate when a new TCP connection is accepted

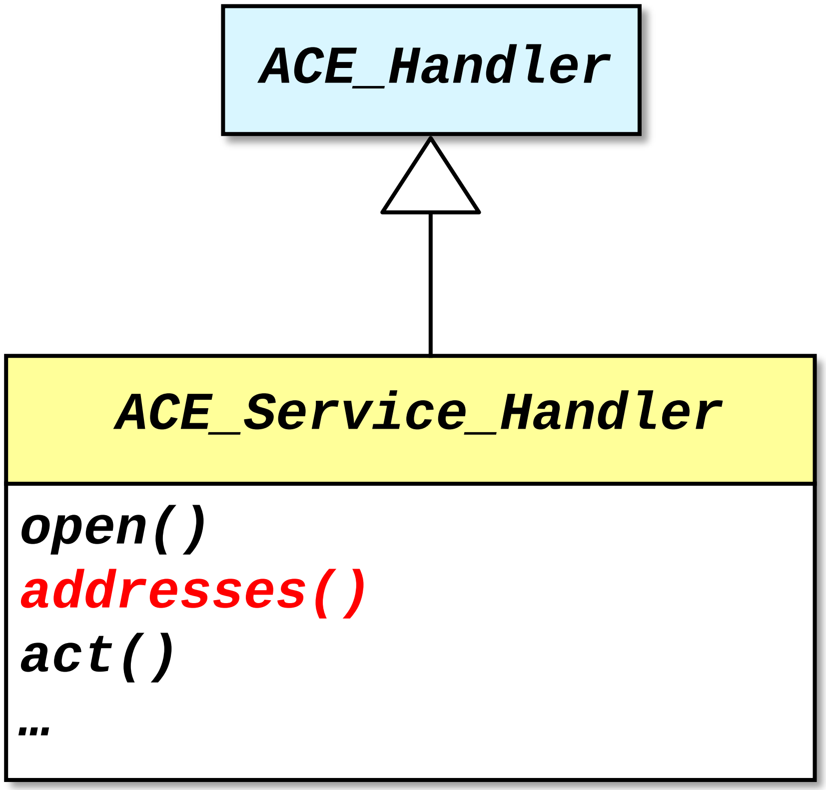
- It provides the basis for initializing & implementing a networked app service
  - Acts as the target of **`ACE_Asynch_Connector`** & **`ACE_Asynch_Acceptor`** connection factories



The ACE_Service_Handler Class

This class defines the interface for the **ACE_Asynch_Acceptor** & **ACE_Asynch_Connector** to activate when a new TCP connection is accepted

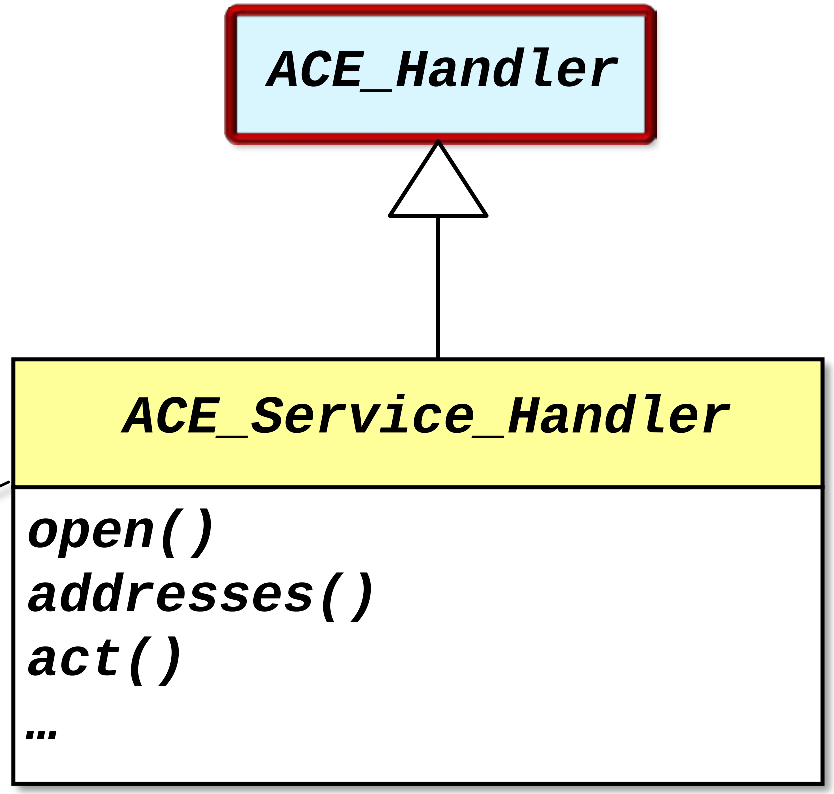
- It provides the basis for initializing & implementing a networked app service
  - Acts as the target of **ACE_Asynch_Connector** & **ACE_Asynch_Acceptor** connection factories
- It receives the connected peer's address



The ACE_Service_Handler Class

This class defines the interface for the **`ACE_Asynch_Acceptor`** & **`ACE_Asynch_Connector`** to activate when a new TCP connection is accepted

- It provides the basis for initializing & implementing a networked app service
  - Acts as the target of **`ACE_Asynch_Connector`** & **`ACE_Asynch_Acceptor`** connection factories
- It receives the connected peer's address



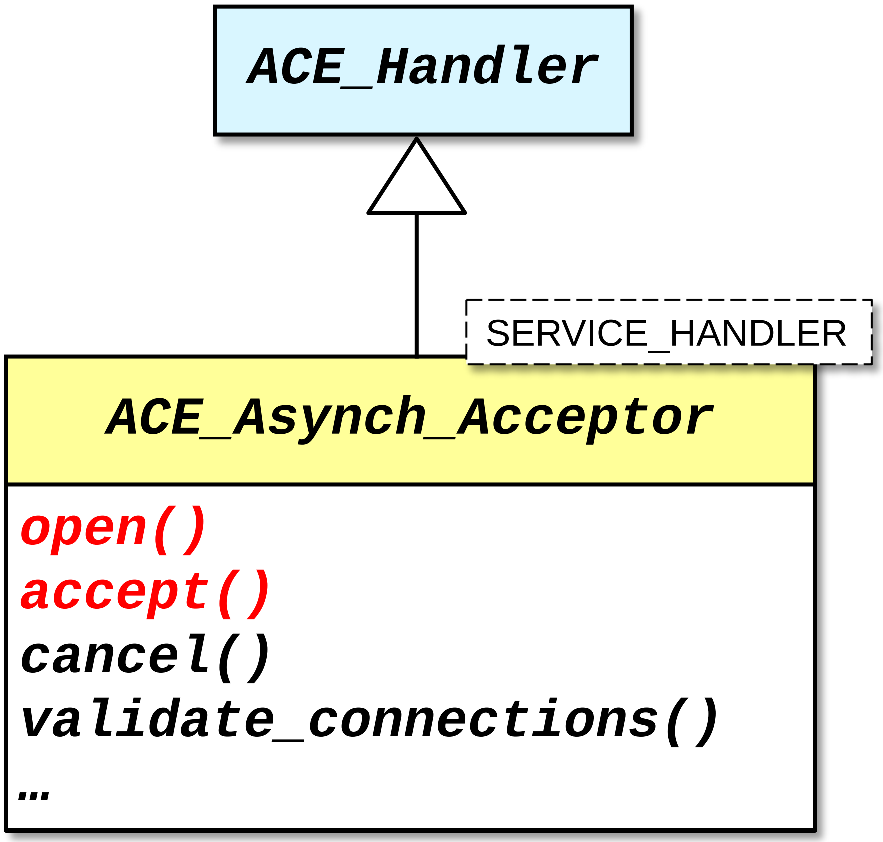
It inherits the ability to handle asynchronous I/O completion events since it derives from ***`ACE_Handler`***

Handles *variability* of asynchronous network I/O via a *common* API

The ACE_Asynch_Acceptor Class

This class provides an implementation of asynchronous *Acceptor* capability in the *Acceptor-Connector* pattern:

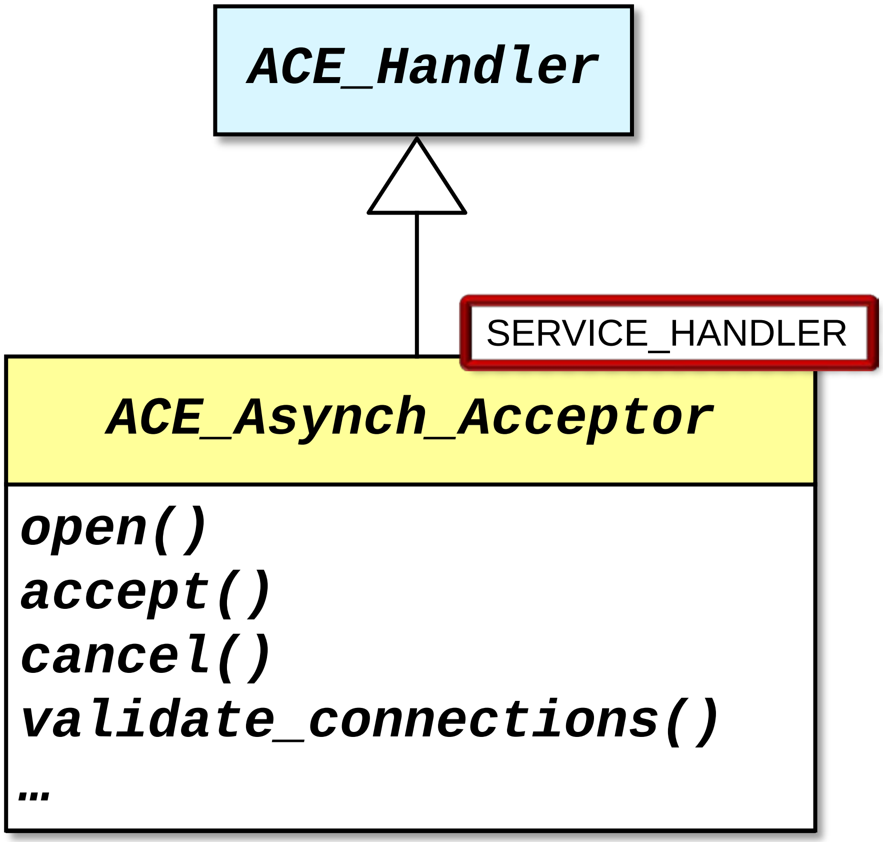
- It initiates asynchronous passive connection establishment



The ACE_Asynch_Acceptor Class

This class provides an implementation of asynchronous *Acceptor* capability in the *Acceptor-Connector* pattern:

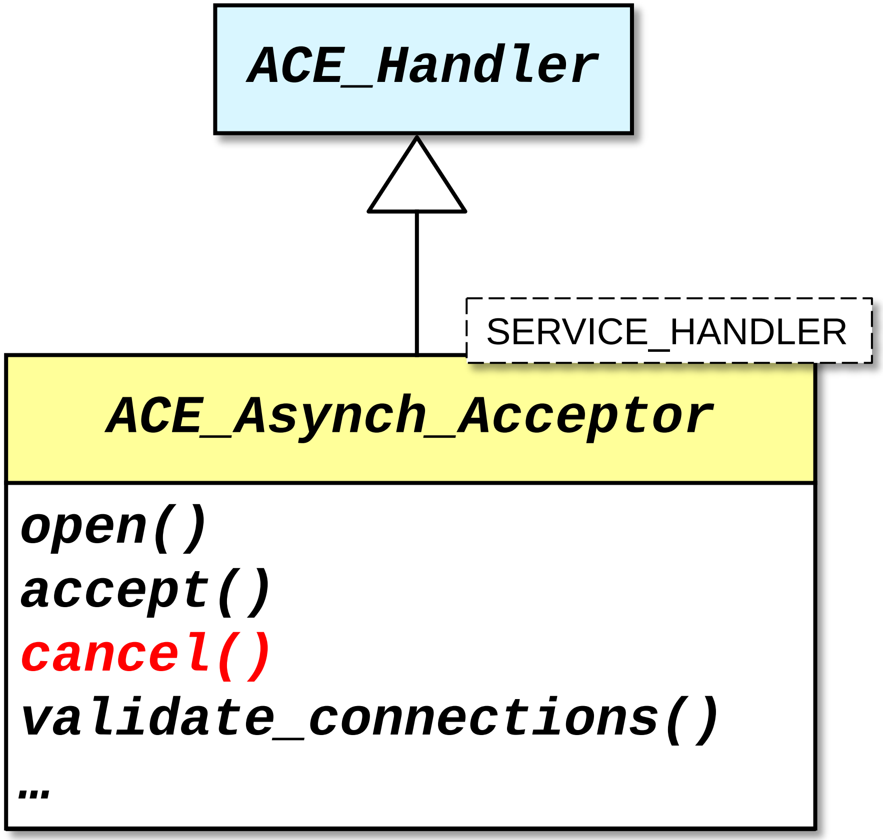
- It initiates asynchronous passive connection establishment
- It acts as a factory, creating a new **`ACE_Service_Handler`** for each accepted connection



The ACE_Asynch_Acceptor Class

This class provides an implementation of asynchronous *Acceptor* capability in the *Acceptor-Connector* pattern:

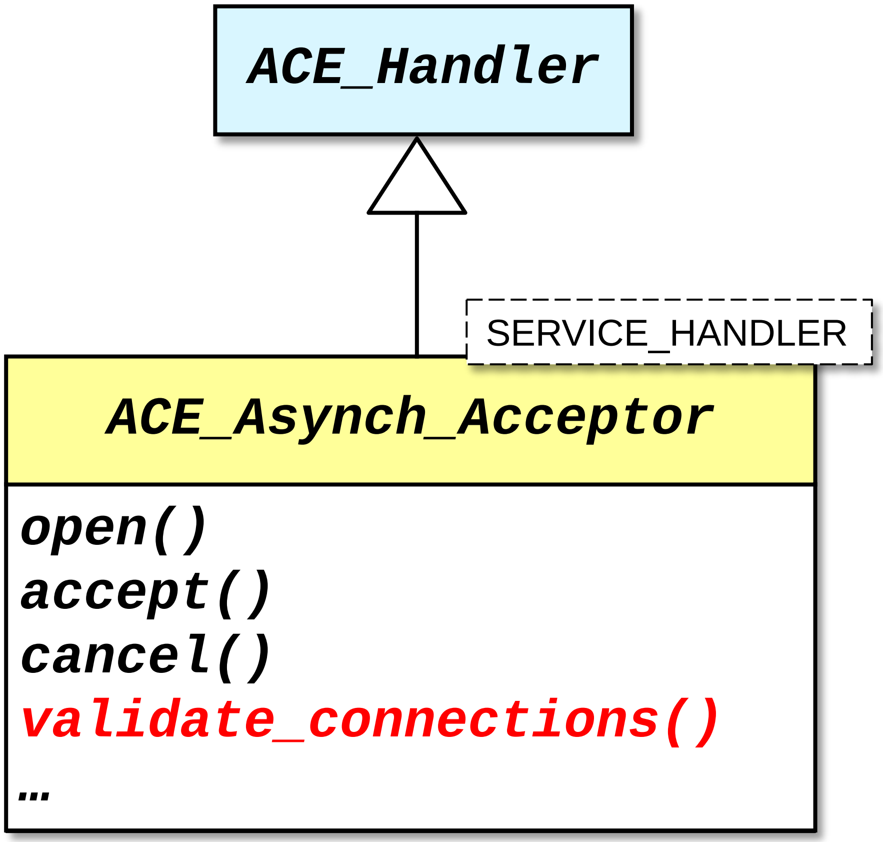
- It initiates asynchronous passive connection establishment
- It acts as a factory, creating a new **`ACE_Service_Handler`** for each accepted connection
- It can cancel a previously initiated asynchronous **`accept()`** operation



The ACE_Asynch_Acceptor Class

This class provides an implementation of asynchronous *Acceptor* capability in the *Acceptor-Connector* pattern:

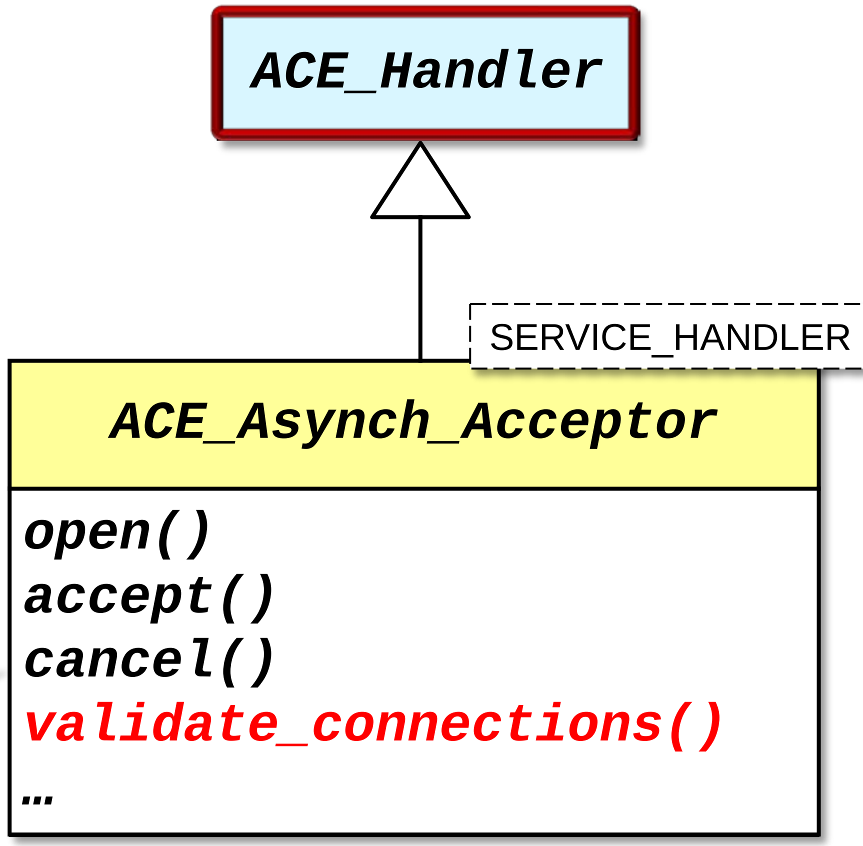
- It initiates asynchronous passive connection establishment
- It acts as a factory, creating a new **`ACE_Service_Handler`** for each accepted connection
- It can cancel a previously initiated asynchronous **`accept()`** operation
- It provides a hook method to validate the peer before initializing a new **`ACE_Service_Handler`**



The ACE_Asynch_Acceptor Class

This class provides an implementation of asynchronous *Acceptor* capability in the *Acceptor-Connector* pattern:

- It initiates asynchronous passive connection establishment
- It acts as a factory, creating a new **`ACE_Service_Handler`** for each accepted connection
- It can cancel a previously initiated asynchronous **`accept()`** operation
- It provides a hook method to validate the peer before initializing a new **`ACE_Service_Handler`**

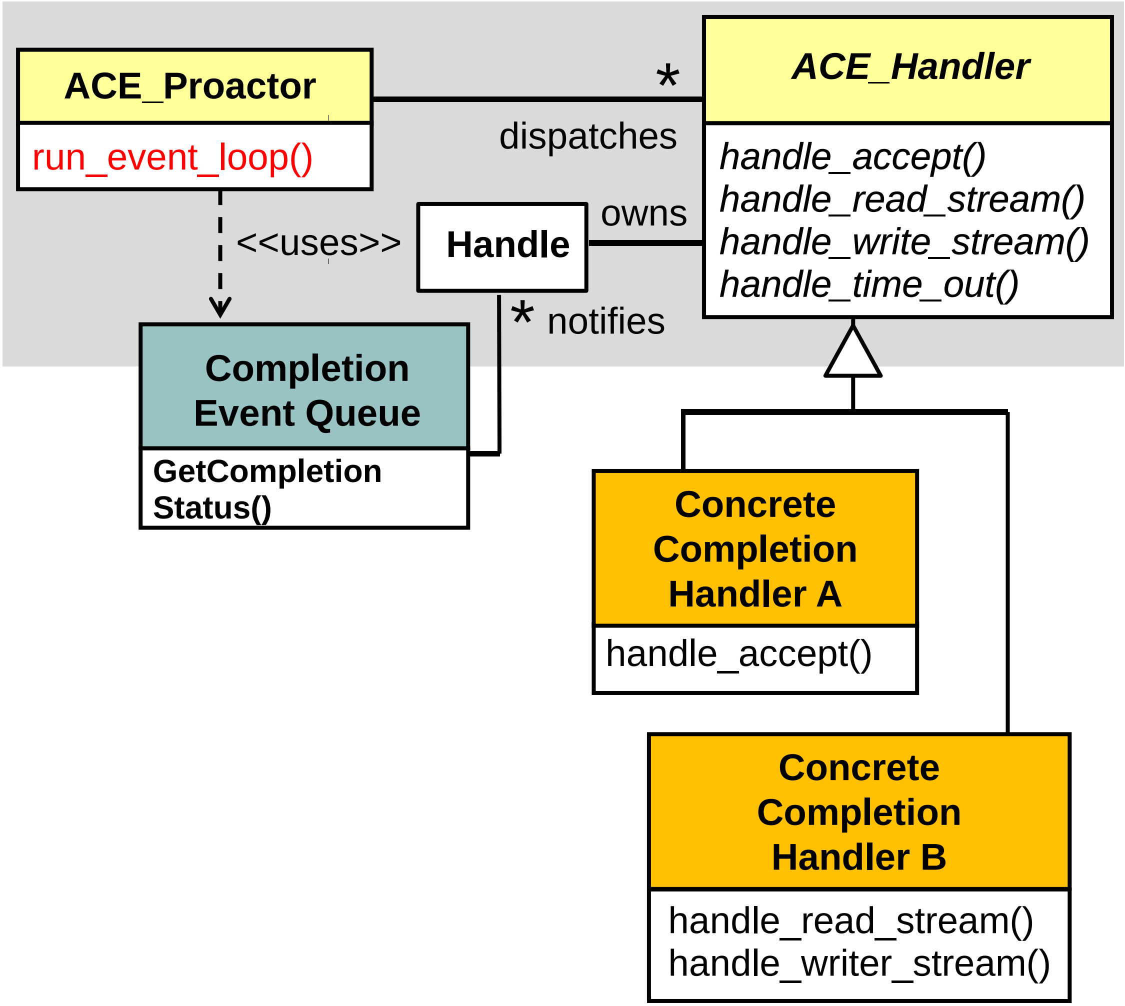


*Inherits from **`ACE_Handler`** to handle asynchronous accept completion events*

The ACE_Proactor Class

Defines an interface for ACE *Proactor* framework capabilities:

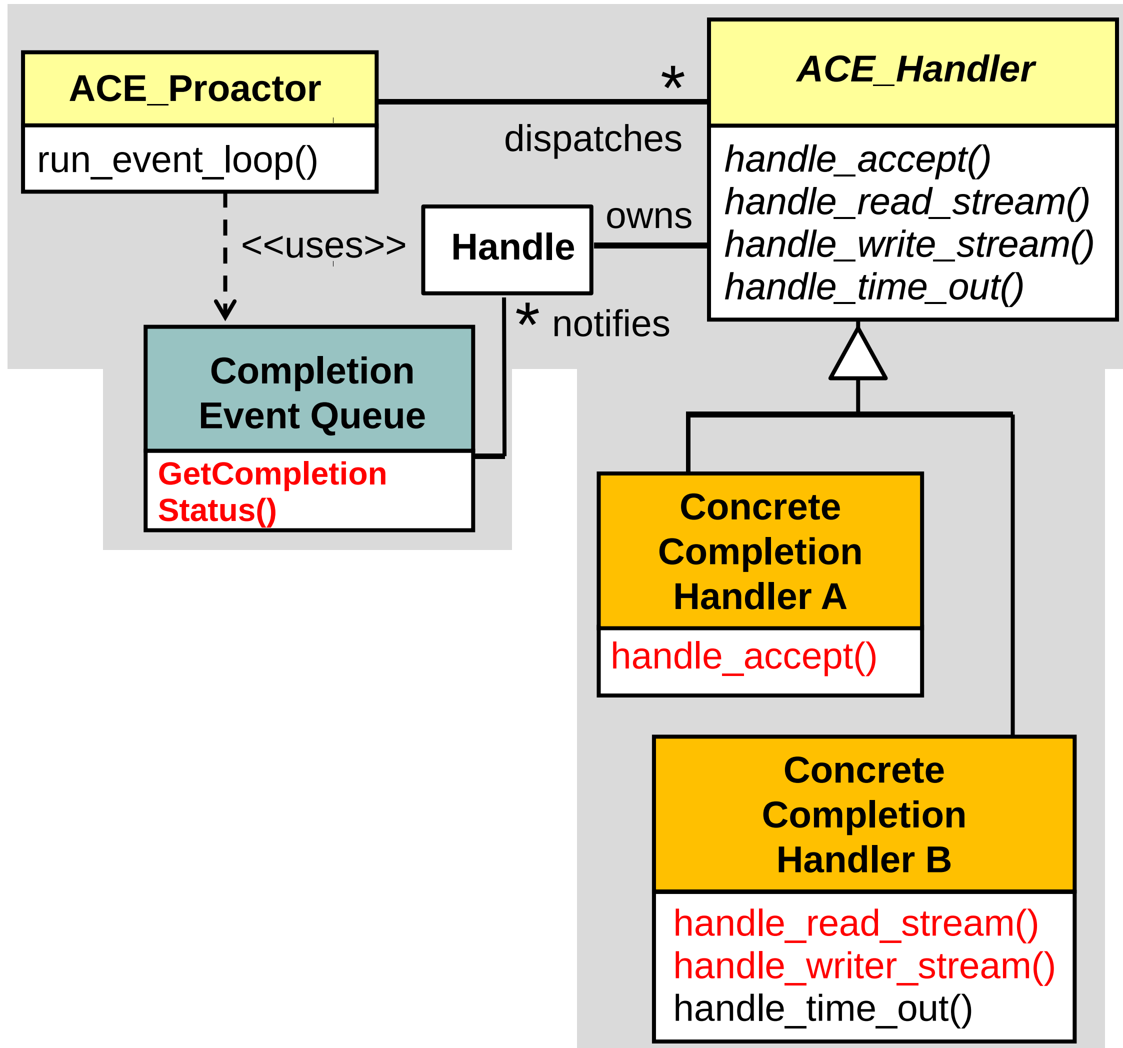
- Centralize event loop processing



The ACE_Proactor Class

Defines an interface for ACE *Proactor* framework capabilities:

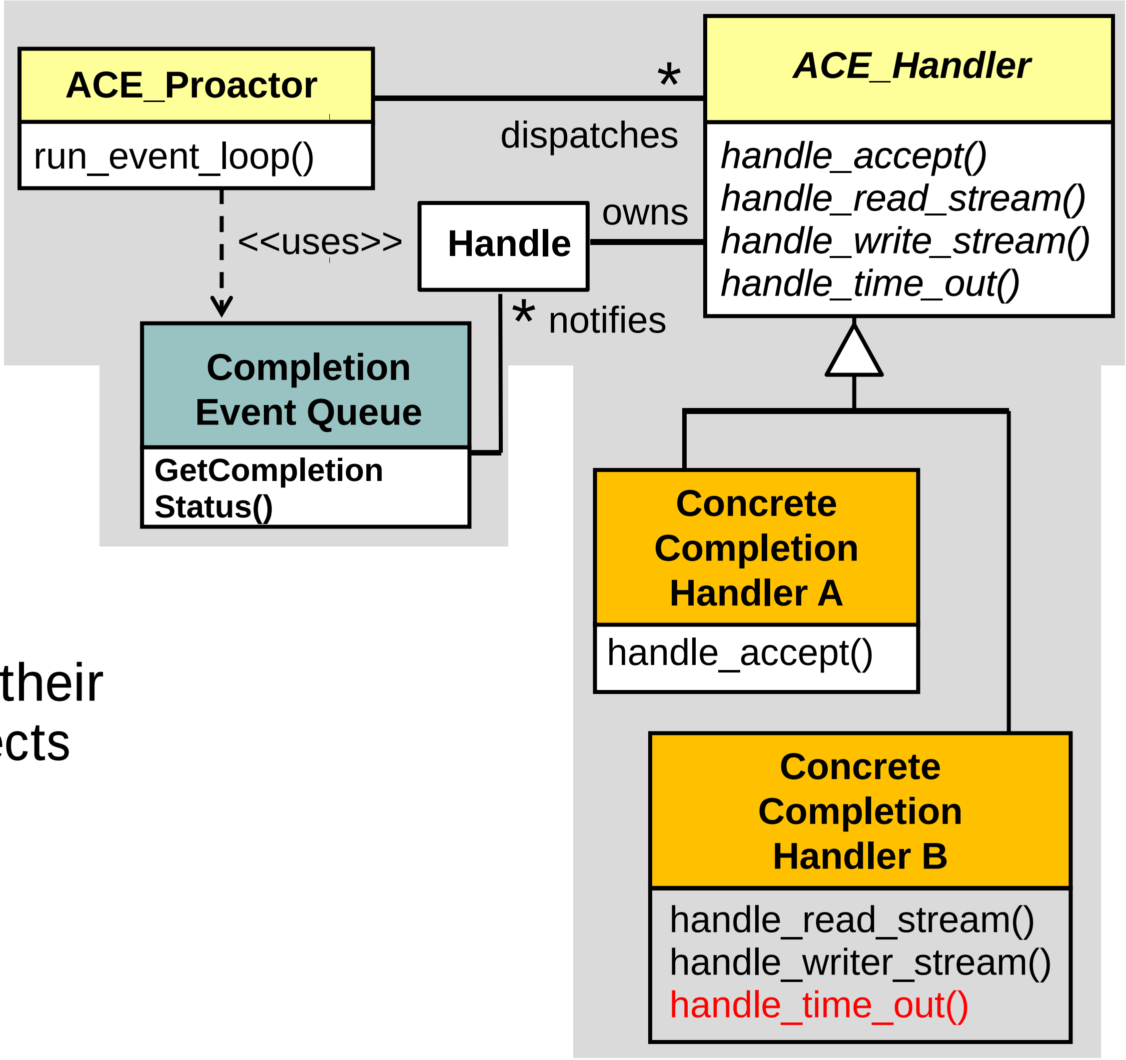
- Centralize event loop processing
- Demuxes completion events to completion handlers & dispatches hook methods on completion handlers



The ACE_Proactor Class

Defines an interface for ACE *Proactor* framework capabilities:

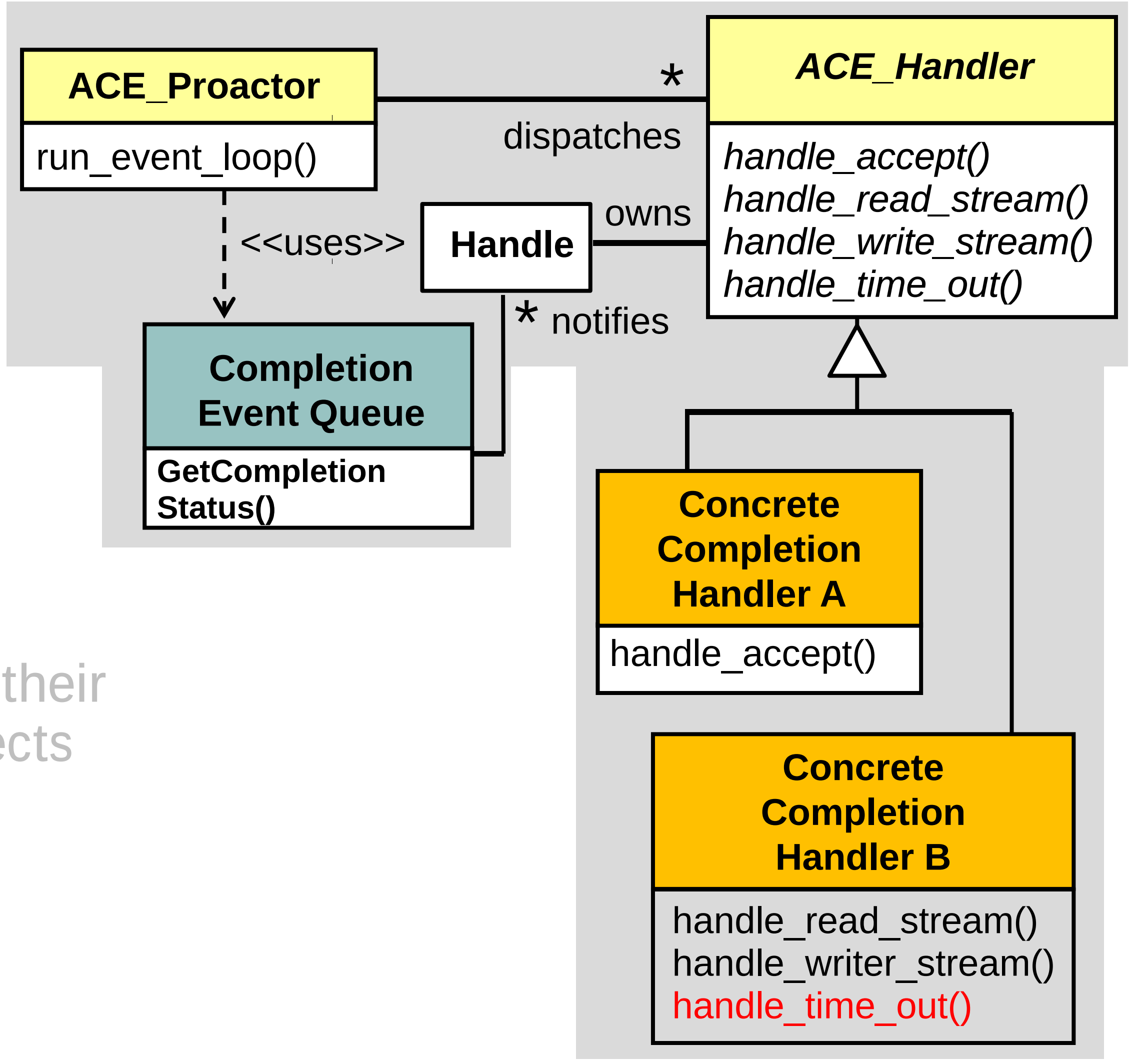
- Centralize event loop processing
- Demuxes completion events to completion handlers & dispatches hook methods on completion handlers
- Dispatch timer expirations to their associated **`ACE_Handle`** objects



The ACE_Proactor Class

Defines an interface for ACE *Proactor* framework capabilities:

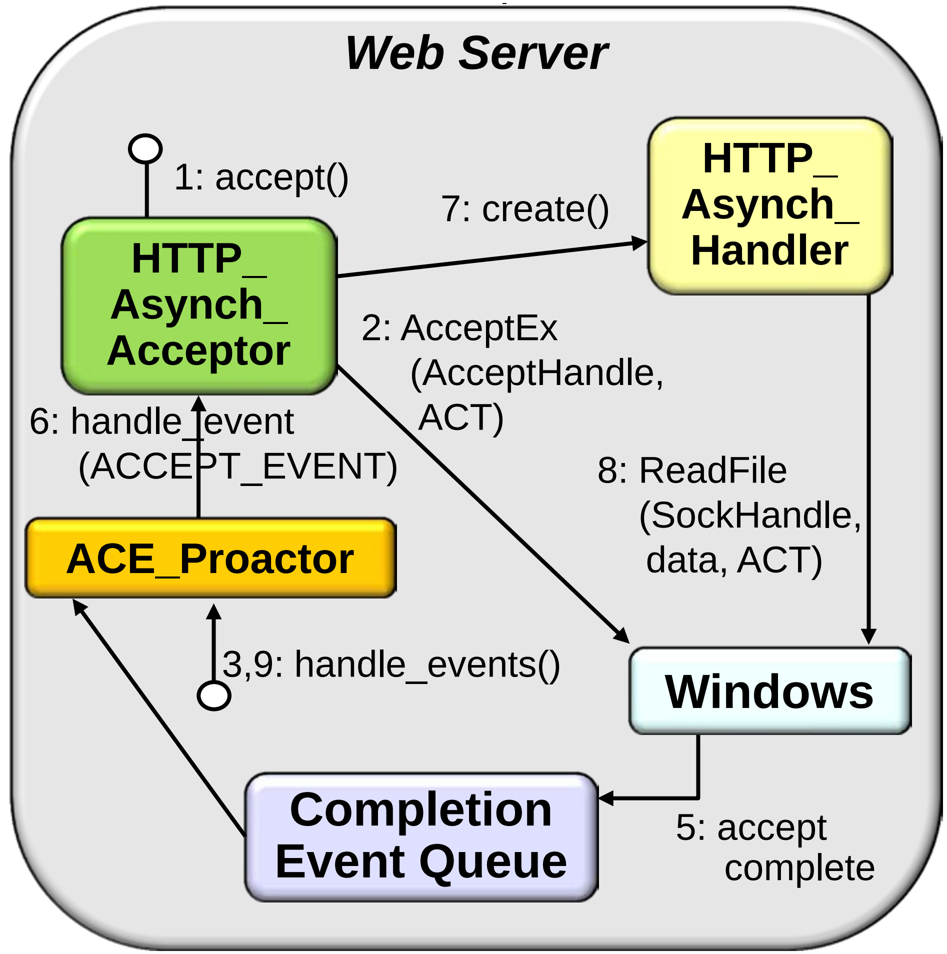
- Centralize event loop processing
- Demuxes completion events to completion handlers & dispatches hook methods on completion handlers
- Dispatch timer expirations to their associated **ACE_Handle** objects



Handles *variability* of asynchronous event handling via a *common* API

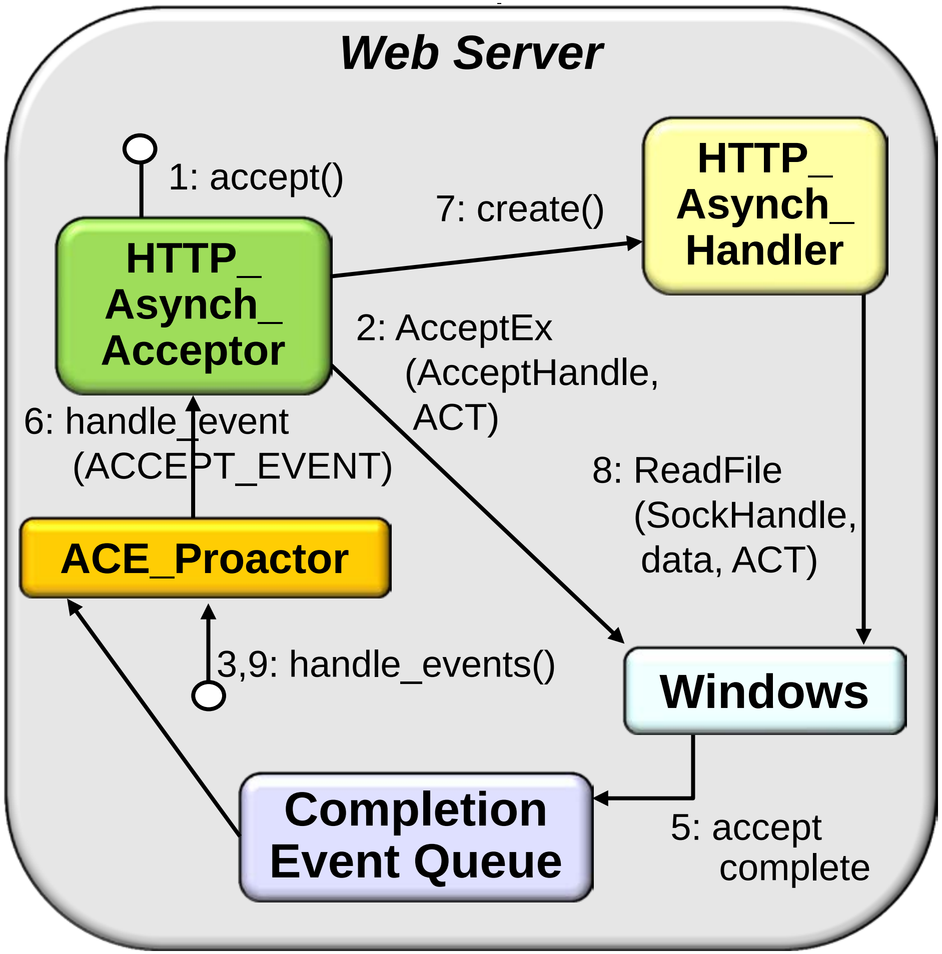
Summary

- A proactive I/O model is harder to program than reactive & synchronous I/O models for several reasons
  - There's a time/space separation between asynchronous invocation & completion handling that requires tricky state management
    - e.g., bookkeeping details & data fragments must be managed explicitly, rather than handled implicitly on the run-time stack



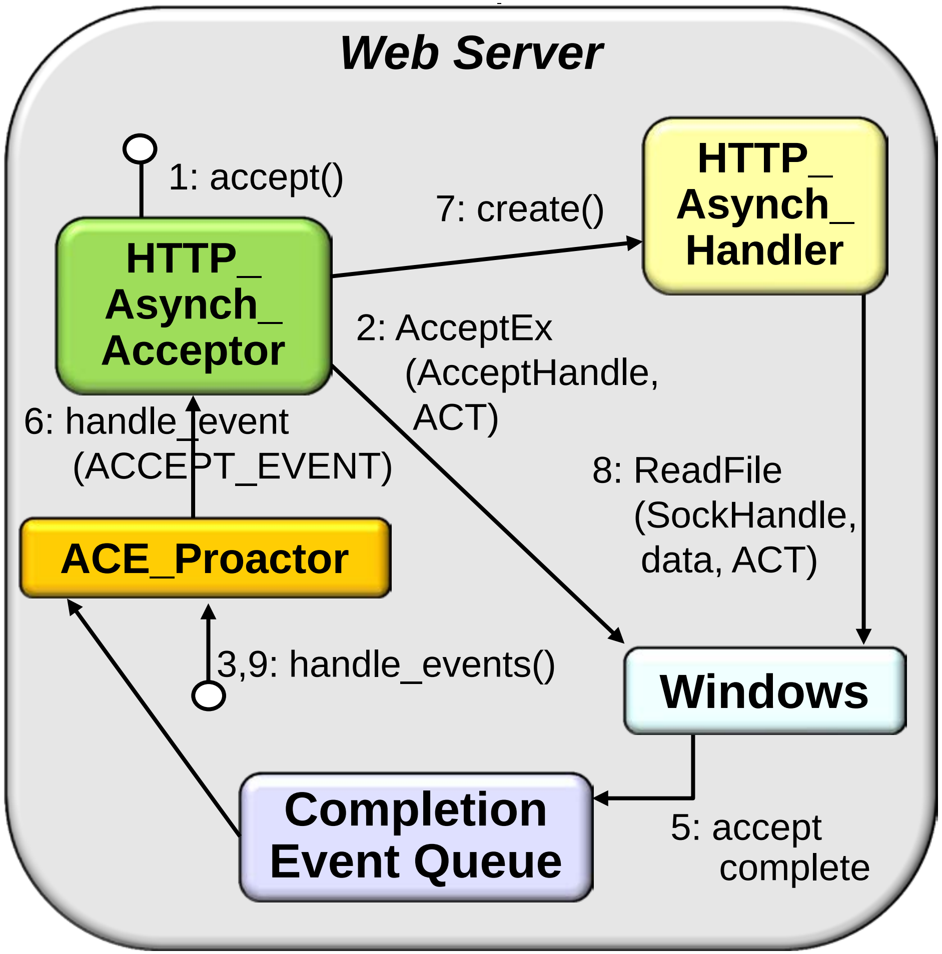
Summary

- A proactive I/O model is harder to program than reactive & synchronous I/O models for several reasons
  - There's a time/space separation between asynchronous invocation & completion handling that requires tricky state management
    - e.g., bookkeeping details & data fragments must be managed explicitly, rather than handled implicitly on the run-time stack
  - There are also significant accidental complexities associated with the quality of asynchronous I/O on many OS platforms



Summary

- A proactive I/O model is harder to program than reactive & synchronous I/O models for several reasons
  - There's a time/space separation between asynchronous invocation & completion handling that requires tricky state management
    - e.g., bookkeeping details & data fragments must be managed explicitly, rather than handled implicitly on the run-time stack
  - There are also significant accidental complexities associated with the quality of asynchronous I/O on many OS platforms



- The ACE *Proactor* framework helps to alleviate many of these complexities

Patterns & Frameworks for Asynchronous Event Handling: Part 3

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

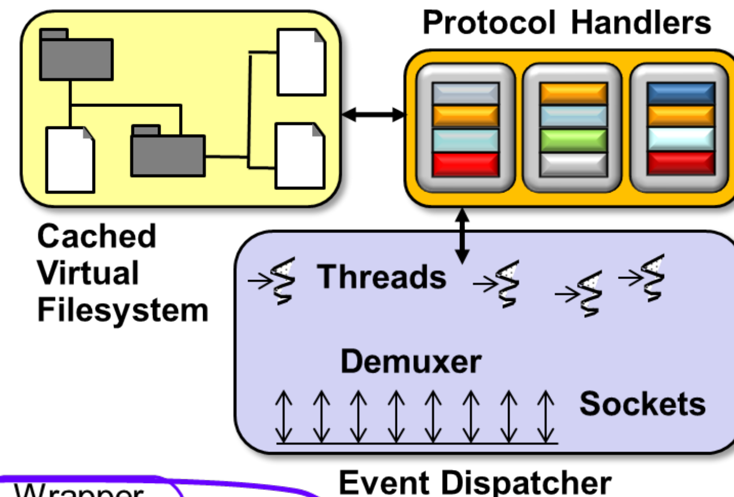
Professor of Computer Science

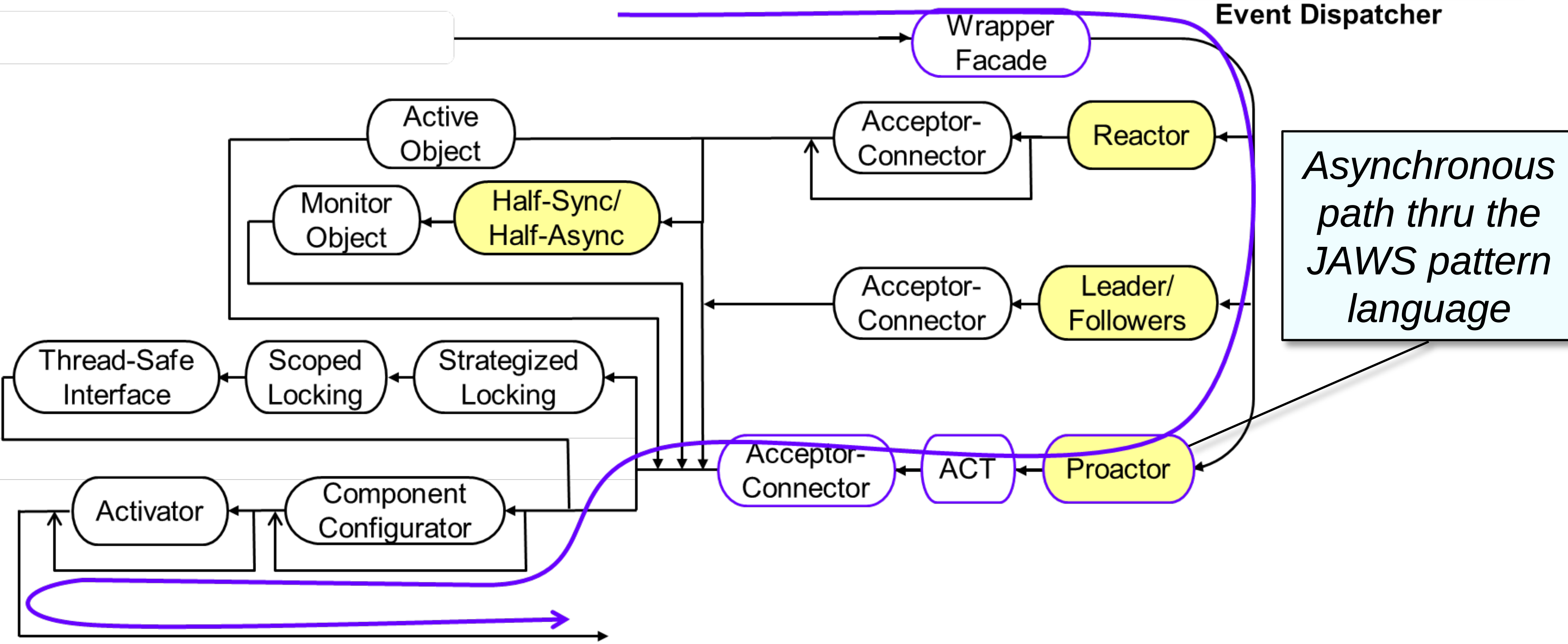
Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

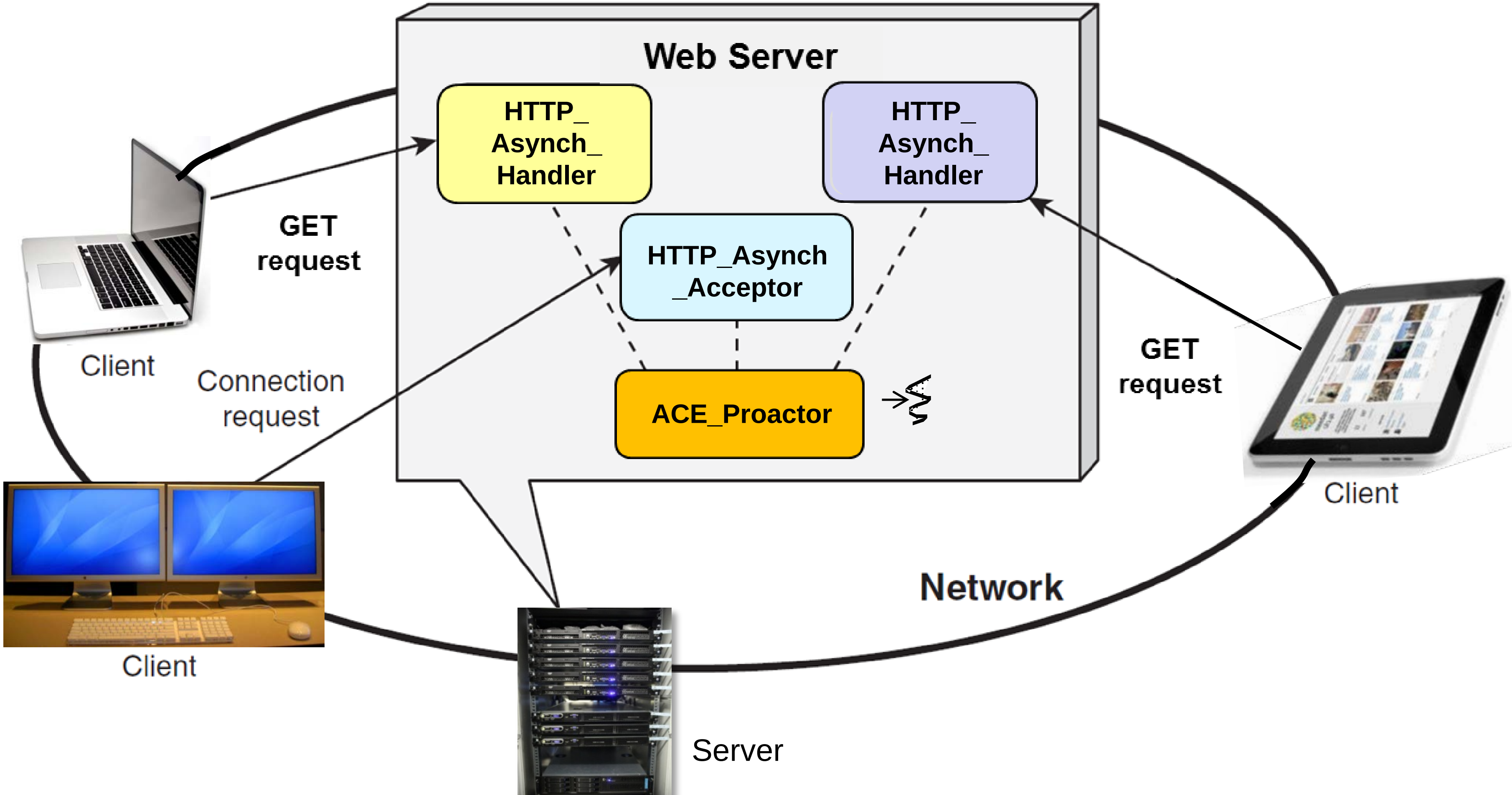
- Describe the *Proactor* pattern
- Describe the ACE *Proactor* framework
- Apply the ACE *Proactor* framework to JAWS





Proactive Processing w/ACE Proactor Framework

Use **`ACE_Service_Handler`** & **`ACE_Asynch_Acceptor`** to implement a JAWS web server based on the *Proactor* pattern



This implementation only uses a single thread, but is still scalable on Windows

Proactive Processing with ACE_Service_Handler

Implements HTTP using asynchronous operations

```
class HTTP_Asynch_Handler : public ACE_Service_Handler {
private:
  ACE_Proactor *proactor_; // Cached Proactor.
  ACE_Mem_Map file_; // Memory-mapped file
  ACE_HANDLE handle_; // Socket endpoint

  HTTP_Request request_;

  ACE_Asynch_Read_Stream read_stream_;
  ACE_Asynch_Write_Stream write_stream_;

public:
  HTTP_Asynch_Handler (ACE_Proactor *proactor)
                                        : proactor_ (proactor) {}
// ... Continued below
```

Hold HTTP request while it's being processed

Read/write asynchronous socket I/O

Proactive Processing with ACE_Service_Handler

Hook method invoked by `HTTP_Asynch_Acceptor`

```
virtual void open (ACE_HANDLE new_handle,
                   ACE_Message_Block &mb) {
  request_.state_ = INCOMPLETE; // Initialize state for request
  io_handle_ = new_handle; // Store handle to the open socket


  read_stream_.open
    (*this,
     io_handle_,
     0, proactor_);
```

Initialize `ACE_Async_Read_Stream`, with `*this` as completion handler

Start asynchronous read operation on connected socket

```
  read_stream_.read
    (request_.message (), request_.size ());
}
```

Proactive Processing with ACE_Service_Handler

Completion event handling method dispatched by ACE *Proactor* framework

```
virtual void handle_read_stream
          (const ACE_Asynch_Read_Stream::Result &result) {

  if (request_complete (result))
    handle_request ();
```

Got the entire read request, so handle it

```
  else
    read_stream_.read (request_.message (),
                       request_.size ());
}
  // ...
}
```

Didn't get entire request, so initiate a new asynchronous `read()` operation to try & get the remainder

Proactive Processing with ACE_Service_Handler

Handle processing of a completed request

Switch on the HTTP command type

Memory map the requested content & invoke an asynchronous write operation to transmit it to the client

```
void handle_request () {

  switch (request_.command ()) {

  case HTTP_Request::GET: // Request to download a file

    file_.map (request_.filename ());
    write_stream_.write (file_.addr (),
                         file_.size ());
    break;

  case HTTP_Request::PUT: // Request to upload a file
  ...
```

We could also use **transmit_file()** here

Proactive Processing with ACE_Asynch_Acceptor

Implements HTTP using asynchronous operations

```
class HTTP_Asynch_Acceptor
 : public ACE_Asynch_Acceptor<HTTP_Asynch_Handler> {
public:
  HTTP_Asynch_Acceptor (ACE_INET_Addr addr,
                        ACE_Proactor *proactor) {

    open (addr, 0, false, ACE_DEFAULT_ASYNCH_BACKLOG, 1,
          proactor);
 }
```

Starts multiple asynchronous accept requests on addr

Applying the ACE Proactor framework to JAWS

```
const u_short PORT = 80;

int main (int argc, char *argv[]) {
  ACE_INET_Addr addr (argc == 1 ? PORT : atoi (argv[1]));
```

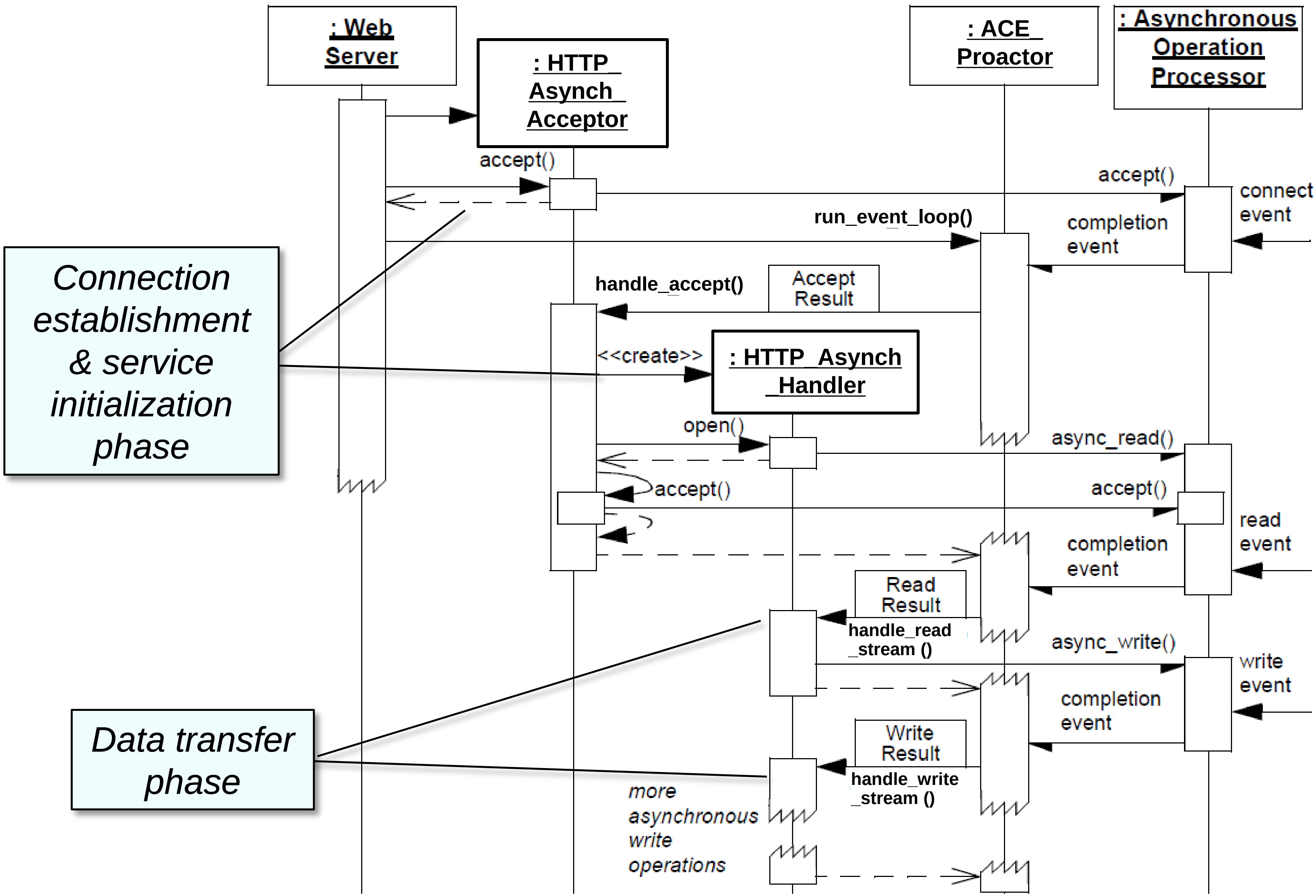
Associate the `HTTP_Asynch_Acceptor`'s passive-mode socket handle with the `ACE_Proactor` singleton's completion port & invoke multiple asynchronous accept operations to initiative proactive web serve processing

```
  HTTP_Asynch_Acceptor acceptor (addr,
                                 ACE_Proactor::instance ());
```

Event loop processes client connection requests & HTTP requests proactively

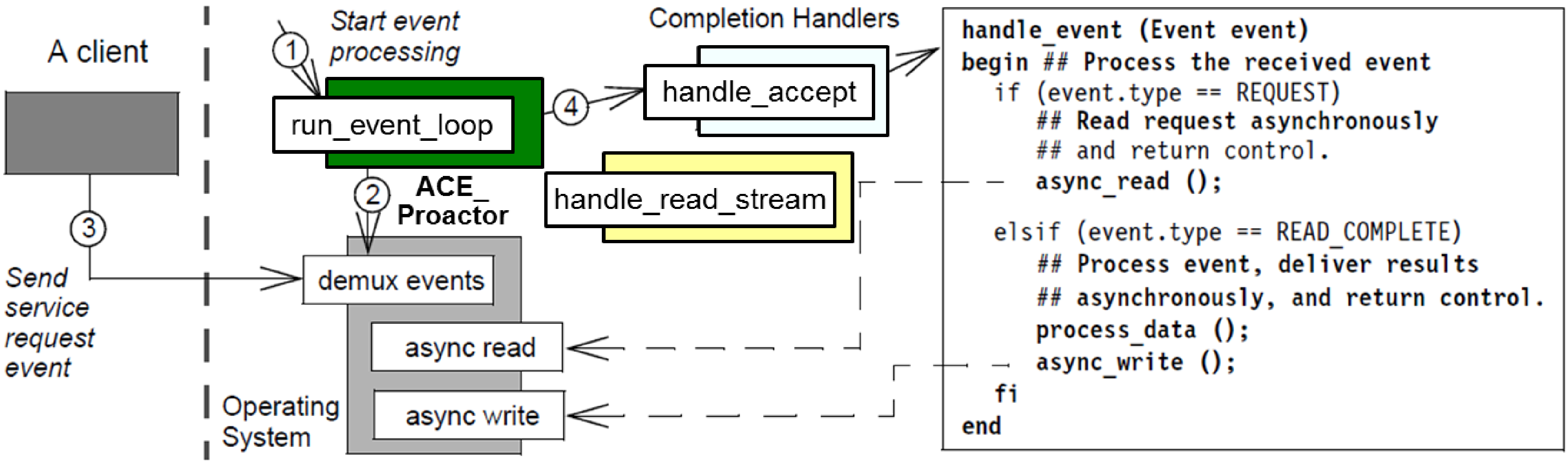
```
  for (;;)
    ACE_Proactor::instance ()->run_proactor_event_loop ();
  ...
```

Sequence Diagram of ACE Proactor Web Server



Summary

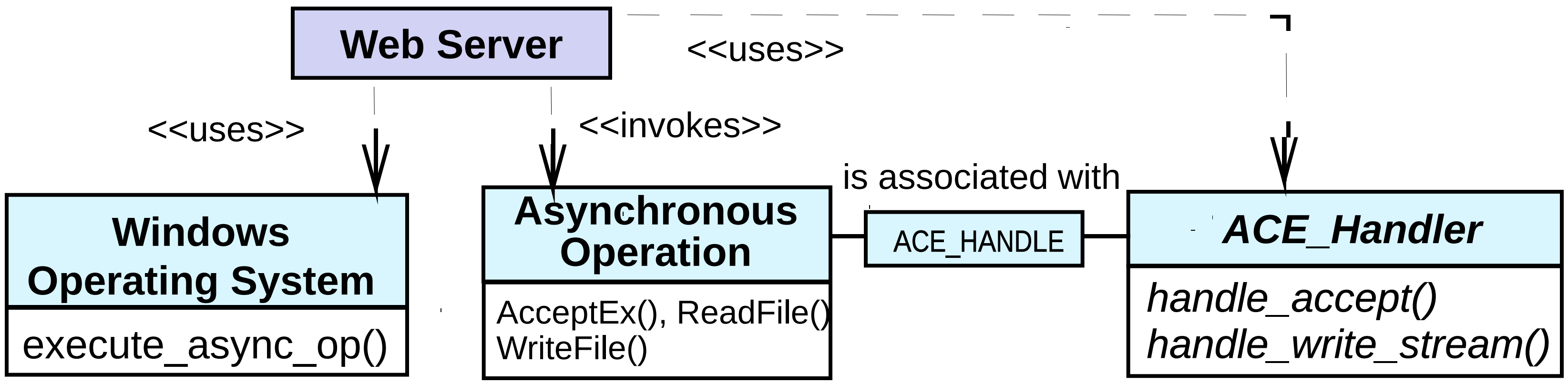
- The ACE *Proactor* framework alleviates reactive I/O bottlenecks without introducing the complexity & overhead of synchronous I/O & multi-threading



Summary

- The ACE *Proactor* framework alleviates reactive I/O bottlenecks without introducing the complexity & overhead of synchronous I/O & multi-threading
- This framework allows an app to execute I/O operations via two phases:

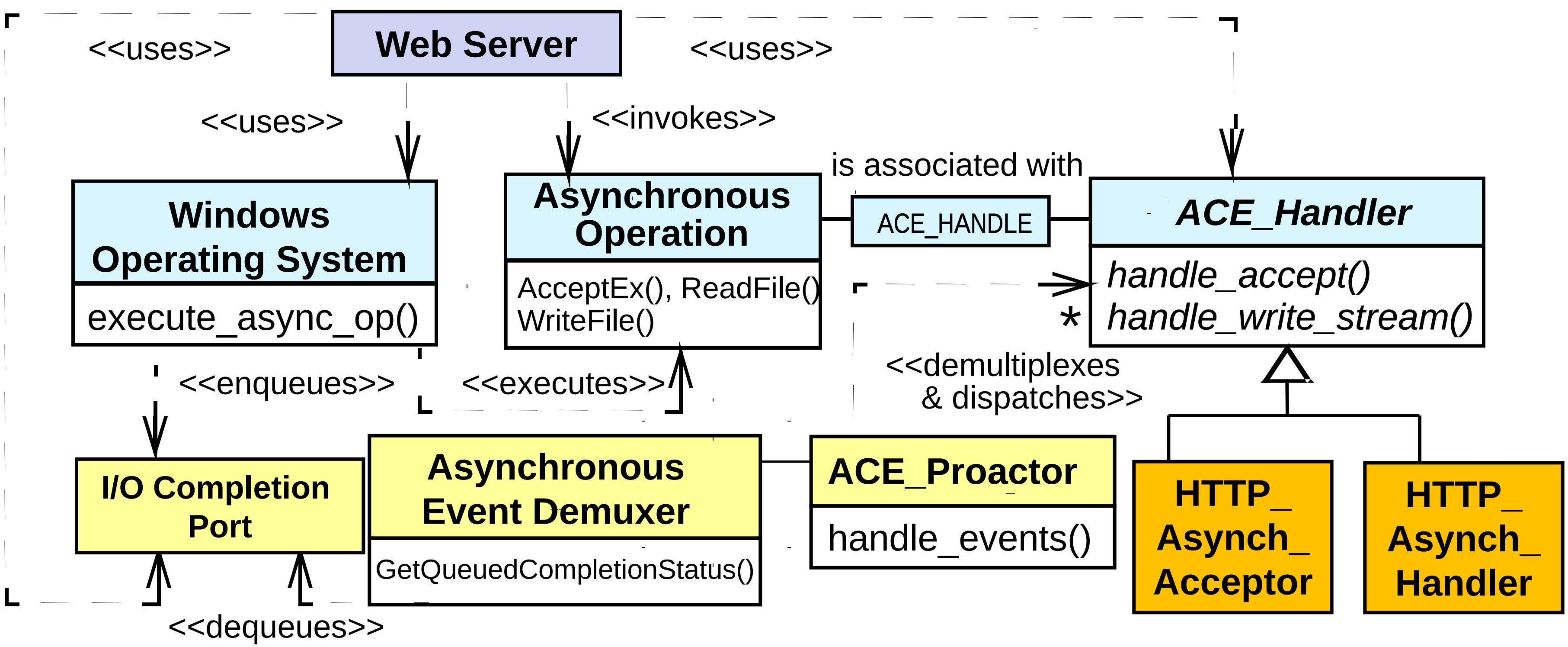
1. An app can initiate one or more asynchronous I/O operations on multiple I/O handles in parallel without having to wait until they complete



Summary

- The ACE *Proactor* framework alleviates reactive I/O bottlenecks without introducing the complexity & overhead of synchronous I/O & multithreading
- This framework allows an app to execute I/O operations via two phases:

1. An app can initiate one or more asynchronous I/O operations on multiple I/O handles in parallel without having to wait until they complete



2. As each operation completes, the OS notifies an app-defined completion handler that then processes the results from the completed I/O operation

Patterns & Frameworks for Asynchronous Event Handling: Part 4

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

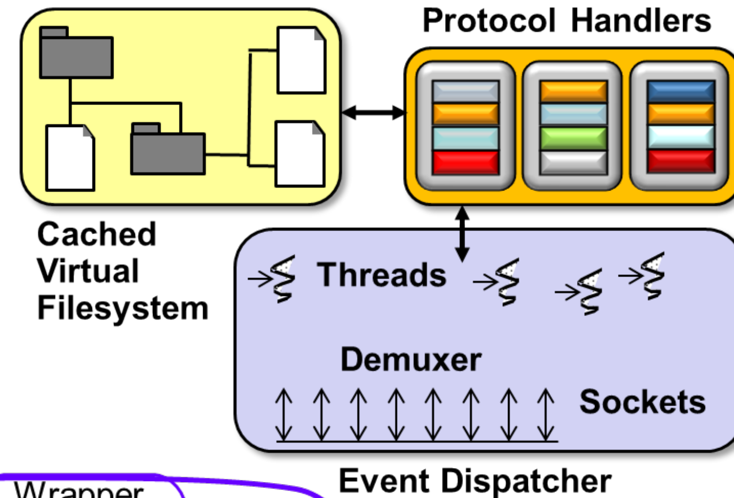
Professor of Computer Science

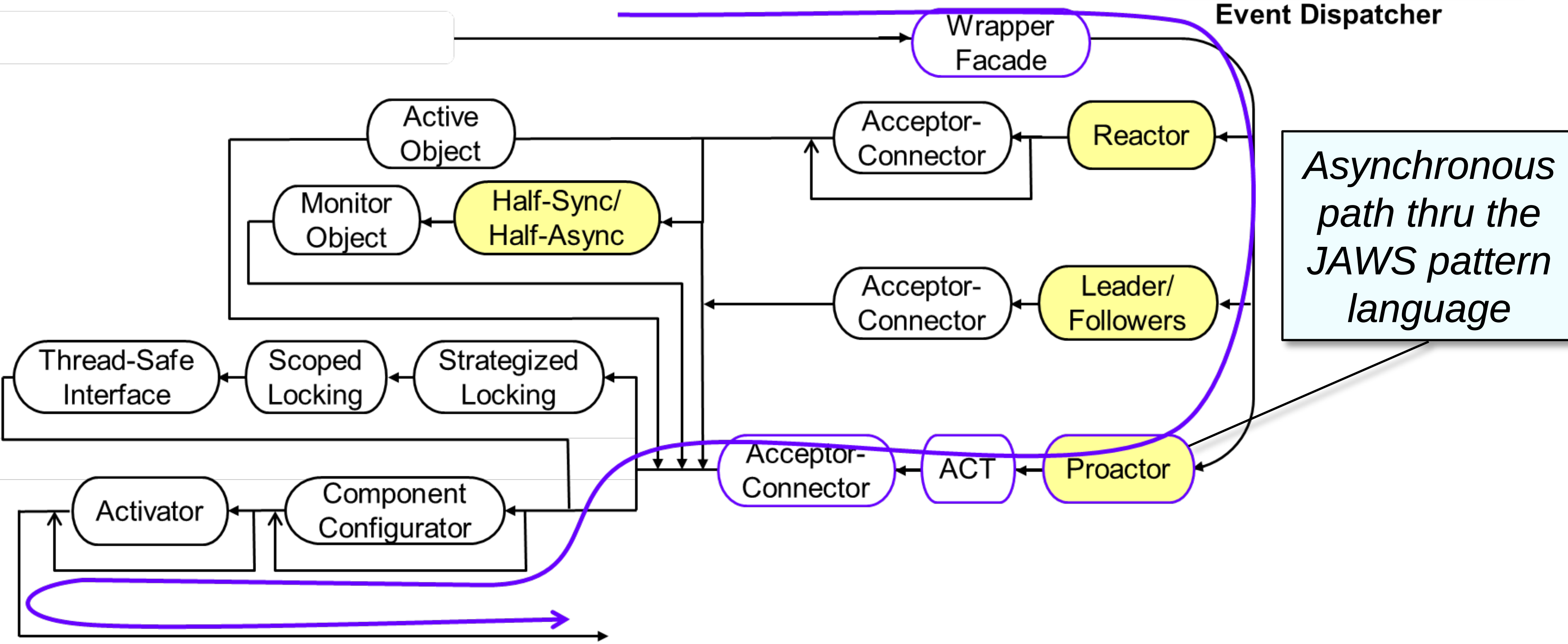
Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

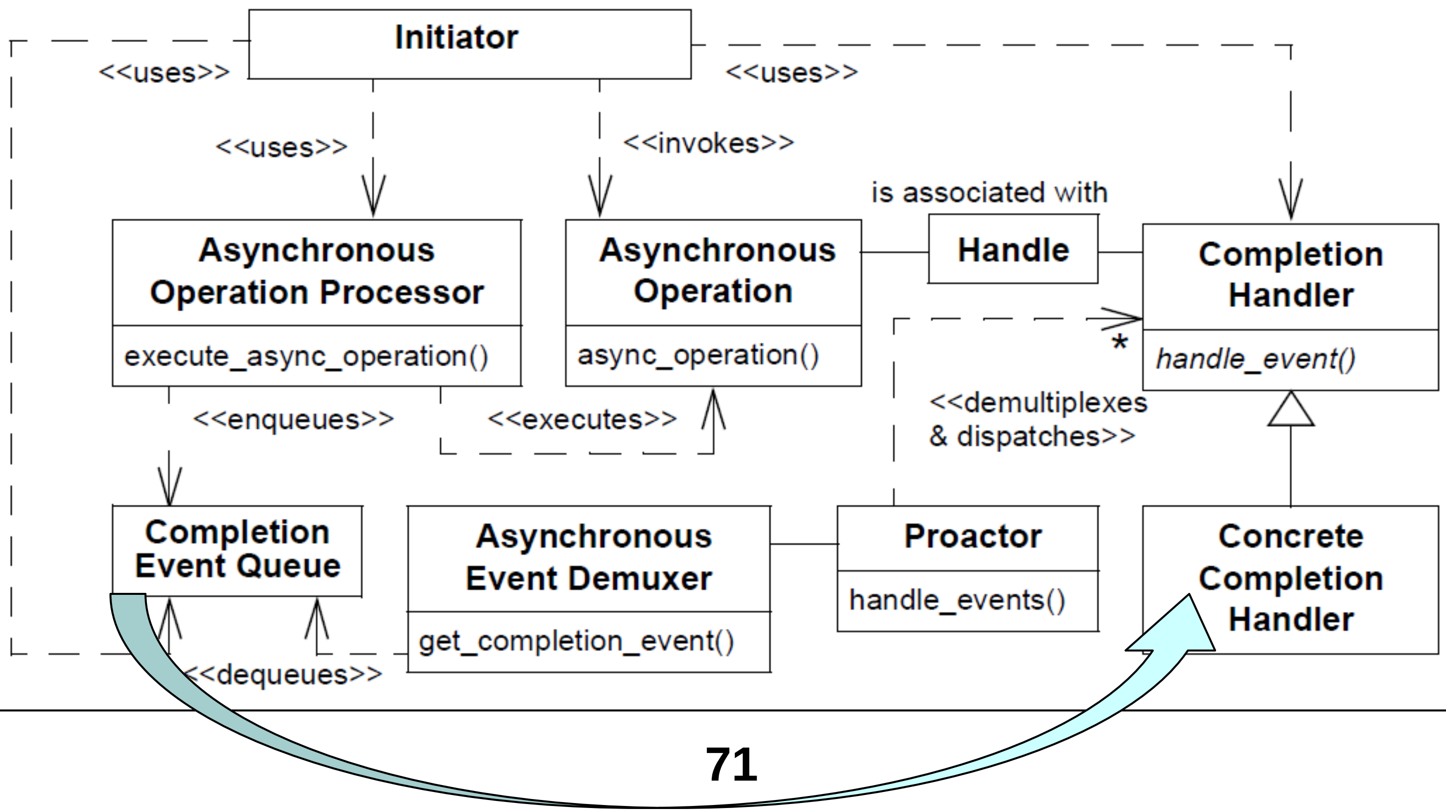
- Describe the *Proactor* pattern
- Describe the ACE *Proactor* framework
- Apply the ACE *Proactor* framework to JAWS
- Describe the *Asynchronous Completion Token* pattern & apply it to JAWS





Efficiently Demuxing Asynch Event Completions

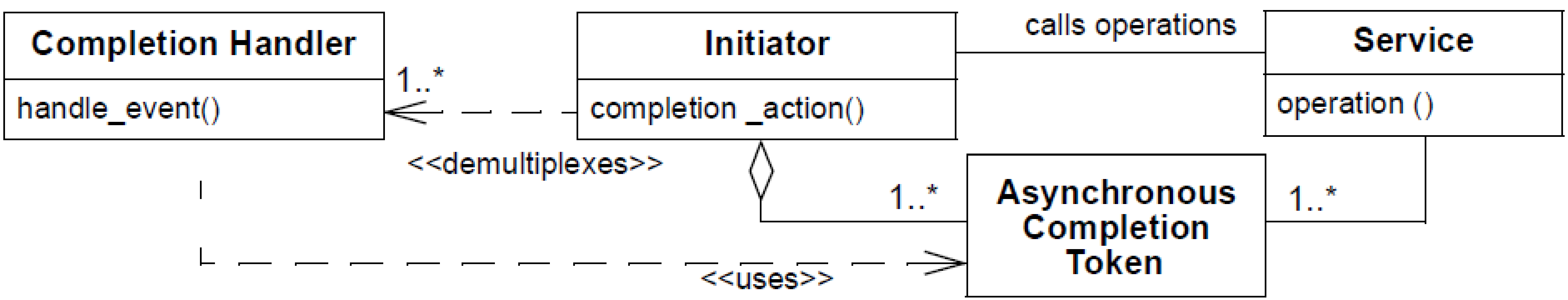
Context	Problem
• In a proactive web server async I/O operations will yield I/O completion event responses that must be processed efficiently	• Need to minimize time/ space used to demux completion events to their associated completion handler



Efficiently Demuxing Asynch Event Completions

Context	Problem	Solution
• In a proactive web server async I/O operations will yield I/O completion event responses that must be processed efficiently	• Need to minimize time/ space used to demux completion events to their associated completion handler	• Apply the *Asynchronous Completion Token* pattern to demux & process the responses of asynchronous operations efficiently

Asynchronous Completion Token allows an app to efficiently demux & process the responses of asynchronous operations it invokes on services

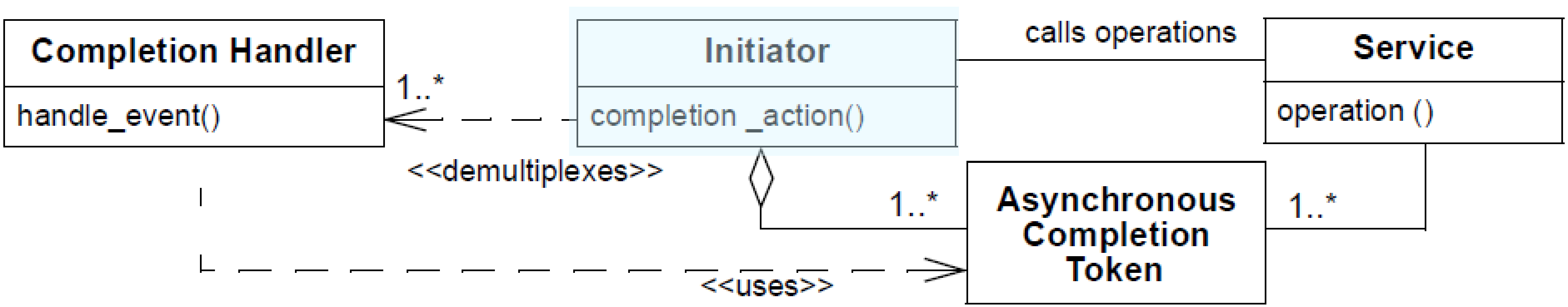


Structure

Efficiently Demuxing Asynch Event Completions

Context	Problem	Solution
• In a proactive web server async I/O operations will yield I/O completion event responses that must be processed efficiently	• Need to minimize time/ space used to demux completion events to their associated completion handler	• Apply the *Asynchronous Completion Token* pattern to demux & process the responses of asynchronous operations efficiently

Asynchronous Completion Token allows an app to efficiently demux & process the responses of asynchronous operations it invokes on services

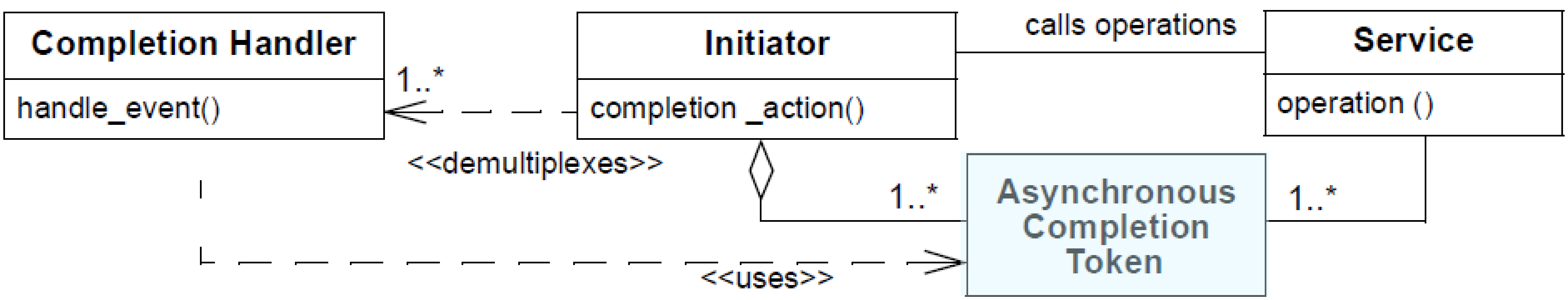


Structure

Efficiently Demuxing Asynch Event Completions

Context	Problem	Solution
• In a proactive web server async I/O operations will yield I/O completion event responses that must be processed efficiently	• Need to minimize time/ space used to demux completion events to their associated completion handler	• Apply the *Asynchronous Completion Token* pattern to demux & process the responses of asynchronous operations efficiently

Asynchronous Completion Token allows an app to efficiently demux & process the responses of asynchronous operations it invokes on services

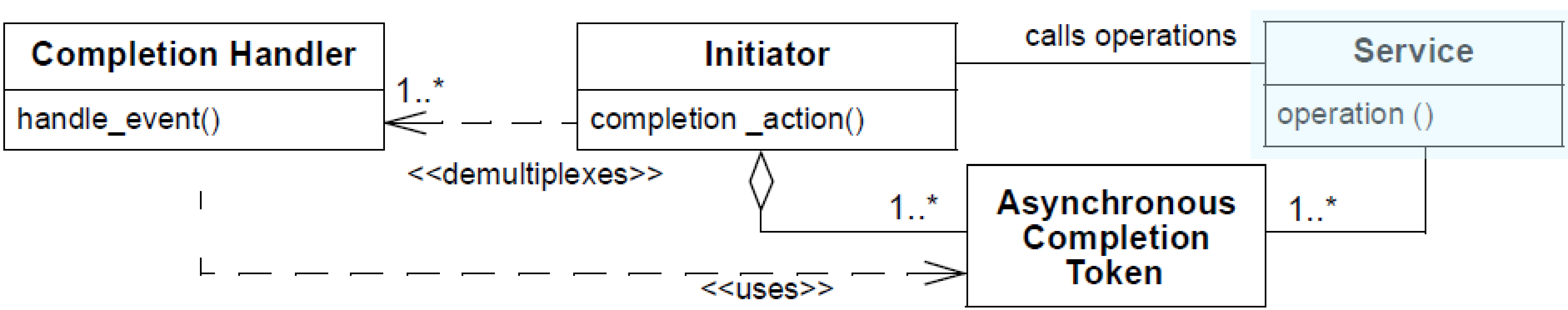


Structure

Efficiently Demuxing Asynch Event Completions

Context	Problem	Solution
• In a proactive web server async I/O operations will yield I/O completion event responses that must be processed efficiently	• Need to minimize time/ space used to demux completion events to their associated completion handler	• Apply the *Asynchronous Completion Token* pattern to demux & process the responses of asynchronous operations efficiently

Asynchronous Completion Token allows an app to efficiently demux & process the responses of asynchronous operations it invokes on services

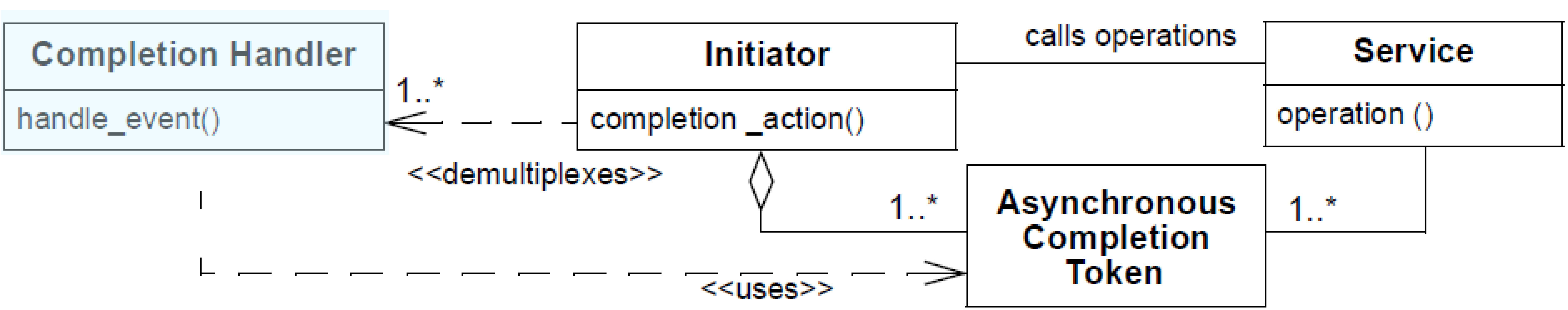


Structure

Efficiently Demuxing Asynch Event Completions

Context	Problem	Solution
• In a proactive web server async I/O operations will yield I/O completion event responses that must be processed efficiently	• Need to minimize time/ space used to demux completion events to their associated completion handler	• Apply the *Asynchronous Completion Token* pattern to demux & process the responses of asynchronous operations efficiently

Asynchronous Completion Token allows an app to efficiently demux & process the responses of asynchronous operations it invokes on services

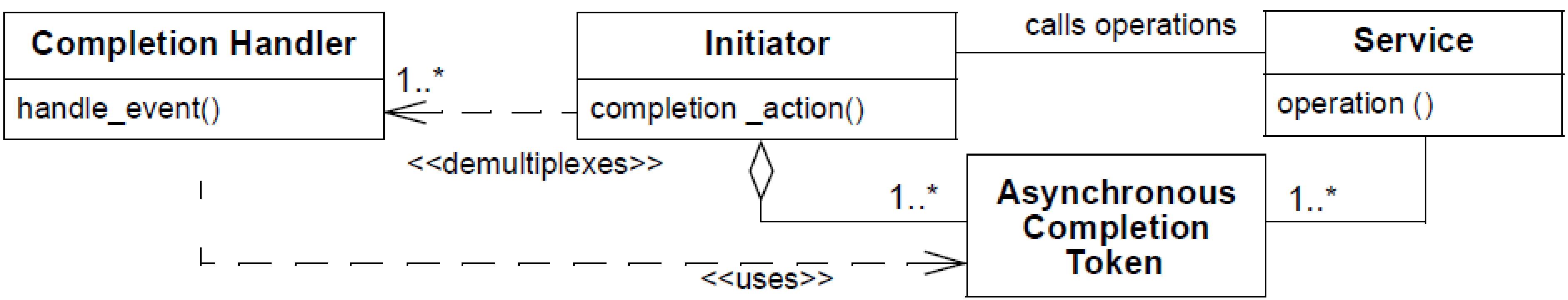


Structure

Efficiently Demuxing Asynch Event Completions

Context	Problem	Solution
• In a proactive web server async I/O operations will yield I/O completion event responses that must be processed efficiently	• Need to minimize time/ space used to demux completion events to their associated completion handler	• Apply the *Asynchronous Completion Token* pattern to demux & process the responses of asynchronous operations efficiently

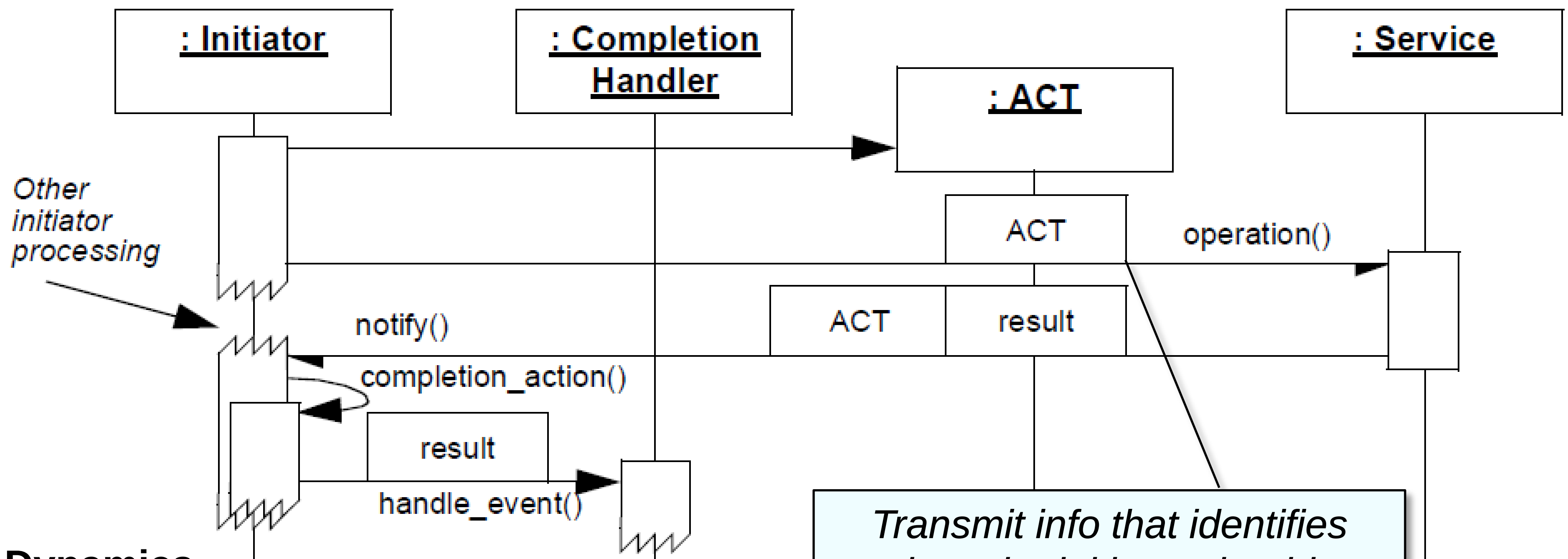
Asynchronous Completion Token allows an app to efficiently demux & process the responses of asynchronous operations it invokes on services



Structure

Efficiently Demuxing Asynch Event Completions

Context	Problem	Solution
• In a proactive web server async I/O operations will yield I/O completion event responses that must be processed efficiently	• Need to minimize time/ space used to demux completion events to their associated completion handler	• Apply the *Asynchronous Completion Token* pattern to demux & process the responses of asynchronous operations efficiently

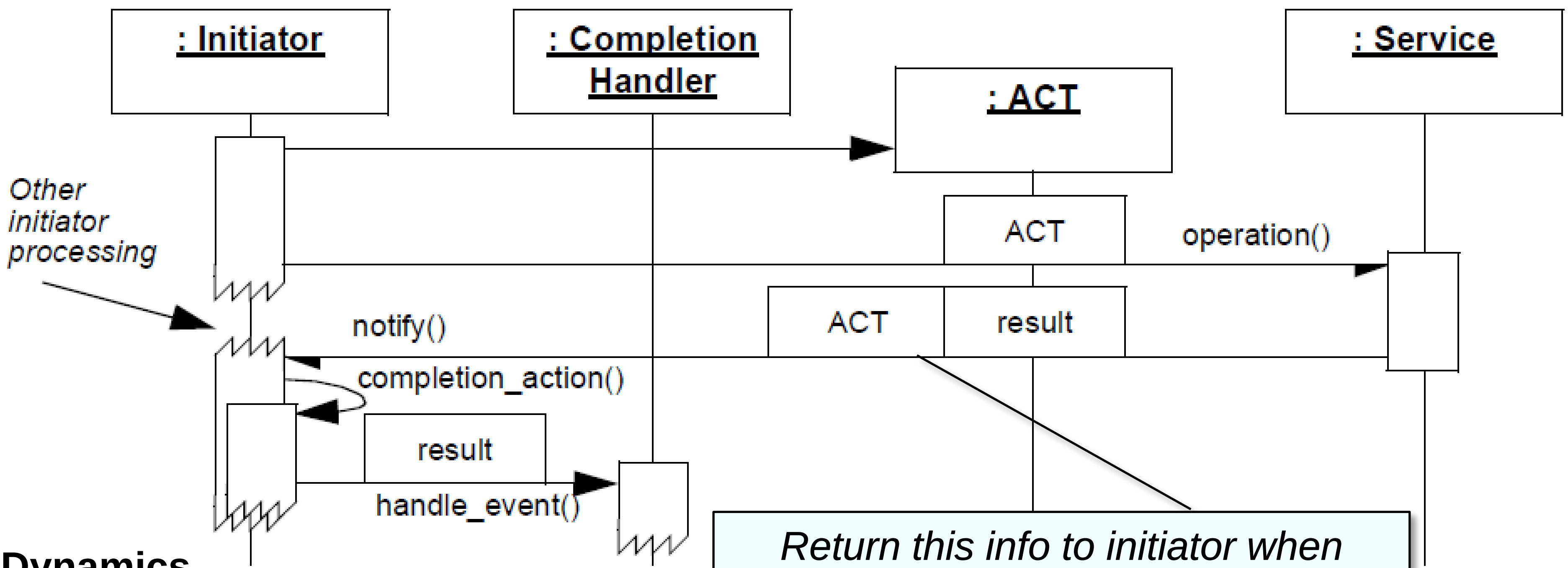


Dynamics

Transmit info that identifies how the initiator should process the service's response

Efficiently Demuxing Asynch Event Completions

Context	Problem	Solution
• In a proactive web server async I/O operations will yield I/O completion event responses that must be processed efficiently	• Need to minimize time/ space used to demux completion events to their associated completion handler	• Apply the *Asynchronous Completion Token* pattern to demux & process the responses of asynchronous operations efficiently



Return this info to initiator when operation finishes, so it can be used to demux the response efficiently

Dynamics

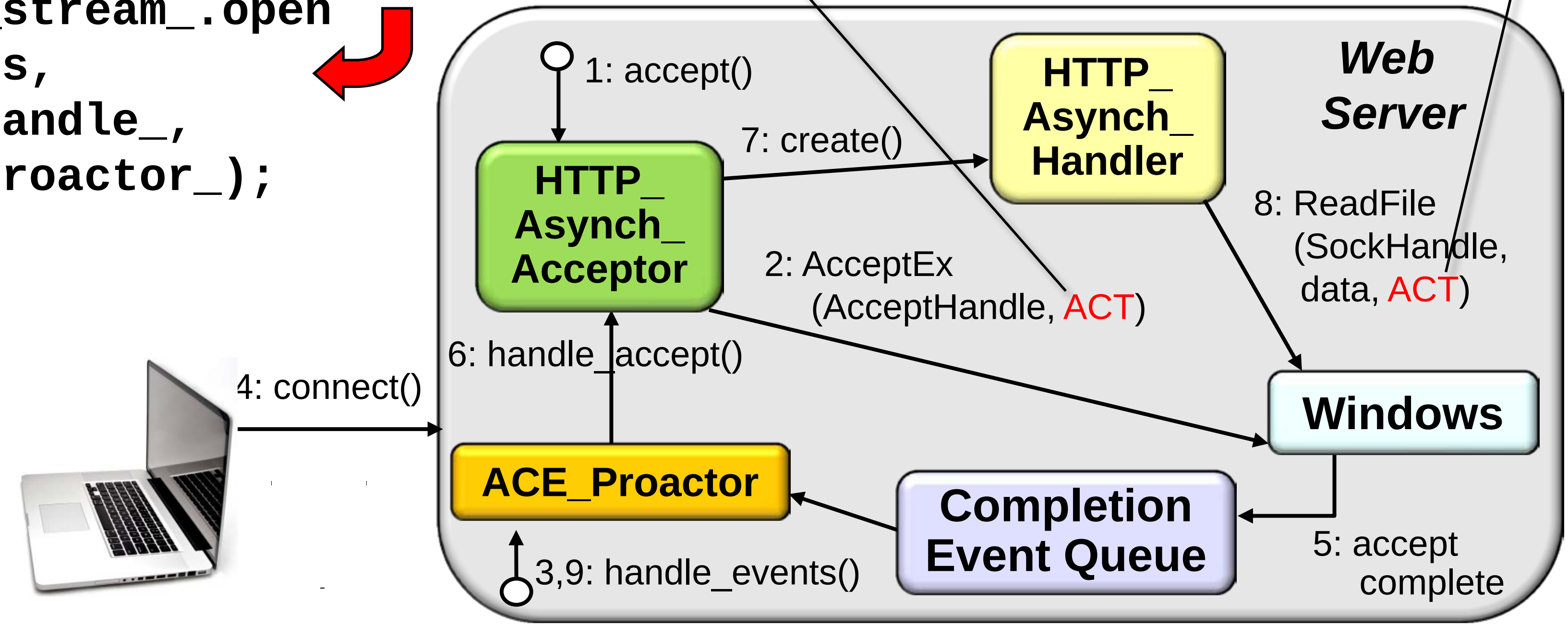
Applying the Asynchronous Completion Token Pattern in JAWS

```
class HTTP_Asynch_Handler :
     public ACE_Service_Handler {

  virtual void open (.....) {
    ...
```

Pass this as ACT

```
    read_stream_.open
    (*this,
     io_handle_,
     0, proactor_);

    ...
```

Asynchronous completion tokens (ACTs) are passed to all asynchronous operations to efficiently indicate completion handlers

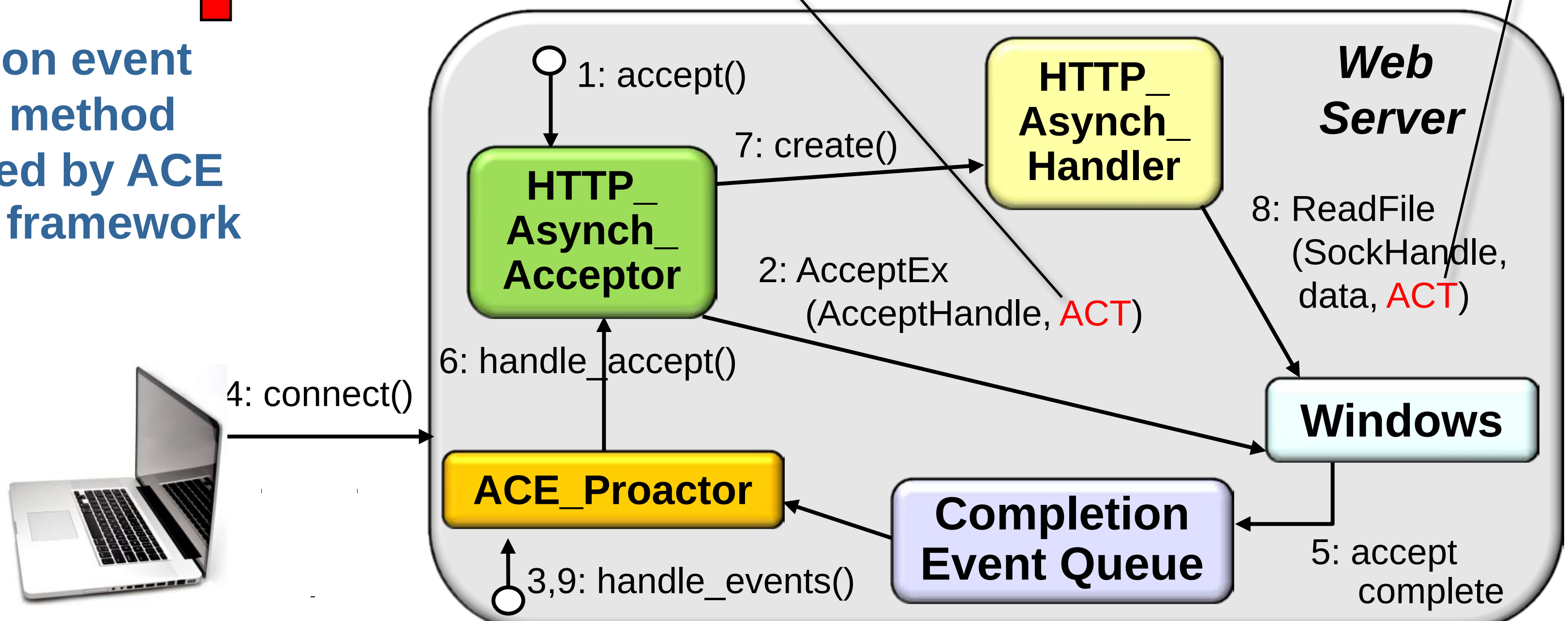


Applying the Asynchronous Completion Token Pattern in JAWS

```
class HTTP_Asynch_Handler :
    public ACE_Service_Handler {

  virtual void handle_read_stream
                          (.....) {
  ...
}
```

Completion event handling method dispatched by ACE *Proactor* framework

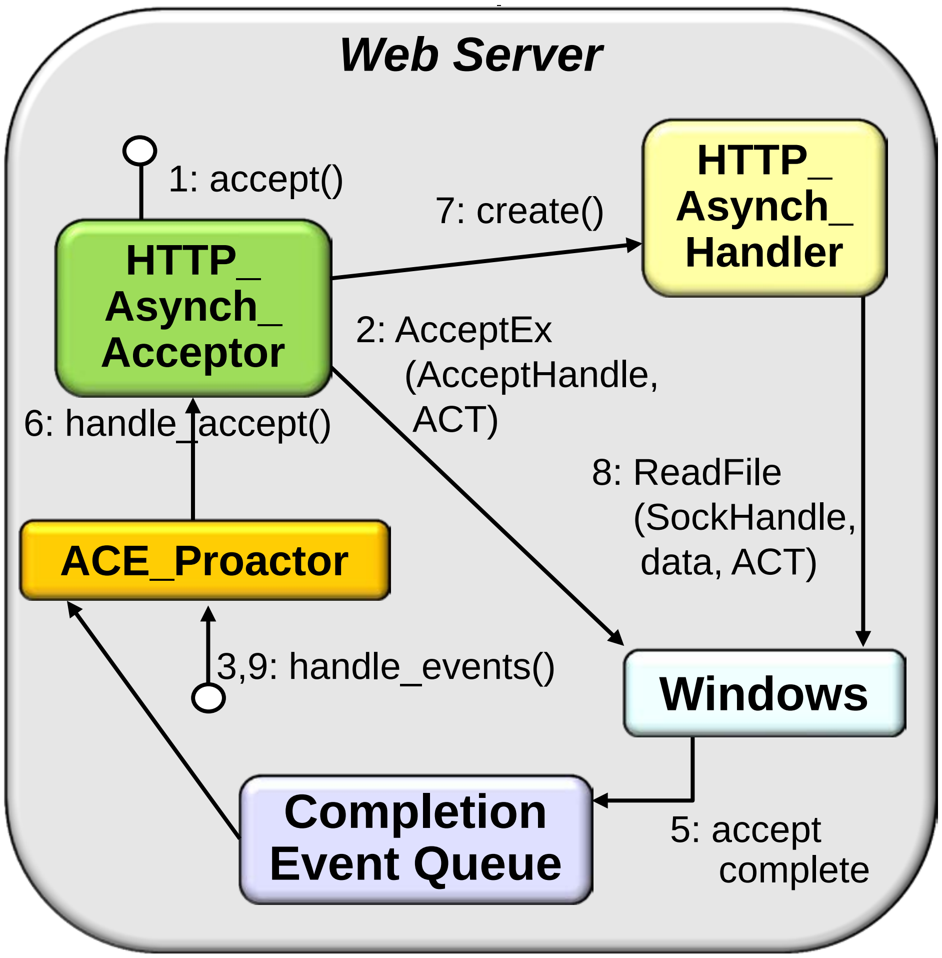
Asynchronous completion tokens (ACTs) are passed to all asynchronous operations to efficiently indicate completion handlers



Benefits of Asynchronous Completion Token

Simplified initiator data structures

- Initiators need not maintain complex data structures to associate responses with completion handlers



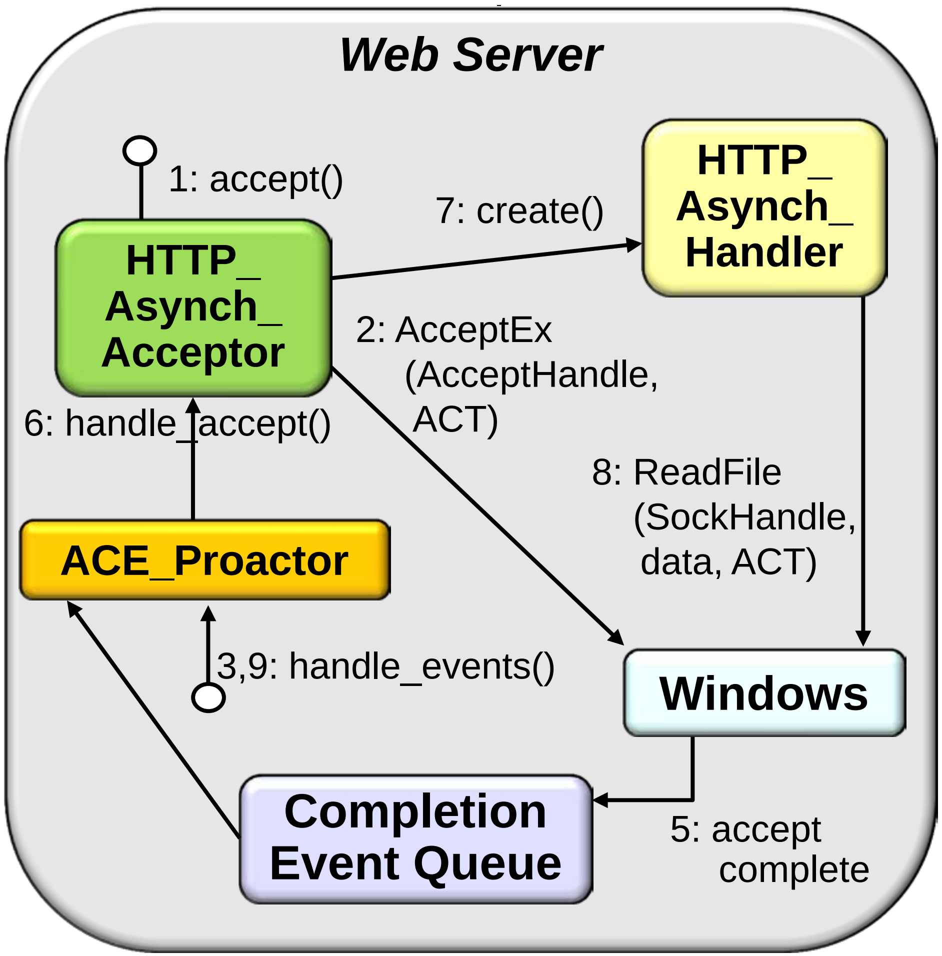
Benefits of Asynchronous Completion Token

Simplified initiator data structures

- Initiators need not maintain complex data structures to associate responses with completion handlers

Efficient state acquisition

- ACTs are time efficient because they need not require complex parsing of data returned with service response



Benefits of Asynchronous Completion Token

Simplified initiator data structures

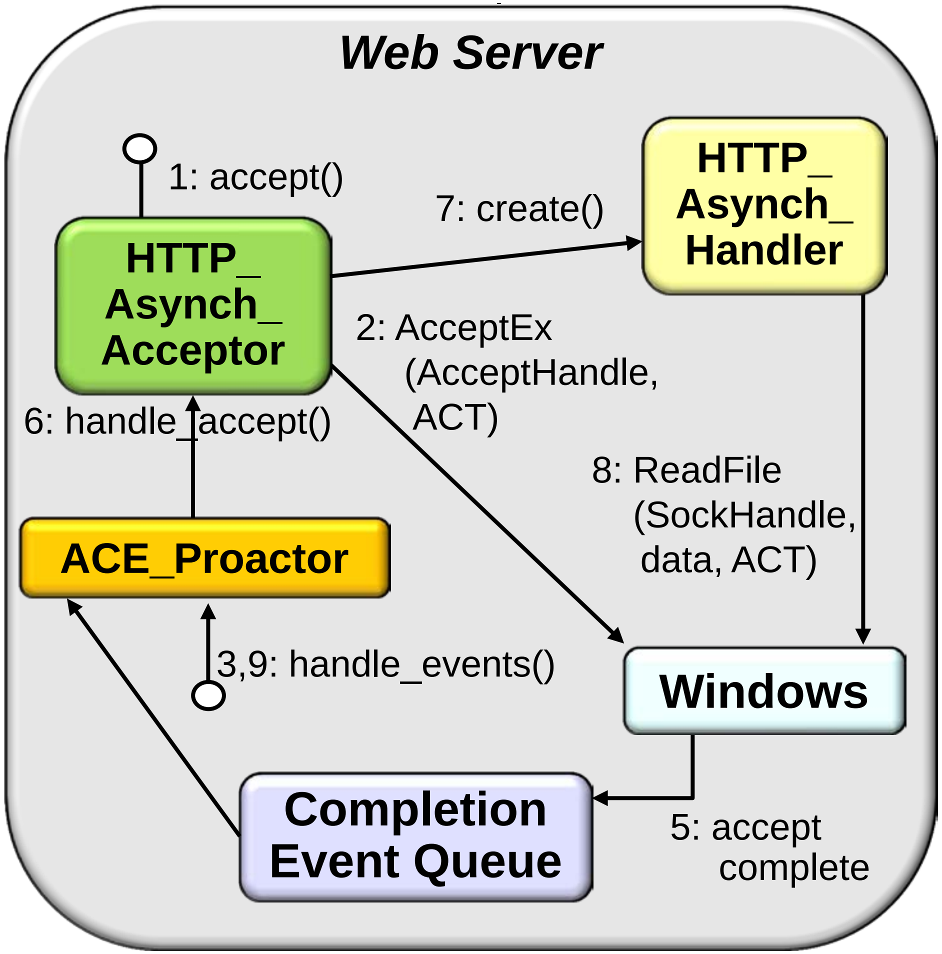
- Initiators need not maintain complex data structures to associate responses with completion handlers

Efficient state acquisition

- ACTs are time efficient because they need not require complex parsing of data returned with service response

Space efficiency

- ACTs can consume minimal space



Benefits of Asynchronous Completion Token

Simplified initiator data structures

- Initiators need not maintain complex data structures to associate responses with completion handlers

Efficient state acquisition

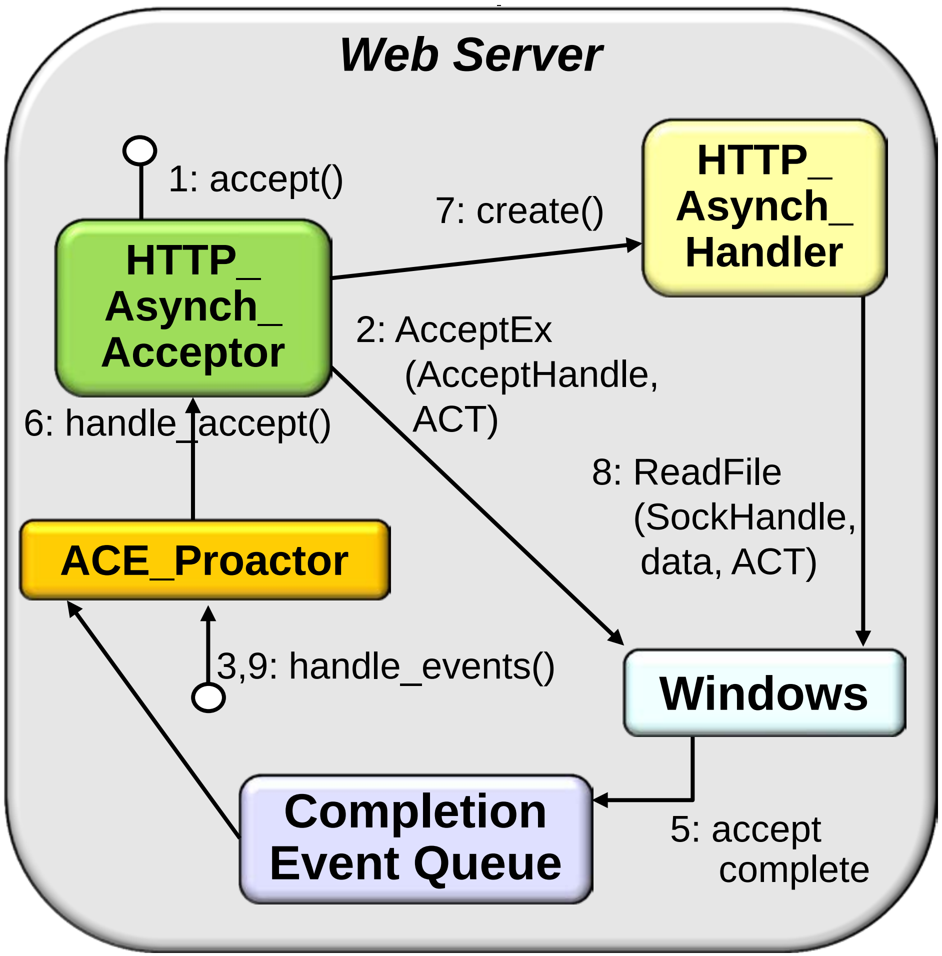
- ACTs are time efficient because they need not require complex parsing of data returned with service response

Space efficiency

- ACTs can consume minimal space

Flexibility

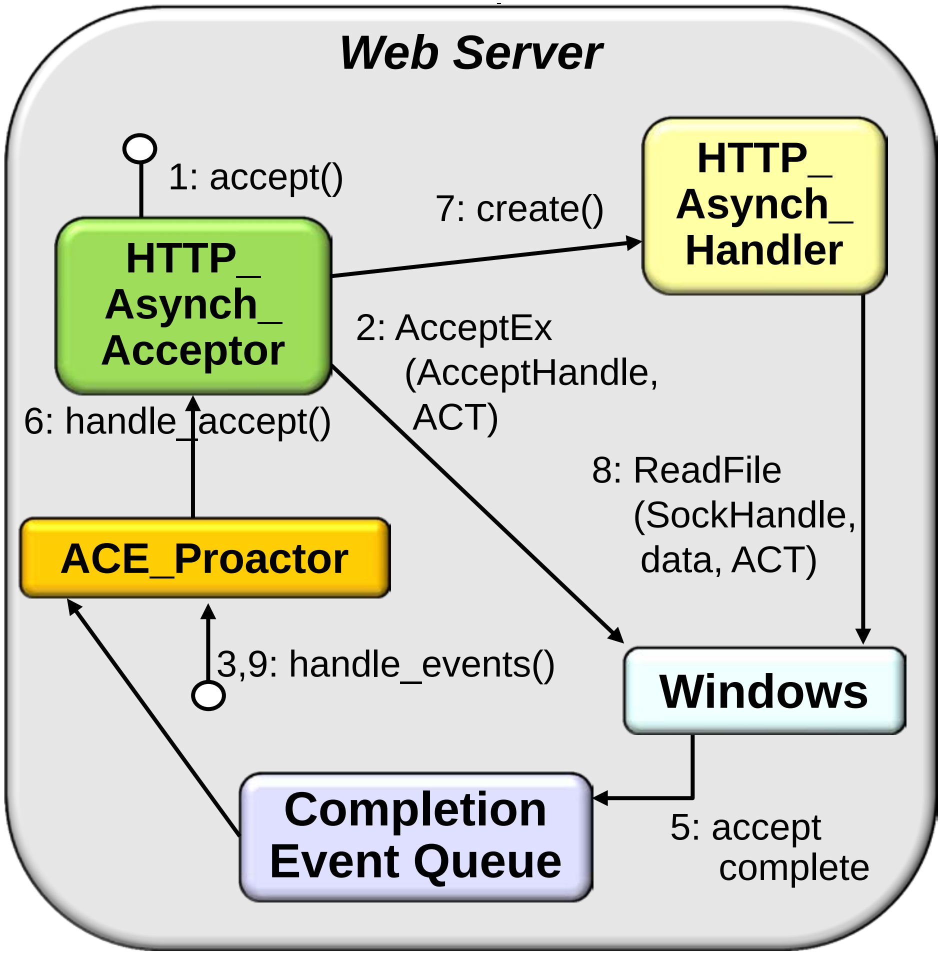
- User-defined ACTs are not forced to inherit from an interface to use the service's ACTs



Limitations of Asynchronous Completion Token

Memory leaks

- Memory leaks can result if initiators use ACTs as pointers to dynamically allocated memory & services fail to return the ACTs



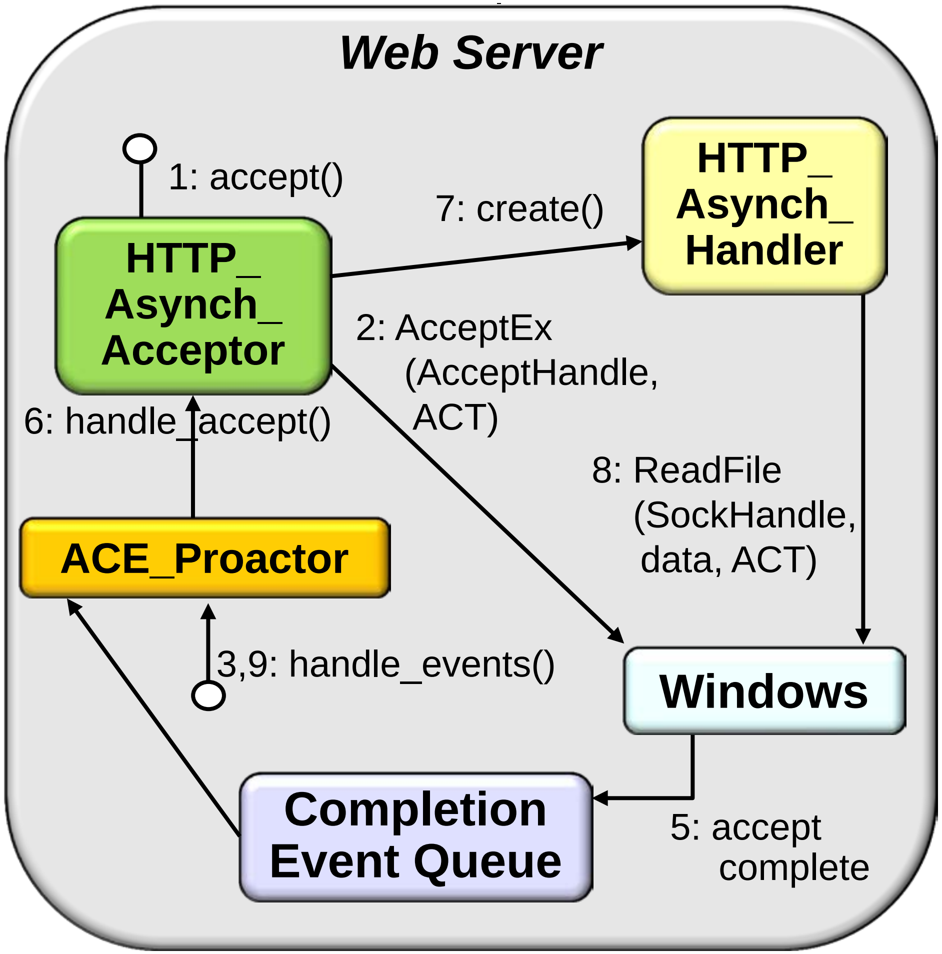
Limitations of Asynchronous Completion Token

Memory leaks

- Memory leaks can result if initiators use ACTs as pointers to dynamically allocated memory & services fail to return the ACTs

Authentication

- When an ACT is returned to an initiator on completion of an asynch event, the initiator may need to authenticate the ACT before using it



Limitations of Asynchronous Completion Token

Memory leaks

- Memory leaks can result if initiators use ACTs as pointers to dynamically allocated memory & services fail to return the ACTs

Authentication

- When an ACT is returned to an initiator on completion of an asynch event, the initiator may need to authenticate the ACT before using it

Application re-mapping

- If ACTs are used as direct pointers to memory, errors can occur if part of the application is re-mapped in virtual memory