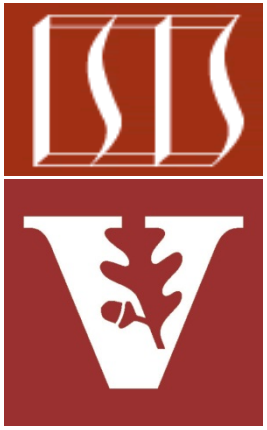
Patterns & Frameworks for Concurrency & Synchronization: Part 1

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



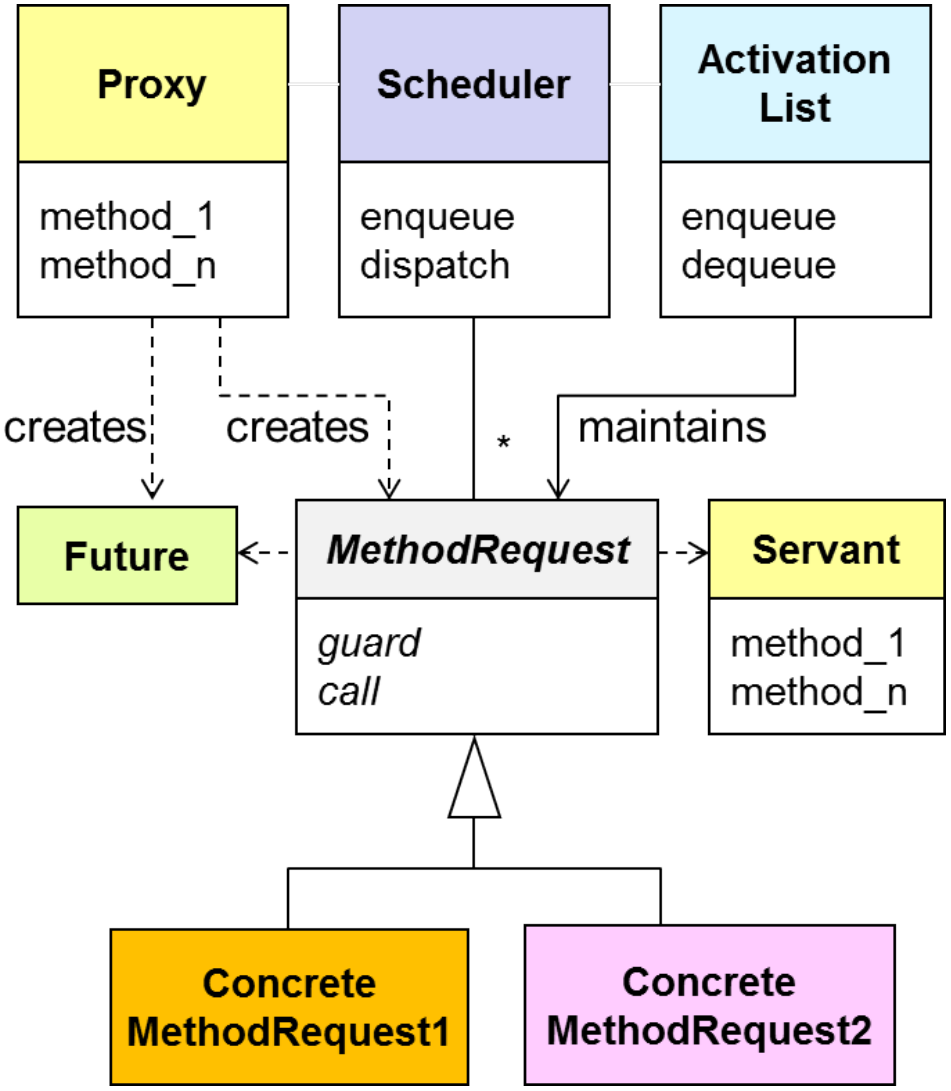
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

- Describe the *Active Object* pattern



Improving Upon Reactive Event Handling

Context	Problem
• High-performance web servers that need to leverage advances in hardware & software	• To improve quality-of-service (QoS) for all connected clients, a web server must not block while waiting for connection flow control to abate & must fully leverage parallelism

HTTP runs over TCP, where flow control ensures that senders don't produce data faster than slow receivers or congested networks can handle

Improving Upon Reactive Event Handling

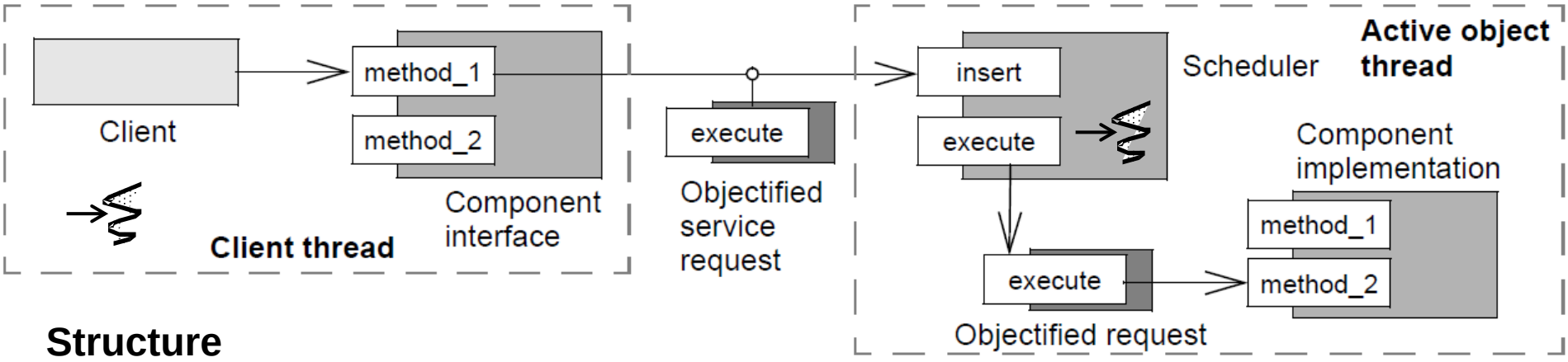
Context	Problem
• High-performance web servers that need to leverage advances in hardware & software	• To improve quality-of-service (QoS) for all connected clients, a web server must not block while waiting for connection flow control to abate & must fully leverage parallelism

A web server must also scale efficiently as # of clients grows

Improving Upon Reactive Event Handling

Context	Problem	Solution
• High-performance web servers that need to leverage advances in hardware & software	• To improve quality-of-service (QoS) for all connected clients, a web server must not block while waiting for connection flow control to abate & must fully leverage parallelism	• Apply the *Active Object* pattern to scale up server performance by processing each HTTP request concurrently in its own thread

Active Object defines units of concurrency as service requests on components & runs service requests on a component in a different thread from the requesting client thread



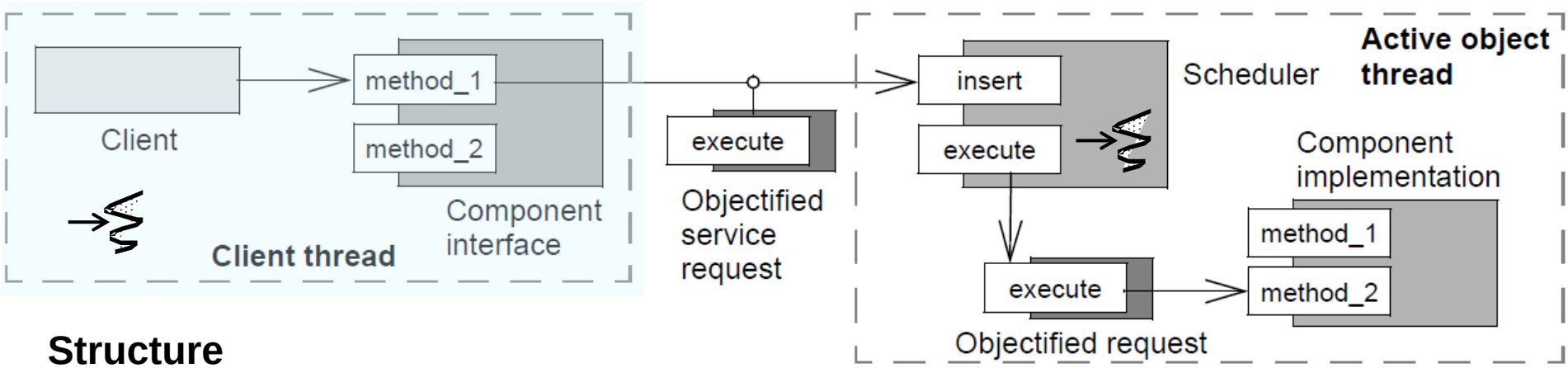
Structure

We'll describe the complete Active Object pattern, but just use a subset of it

Improving Upon Reactive Event Handling

Context	Problem	Solution
• High-performance web servers that need to leverage advances in hardware & software	• To improve quality-of-service (QoS) for all connected clients, a web server must not block while waiting for connection flow control to abate & must fully leverage parallelism	• Apply the *Active Object* pattern to scale up server performance by processing each HTTP request concurrently in its own thread

Active Object defines units of concurrency as service requests on components & runs service requests on a component in a different thread from the requesting client thread

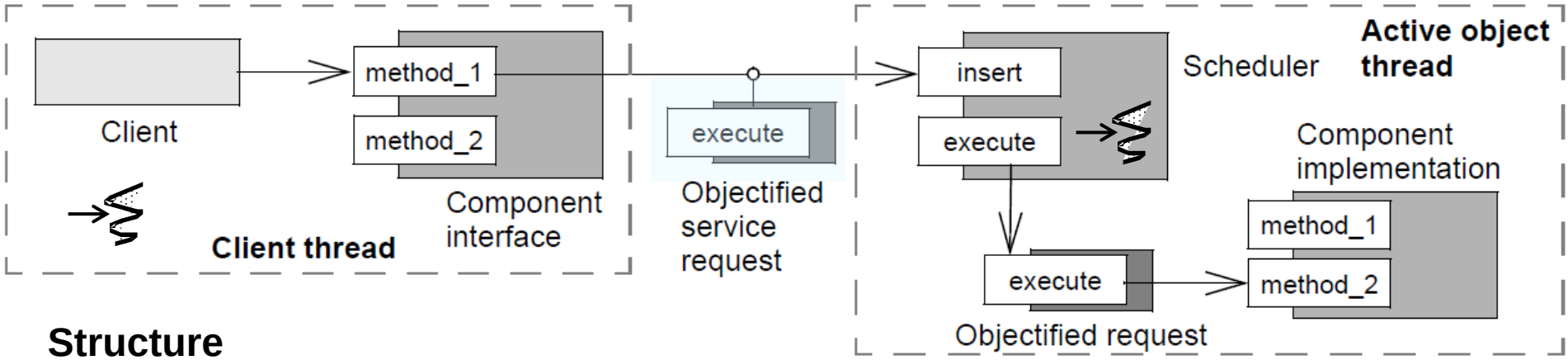


Structure

Improving Upon Reactive Event Handling

Context	Problem	Solution
• High-performance web servers that need to leverage advances in hardware & software	• To improve quality-of-service (QoS) for all connected clients, a web server must not block while waiting for connection flow control to abate & must fully leverage parallelism	• Apply the *Active Object* pattern to scale up server performance by processing each HTTP request concurrently in its own thread

Active Object defines units of concurrency as service requests on components & runs service requests on a component in a different thread from the requesting client thread

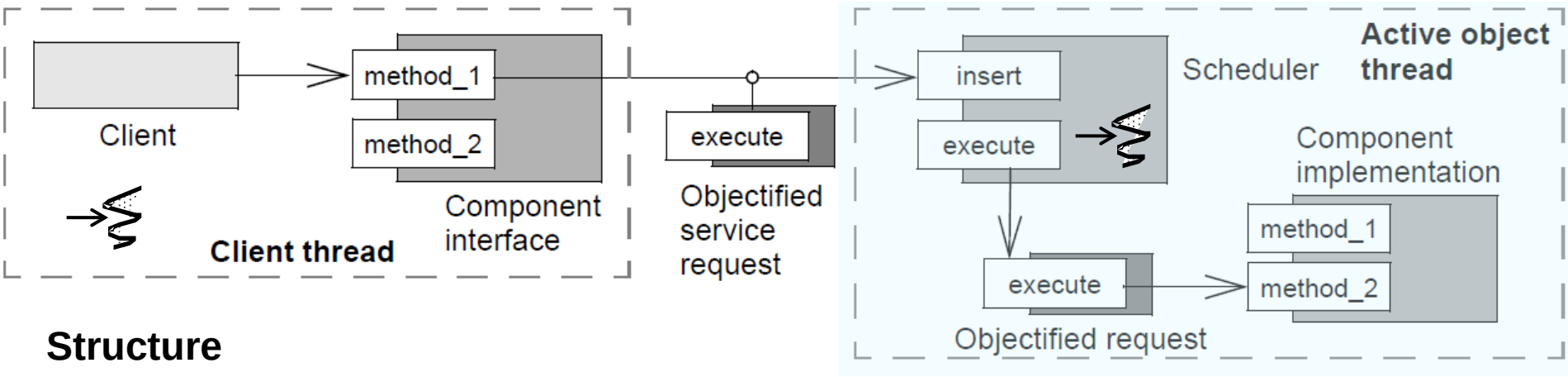


Structure

Improving Upon Reactive Event Handling

Context	Problem	Solution
• High-performance web servers that need to leverage advances in hardware & software	• To improve quality-of-service (QoS) for all connected clients, a web server must not block while waiting for connection flow control to abate & must fully leverage parallelism	• Apply the *Active Object* pattern to scale up server performance by processing each HTTP request concurrently in its own thread

Active Object defines units of concurrency as service requests on components & runs service requests on a component in a different thread from the requesting client thread



Structure

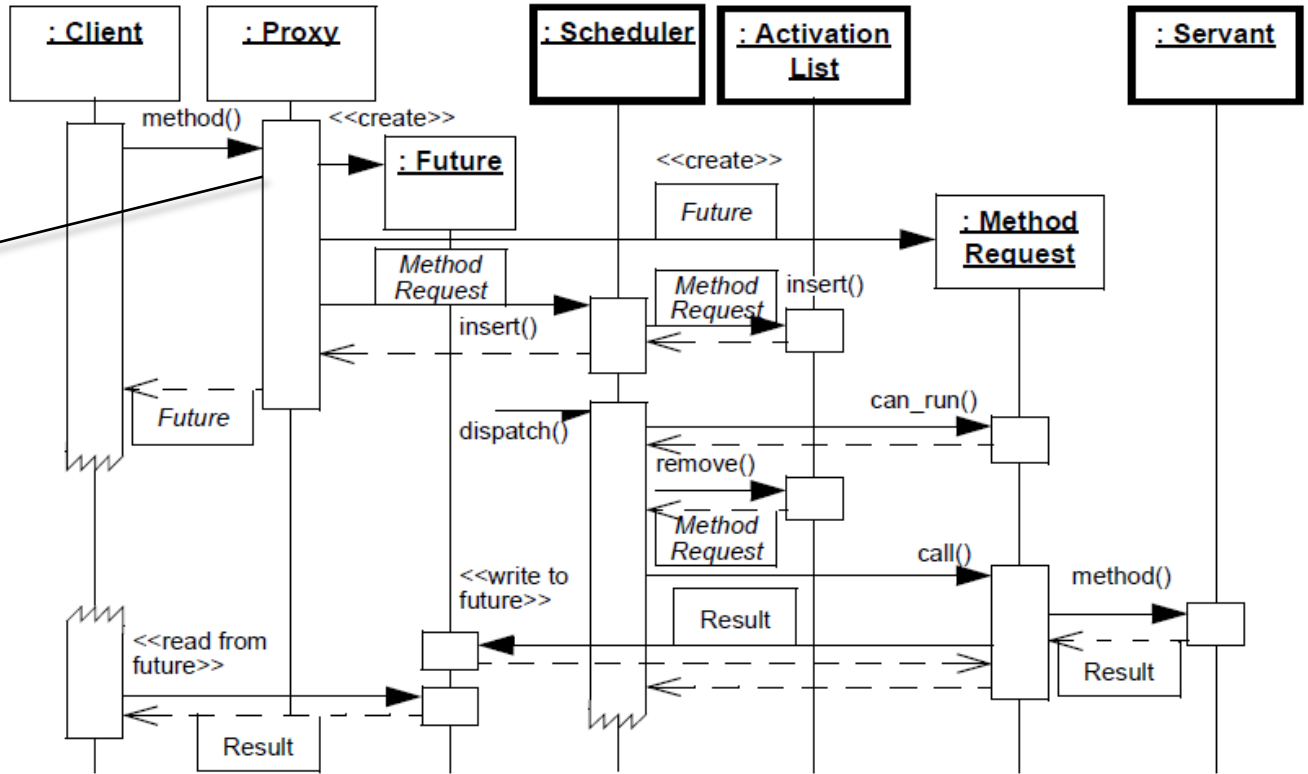
See www.dre.vanderbilt.edu/~schmidt/PDF/Active-Objects.pdf for more info

Improving Upon Reactive Event Handling

Context	Problem	Solution
• High-performance web servers that need to leverage advances in hardware & software	• To improve quality-of-service (QoS) for all connected clients, a web server must not block while waiting for connection flow control to abate & must fully leverage parallelism	• Apply the *Active Object* pattern to scale up server performance by processing each HTTP request concurrently in its own thread

Dynamics

Client invokes a method call on a proxy

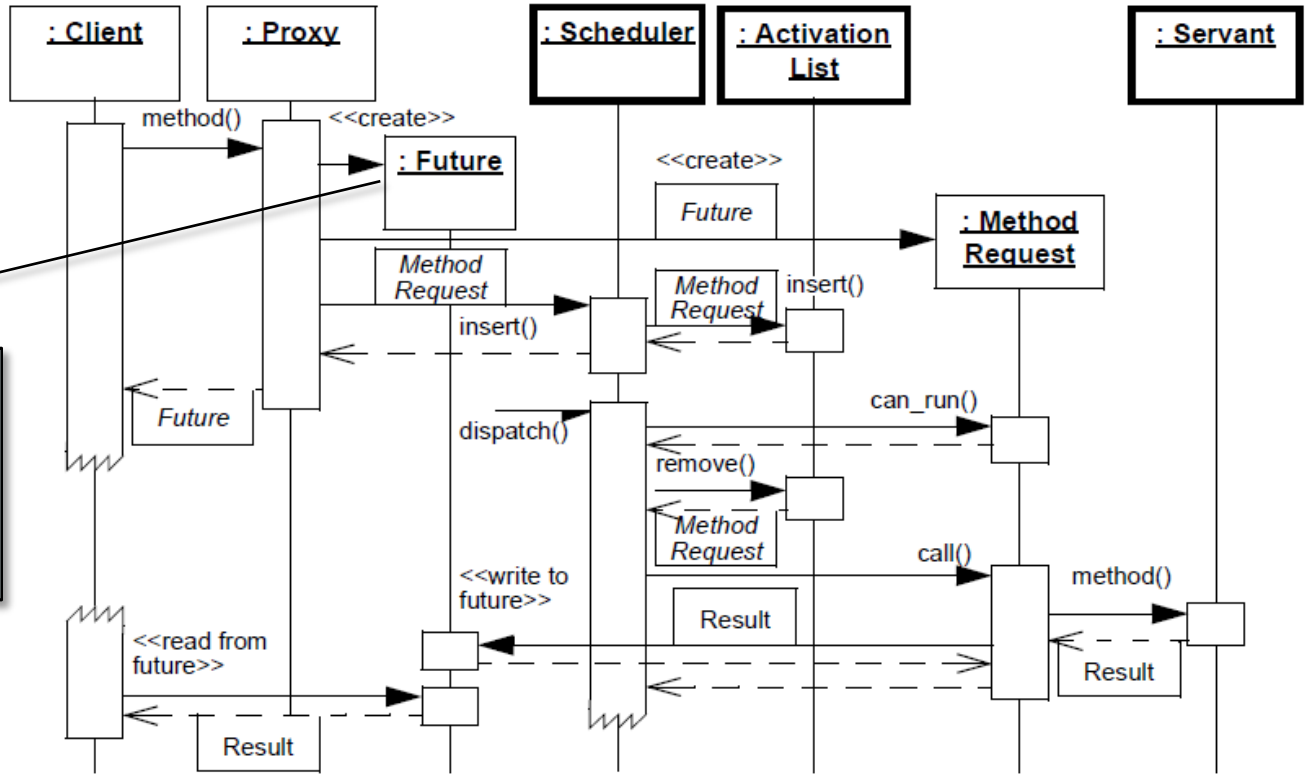


Improving Upon Reactive Event Handling

Context	Problem	Solution
• High-performance web servers that need to leverage advances in hardware & software	• To improve quality-of-service (QoS) for all connected clients, a web server must not block while waiting for connection flow control to abate & must fully leverage parallelism	• Apply the *Active Object* pattern to scale up server performance by processing each HTTP request concurrently in its own thread

Dynamics

Proxy converts method call into method request, passes to scheduler, & returns future to client

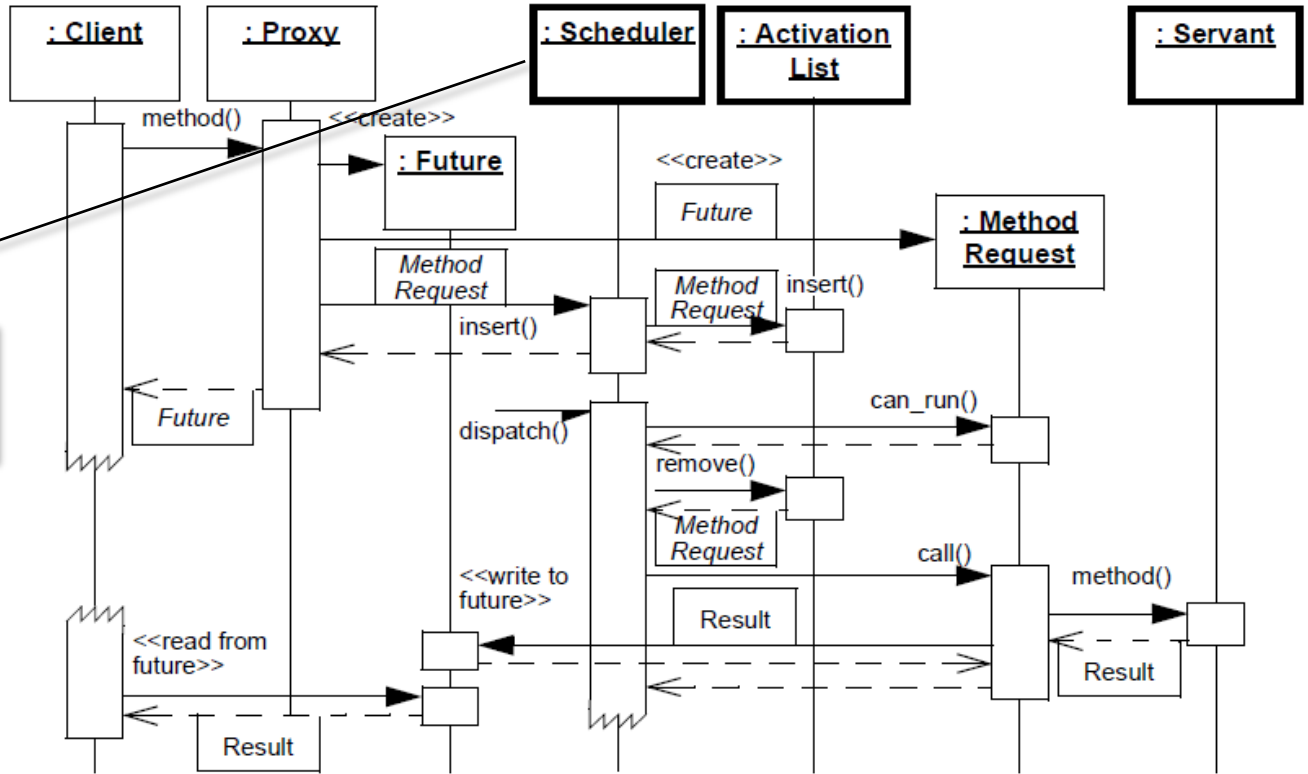


Improving Upon Reactive Event Handling

Context	Problem	Solution
• High-performance web servers that need to leverage advances in hardware & software	• To improve quality-of-service (QoS) for all connected clients, a web server must not block while waiting for connection flow control to abate & must fully leverage parallelism	• Apply the *Active Object* pattern to scale up server performance by processing each HTTP request concurrently in its own thread

Dynamics

Scheduler enqueues method request

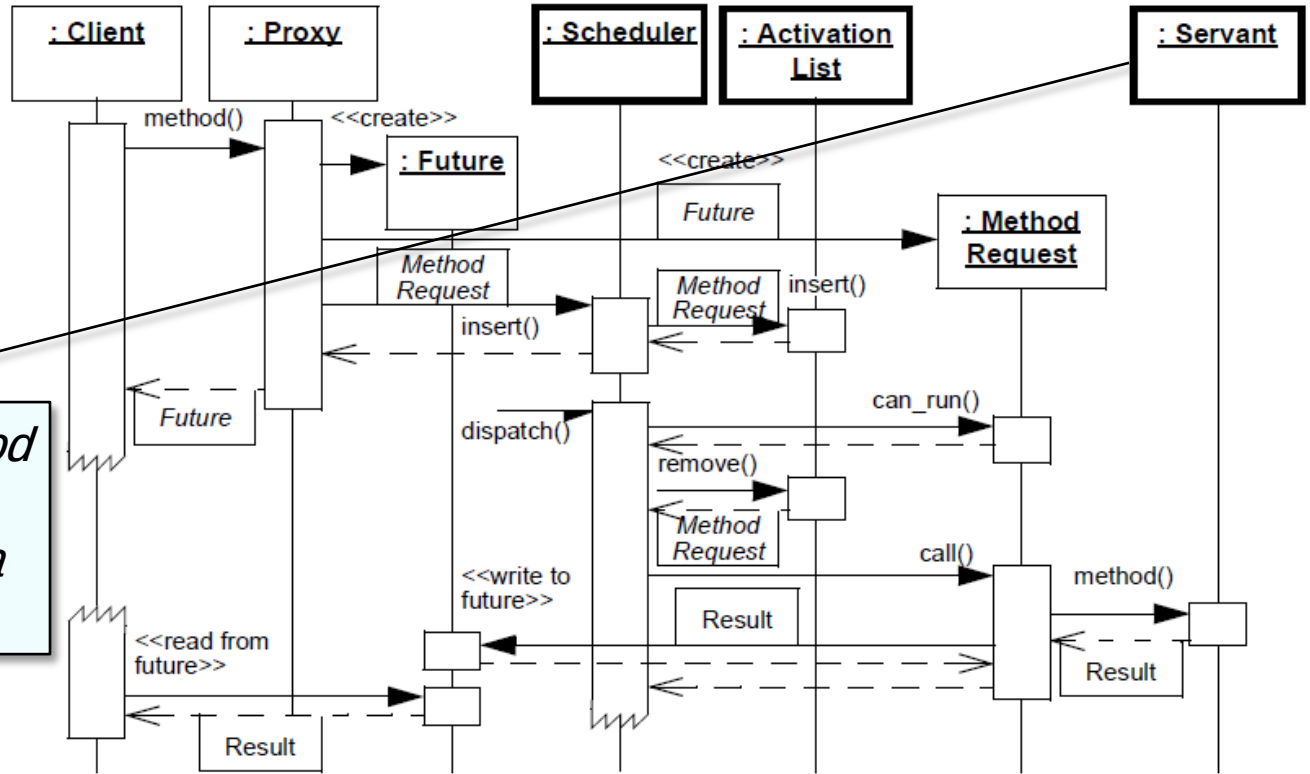


Improving Upon Reactive Event Handling

Context	Problem	Solution
• High-performance web servers that need to leverage advances in hardware & software	• To improve quality-of-service (QoS) for all connected clients, a web server must not block while waiting for connection flow control to abate & must fully leverage parallelism	• Apply the *Active Object* pattern to scale up server performance by processing each HTTP request concurrently in its own thread

Dynamics

Scheduler dequeues method request at some point & runs it on the servant in a separate thread

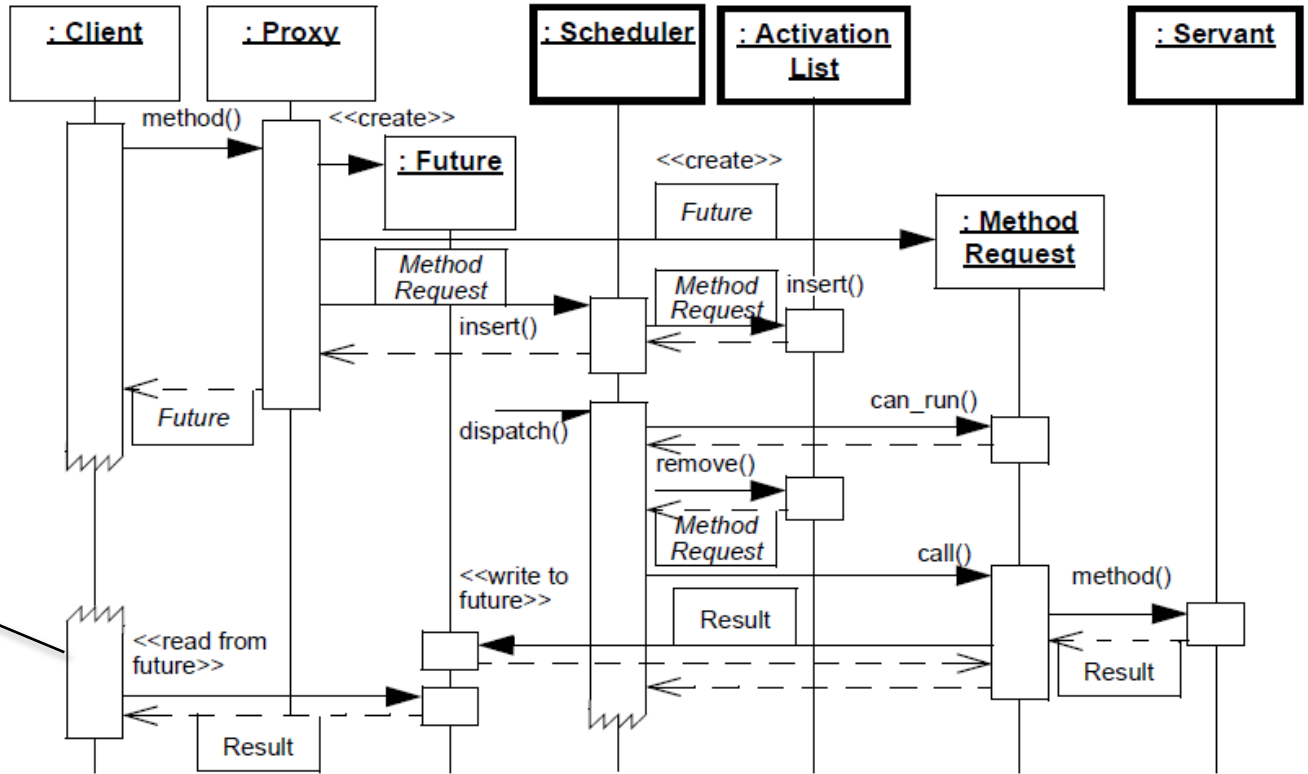


Improving Upon Reactive Event Handling

Context	Problem	Solution
• High-performance web servers that need to leverage advances in hardware & software	• To improve quality-of-service (QoS) for all connected clients, a web server must not block while waiting for connection flow control to abate & must fully leverage parallelism	• Apply the *Active Object* pattern to scale up server performance by processing each HTTP request concurrently in its own thread

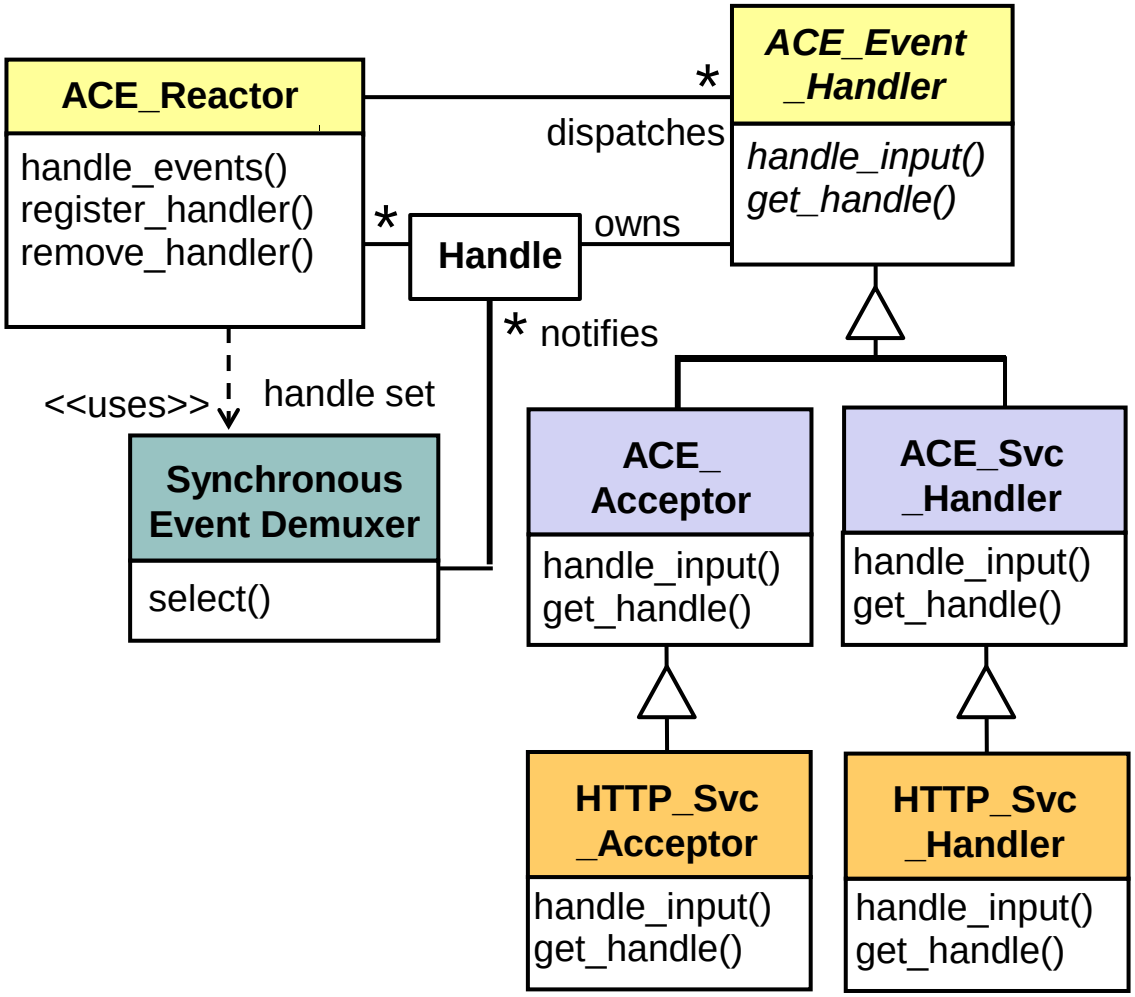
Dynamics

Clients can obtain result from futures via polling, or callbacks



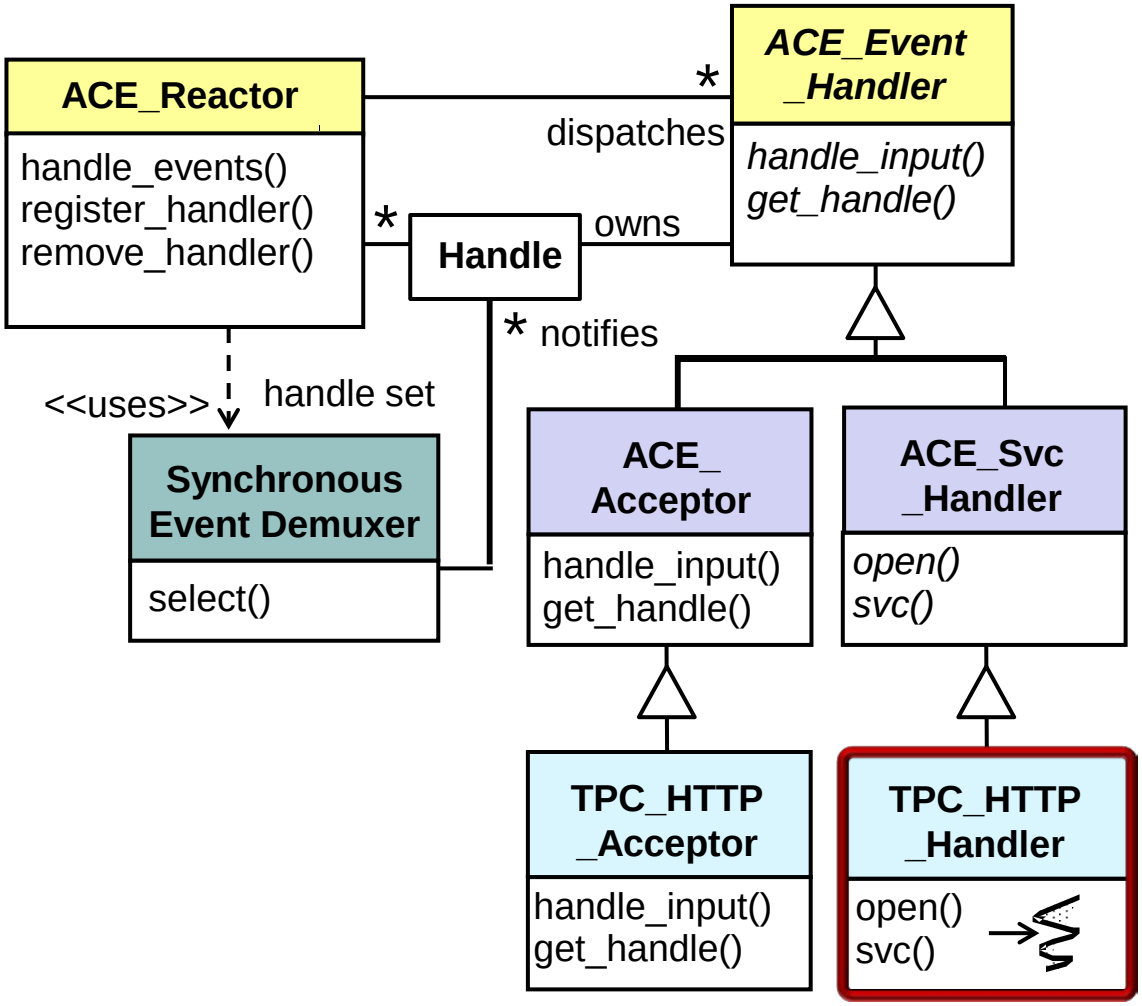
Applying the Active Object Pattern in JAWS

- One way to apply the *Active Object* pattern to JAWS requires a minor extension to our reactive *Acceptor-Connector*-based version



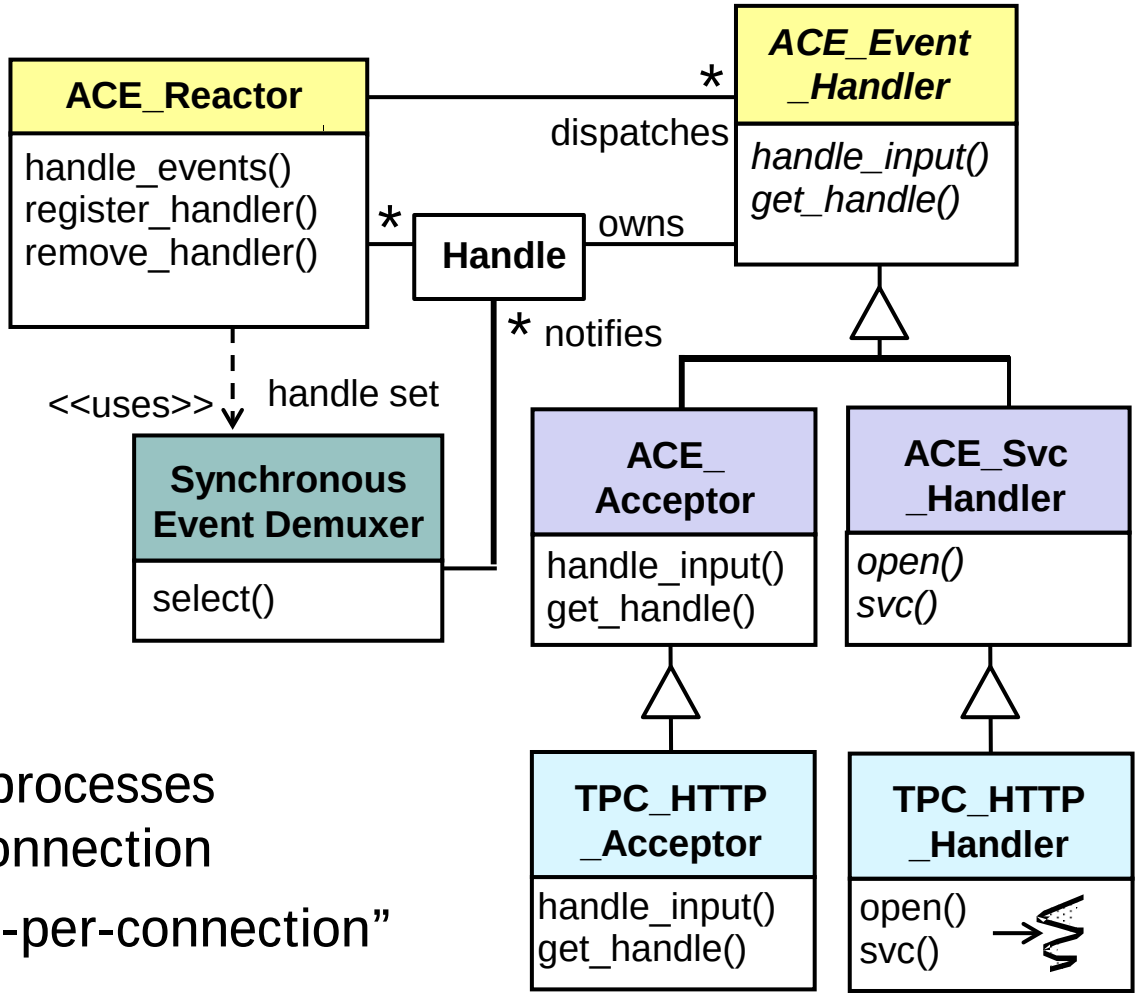
Applying the Active Object Pattern in JAWS

- One way to apply the *Active Object* pattern to JAWS requires a minor extension to our reactive *Acceptor-Connector*-based version
- We simply inherit from **`ACE_Acceptor`** & **`ACE_Svc_Handler`** to create a new subclass **`TPC_HTTP_Handler`** that runs as an active object



Applying the Active Object Pattern in JAWS

- One way to apply the *Active Object* pattern to JAWS requires a minor extension to our reactive *Acceptor-Connector*-based version
- We simply inherit from **`ACE_Acceptor`** & **`ACE_Svc_Handler`** to create a new subclass **`TPC_HTTP_Handler`** that runs as an active object



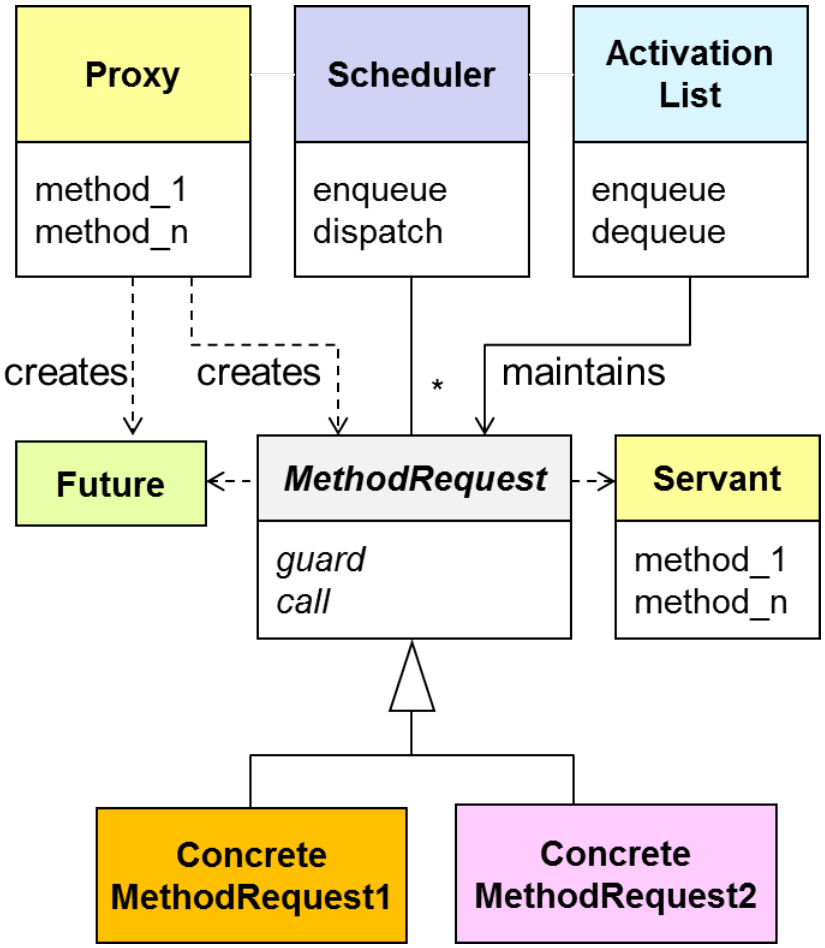
- This active object receives & processes requests from a designated connection
  - i.e., it implements a "thread-per-connection" concurrency model

This *Active Object* variant spans user/kernel boundary & omits futures/servants

Benefits of the Active Object Pattern

Enhances concurrency & simplifies synchronized complexity

- Client threads & asynchronous method executions can run concurrently
- A scheduler can evaluate synchronization constraints to serialize access to servants



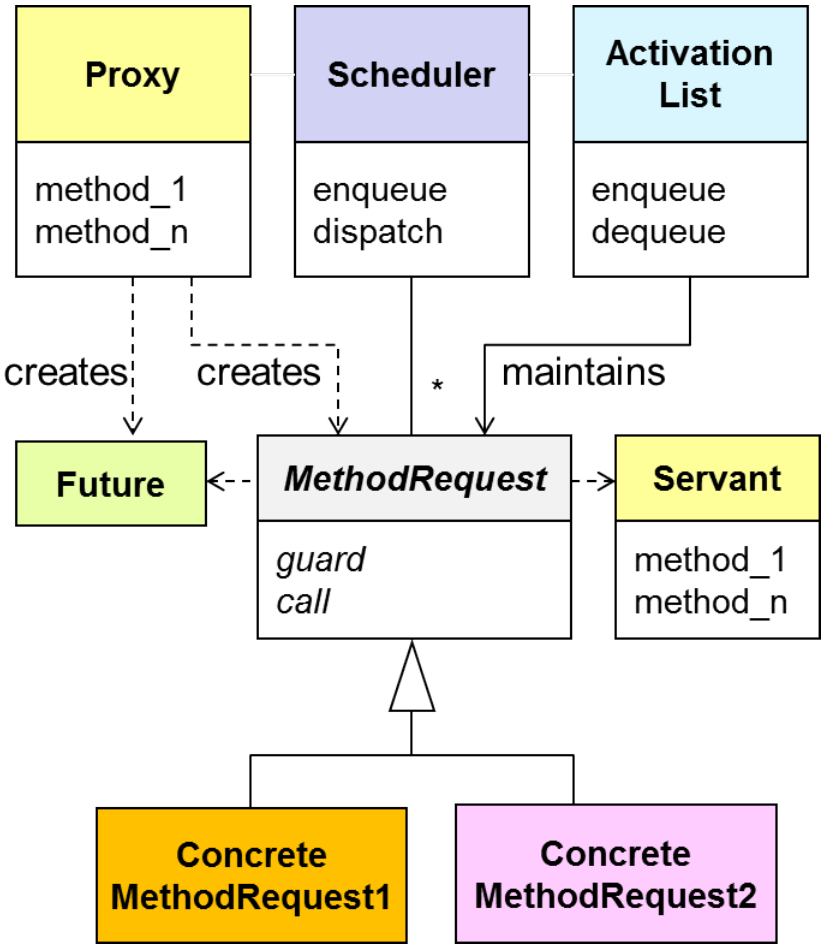
Benefits of the Active Object Pattern

Enhances concurrency & simplifies synchronized complexity

- Client threads & asynchronous method executions can run concurrently
- A scheduler can evaluate synchronization constraints to serialize access to servants

Transparent leveraging of available parallelism

- Multiple active object methods can execute in parallel if supported by the OS/hardware



Benefits of the Active Object Pattern

Enhances concurrency & simplifies synchronized complexity

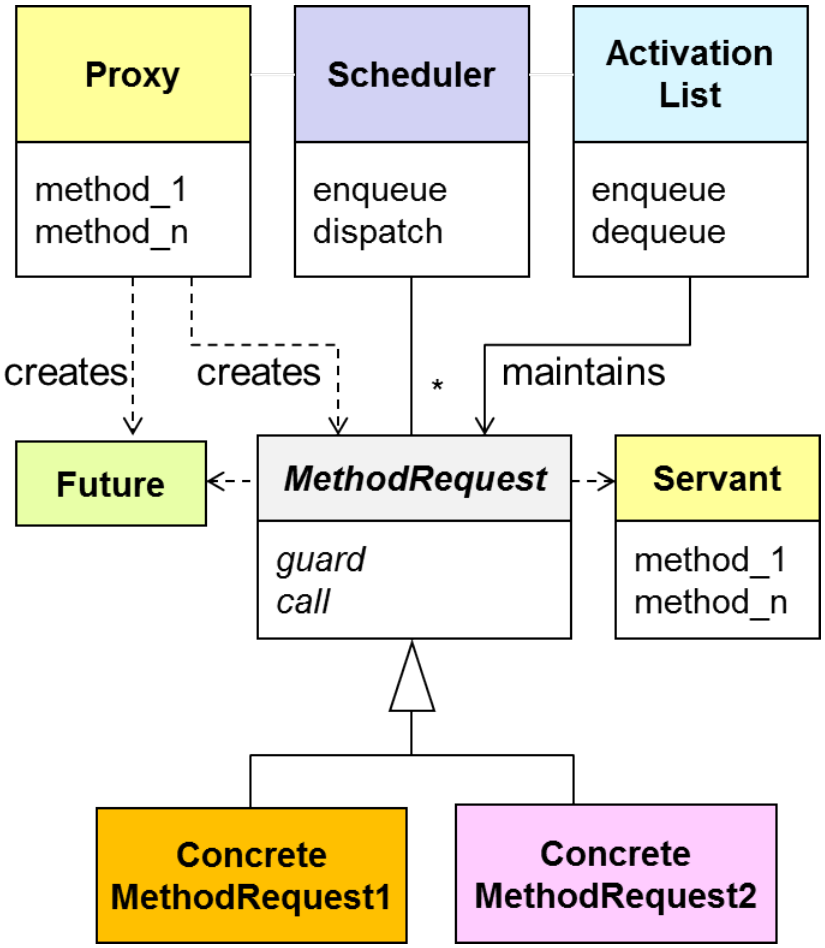
- Client threads & asynchronous method executions can run concurrently
- A scheduler can evaluate synchronization constraints to serialize access to servants

Transparent leveraging of available parallelism

- Multiple active object methods can execute in parallel if supported by the OS/hardware

Method execution order can differ from method invocation order

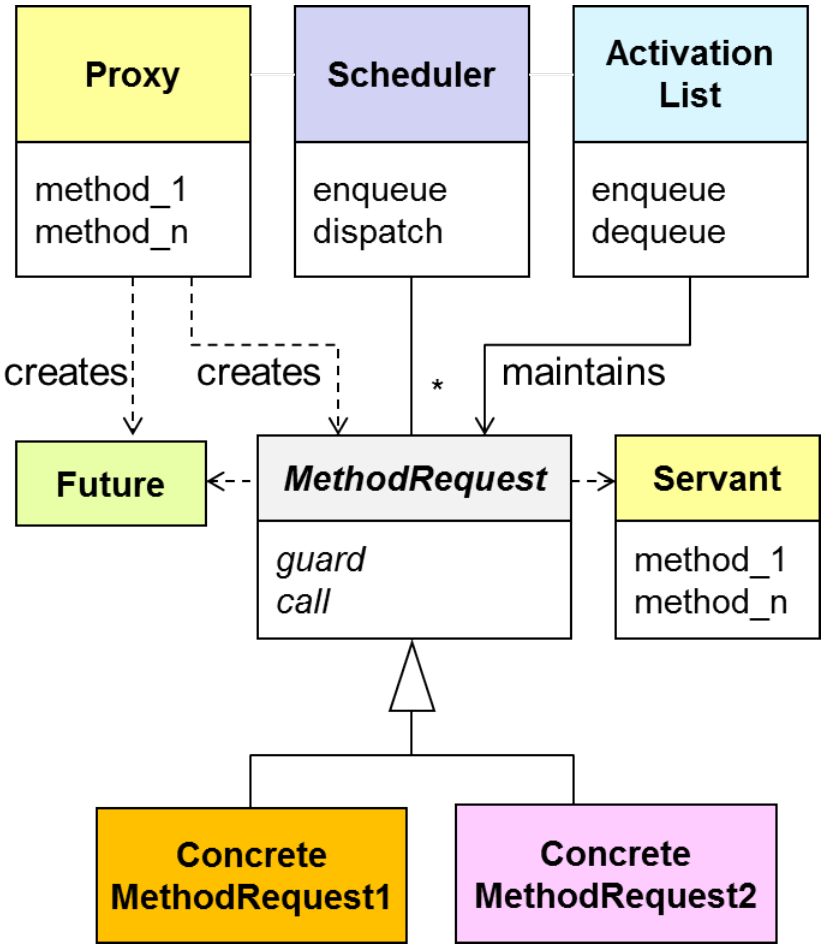
- Methods invoked asynchronous can be executed according to synchronization constraints defined by guards & scheduling policies



Limitations of the Active Object Pattern

Higher overhead

- Depending on how an active object is implemented, context switching, synchronization, & data movement/ copying overhead may occur when invoking, scheduling, & executing active object method calls



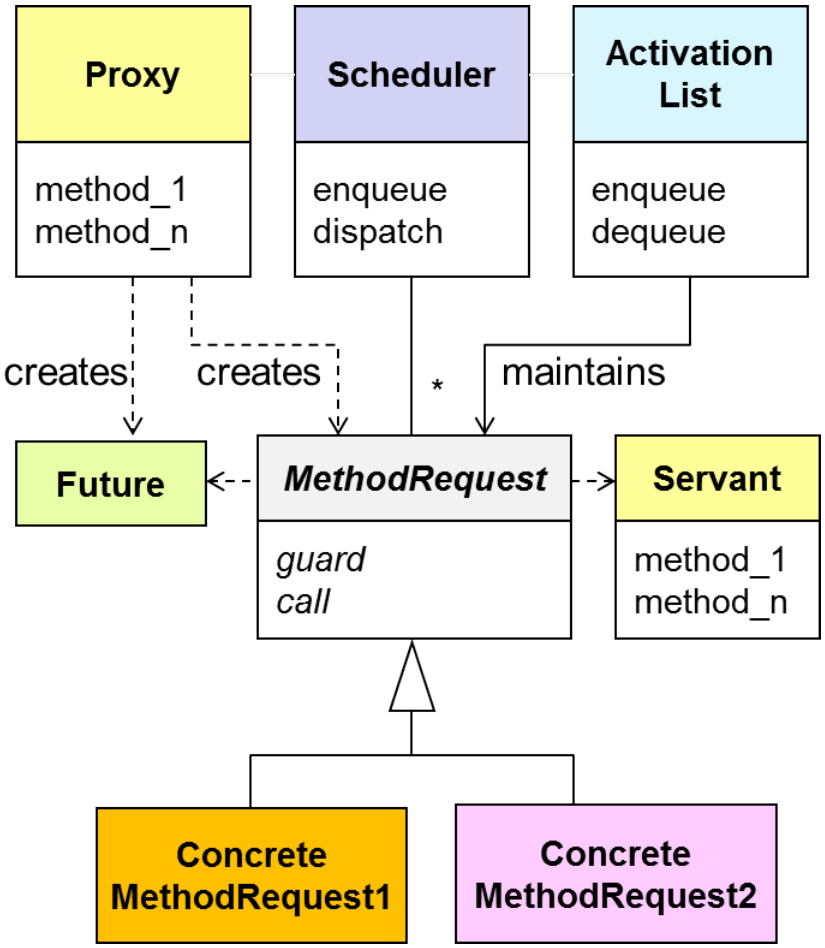
Limitations of the Active Object Pattern

Higher overhead

- Depending on how an active object is implemented, context switching, synchronization, & data movement/ copying overhead may occur when invoking, scheduling, & executing active object method calls

Complicated debugging

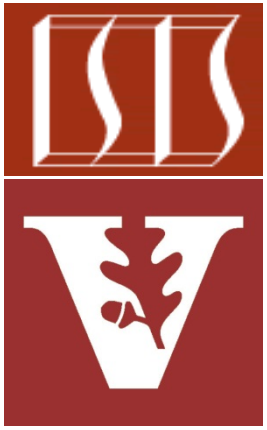
- It is harder to debug programs that use concurrency due to non-determinism of the various schedulers



Subsets of *Active Object* are often used to workaround these limitations

Patterns & Frameworks for Concurrency & Synchronization: Part 2

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



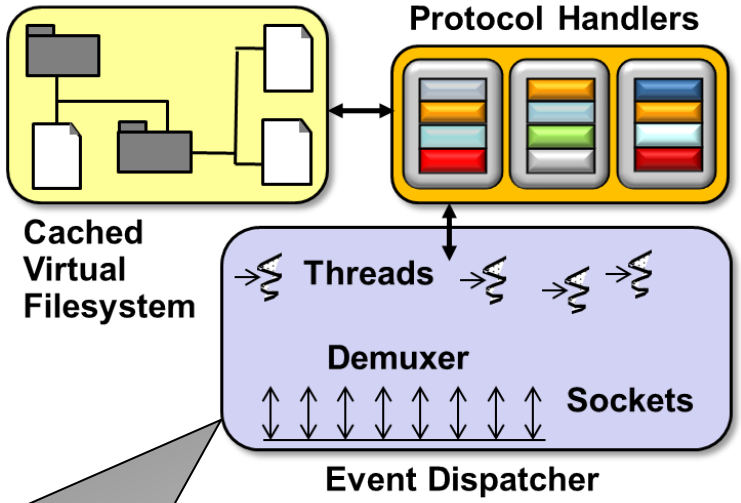
Professor of Computer Science

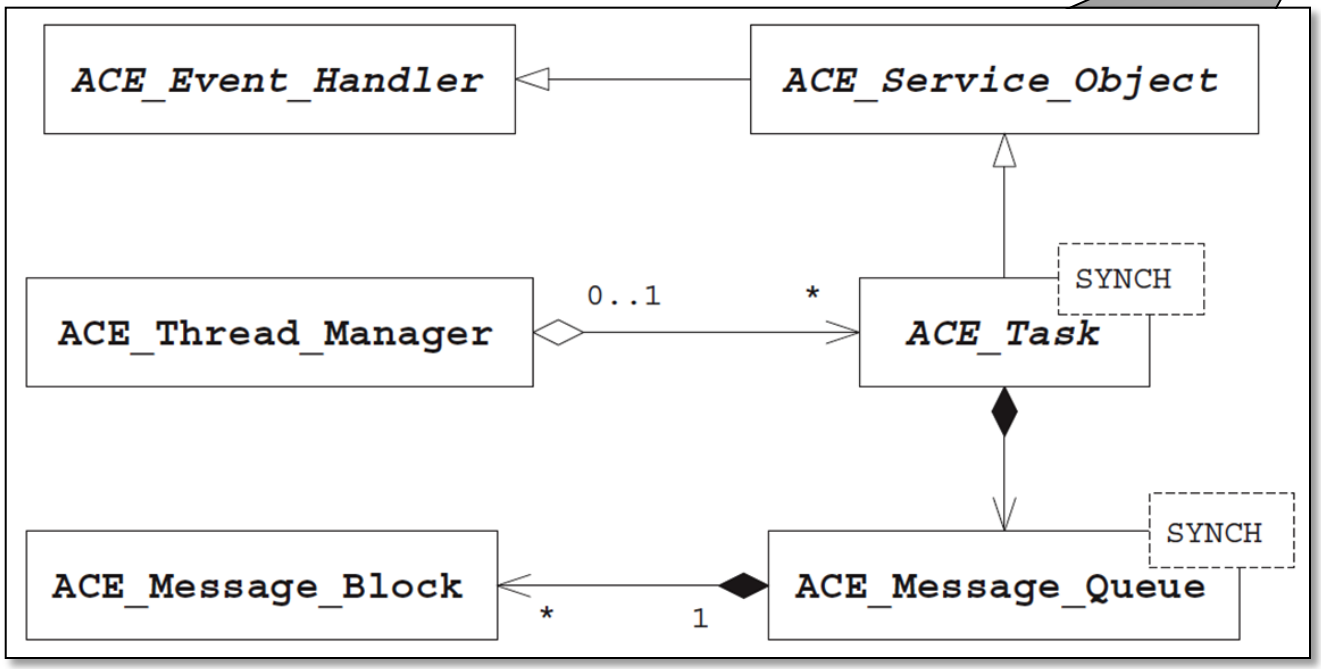
Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

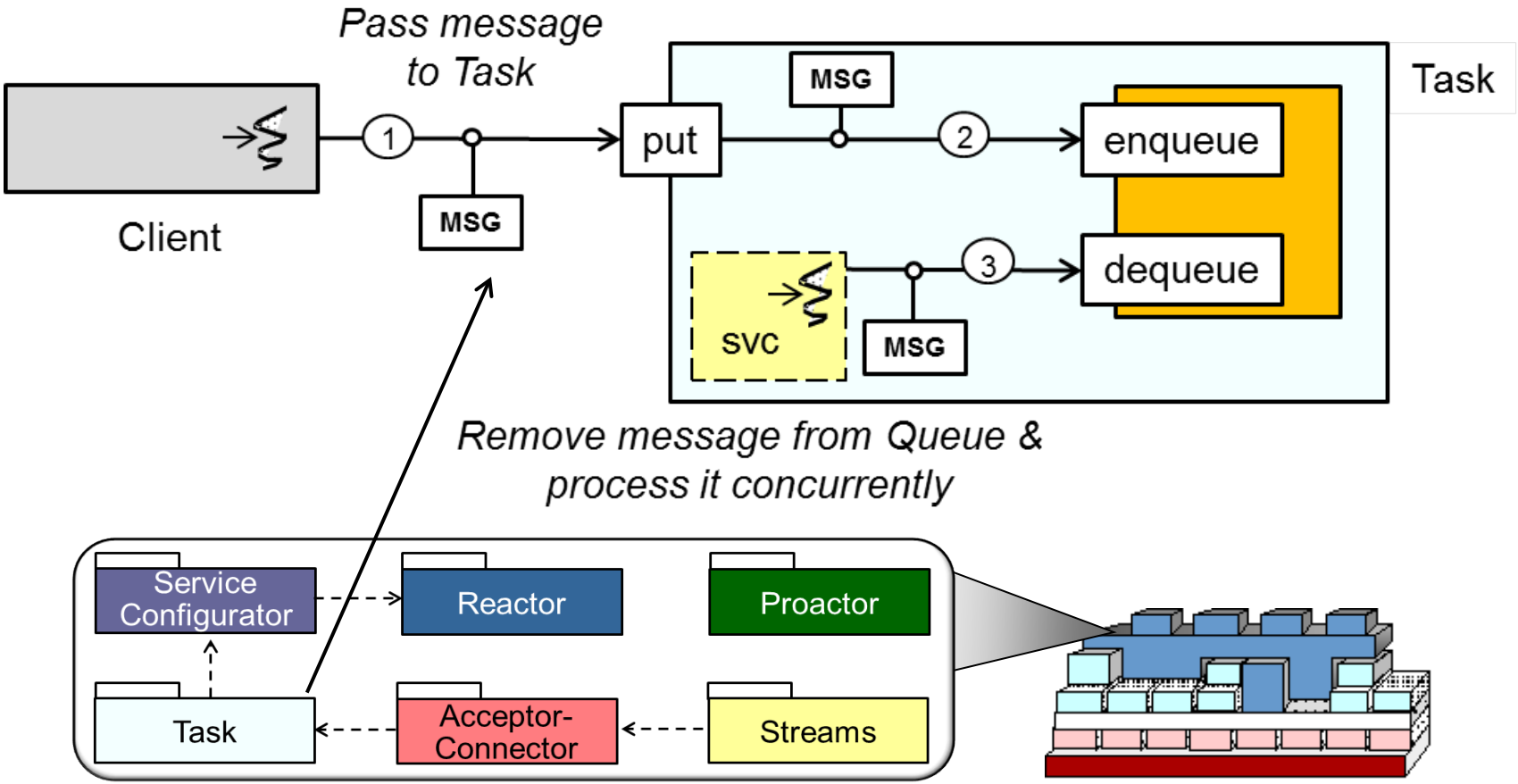
- Describe the *Active Object* pattern
- Describe the ACE *Task* framework





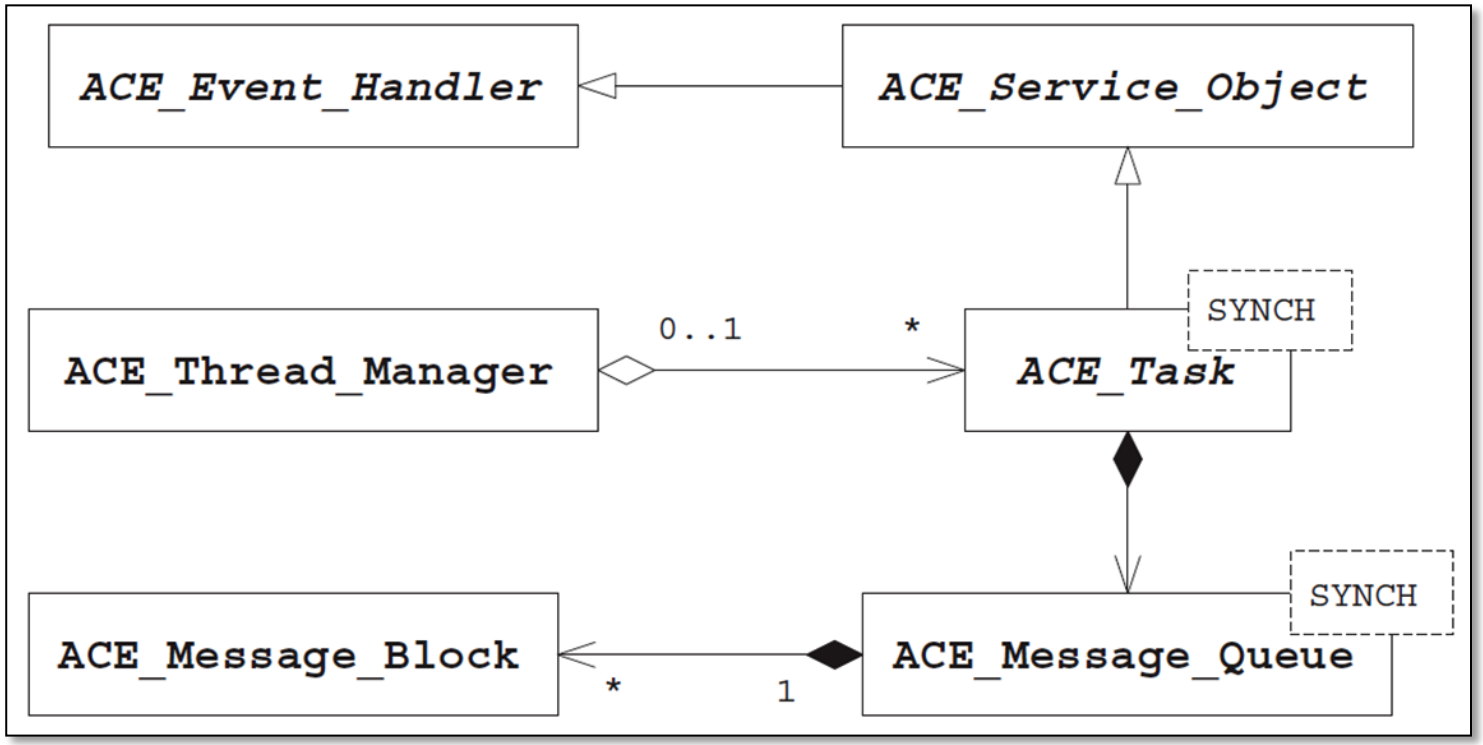
Overview of the ACE Task Framework

- Classes in this framework provide a producer/consumer concurrency model that can associate one or more threads with a message queue & other service-related information, such as methods & state



Overview of the ACE Task Framework

- Classes in this framework provide a producer/consumer concurrency model that can associate one or more threads with a message queue & other service-related information, such as methods & state
- Apps inherit from **`ACE_Task`** & override its hook methods, which the ACE *Task* framework dispatch to decouple invocations from executions



These classes are designed in accordance with the *Active Object* pattern

Overview of the ACE Task Framework

- Classes in this framework provide a producer/consumer concurrency model that can associate one or more threads with a message queue & other service-related information, such as methods & state
- Apps inherit from **`ACE_Task`** & override its hook methods, which the ACE *Task* framework dispatch to decouple invocations from executions
- Key classes in the ACE Task framework include

ACE Class	Description
`ACE_Message_Block`	Implements the *Composite* pattern to enable efficient manipulation of fixed- & variable-sized messages
`ACE_Message_Queue`	Provides an intra-process message queue that enables applications to pass & buffer messages between threads in a process
`ACE_Thread_Manager`	Allows applications to portably create & manage the lifetime, synchronization, & properties of one or more threads
`ACE_Task`	Allows applications to create passive or active objects that decouple different units of processing; use messages to communicate requests, responses, data, & control information; & can queue & process messages sequentially or concurrently

Overview of the ACE Task Framework

- Classes in this framework provide a producer/consumer concurrency model that can associate one or more threads with a message queue & other service-related information, such as methods & state
- Apps inherit from **`ACE_Task`** & override its hook methods, which the ACE *Task* framework dispatch to decouple invocations from executions
- Key classes in the ACE Task framework include

ACE Class	Description
`ACE_Message_Block`	Implements the *Composite* pattern to enable efficient manipulation of fixed- & variable-sized messages
`ACE_Message_Queue`	Provides an intra-process message queue that enables applications to pass & buffer messages between threads in a process
`ACE_Thread_Manager`	Allows applications to portably create & manage the lifetime, synchronization, & properties of one or more threads
`ACE_Task`	Allows applications to create passive or active objects that decouple different units of processing; use messages to communicate requests, responses, data, & control information; & can queue & process messages sequentially or concurrently

Overview of the ACE Task Framework

- Classes in this framework provide a producer/consumer concurrency model that can associate one or more threads with a message queue & other service-related information, such as methods & state
- Apps inherit from **`ACE_Task`** & override its hook methods, which the ACE *Task* framework dispatch to decouple invocations from executions
- Key classes in the ACE Task framework include

ACE Class	Description
`ACE_Message_Block`	Implements the *Composite* pattern to enable efficient manipulation of fixed- & variable-sized messages
`ACE_Message_Queue`	Provides an intra-process message queue that enables applications to pass & buffer messages between threads in a process
`ACE_Thread_Manager`	Allows applications to portably create & manage the lifetime, synchronization, & properties of one or more threads
`ACE_Task`	Allows applications to create passive or active objects that decouple different units of processing; use messages to communicate requests, responses, data, & control information; & can queue & process messages sequentially or concurrently

Overview of the ACE Task Framework

- Classes in this framework provide a producer/consumer concurrency model that can associate one or more threads with a message queue & other service-related information, such as methods & state
- Apps inherit from **`ACE_Task`** & override its hook methods, which the ACE *Task* framework dispatch to decouple invocations from executions
- Key classes in the ACE Task framework include

ACE Class	Description
`ACE_Message_Block`	Implements the *Composite* pattern to enable efficient manipulation of fixed- & variable-sized messages
`ACE_Message_Queue`	Provides an intra-process message queue that enables applications to pass & buffer messages between threads in a process
`ACE_Thread_Manager`	Allows applications to portably create & manage the lifetime, synchronization, & properties of one or more threads
`ACE_Task`	Allows applications to create passive or active objects that decouple different units of processing; use messages to communicate requests, responses, data, & control information; & can queue & process messages sequentially or concurrently

Overview of the ACE Task Framework

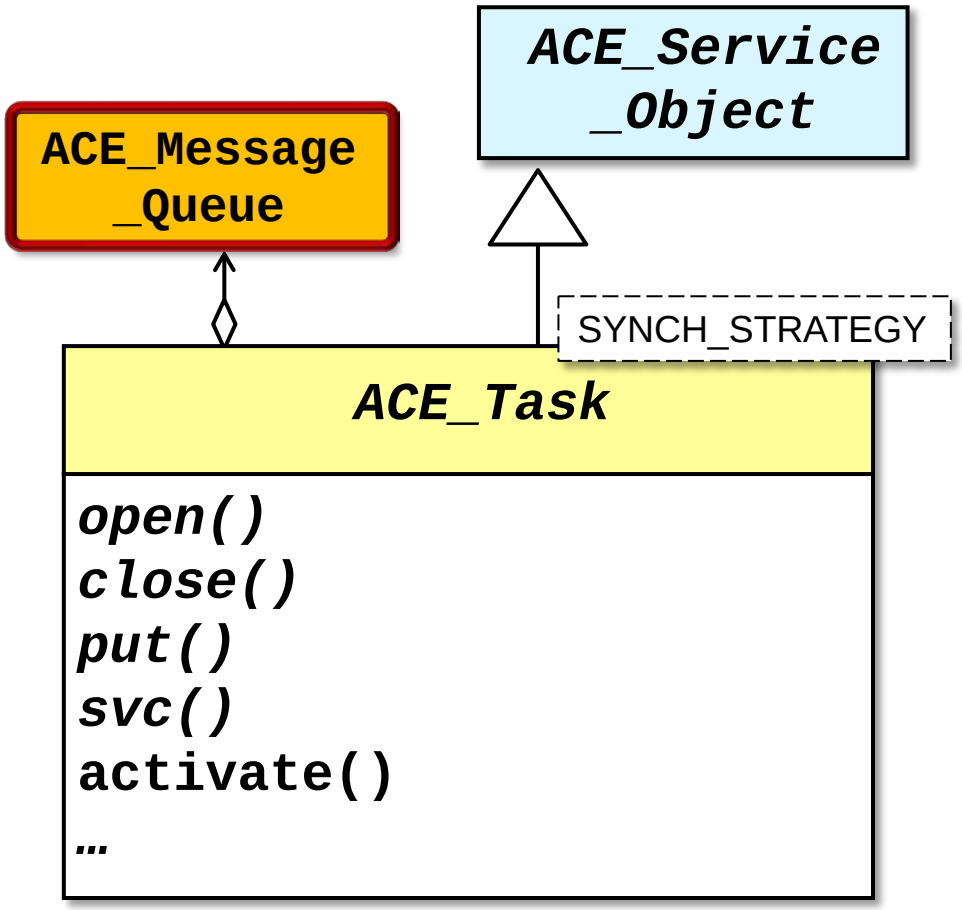
- Classes in this framework provide a producer/consumer concurrency model that can associate one or more threads with a message queue & other service-related information, such as methods & state
- Apps inherit from **`ACE_Task`** & override its hook methods, which the ACE *Task* framework dispatch to decouple invocations from executions
- Key classes in the ACE Task framework include

ACE Class	Description
`ACE_Message_Block`	Implements the *Composite* pattern to enable efficient manipulation of fixed- & variable-sized messages
`ACE_Message_Queue`	Provides an intra-process message queue that enables applications to pass & buffer messages between threads in a process
`ACE_Thread_Manager`	Allows applications to portably create & manage the lifetime, synchronization, & properties of one or more threads
`ACE_Task`	Allows applications to create passive or active objects that decouple different units of processing; use messages to communicate requests, responses, data, & control information; & can queue & process messages sequentially or concurrently

The ACE_Task Class

Basis of ACE's OO concurrency framework that provides the following capabilities:

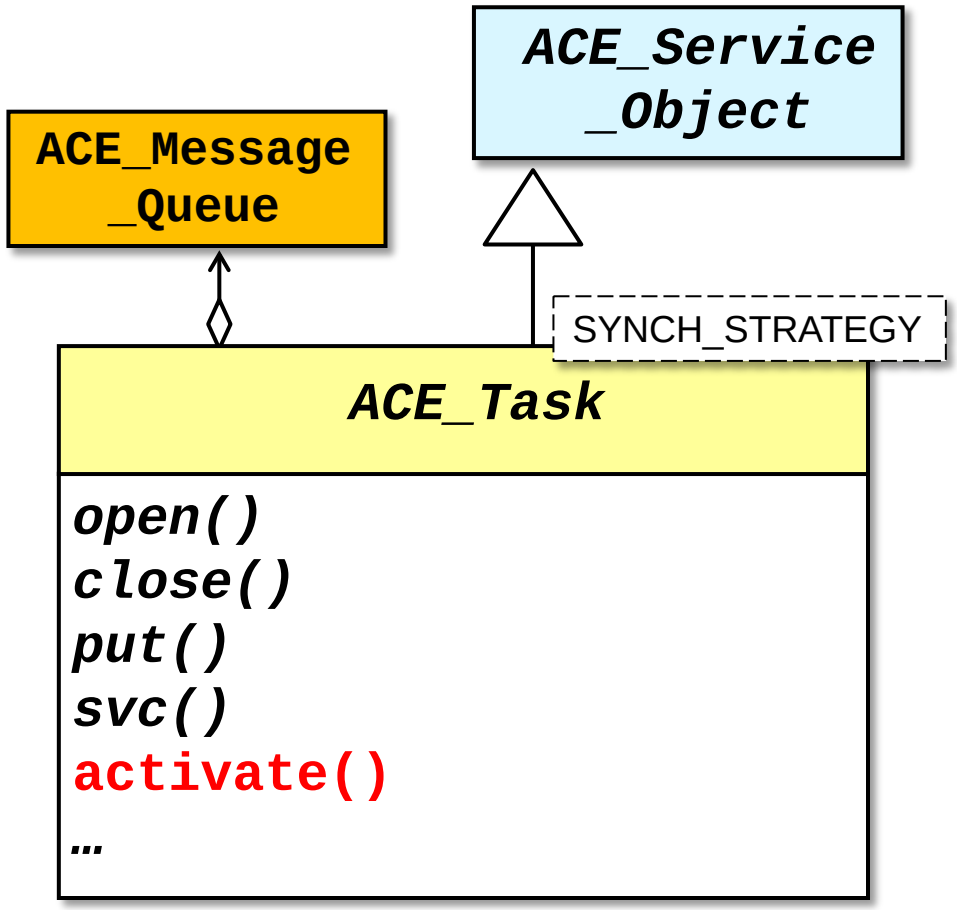
- Its message queue can decouple data & requests from *when* they are processed



The ACE_Task Class

Basis of ACE's OO concurrency framework that provides the following capabilities:

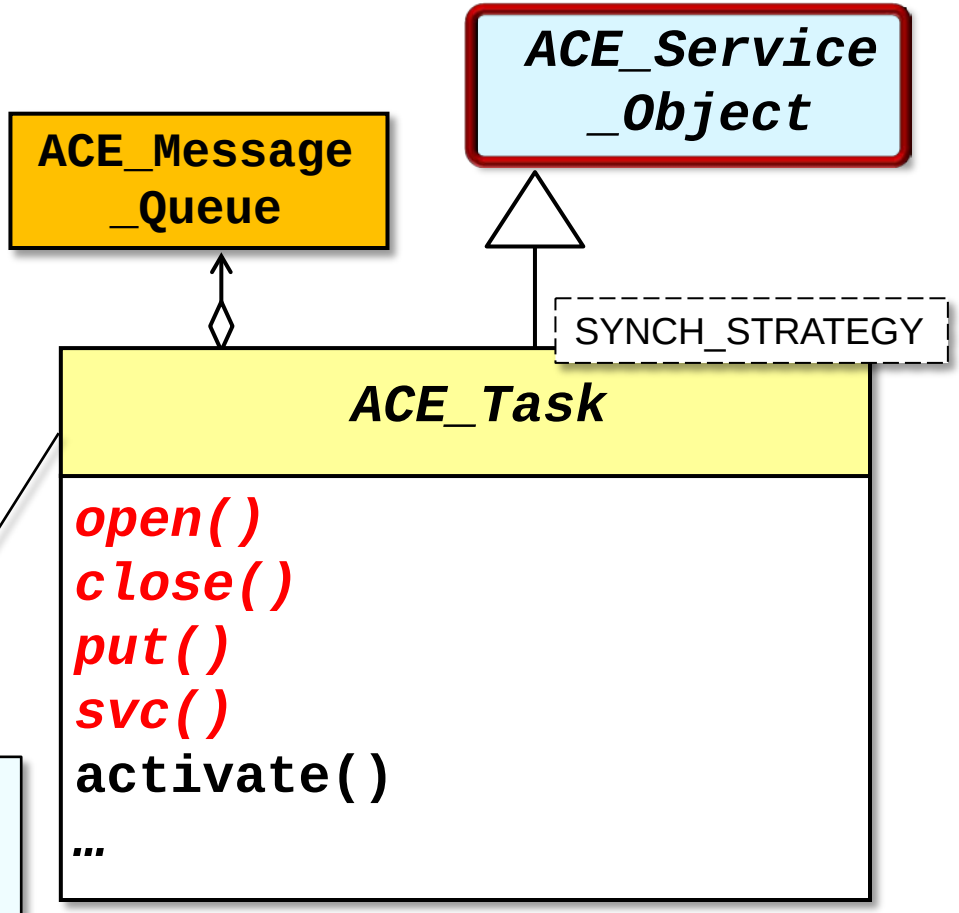
- Its message queue can decouple data & requests from *when* they are processed
- Can be activated to run as an *active object* that processes its queued messages in one or more threads



The ACE_Task Class

Basis of ACE's OO concurrency framework that provides the following capabilities:

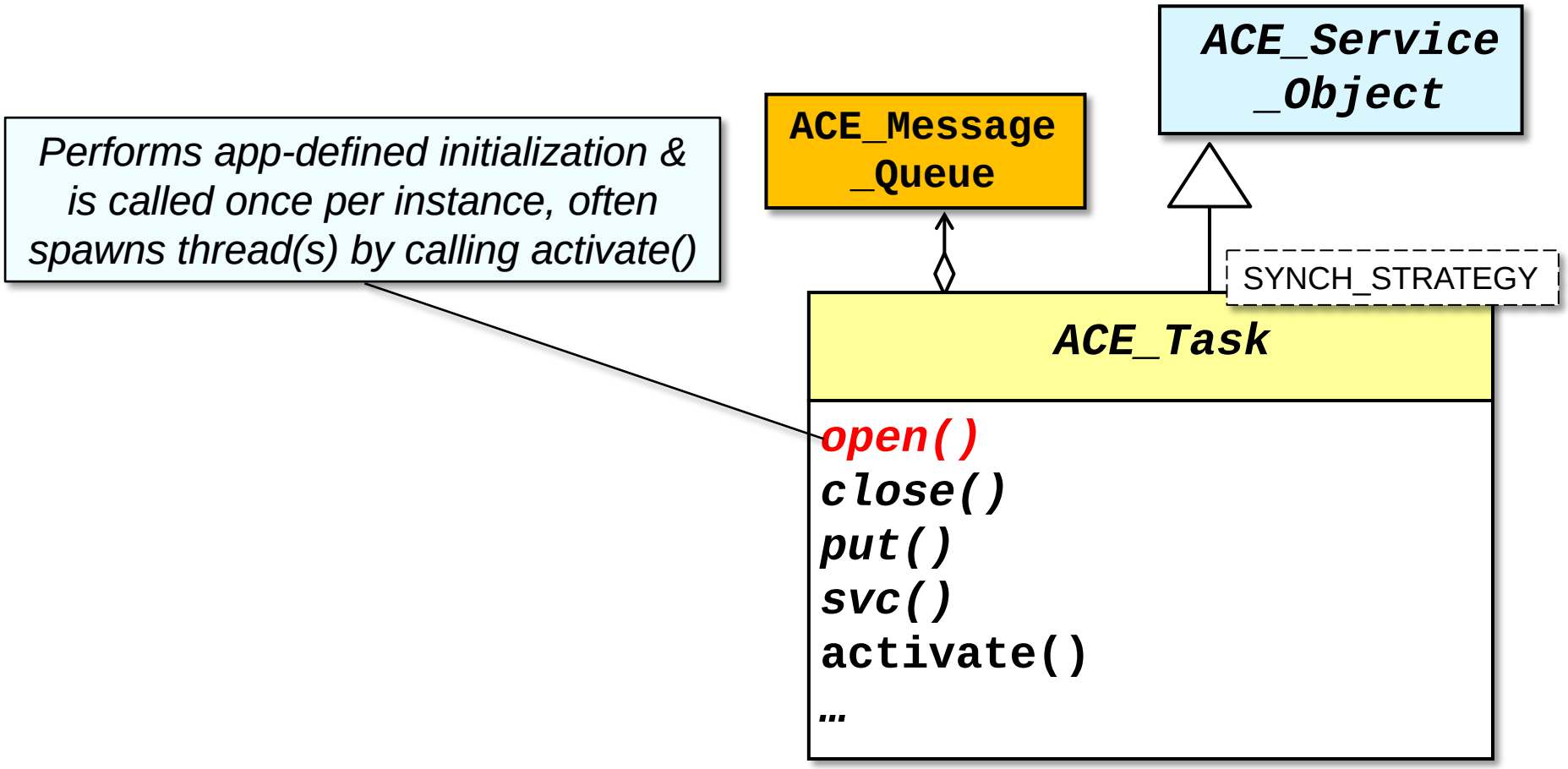
- Its message queue can decouple data & requests from *when* they are processed
- Can be activated to run as an *active object* that processes its queued messages in one or more threads
- Its hook methods can be overridden for task-specific service execution & message handling



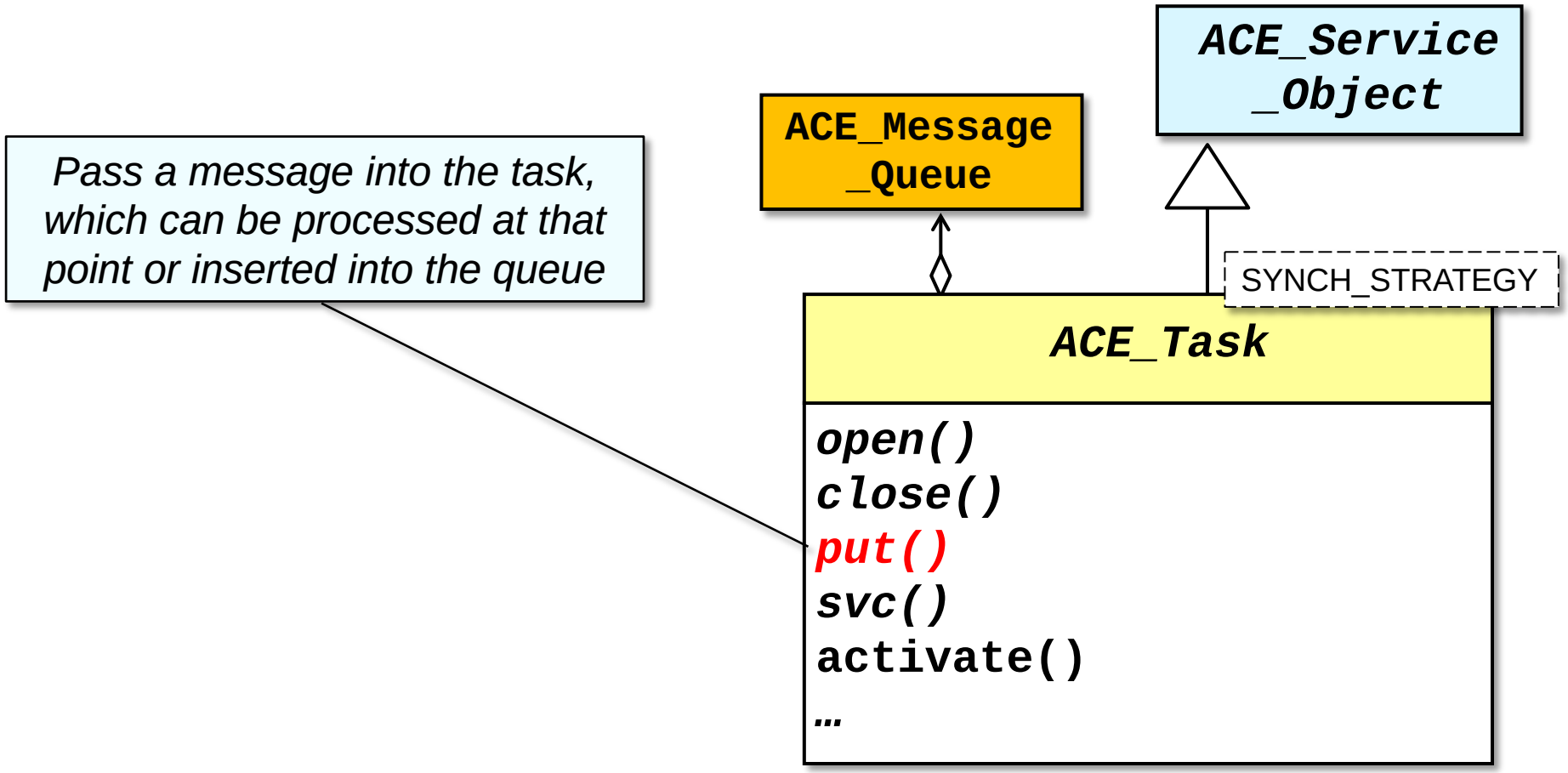
ACE_Task *inherits ACE dynamic configuration & event handling framework capabilities from its base classes*

Handles *variability* of concurrency & queueing via a *common* API

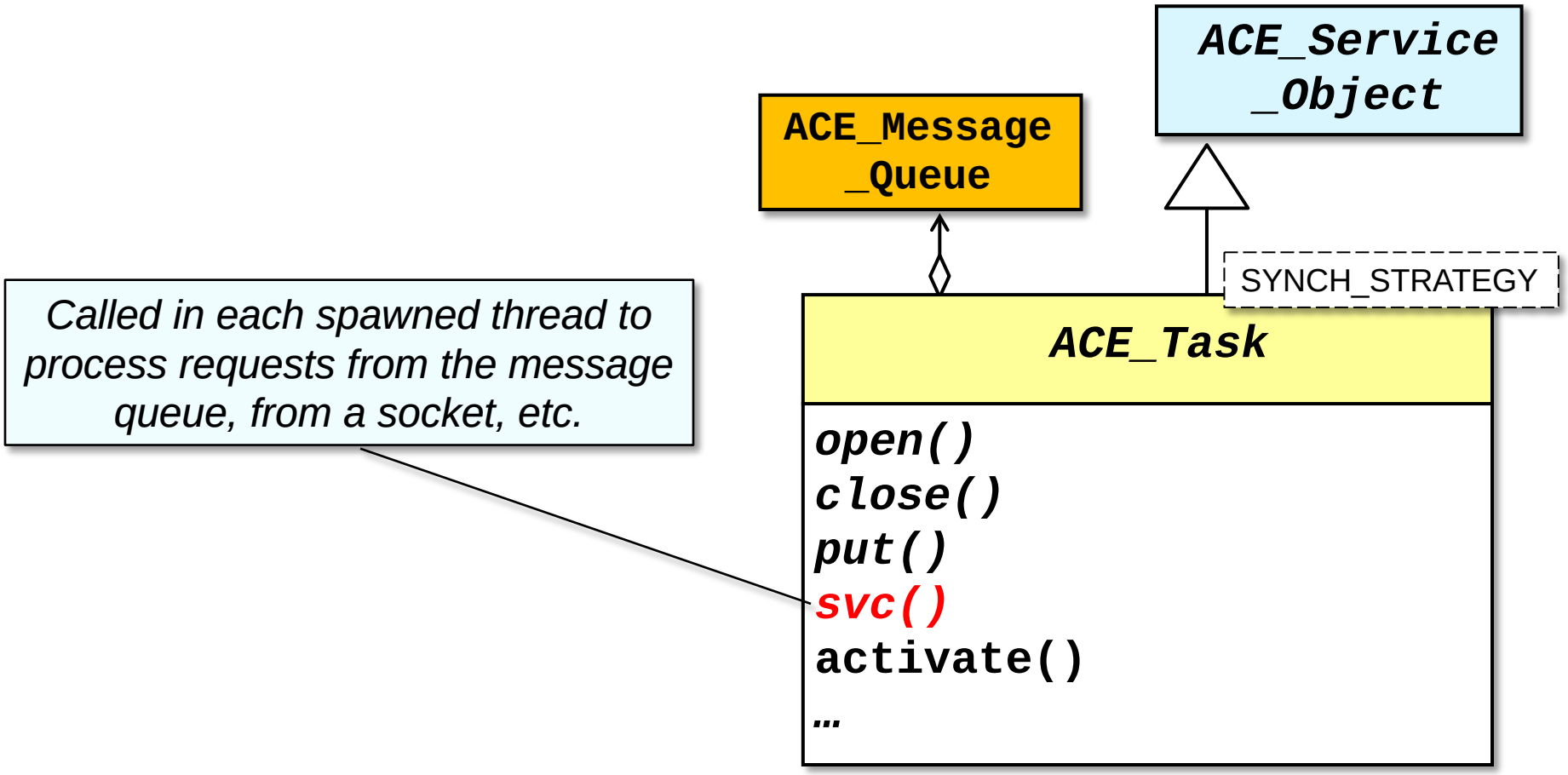
ACE_Task Hook Methods



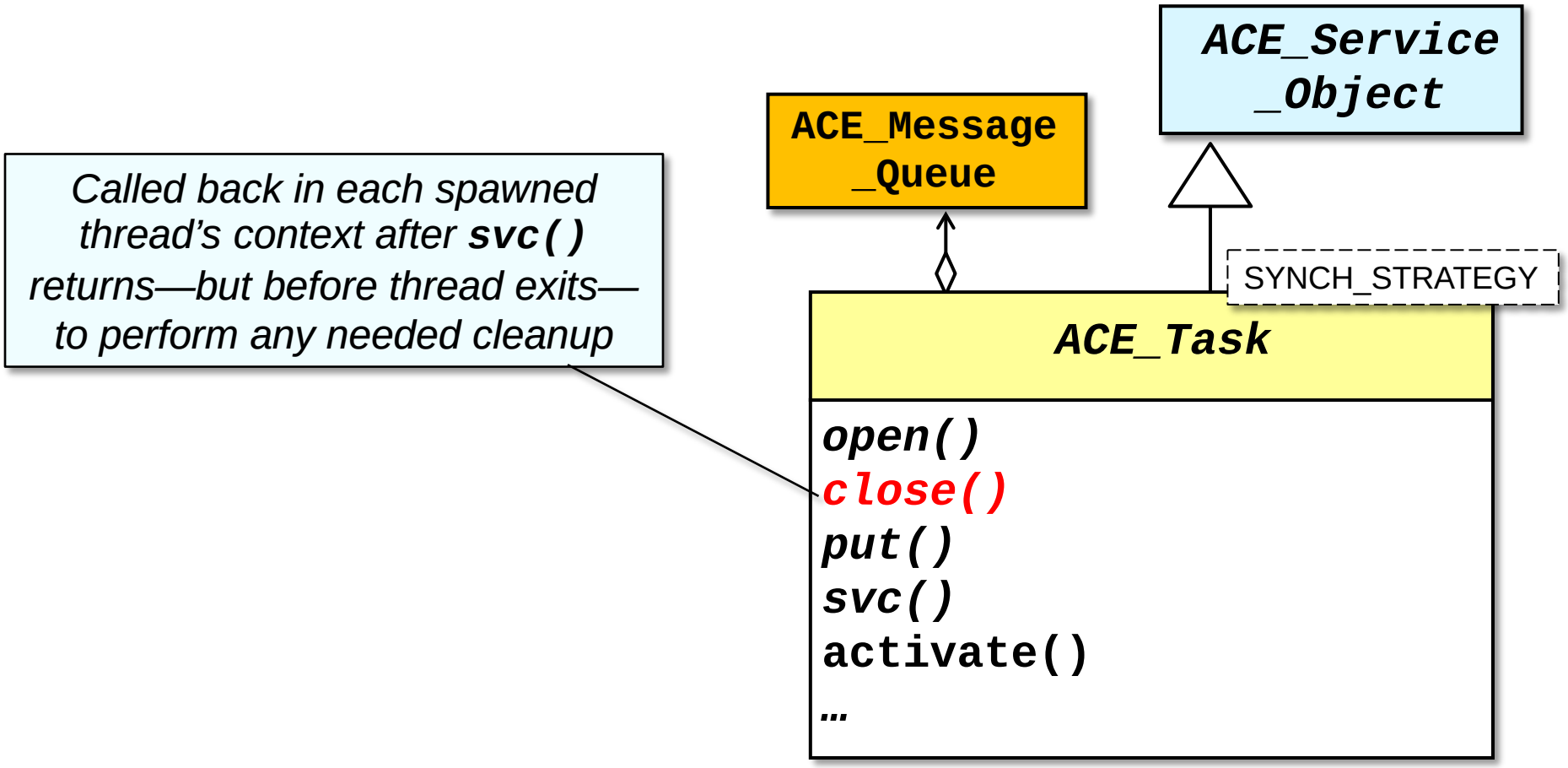
ACE_Task Hook Methods



ACE_Task Hook Methods



ACE_Task Hook Methods



Activating an ACE_Task

- The **`activate()`** method converts an **`ACE_Task`** into an active object

```
class TPC_HTTP_Handler
  : public ACE_Svc_Handler<ACE_SOCK_Stream, ACE_NULL_SYNCH> {

public:
  virtual int open (void *) { return activate (); }
```

Become an active object & calls the `svc()` hook method in 1 thread

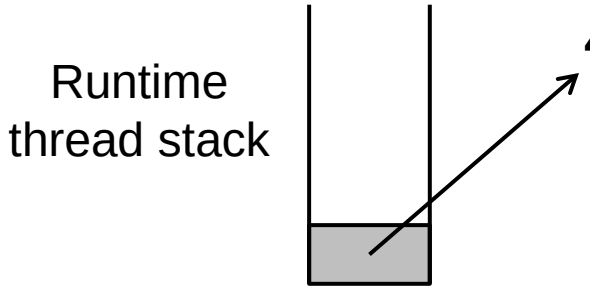
```
class TP_HTTP_Task : public ACE_Task<ACE_MT_SYNCH> {
public:
  enum { THREAD_POOL_SIZE = 8 };

  virtual int open (void *) {
    return activate (THR_BOUND, THREAD_POOL_SIZE);
  }
```

Become an active object & calls the `svc()` hook method in 8 threads

Activating an ACE_Task

- The **`activate()`** method converts an **`ACE_Task`** into an active object
- It uses the **`ACE_Task::svc_run()`** static method as an adapter function to runs in newly spawned thread(s) of control & provide an execution context for the **`svc()`** hook method

```
1. ACE_Task_Base::activate()
2. ACE_Thread_Manager::spawn(&ACE_Task_Base::svc_run, this);
3. _beginthreadex(0, 0, &ACE_Task_Base::svc_run, this,
                  0, &threadId);
```

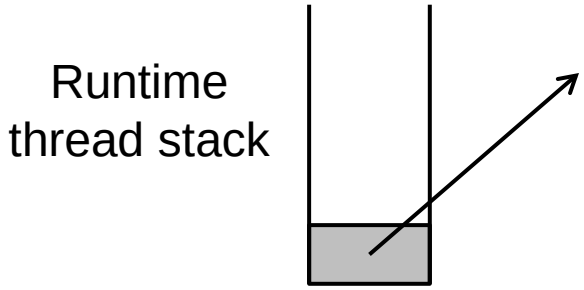



```
4. void *ACE_Task_Base::svc_run
                        (ACE_Task_Base *t) {
     ...
     t->svc();
     ...
   }
```

Activating an ACE_Task

- The **`activate()`** method converts an **`ACE_Task`** into an active object
- It uses the **`ACE_Task::svc_run()`** static method as an adapter function to runs in newly spawned thread(s) of control & provide an execution context for the **`svc()`** hook method

```
1. ACE_Task_Base::activate()
2. ACE_Thread_Manager::spawn(&ACE_Task_Base::svc_run, this);
3. _beginthreadex(0, 0, &ACE_Task_Base::svc_run, this,
                  0, &threadId);
```

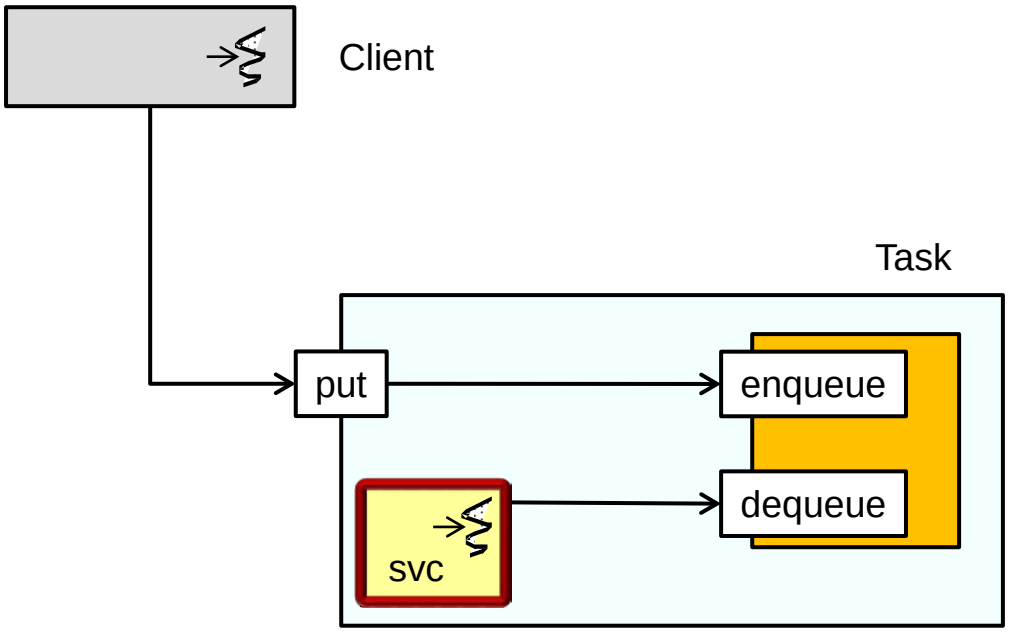



```
4. void *ACE_Task_Base::svc_run
                        (ACE_Task_Base *t) {
     ...
     t->svc();
     ...
   }
```

- The **`ACE_Task`** class shields apps from OS-specific concurrency details
  - e.g., it's simple to replace **`_beginthreadex()`** with **`pthread_create()`**

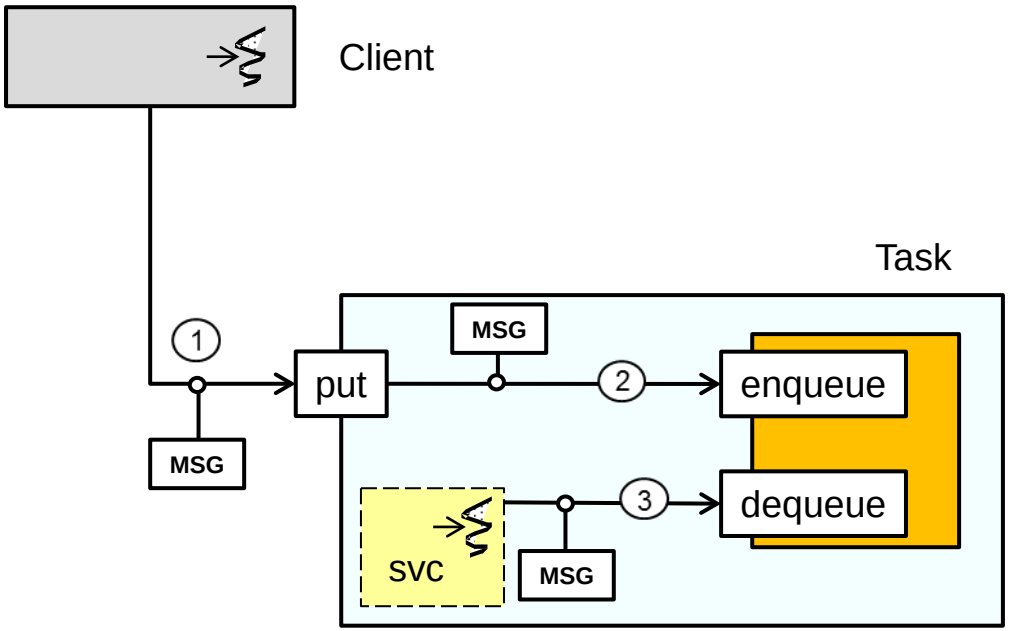
Summary

- The ACE *Task* framework provides extensible object-oriented concurrency capabilities that can
  - Spawn thread(s) in the context of an object



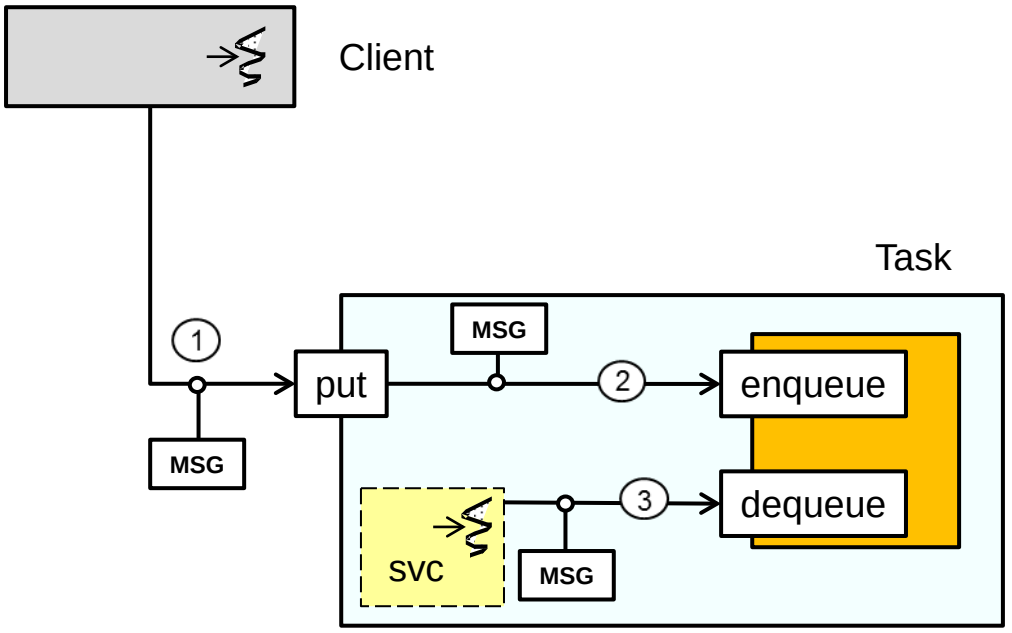
Summary

- The ACE *Task* framework provides extensible object-oriented concurrency capabilities that can
  - Spawn thread(s) in the context of an object
  - Transfer & queue messages between objects executing in separate threads



Summary

- The ACE *Task* framework provides extensible object-oriented concurrency capabilities that can
  - Spawn thread(s) in the context of an object
  - Transfer & queue messages between objects executing in separate threads

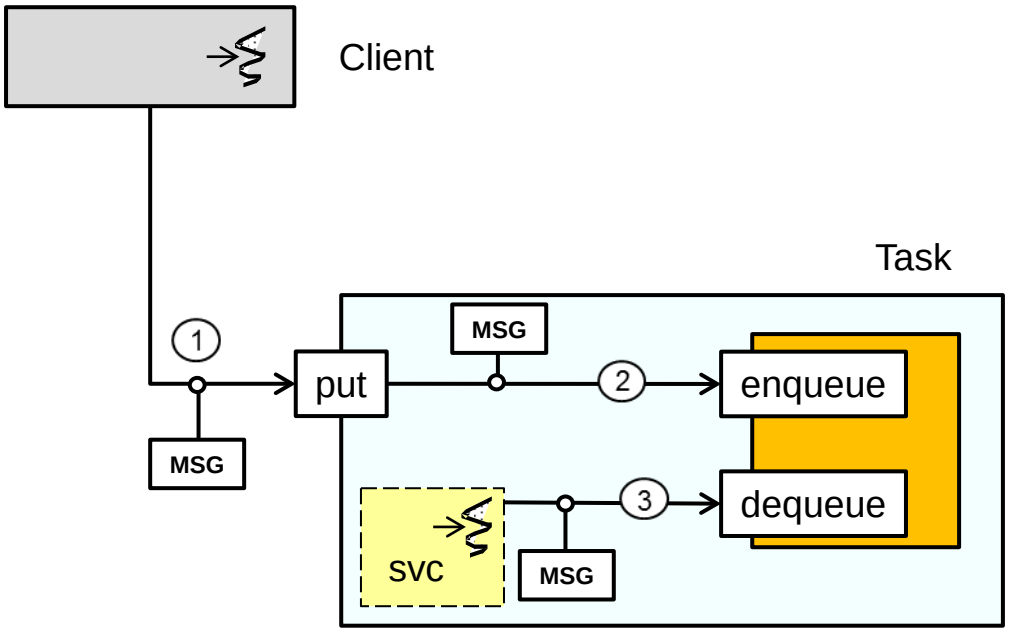


- This framework can be applied to implement key concurrency patterns, including
  - *Active Object* – which decouples the thread that invokes a method from the thread that executes the method

We'll show a subset of the complete *Active Object* pattern shortly

Summary

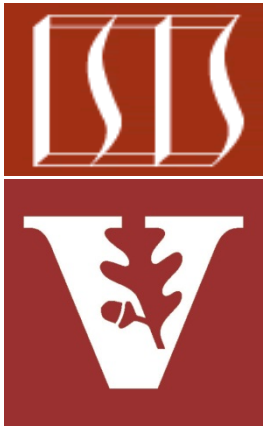
- The ACE *Task* framework provides extensible object-oriented concurrency capabilities that can
  - Spawn thread(s) in the context of an object
  - Transfer & queue messages between objects executing in separate threads



- This framework can be applied to implement key concurrency patterns, including
  - *Active Object* – which decouples the thread that invokes a method from the thread that executes the method
  - *Half-Sync/Half-Async* – which decouples asynchronous & synchronous processing in concurrent systems to simplify programming without unduly reducing performance

Patterns & Frameworks for Concurrency & Synchronization: Part 3

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



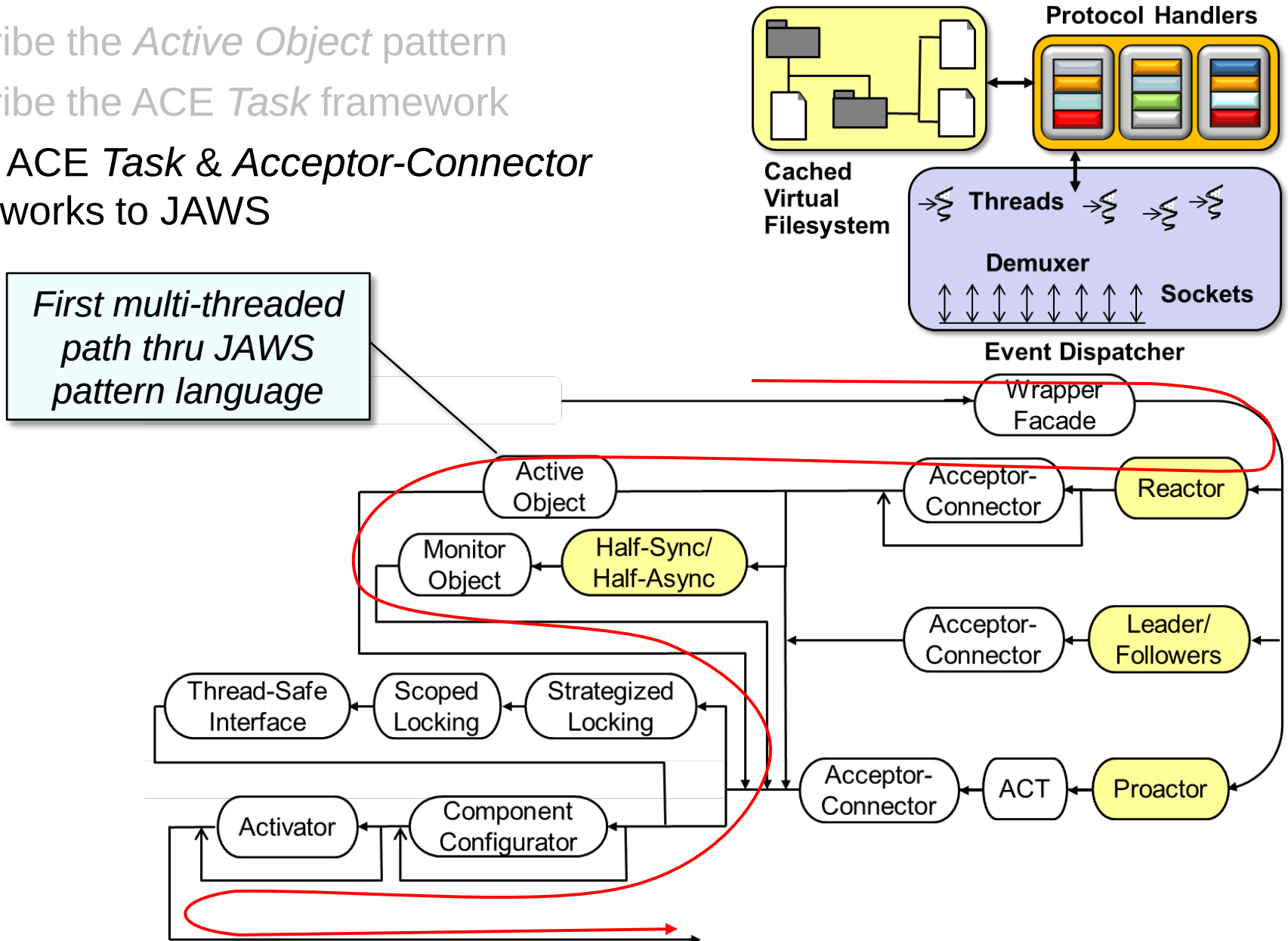
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

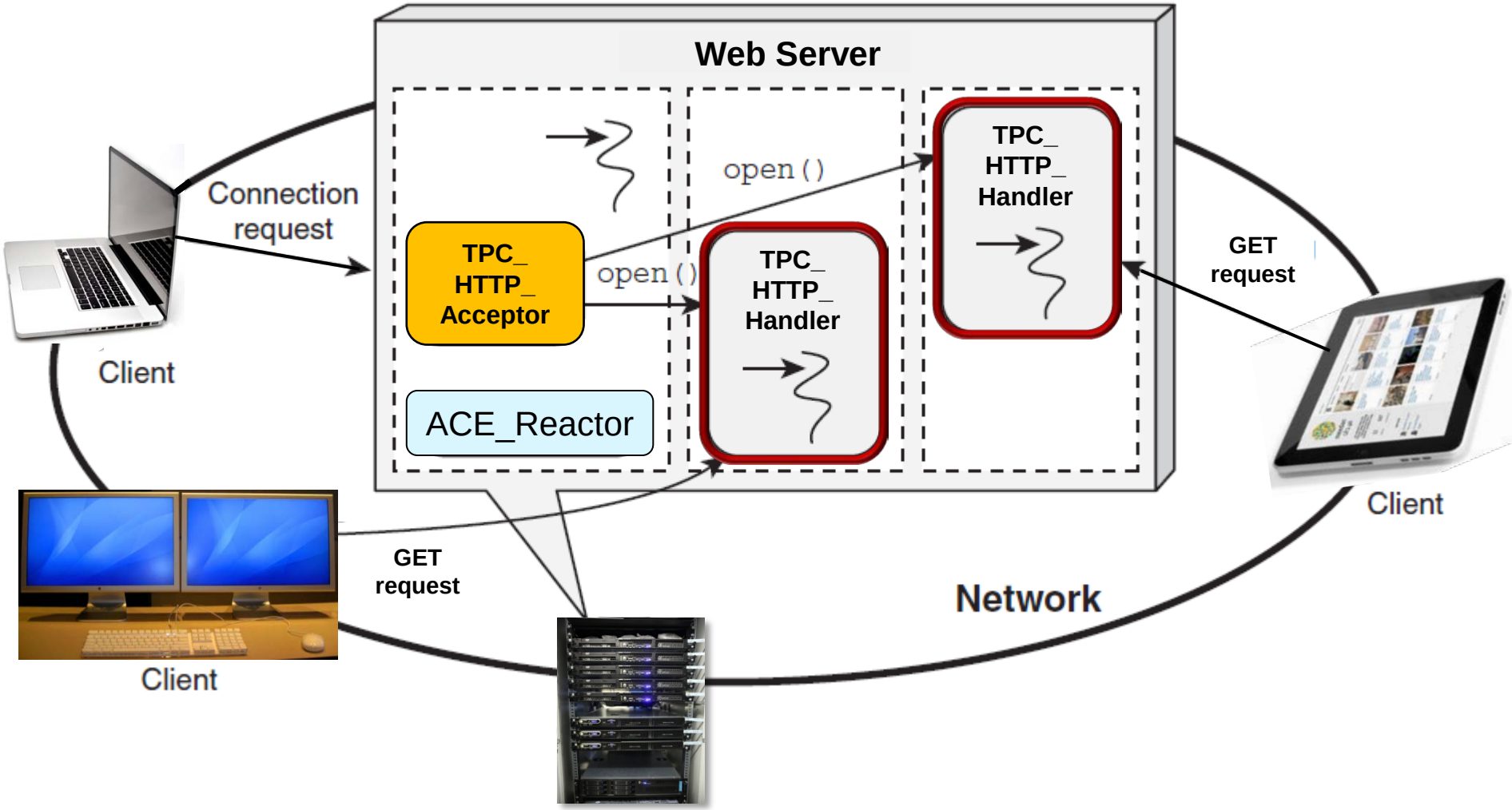
Topics Covered in this Part of the Module

- Describe the *Active Object* pattern
- Describe the ACE *Task* framework
- Apply ACE *Task* & *Acceptor-Connector* frameworks to JAWS



Thread-per-Connection with ACE_Svc_Handler

Use **`ACE_Svc_Handler`** (which inherits from **`ACE_Task`**) to implement a web server using *Active Object* pattern & *thread-per-connection* concurrency model



Thread-per-Connection with ACE_Svc_Handler

```
class TPC_HTTP_Handler
  : public ACE_Svc_Handler<ACE_SOCK_Stream, ACE_NULL_SYNCH> {

public:
  virtual int open (void *) { return activate (); }
```

Be a service handler

Hook method called by ACE_Acceptor for each connection

Thread-per-Connection with ACE_Svc_Handler

```
class TPC_HTTP_Handler
  : public ACE_Svc_Handler<ACE_SOCK_Stream, ACE_NULL_SYNCH> {

public:
  virtual int open (void *) { return activate (); }
```

Become an active object & call the `svc()` hook method

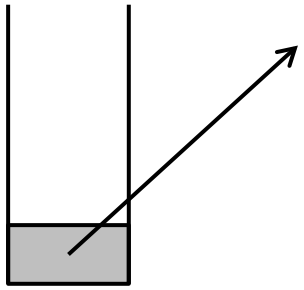
Thread-per-Connection with ACE_Svc_Handler

```
class TPC_HTTP_Handler
  : public ACE_Svc_Handler<ACE_SOCK_Stream, ACE_NULL_SYNCH> {

public:
  virtual int open (void *) { return activate (); }
```

Become an active object & call the `svc()` hook method

1. `ACE_Task_Base::activate()`
2. `ACE_Thread_Manager::spawn(&ACE_Task_Base::svc_run, this);`
3. `pthread_create(0, 0, &ACE_Task_Base::svc_run, this)`

```
4. void *ACE_Task_Base::svc_run
                    (ACE_Task_Base *t) {
     ...
     t->svc();
     ...
   }
```



Thread-per-Connection with ACE_Svc_Handler

```
class TPC_HTTP_Handler
  : public ACE_Svc_Handler<ACE_SOCK_Stream, ACE_NULL_SYNCH> {

public:
  virtual int open (void *) { return activate (); }

  virtual int svc () {
    for (;;) {
      std::string pathname (get_pathname (peer ()));
      ACE_Mem_Map mapped_file (pathname.c_str ());

      switch (peer.send_n (mapped_file.addr (),
                           mapped_file.size ()) {
        case -1: return -1; // Error.
        case 0: return 0; // Client closed connection.
        default: continue; // Default case.
        }
    }
    return 0; /* NOTREACHED */
}
```

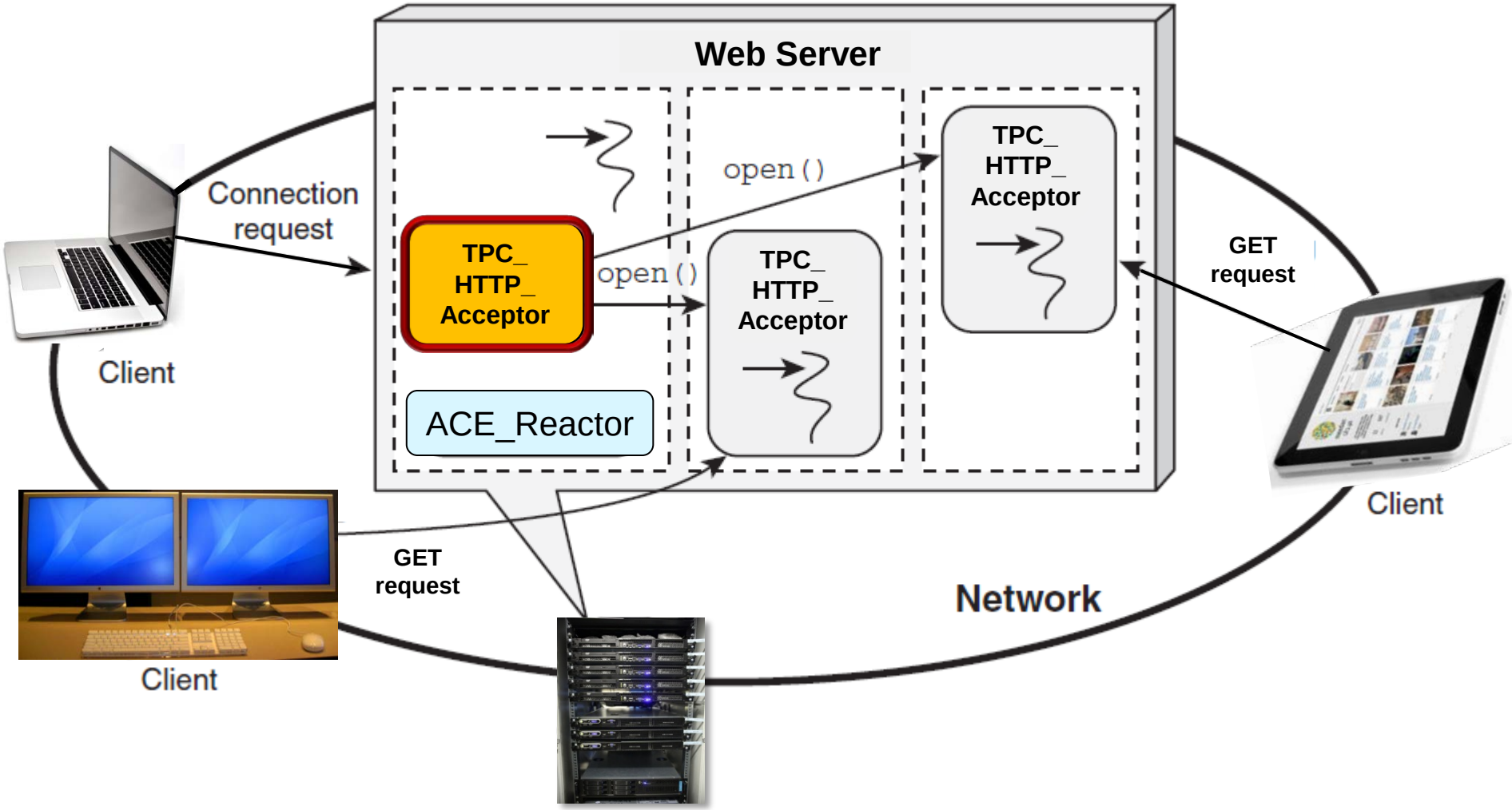
Runs in a separate thread of control

Memory map requested content

Send the requested content back to client

Thread-per-Connection with ACE_Acceptor

We now show how to use **`ACE_Acceptor`** to implement a web server based on the *Active Object* pattern & *thread-per-connection* concurrency model



Thread-per-Connection with ACE_Acceptor

Instantiate ACE_Acceptor template

```
1 typedef ACE_Acceptor<TPC_HTTP_Handler, ACE_SOCK_Acceptor>
2         TPC_HTTP_Acceptor;

3 typedef Reactor_HTTP_Server<TPC_HTTP_Acceptor>
4         HTTP_Server_Daemon;
```

Reuse our earlier template driver & Instantiate it with the TPC_HTTP_Acceptor

```
5 int main (int argc, char *argv[]) {
6   ACE_Reactor reactor;
7   new HTTP_Server_Daemon (argc, argv, &reactor);
```

Event loop controller object

Dynamic allocation ensures proper deletion semantics

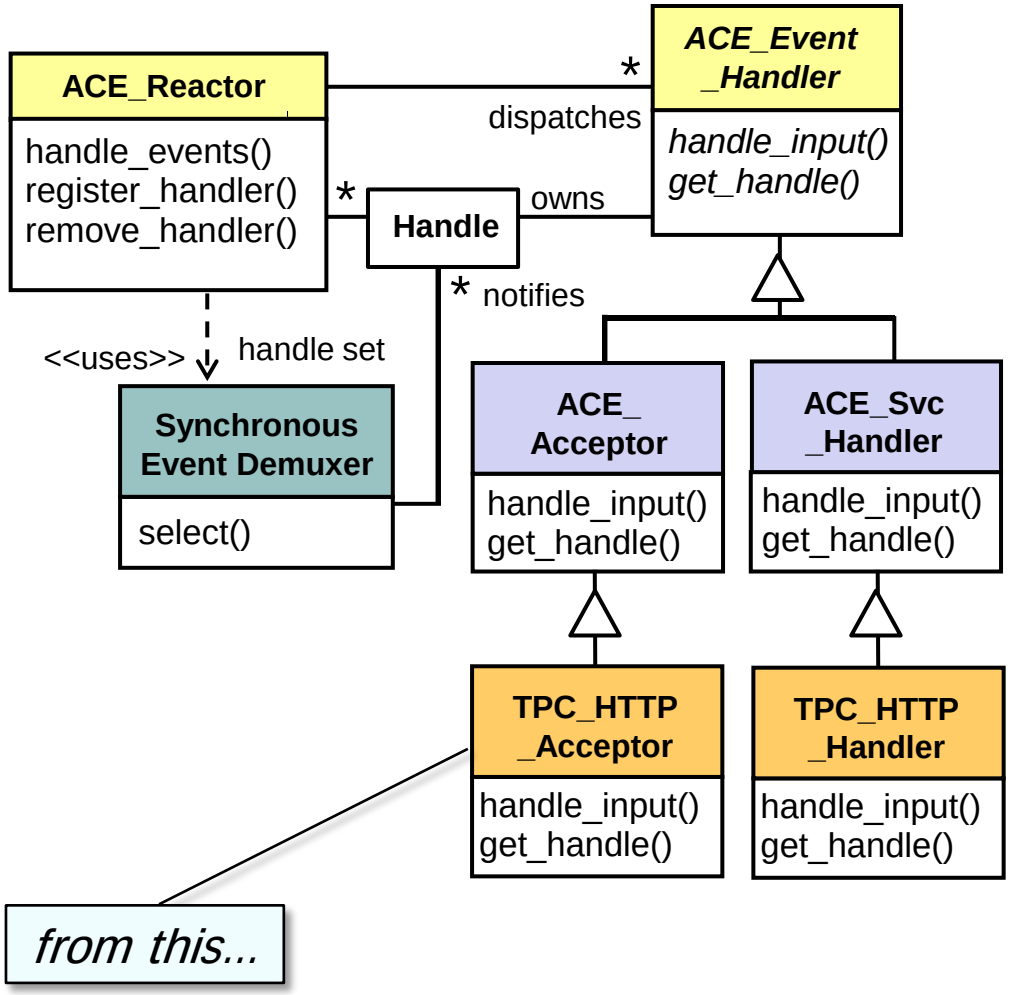
```
8   reactor.run_reactor_event_loop ();
    ...
```

Run server event loop

Note the extensive design & code reuse here!

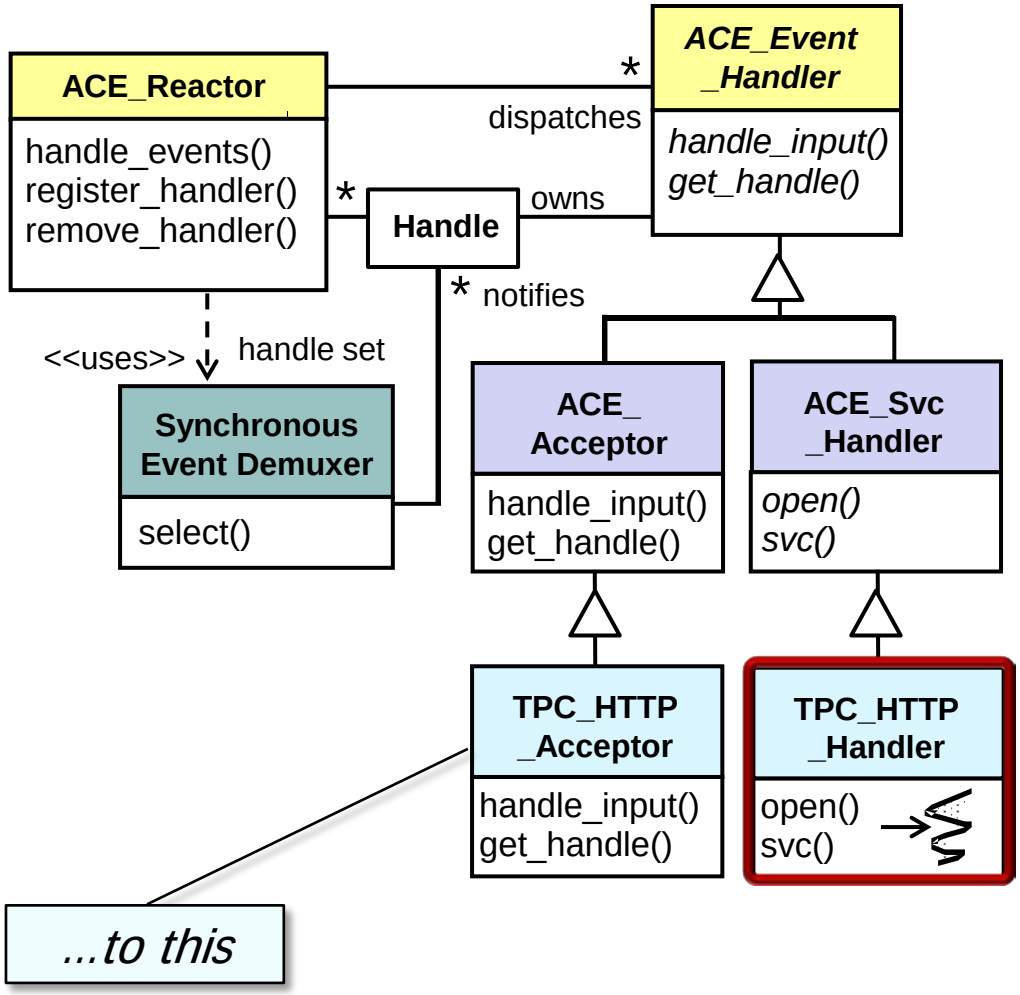
Summary

- Only minor changes were required to extend the *Reactor* & *Acceptor-Connector* version of the JAWS web server to use *Active Object*



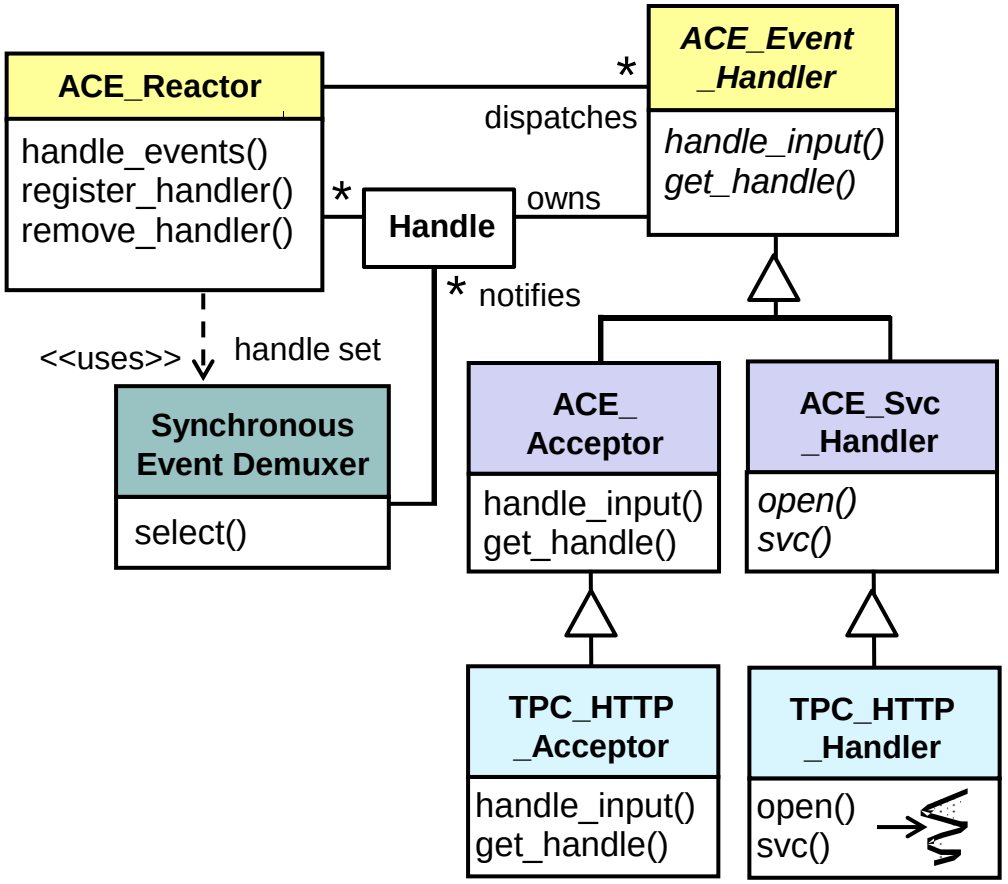
Summary

- Only minor changes were required to extend the *Reactor* & *Acceptor-Connector* version of the JAWS web server to use *Active Object*



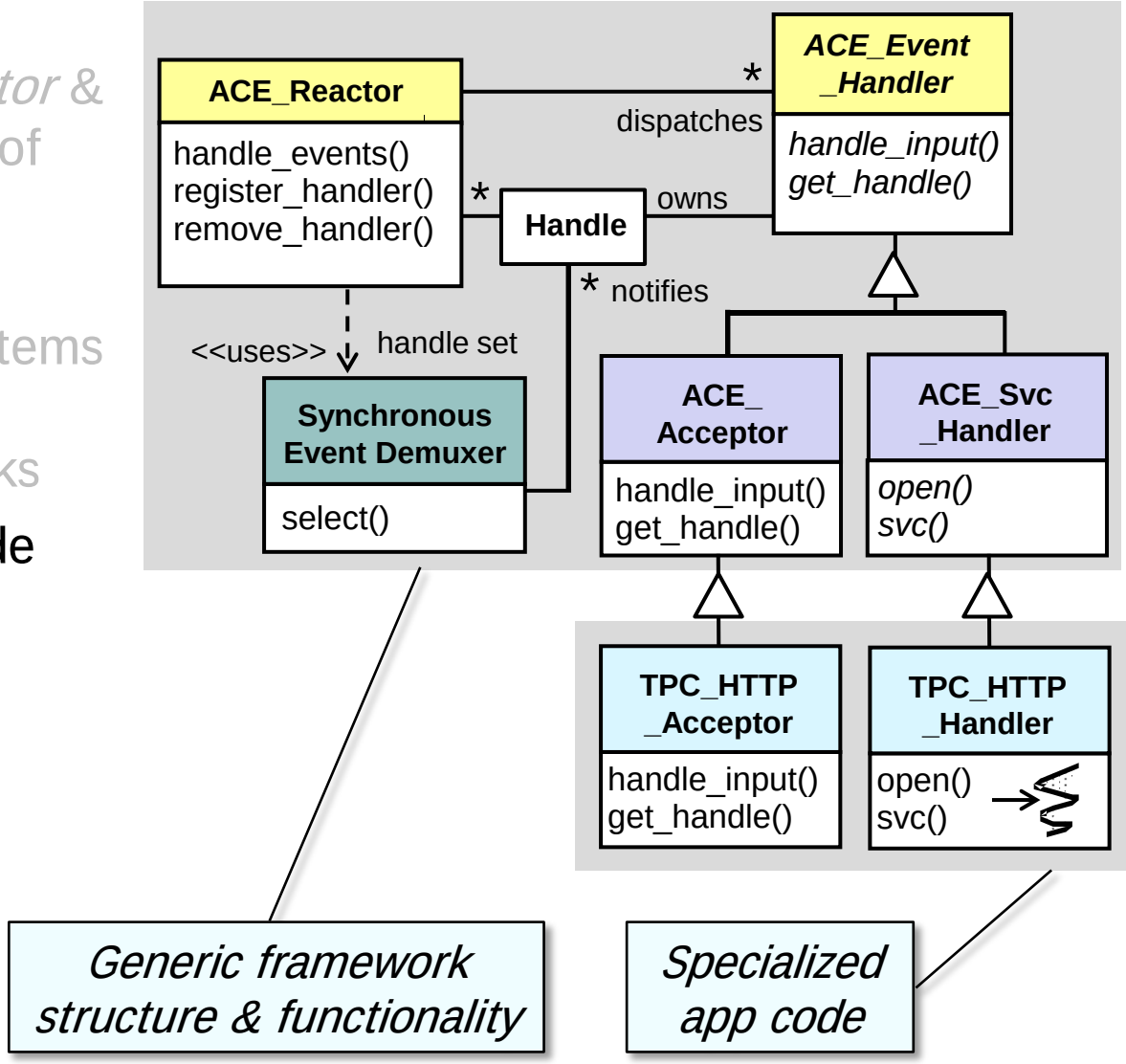
Summary

- Only minor changes were required to extend the *Reactor* & *Acceptor-Connector* version of the JAWS web server to use *Active Object*
- The minimal perturbations stems from the variation-oriented design of the ACE frameworks



Summary

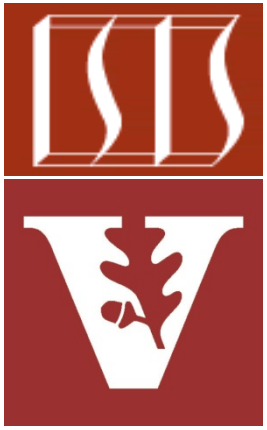
- Only minor changes were required to extend the *Reactor & Acceptor-Connector* version of the JAWS web server to use *Active Object*
- The minimal perturbations stems from the variation-oriented design of the ACE frameworks
- As before, very little app code was required since the ACE frameworks took care of the details of
  - event handling
  - connection establishment
  - service initialization
  - (service configuration)
  - concurrency



There's still a role for the *Reactor* pattern, just not in the main data path!

Patterns & Frameworks for Concurrency & Synchronization: Part 4

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



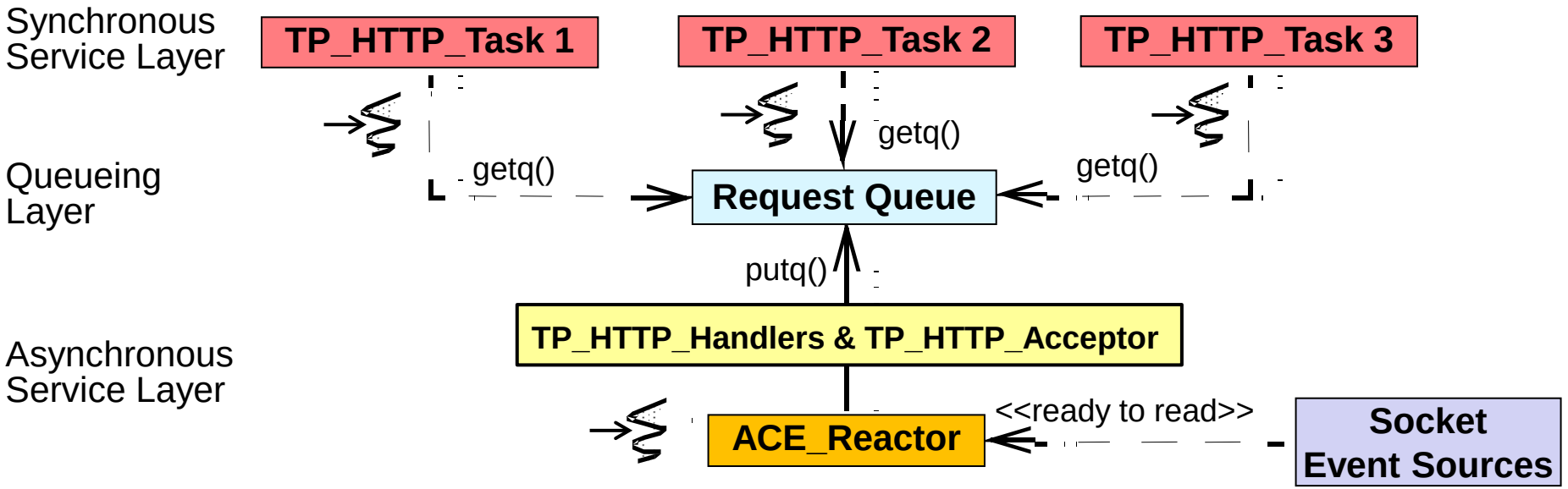
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

- Describe the *Active Object* pattern
- Describe the ACE *Task* framework
- Apply ACE *Task* & *Acceptor-Connector* frameworks to JAWS
- Describe the *Half-Sync/Half-Async* pattern



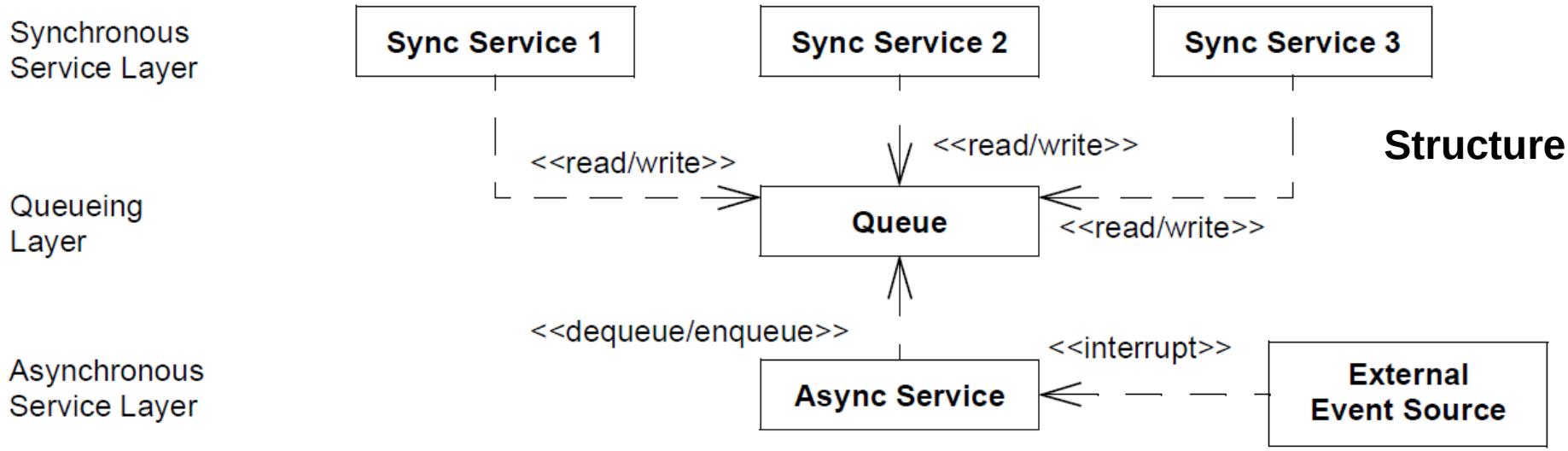
Improving Upon Thread-per-Connection

Context	Problem
• Web servers that run in settings with large # of clients with bursty request patterns	• Allocating an OS thread per connected client doesn't scale without devoting enormous processing capability & memory to the server(s)

Improving Upon Thread-per-Connection

Context	Problem	Solution
• Web servers that run in settings with large # of clients with bursty request patterns	• Allocating an OS thread per connected client doesn't scale without devoting enormous processing capability & memory to the server(s)	• Apply the *Half-Sync/Half-Async* pattern to scale server performance by processing HTTP requests concurrently in multiple threads

The *Half-Sync/Half-Async* pattern decouples async & sync service processing in concurrent systems, to simplify programming without unduly reducing performance

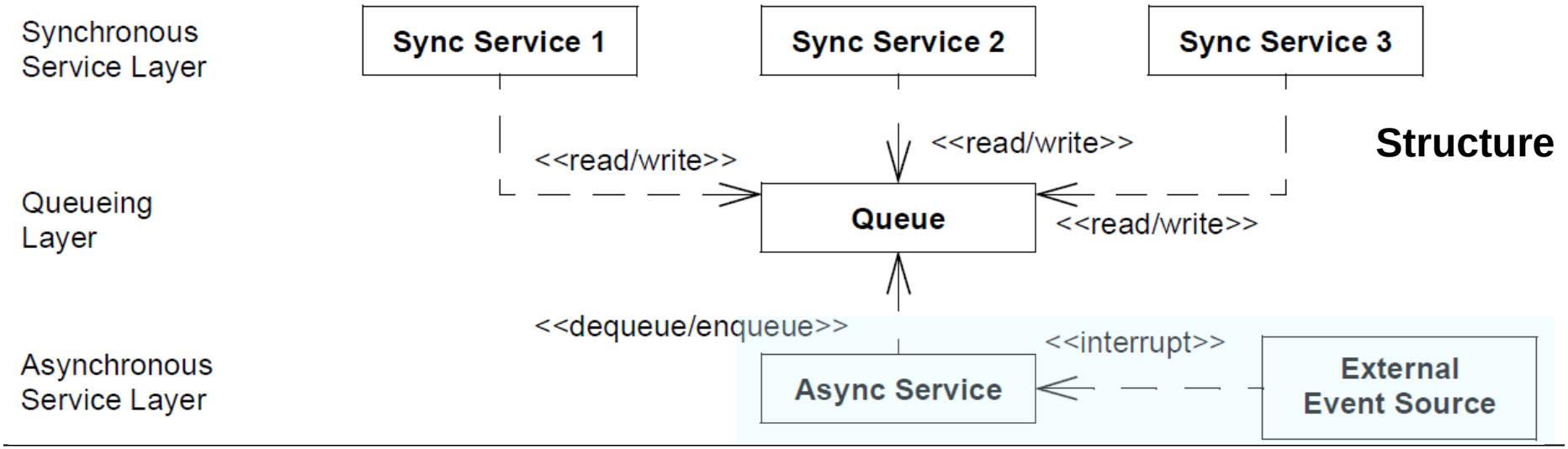


This pattern is widely used in operating systems & GUI frameworks

Improving Upon Thread-per-Connection

Context	Problem	Solution
• Web servers that run in settings with large # of clients with bursty request patterns	• Allocating an OS thread per connected client doesn't scale without devoting enormous processing capability & memory to the server(s)	• Apply the *Half-Sync/Half-Async* pattern to scale server performance by processing HTTP requests concurrently in multiple threads

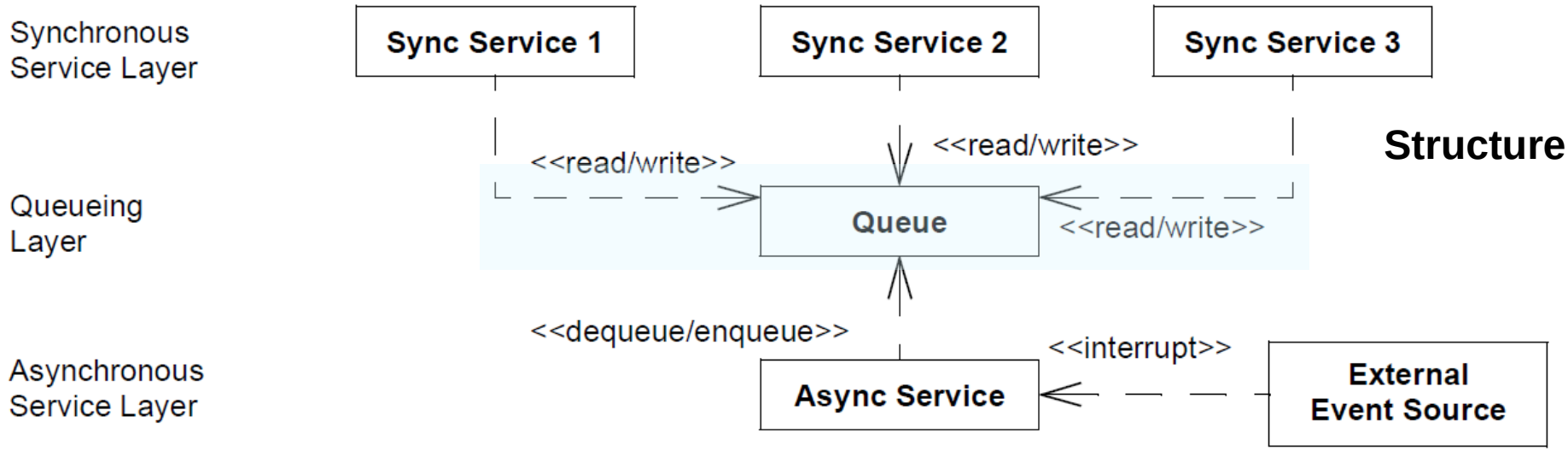
The *Half-Sync/Half-Async* pattern decouples async & sync service processing in concurrent systems, to simplify programming without unduly reducing performance



Improving Upon Thread-per-Connection

Context	Problem	Solution
• Web servers that run in settings with large # of clients with bursty request patterns	• Allocating an OS thread per connected client doesn't scale without devoting enormous processing capability & memory to the server(s)	• Apply the *Half-Sync/Half-Async* pattern to scale server performance by processing HTTP requests concurrently in multiple threads

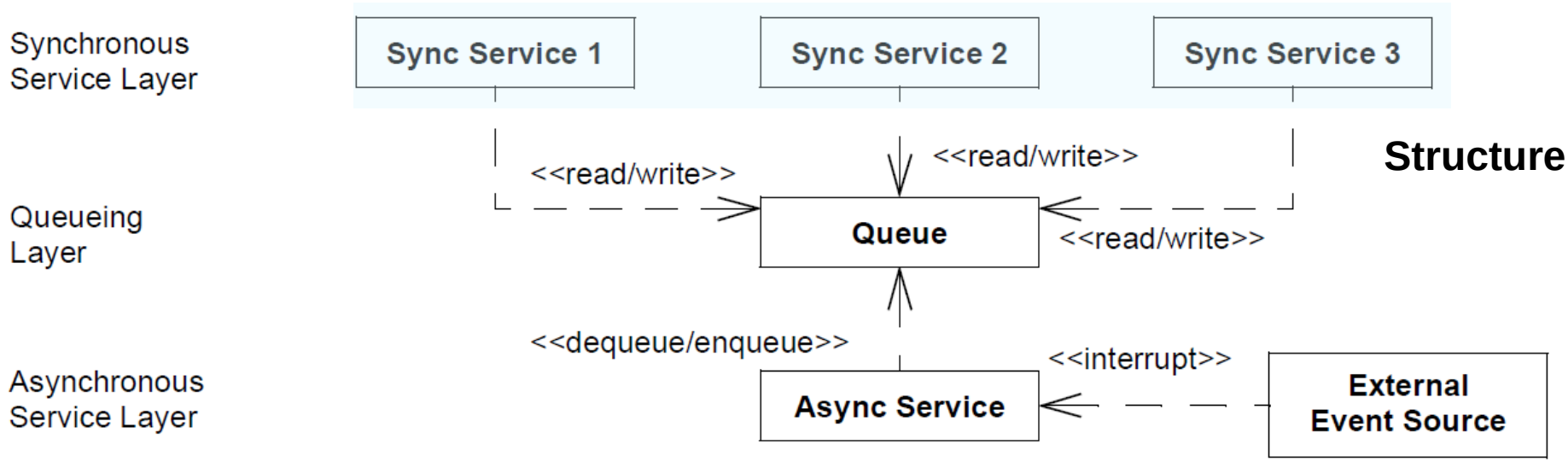
The *Half-Sync/Half-Async* pattern decouples async & sync service processing in concurrent systems, to simplify programming without unduly reducing performance



Improving Upon Thread-per-Connection

Context	Problem	Solution
• Web servers that run in settings with large # of clients with bursty request patterns	• Allocating an OS thread per connected client doesn't scale without devoting enormous processing capability & memory to the server(s)	• Apply the *Half-Sync/Half-Async* pattern to scale server performance by processing HTTP requests concurrently in multiple threads

The *Half-Sync/Half-Async* pattern decouples async & sync service processing in concurrent systems, to simplify programming without unduly reducing performance



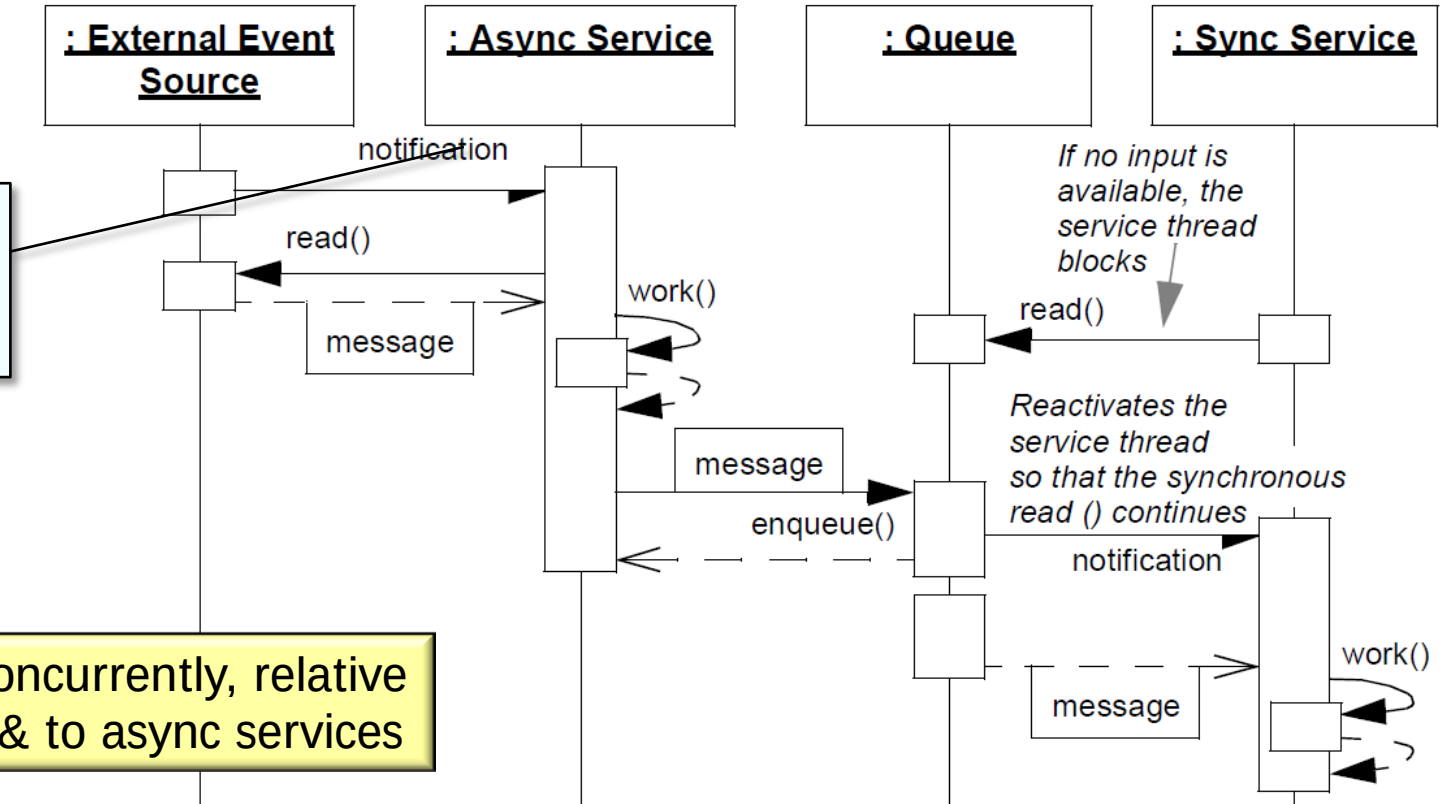
See www.dre.vanderbilt.edu/~schmidt/PDF/HS-HA.pdf for more info

Improving Upon Thread-per-Connection

Context	Problem	Solution
• Web servers that run in settings with large # of clients with bursty request patterns	• Allocating an OS thread per connected client doesn't scale without devoting enormous processing capability & memory to the server(s)	• Apply the *Half-Sync/Half-Async* pattern to scale server performance by processing HTTP requests concurrently in multiple threads

Dynamics

Event handling runs reactively/ asynchronously

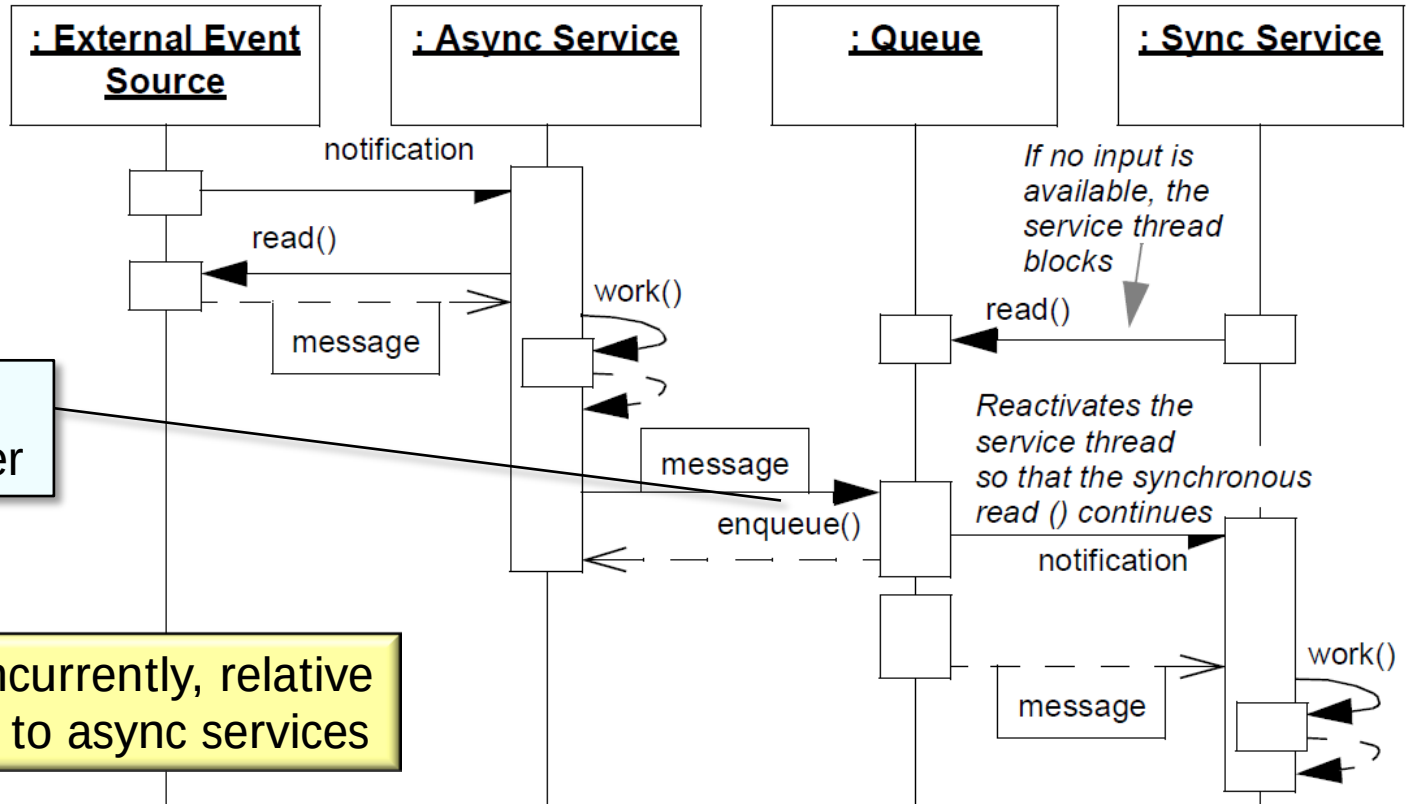


Sync services run concurrently, relative both to each other & to async services

Improving Upon Thread-per-Connection

Context	Problem	Solution
• Web servers that run in settings with large # of clients with bursty request patterns	• Allocating an OS thread per connected client doesn't scale without devoting enormous processing capability & memory to the server(s)	• Apply the *Half-Sync/Half-Async* pattern to scale server performance by processing HTTP requests concurrently in multiple threads

Dynamics



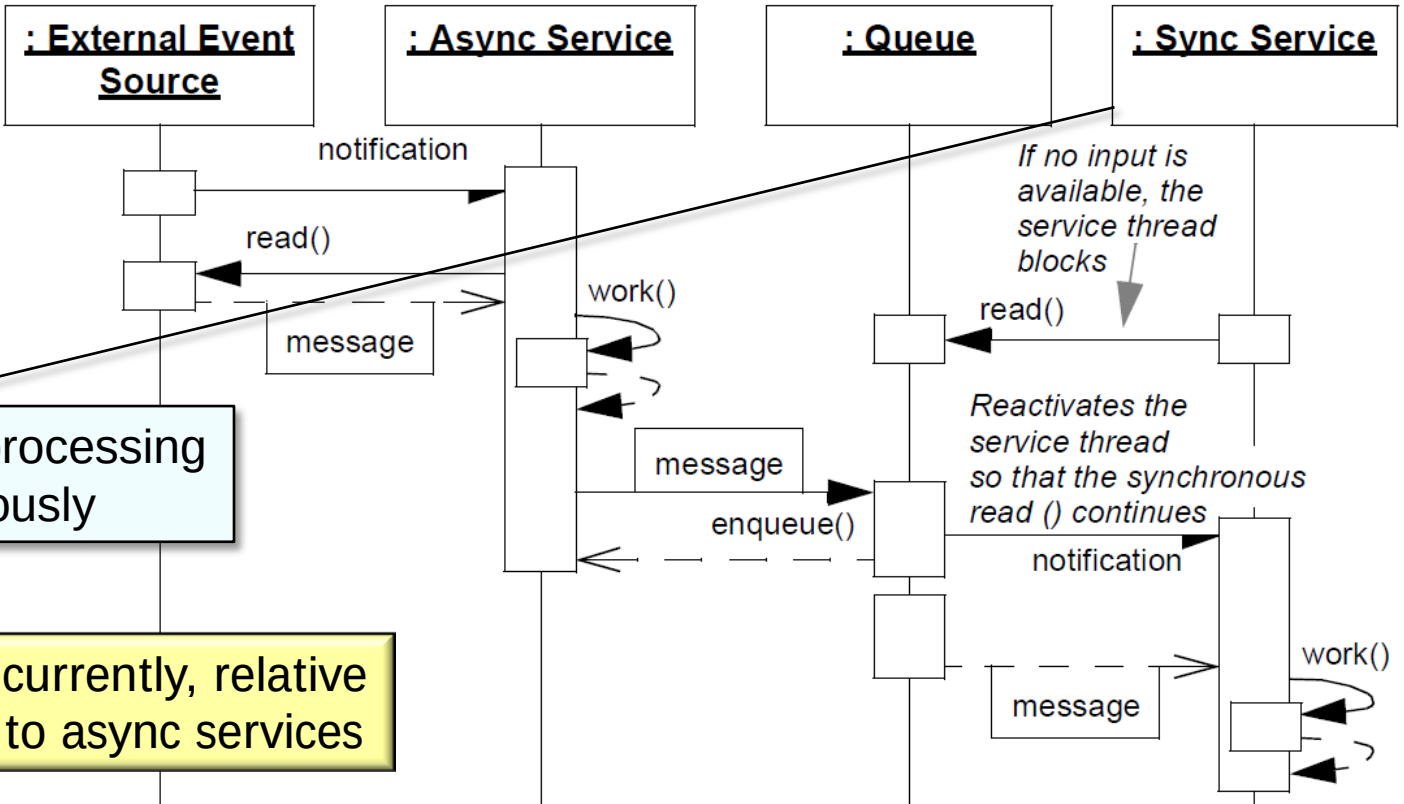
Queue requests without blocking caller

Sync services run concurrently, relative both to each other & to async services

Improving Upon Thread-per-Connection

Context	Problem	Solution
• Web servers that run in settings with large # of clients with bursty request patterns	• Allocating an OS thread per connected client doesn't scale without devoting enormous processing capability & memory to the server(s)	• Apply the *Half-Sync/Half-Async* pattern to scale server performance by processing HTTP requests concurrently in multiple threads

Dynamics

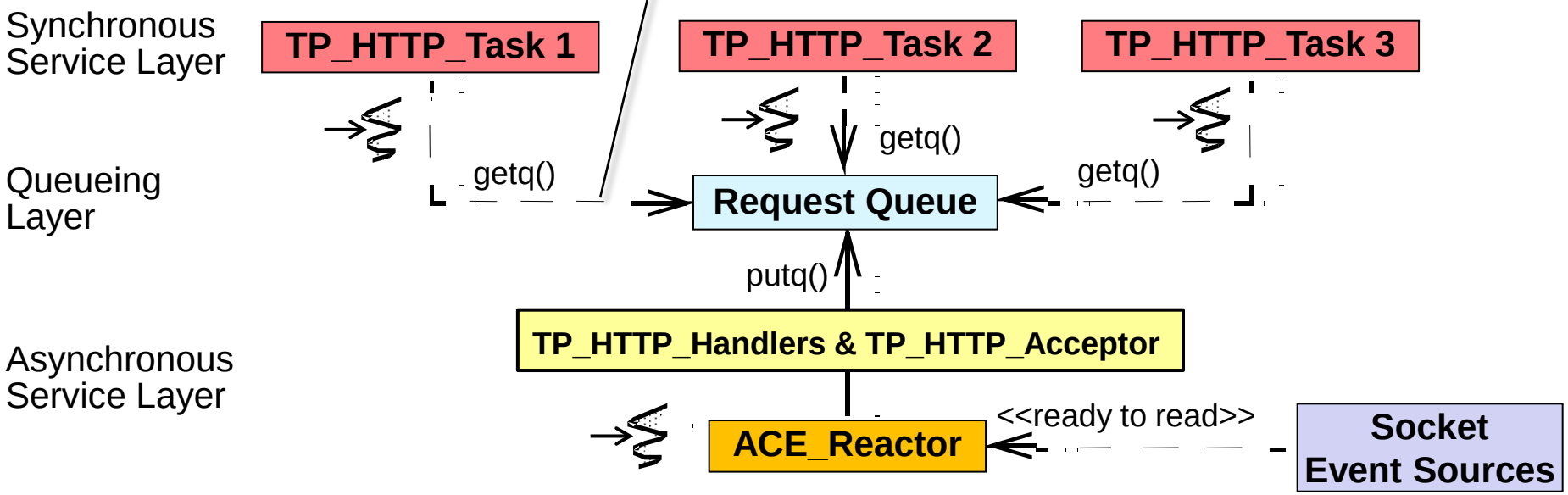


Long-duration app processing runs synchronously

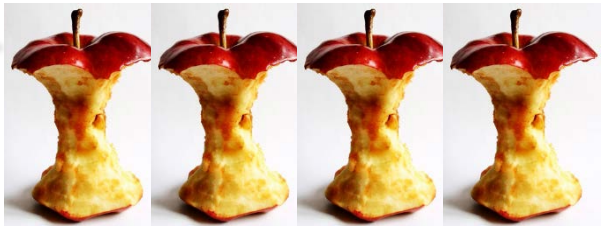
Sync services run concurrently, relative both to each other & to async services

Applying Half-Sync/Half-Async Pattern in JAWS

Each thread in the synchronous service layer blocks independently, which prevents a flow-controlled connection from degrading the performance that other clients receive

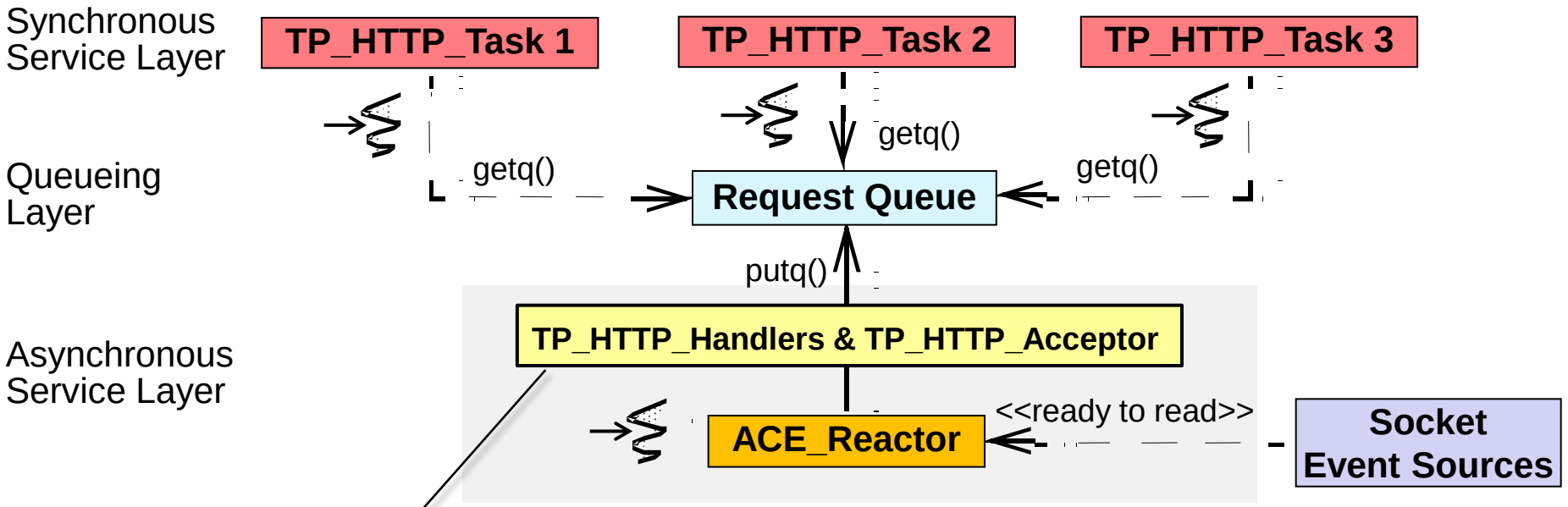


A bounded number of threads can be mapped to separate CPUs/cores to scale up server performance via concurrency

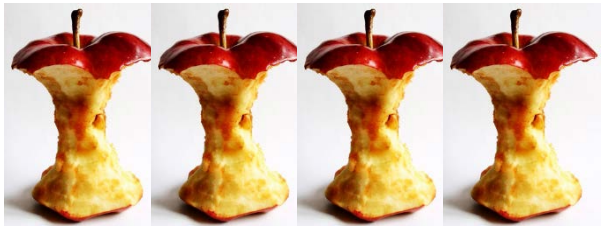


Applying Half-Sync/Half-Async Pattern in JAWS

JAWS can implement the *Half-Sync/Half-Async* pattern by combining the *Reactor* pattern with the *Active Object* pattern

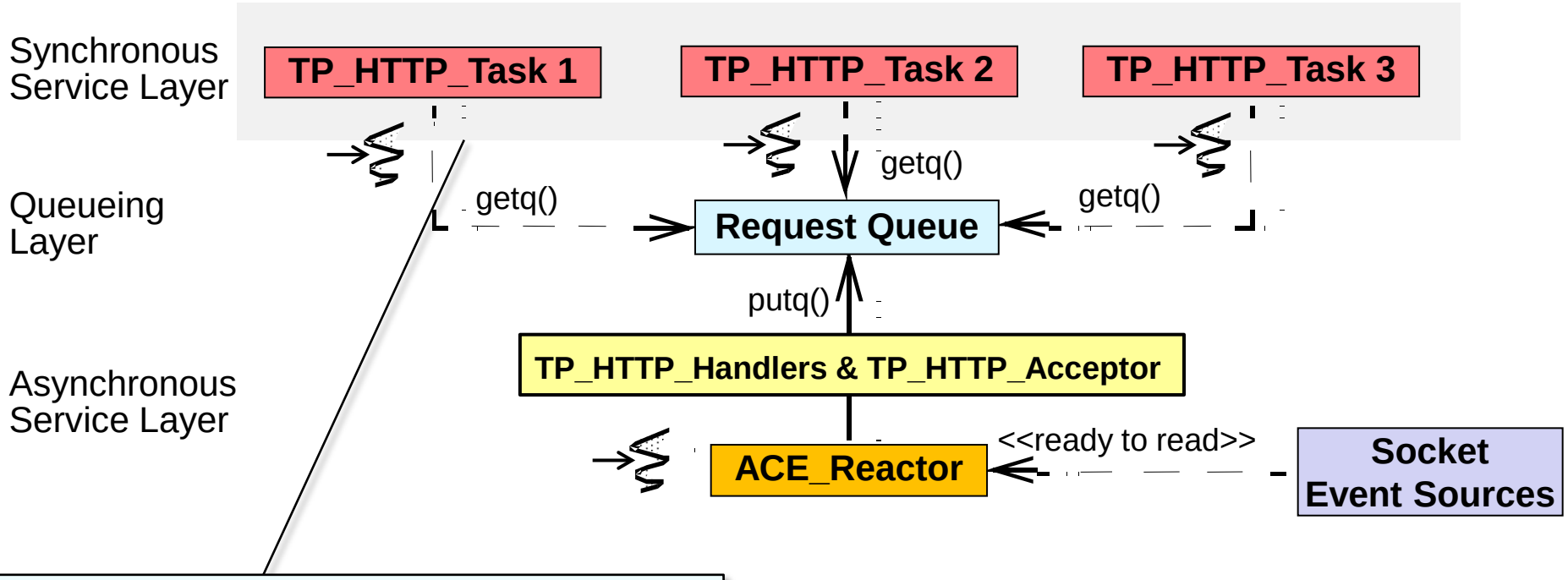


The Reactor processes HTTP GET requests asynchronously/ reactively from multiple clients & put them into the request queue

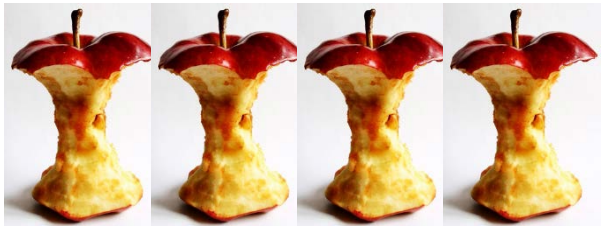


Applying Half-Sync/Half-Async Pattern in JAWS

JAWS can implement the *Half-Sync/Half-Async* pattern by combining the *Reactor* pattern with the *Active Object* pattern

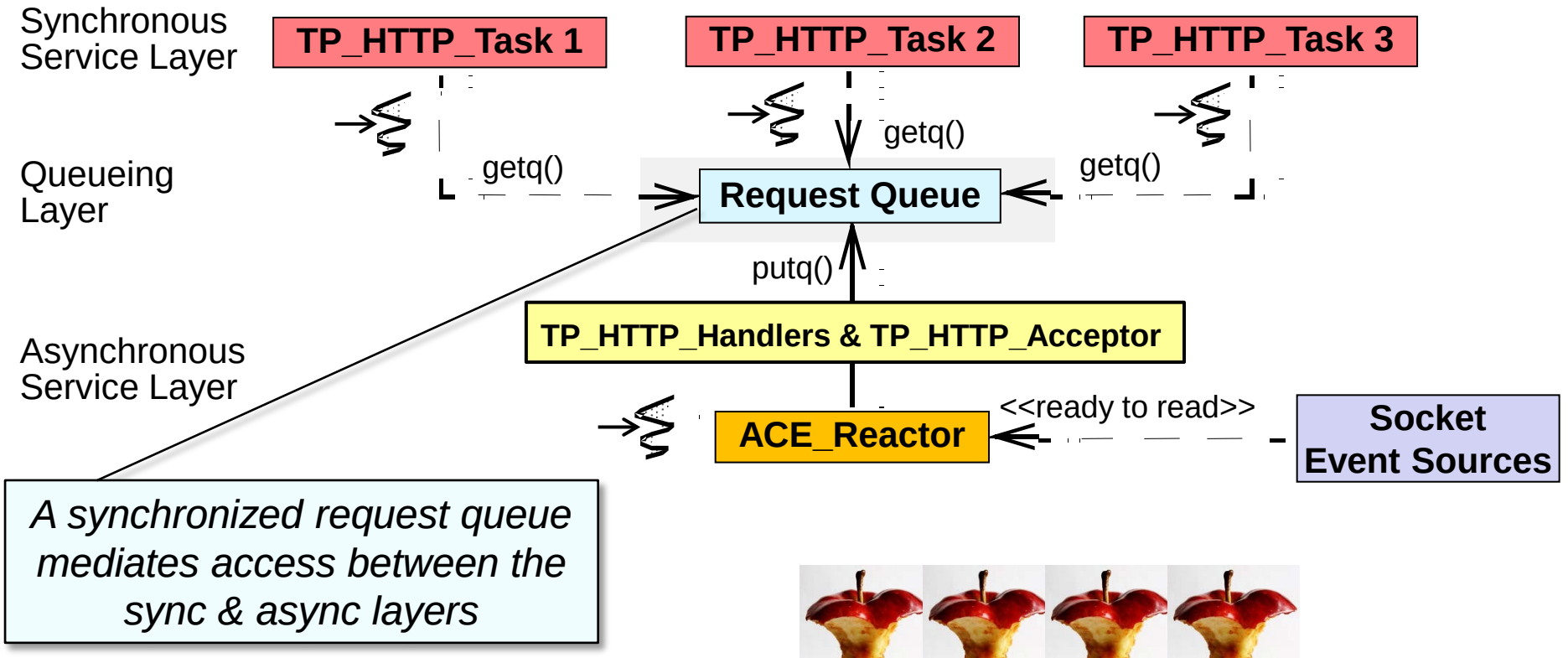


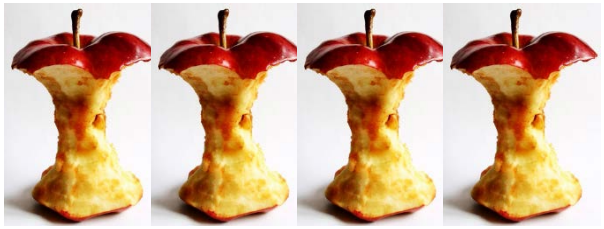
Active Object runs a pool of worker threads that remove requests, performs HTTP processing synchronously, & then transfers file back to client



Applying Half-Sync/Half-Async Pattern in JAWS

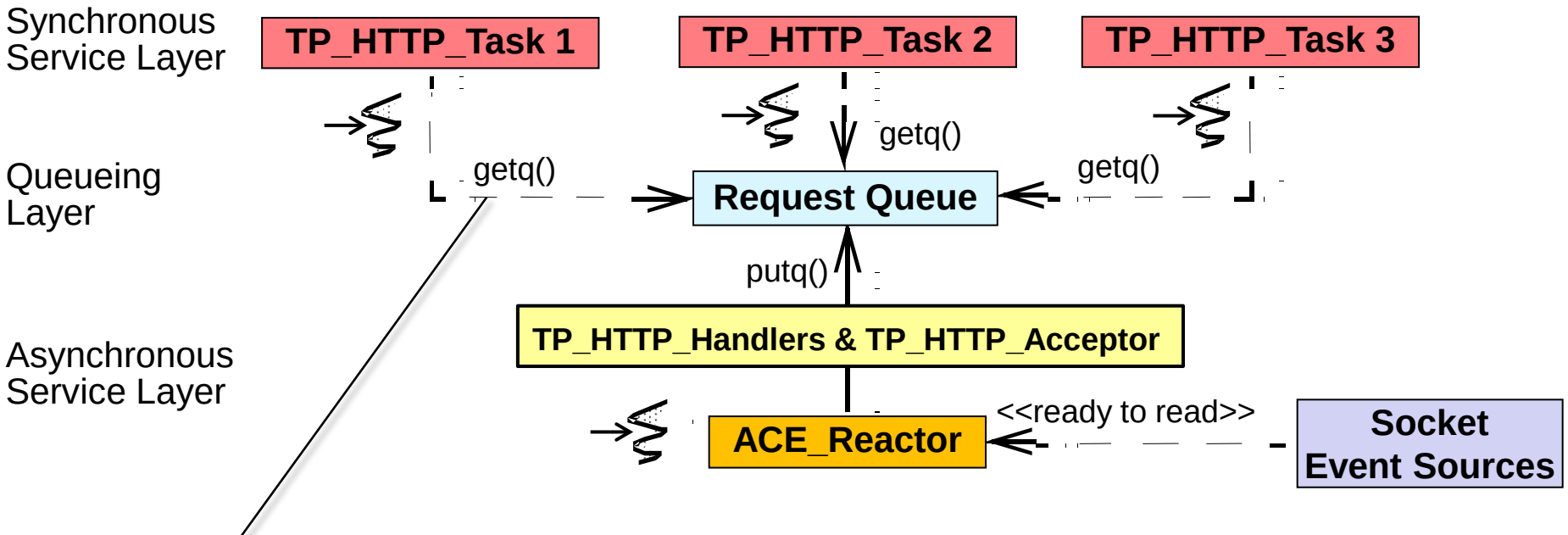
JAWS can implement the *Half-Sync/Half-Async* pattern by combining the *Reactor* pattern with the *Active Object* pattern



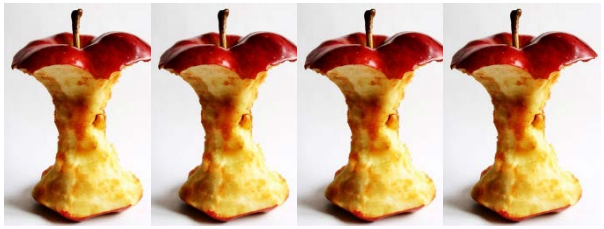


Applying Half-Sync/Half-Async Pattern in JAWS

JAWS can implement the *Half-Sync/Half-Async* pattern by combining the *Reactor* pattern with the *Active Object* pattern



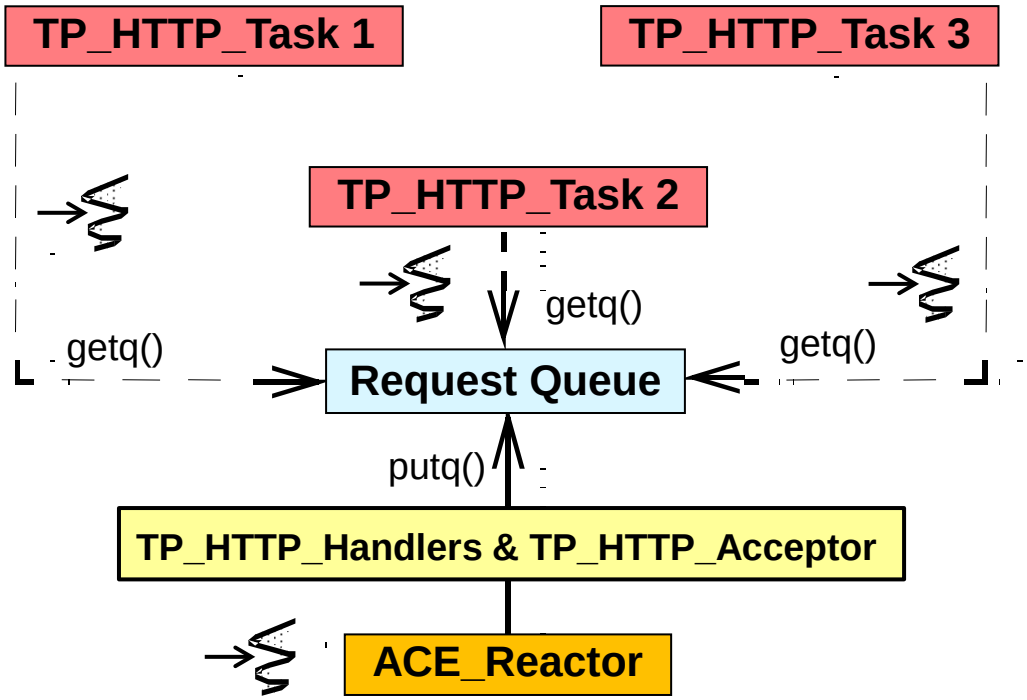
If flow control occurs on a client connection each thread can block without degrading the QoS of other clients in the thread pool



Benefits of the Half-Sync/Half-Async Pattern

Simplification & performance

- The programming of higher-level synchronous processing services are simplified without degrading the performance of lower-level system services



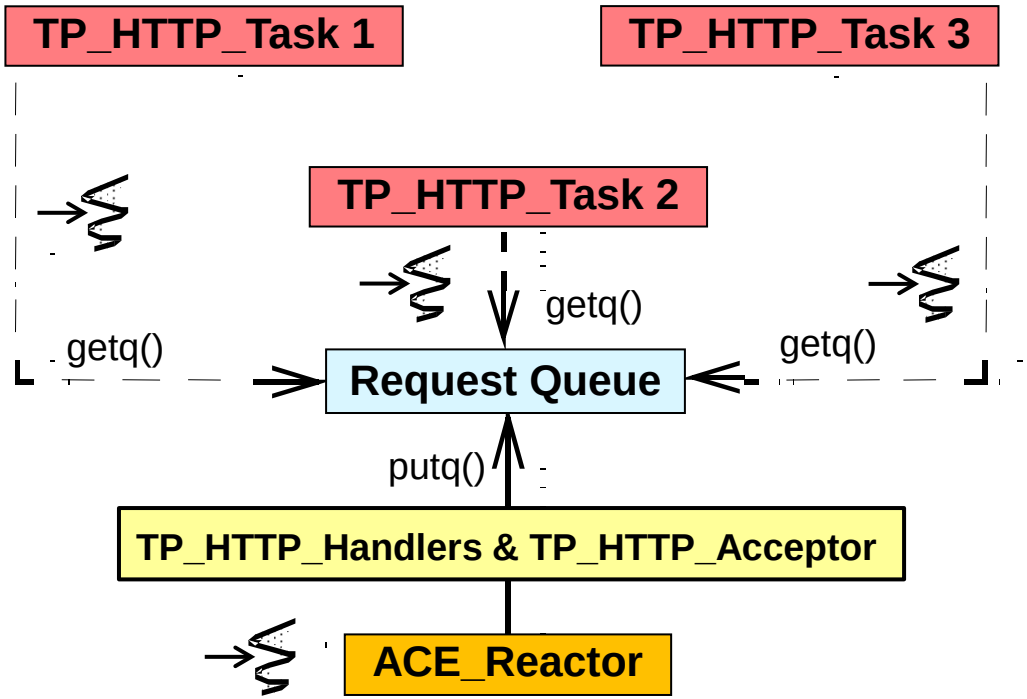
Benefits of the Half-Sync/Half-Async Pattern

Simplification & performance

- The programming of higher-level synchronous processing services are simplified without degrading the performance of lower-level system services

Separation of concerns

- Synchronization policies in each layer are decoupled so that each layer need not use the same concurrency strategies



Benefits of the Half-Sync/Half-Async Pattern

Simplification & performance

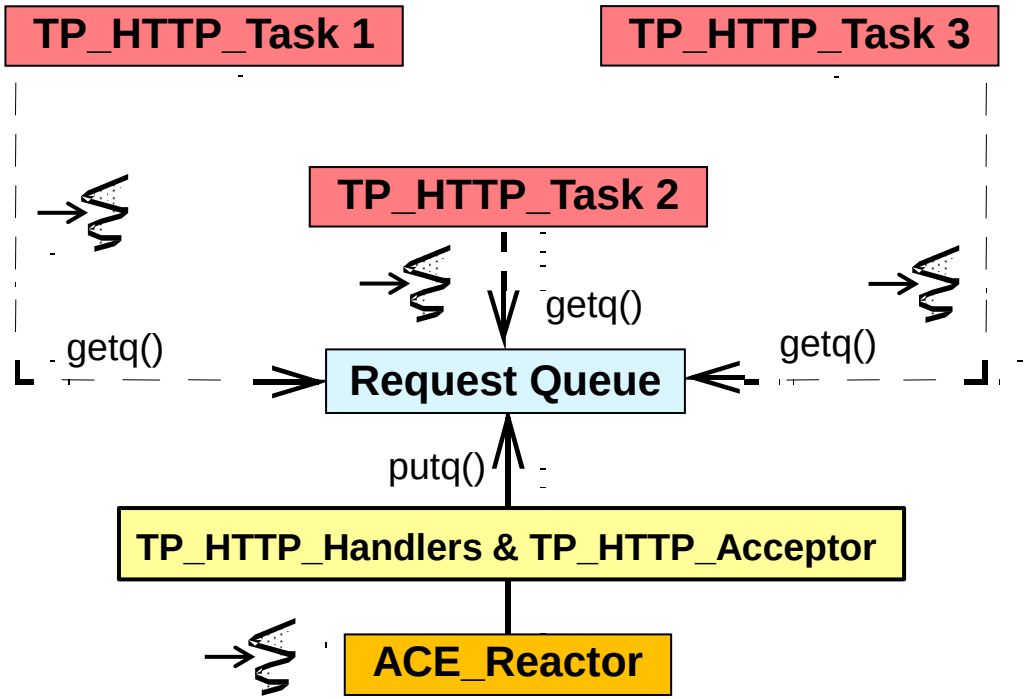
- The programming of higher-level synchronous processing services are simplified without degrading the performance of lower-level system services

Separation of concerns

- Synchronization policies in each layer are decoupled so that each layer need not use the same concurrency strategies

Centralization of inter-layer communication

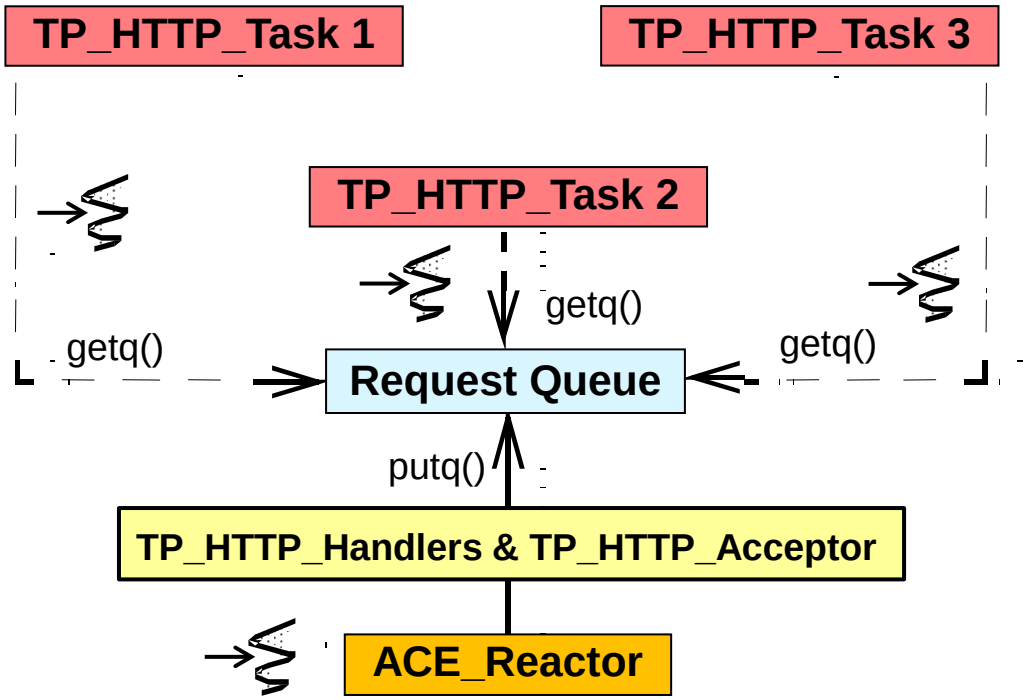
- Inter-layer communication is centralized at a single access point, because all interaction is mediated by the queueing layer



Limitations of the Half-Sync/Half-Async Pattern

May incur a boundary-crossing penalty

- Arising from context switching, synchronization, & data copying overhead when data transferred between sync & async service layers via queueing layer



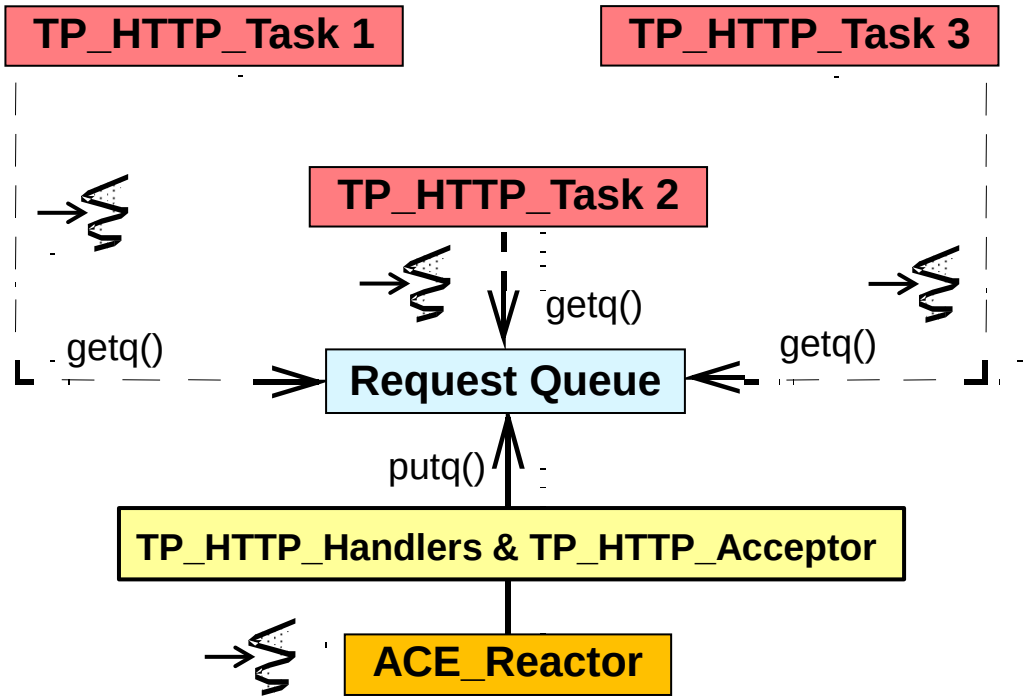
Limitations of the Half-Sync/Half-Async Pattern

May incur a boundary-crossing penalty

- Arising from context switching, synchronization, & data copying overhead when data transferred between sync & async service layers via queueing layer

Higher-level app services may not benefit from async I/O

- Depending on design of OS or application framework interfaces, higher-level services may not use low-level async I/O devices effectively



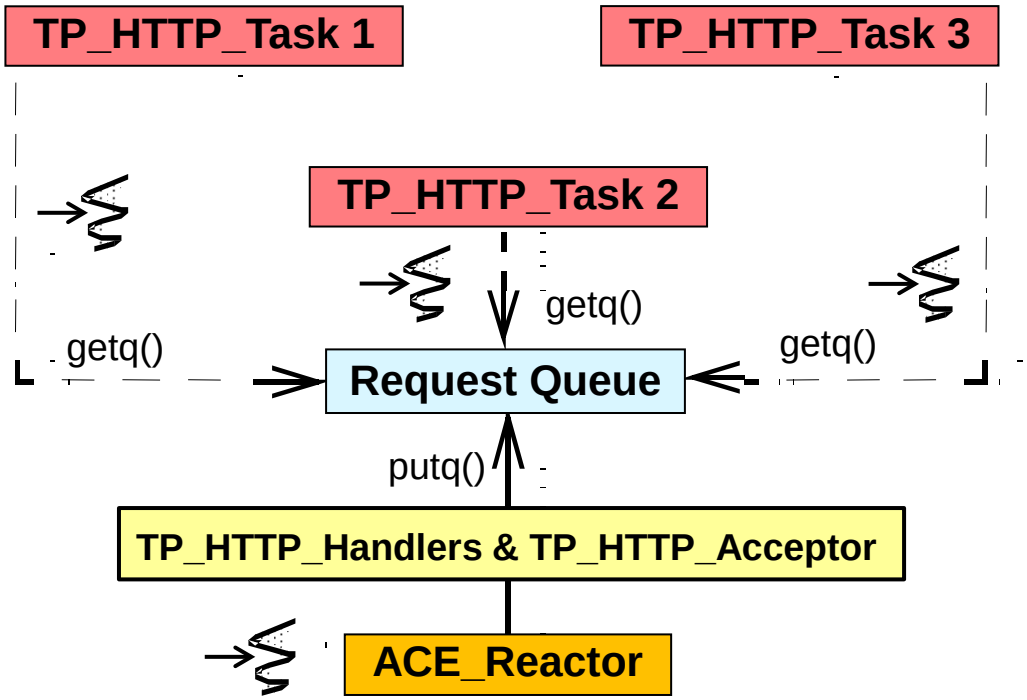
Limitations of the Half-Sync/Half-Async Pattern

May incur a boundary-crossing penalty

- Arising from context switching, synchronization, & data copying overhead when data transferred between sync & async service layers via queueing layer

Higher-level app services may not benefit from async I/O

- Depending on design of OS or application framework interfaces, higher-level services may not use low-level async I/O devices effectively

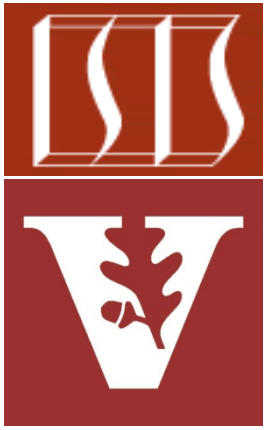


Complexity of debugging & testing

- Apps can be hard to debug due to concurrent execution

Patterns & Frameworks for Concurrency & Synchronization: Part 5

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



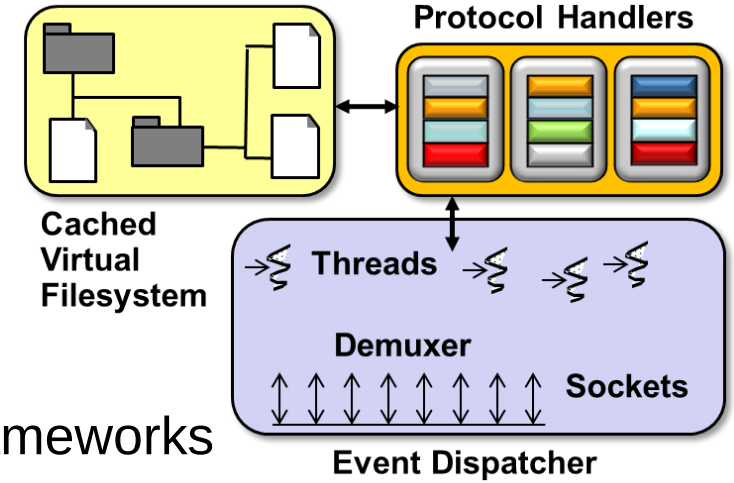
Professor of Computer Science

Institute for Software Integrated Systems

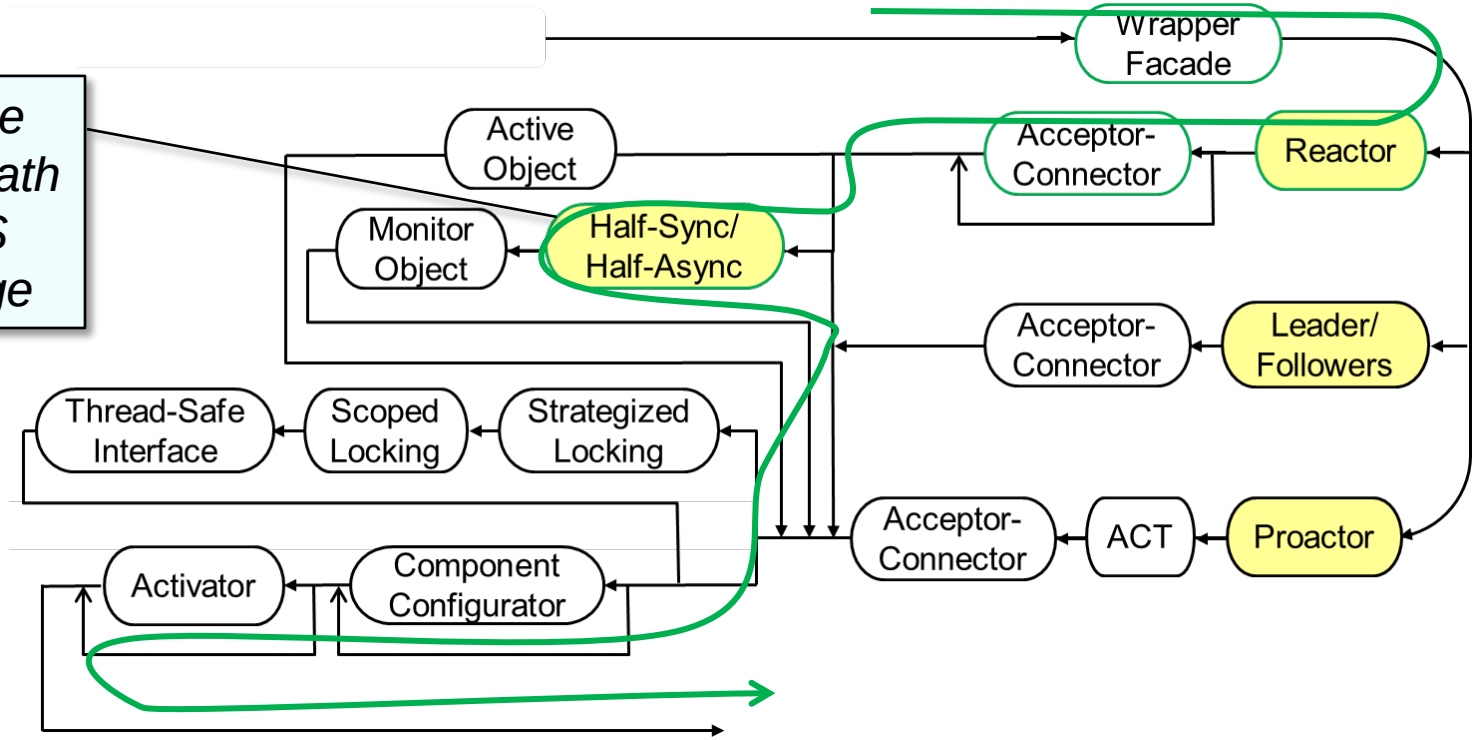
Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

- Describe the *Active Object* pattern
- Describe the ACE *Task* framework
- Apply ACE *Task* & *Acceptor-Connector* frameworks to JAWS
- Describe the *Half-Sync/Half-Async* pattern
- Implement *Half-Sync/Half-Async* using ACE frameworks

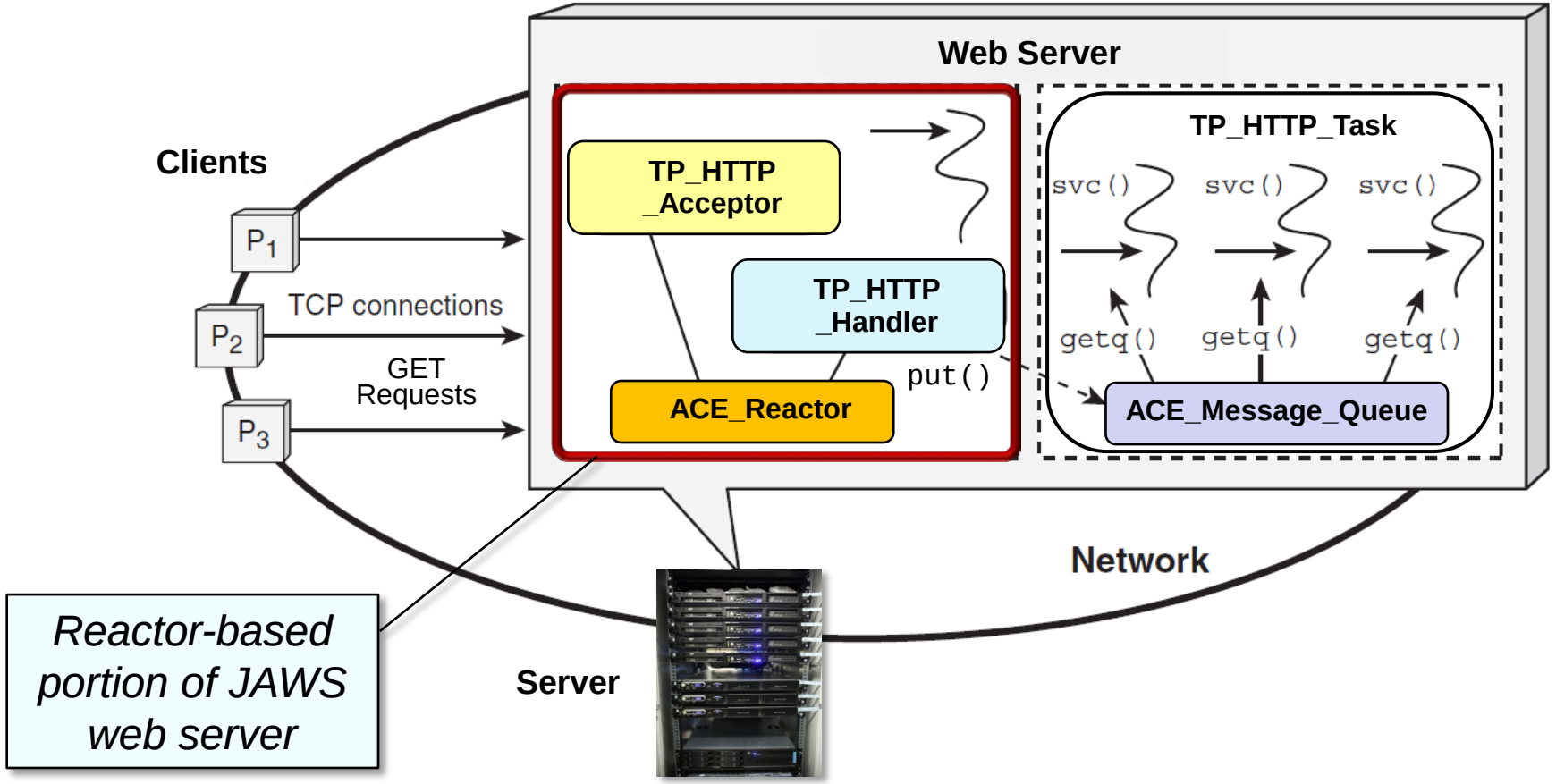


A more scalable multi-threaded path thru the JAWS pattern language



Half-Async with ACE_Svc_Handler Class

Use **`ACE_Svc_Handler`** to implement "half-async" portion of JAWS web server based on the *Half-Sync/Half-Async* pattern



This implementation is the most complex we've seen thus far

Half-Async w/ACE_Svc_Handler Class

Be an ACE_Svc_Handler

```
class TP_HTTP_Handler
  : public ACE_Svc_Handler<ACE_SOCK_Stream, ACE_NULL_SYNCH> {
private:
  TP_HTTP_Task &tp_http_task_;

public:
  TP_HTTP_Handler (TP_HTTP_Task &task): tp_http_task_ (task) {}
```

Reference to active object

Activation hook method called by ACE_Acceptor for each connection

```
  virtual int open (void *) {
    reactor()->register_handler (this,
                                 ACE_Event_Handler::READ_MASK);

    reactor ()->schedule_timer
                  (this, 0, ACE_Time_Value (CLIENT_TIMEOUT));
  }
```

Register with reactor to handle input events

Register timeout, in case a client sends no HTTP request(s)

Half-Async w/ACE_Svc_Handler Class

Method called back by Reactor when clients timeout to remove the handler & conserve resources

```
virtual int handle_timeout(const ACE_Time_Value&,const void*){
  reactor ()->remove_handler (this,
                              ACE_Event_Handler::READ_MASK);
}
```

Reactor dispatches this method when HTTP requests arrive

```
virtual int handle_input (ACE_HANDLE) {
  ACE_Message_Block *msg = 0;
```

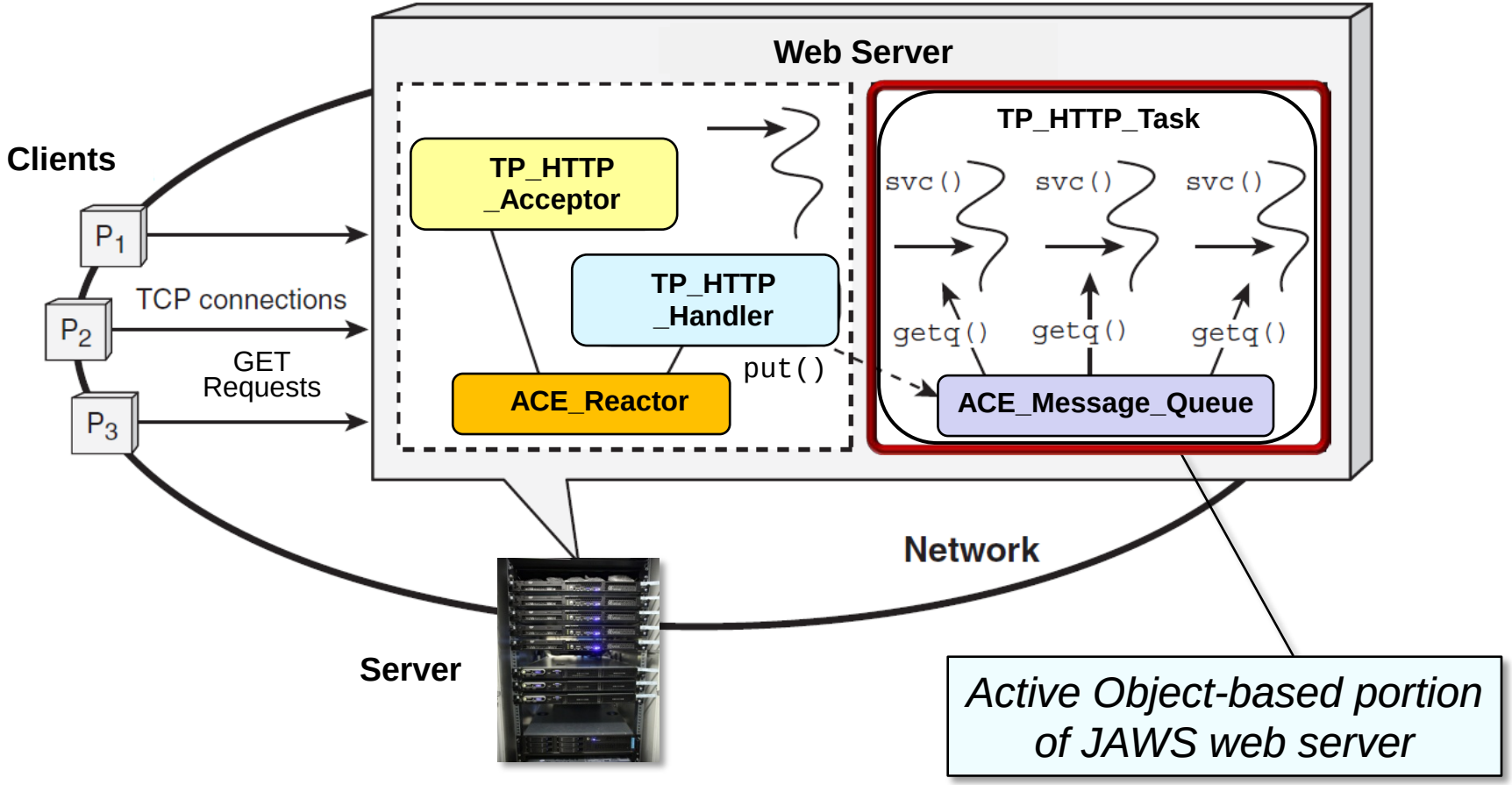
Try to receive complete message

```
  if (recv_request (msg) == HTTP_REQUEST_COMPLETE) {
    reactor ()->remove_handler (this,
                                ACE_Event_Handler::READ_MASK);
    reactor ()->cancel_timer (this);
    tp_http_task_.put (msg, timeout_);
  }
  // ...
}
```

Cleanup resources & pass message to active object for subsequent processing

Half-Sync with ACE_Task Class

Use **`ACE_Task`** to implement "half-sync" portion of JAWS web server based on the *Half-Sync/Half-Async* pattern



Half-Sync with ACE_Task Class

Be an ACE_Task with multi-threaded synchronization trait

```
class TP_HTTP_Task : public ACE_Task<ACE_MT_SYNCH> {
public:
  enum { THREAD_POOL_SIZE = 8 };

  TP_HTTP_Task () {
    activate (THR_BOUND, THREAD_POOL_SIZE);
  }

  virtual int put (ACE_Message_Block *message,
                   ACE_Time_Value *timeout) {
    return putq (message, timeout);
  }

  ... // Continued on next slide
```

Become an active object

Hook method called by half-async layer to pass GET request to half-sync layer

Enqueue message for subsequent processing

We'll explain how the synchronized message queue works in the next part

Half-Sync with ACE_Task Class

This hook method runs in separate thread(s) via ACE *Task* framework

```
int svc () {

  for (ACE_Message_Block *msg; getq (msg) != -1; ) {

    std::string pathname (get_pathname (msg));
    ACE_Mem_Map mapped_file (pathname.c_str ());

    switch (get_stream(msg).send_n (mapped_file.addr (),
                                    mapped_file.size ()) {
    ...
    }
    ...
}
```

This loop blocks until new message arrives from half-async layer

Map file & send the requested web content back to the client

Note how the core web server logic is the same as always

Half-Sync/Half-Async with ACE_Acceptor Class

Be an ACE_Acceptor

```
class TP_HTTP_Acceptor : public ACE_Acceptor<TP_HTTP_Handler,
                                             ACE_SOCK_Acceptor>
{
public:
   TP_HTTP_Acceptor (ACE_SOCK_Acceptor::PEER_ADDR addr):
     ACE_Acceptor<TP_HTTP_Handler, ACE_SOCK_Acceptor> (addr) {}
```

Constructor forwards to ACE_Acceptor

Hook method called by handle_input() to create an TP_HTTP_Handler that receives GET requests from clients

```
   virtual int make_svc_handler (TP_HTTP_Handler &*handler) {
     handler = new TP_HTTP_Handler (tp_http_task_);
     return 0;
   }
private:
    TP_HTTP_Task tp_http_task_;
};
```

Active object (passed by reference)

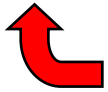
Driver for Half-Sync/Half-Async Web Server

```
1 typedef Reactor_HTTP_Server<TP_HTTP_Acceptor>
2         HTTP_Server_Daemon;
```

Reuse our earlier template driver & Instantiate it with the TP_HTTP_Acceptor

```
3 int main (int argc, char *argv[]) {
4   ACE_Reactor reactor;
5   new HTTP_Server_Daemon (argc, argv, &reactor);
```

Event loop controller object



Dynamic allocation ensures proper deletion semantics

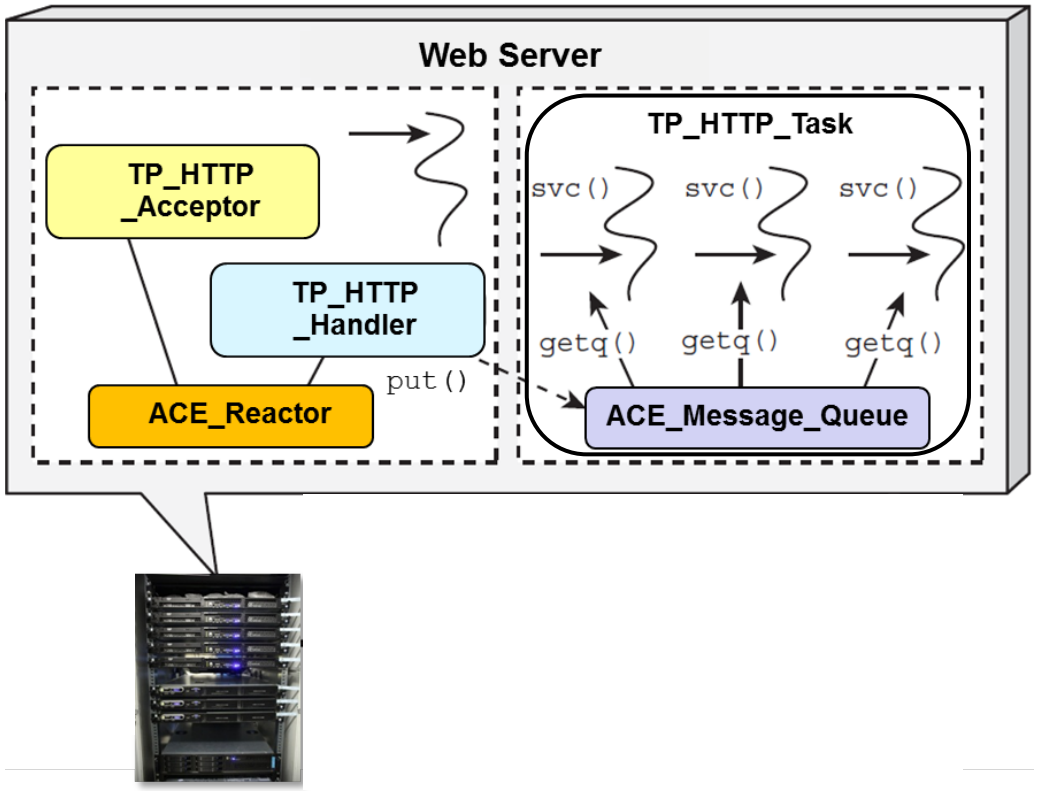
```
5   reactor.run_reactor_event_loop ();
    ...
```

Run server event loop

This main program is nearly identical to the thread-per-connection web server

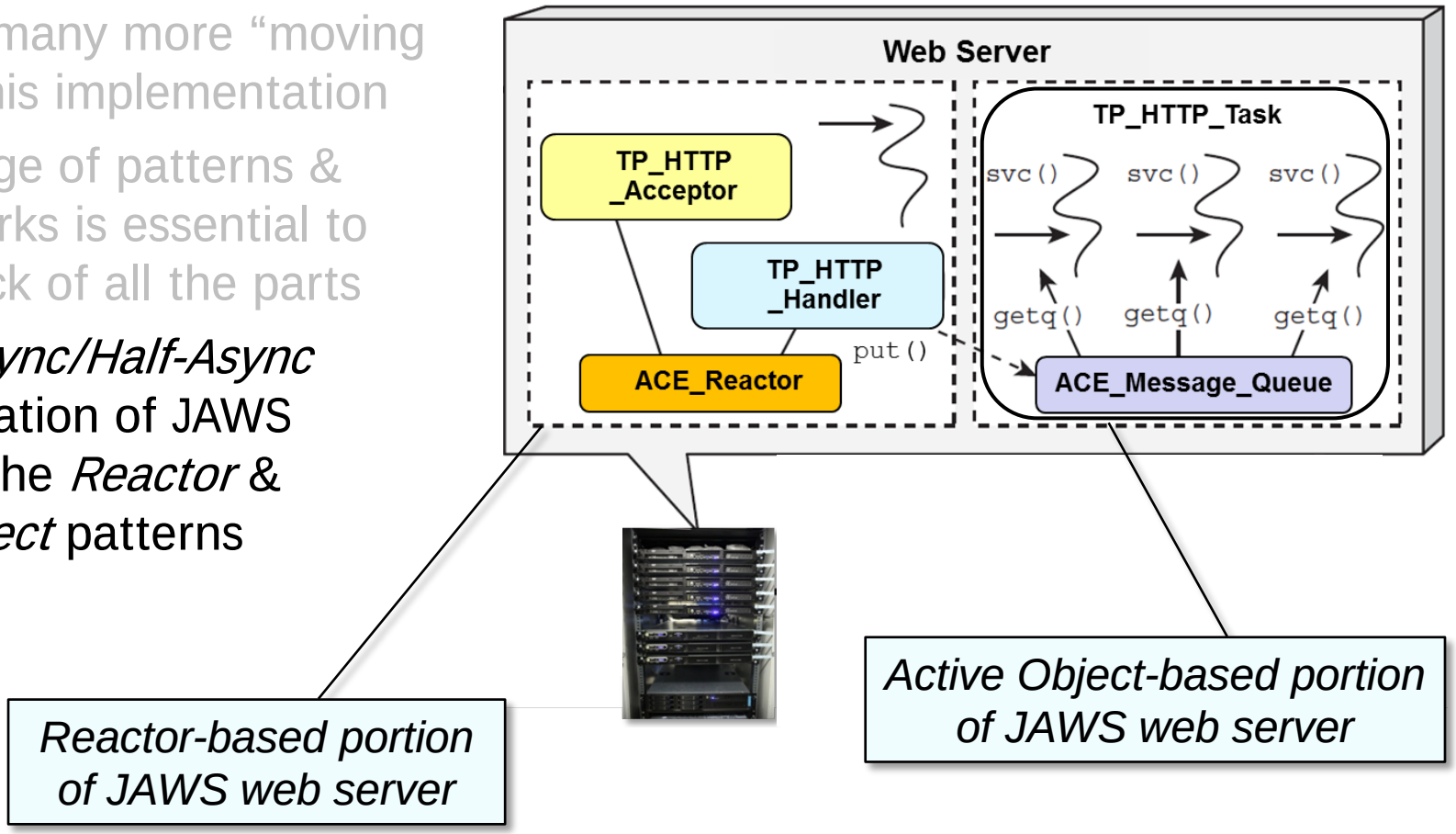
Summary

- There are many more “moving parts” in this implementation
  - Knowledge of patterns & frameworks is essential to keep track of all the parts



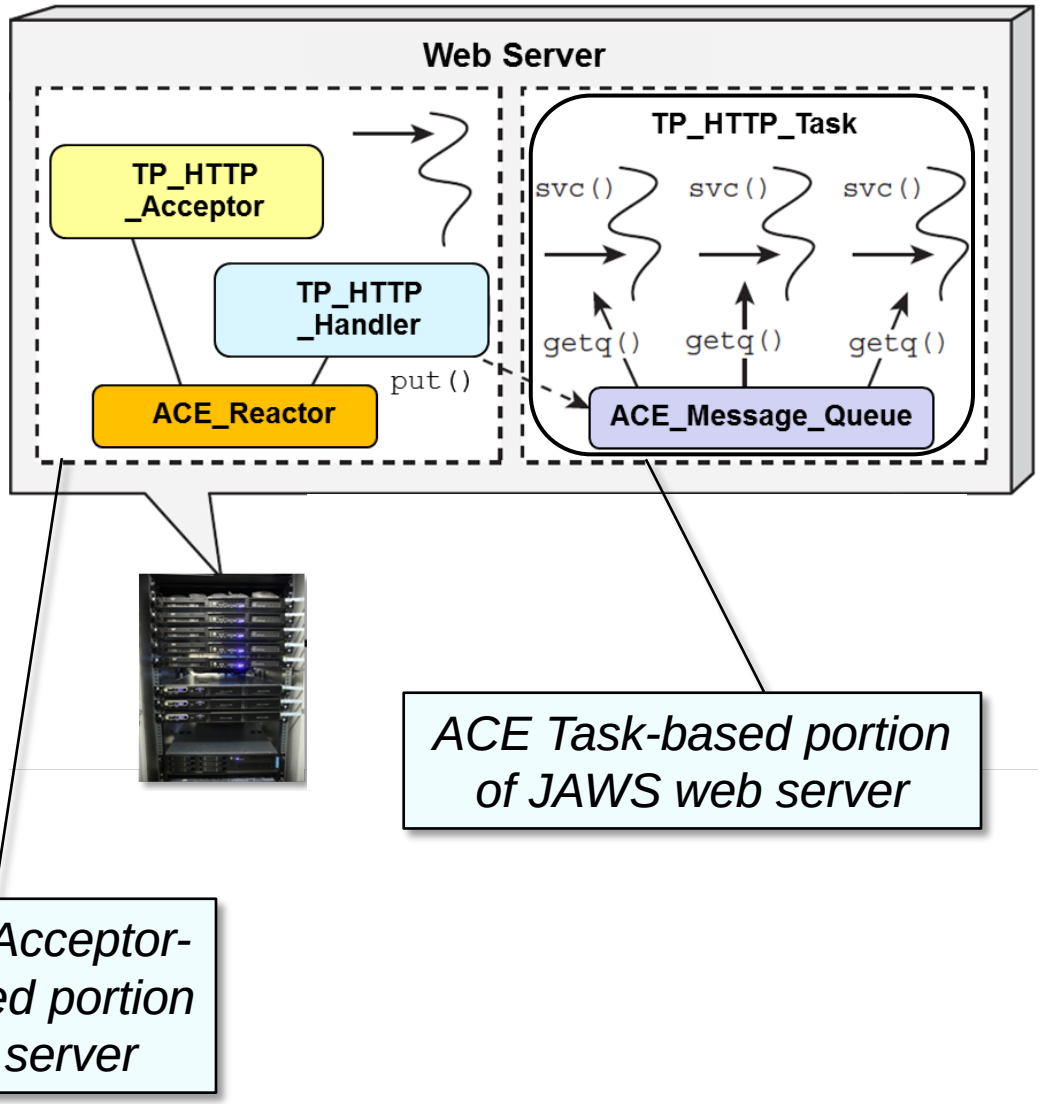
Summary

- There are many more "moving parts" in this implementation
  - Knowledge of patterns & frameworks is essential to keep track of all the parts
- The *Half-Sync/Half-Async* implementation of JAWS combines the *Reactor* & *Active Object* patterns



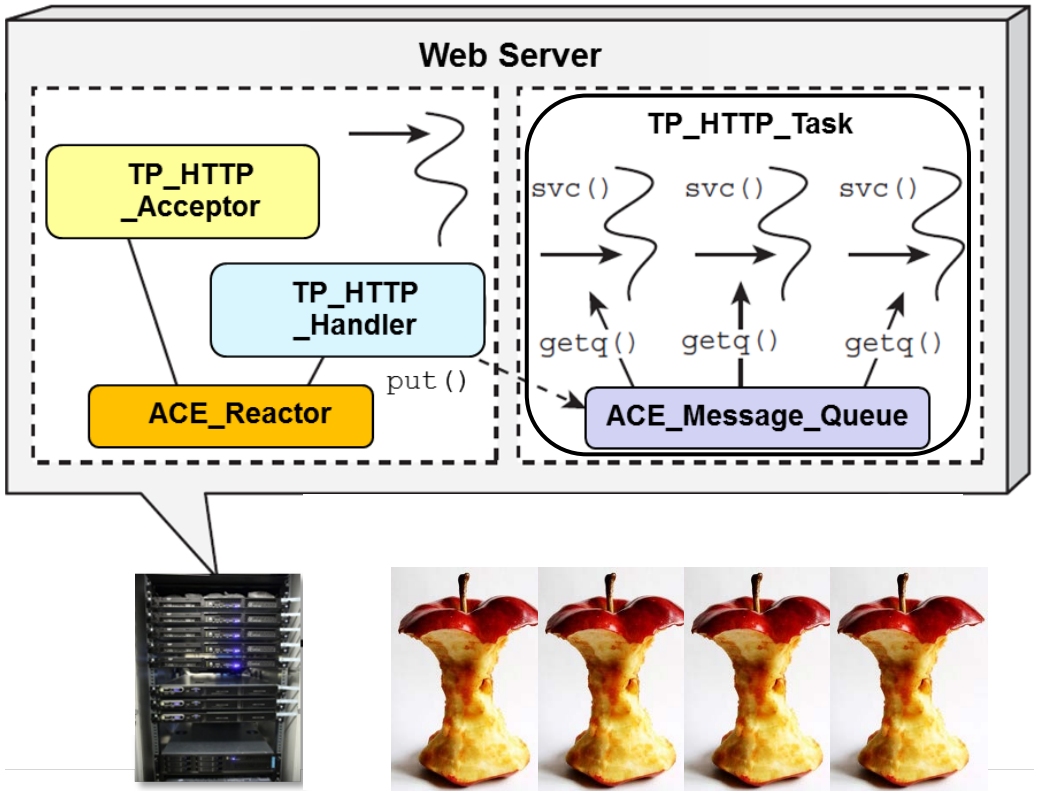
Summary

- There are many more "moving parts" in this implementation
  - Knowledge of patterns & frameworks is essential to keep track of all the parts
- The *Half-Sync/Half-Async* implementation of JAWS combines the *Reactor* & *Active Object* patterns
- Likewise, it combines the ACE *Reactor*, *Acceptor-Connector*, & *Task* frameworks



Summary

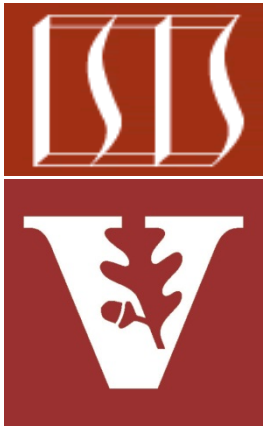
- There are many more "moving parts" in this implementation
  - Knowledge of patterns & frameworks is essential to keep track of all the parts
- The *Half-Sync/Half-Async* implementation of JAWS combines the *Reactor* & *Active Object* patterns
- Likewise, it combines the ACE *Reactor*, *Acceptor-Connector*, & *Task* frameworks



- This pattern-oriented software architecture may scale better to multi-core & multi-CPU platforms
  - e.g., by bounding the number of threads to match the available parallel processing resources more effectively

Patterns & Frameworks for Concurrency & Synchronization: Part 6

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



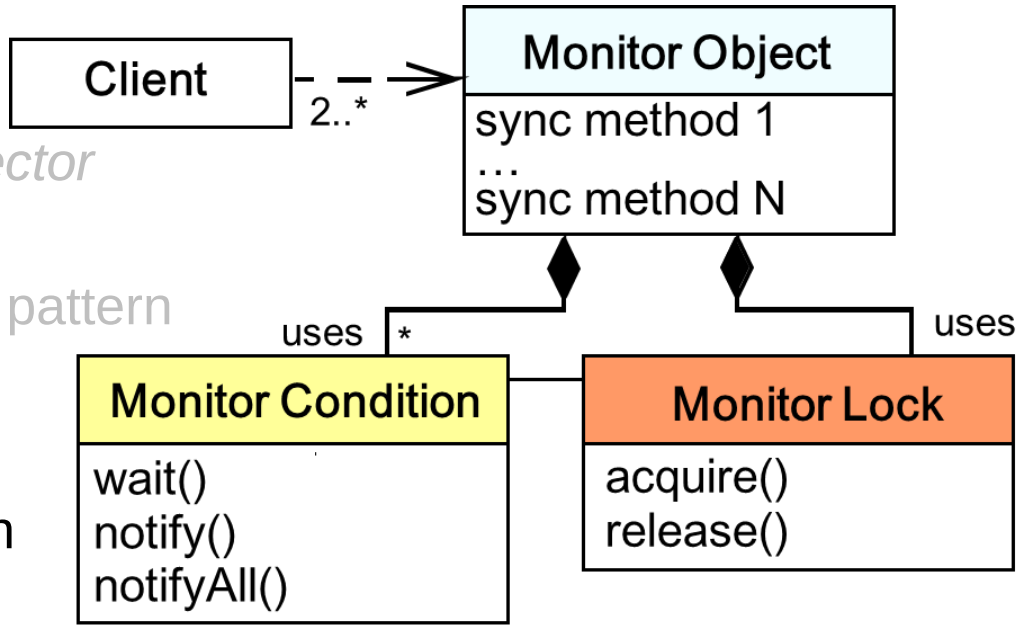
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

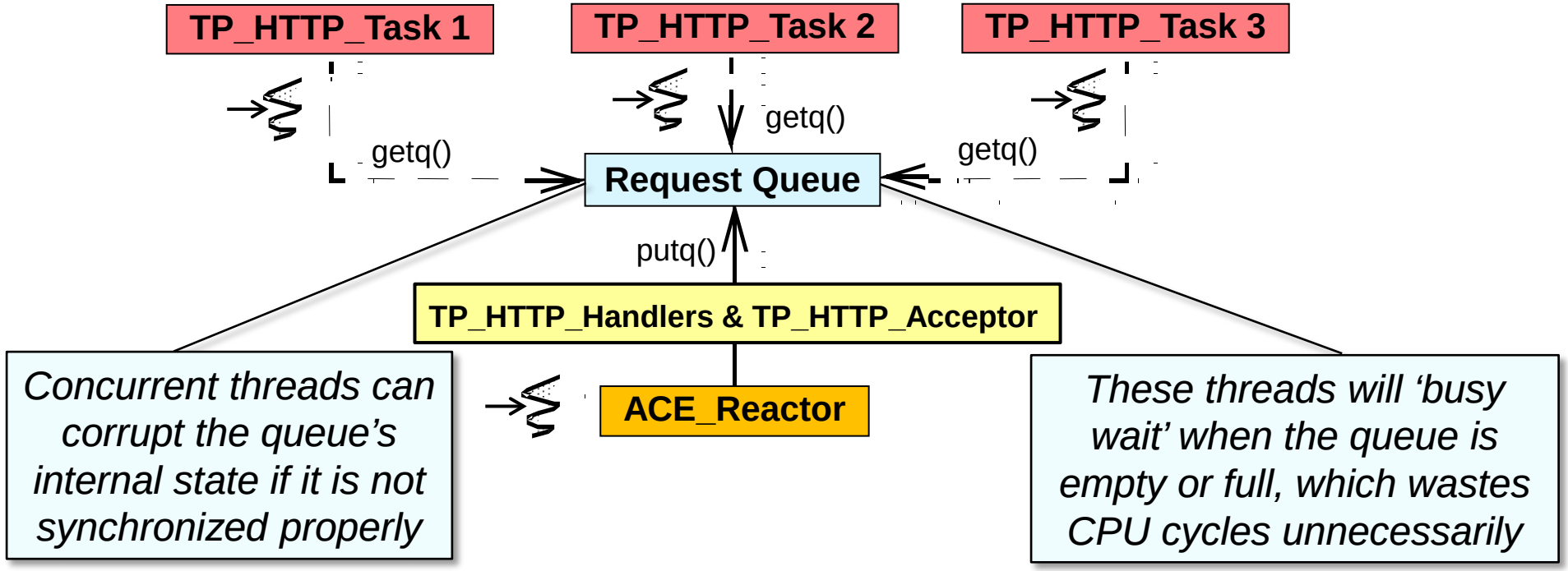
Topics Covered in this Part of the Module

- Describe the *Active Object* pattern
- Describe the ACE *Task* framework
- Apply ACE *Task* & *Acceptor-Connector* frameworks to JAWS
- Describe the *Half-Sync/Half-Async* pattern
- Implement *Half-Sync/Half-Async* using ACE frameworks
- Describe the *Monitor Object* pattern



Implementing a Synchronized Request Queue

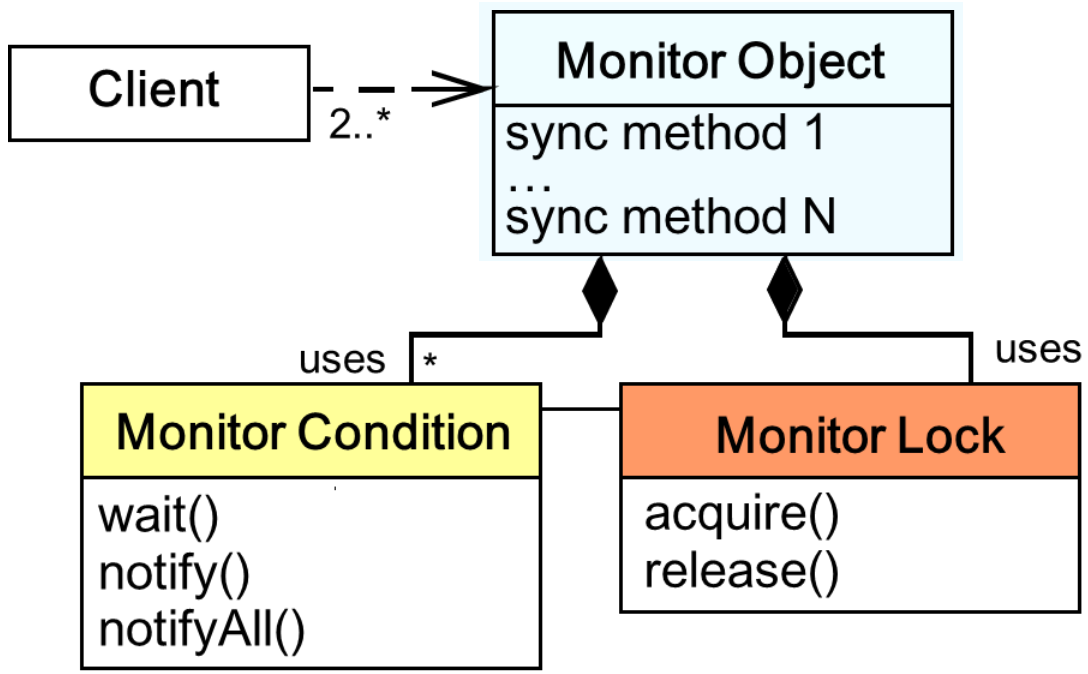
Context	Problem
• JAWS's **ACE_Task** for *Half-Sync/Half-Async* contains a request queue	• Naïve request queues incurs race conditions or "busy waiting" when multiple threads put/get requests



Implementing a Synchronized Request Queue

Context	Problem	Solution
• JAWS's **ACE_Task** for *Half-Sync/Half-Async* contains a request queue	• Naïve request queues incurs race conditions or "busy waiting" when multiple threads put/get requests	• Apply the ***Monitor Object*** pattern to synchronize the request queue efficiently & conveniently

Structure

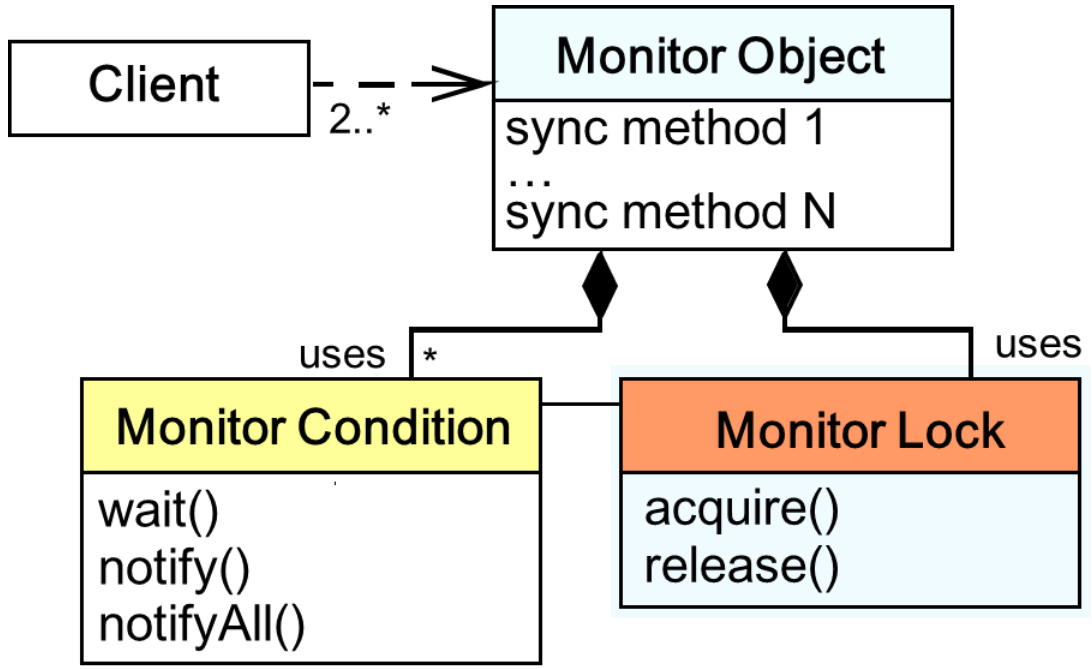


Monitor Object synchronizes concurrent method execution to ensure only one method at a time runs within an object & allows an object's methods to cooperatively schedule their execution sequences

Implementing a Synchronized Request Queue

Context	Problem	Solution
• JAWS's **ACE_Task** for *Half-Sync/Half-Async* contains a request queue	• Naïve request queues incurs race conditions or "busy waiting" when multiple threads put/get requests	• Apply the ***Monitor Object*** pattern to synchronize the request queue efficiently & conveniently

Structure

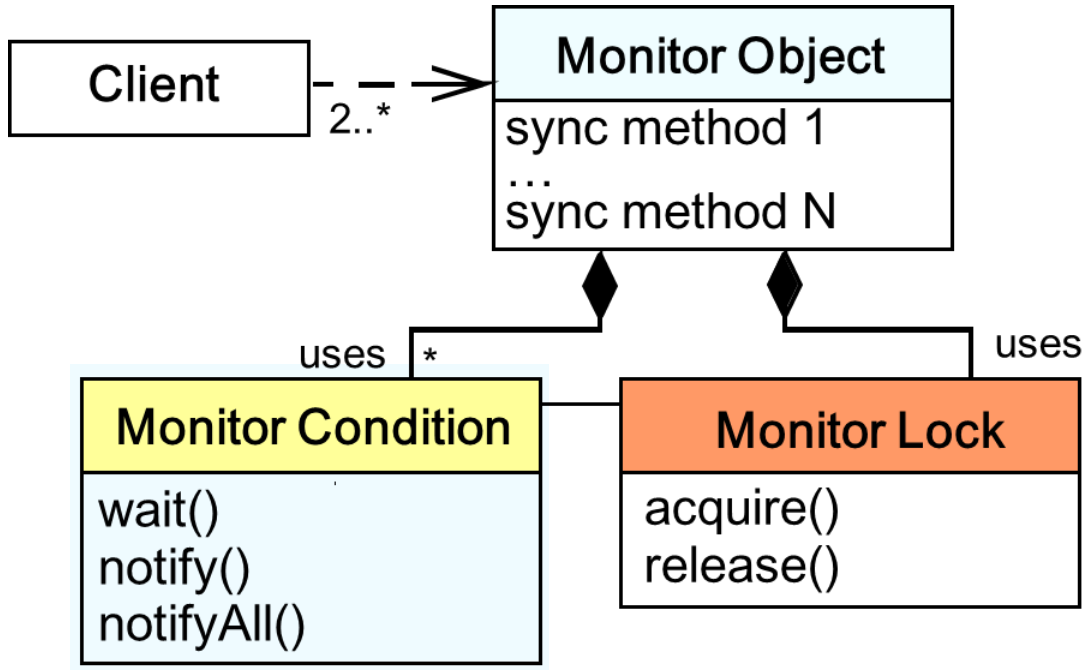


Monitor Object synchronizes concurrent method execution to ensure only one method at a time runs within an object & allows an object's methods to cooperatively schedule their execution sequences

Implementing a Synchronized Request Queue

Context	Problem	Solution
• JAWS's **ACE_Task** for *Half-Sync/Half-Async* contains a request queue	• Naïve request queues incurs race conditions or "busy waiting" when multiple threads put/get requests	• Apply the ***Monitor Object*** pattern to synchronize the request queue efficiently & conveniently

Structure

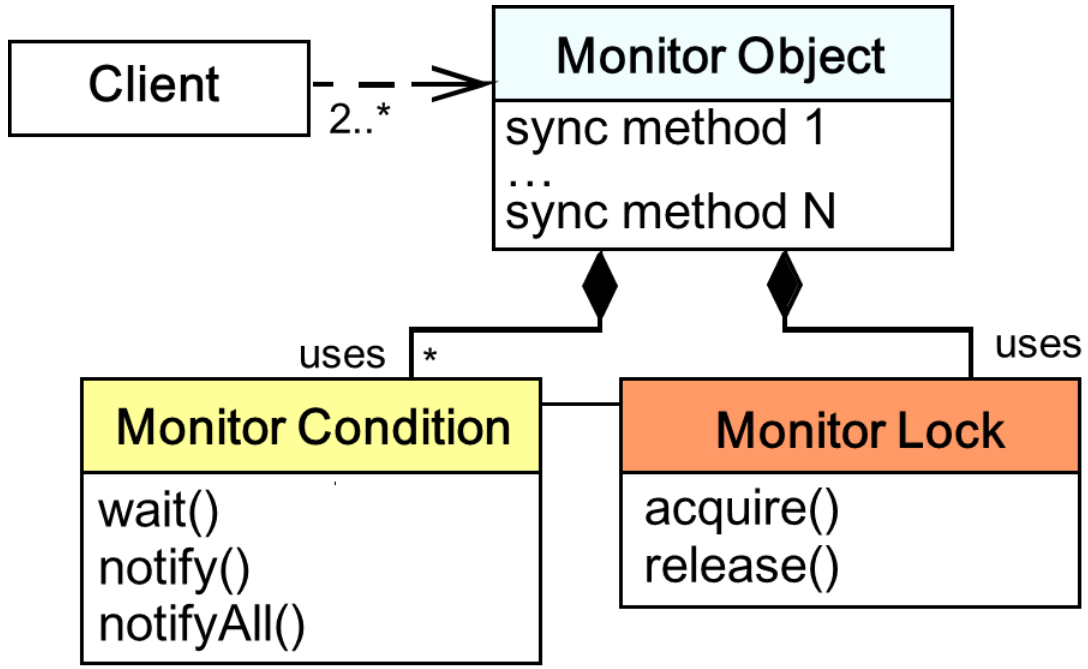


Monitor Object synchronizes concurrent method execution to ensure only one method at a time runs within an object & allows an object's methods to cooperatively schedule their execution sequences

Implementing a Synchronized Request Queue

Context	Problem	Solution
• JAWS's **ACE_Task** for *Half-Sync/Half-Async* contains a request queue	• Naïve request queues incurs race conditions or "busy waiting" when multiple threads put/get requests	• Apply the ***Monitor Object*** pattern to synchronize the request queue efficiently & conveniently

Structure



Monitor Object synchronizes concurrent method execution to ensure only one method at a time runs within an object & allows an object's methods to cooperatively schedule their execution sequences

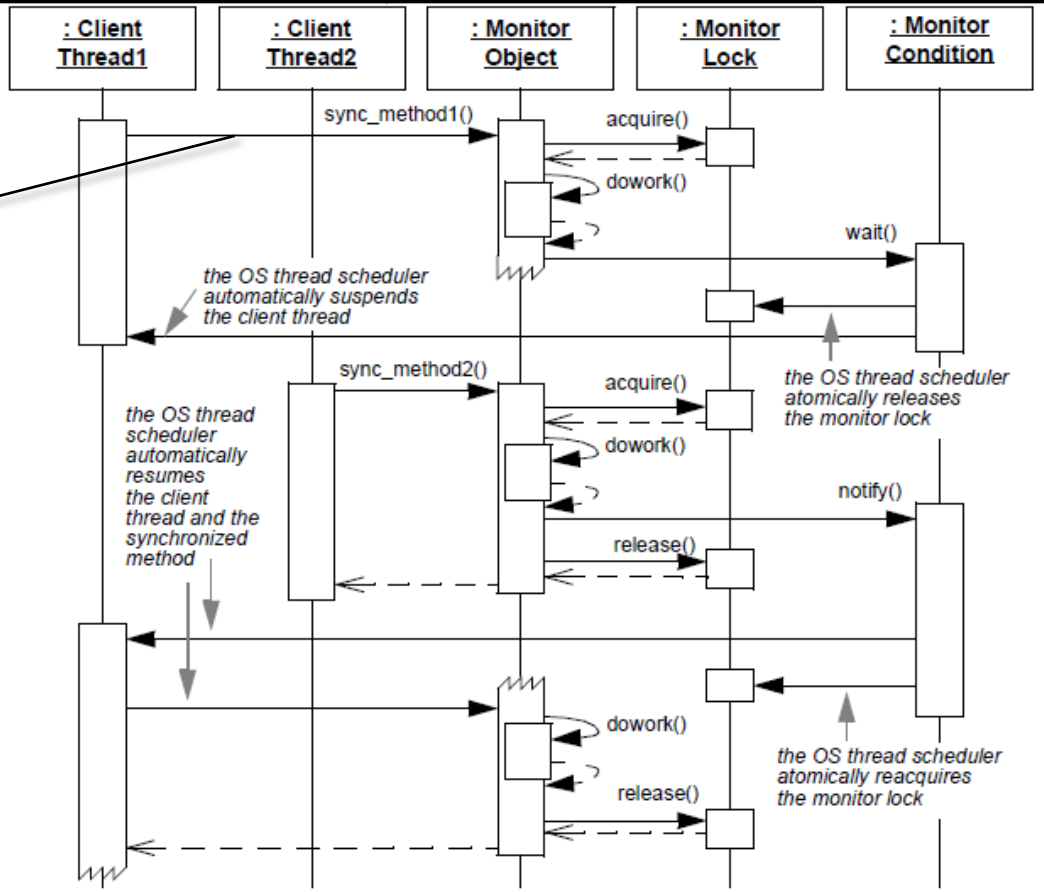
See www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf for more info

Implementing a Synchronized Request Queue

Context	Problem	Solution
• JAWS's **ACE_Task** for *Half-Sync/Half-Async* contains a request queue	• Naïve request queues incurs race conditions or "busy waiting" when multiple threads put/get requests	• Apply the ***Monitor Object*** pattern to synchronize the request queue efficiently & conveniently

Dynamics

Synchronized method invocation & serialization

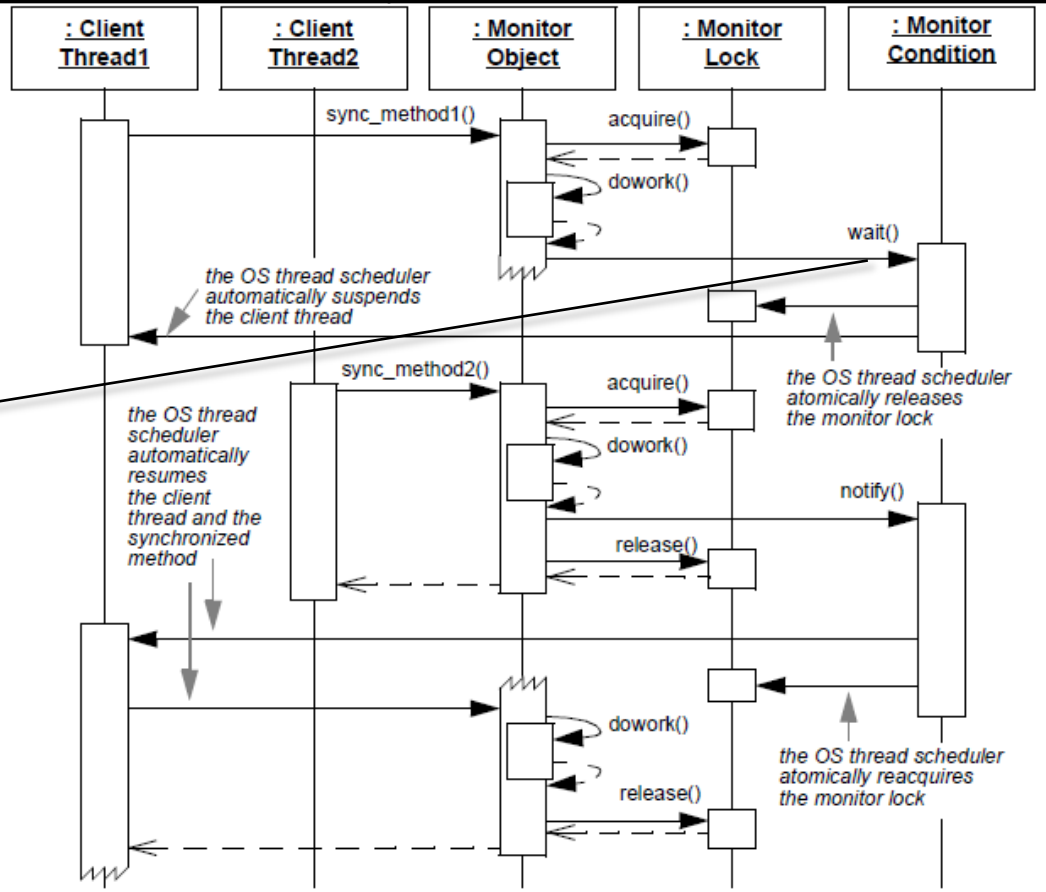


Implementing a Synchronized Request Queue

Context	Problem	Solution
• JAWS's **ACE_Task** for *Half-Sync/Half-Async* contains a request queue	• Naïve request queues incurs race conditions or "busy waiting" when multiple threads put/get requests	• Apply the ***Monitor Object*** pattern to synchronize the request queue efficiently & conveniently

Dynamics

Synchronized method thread suspension

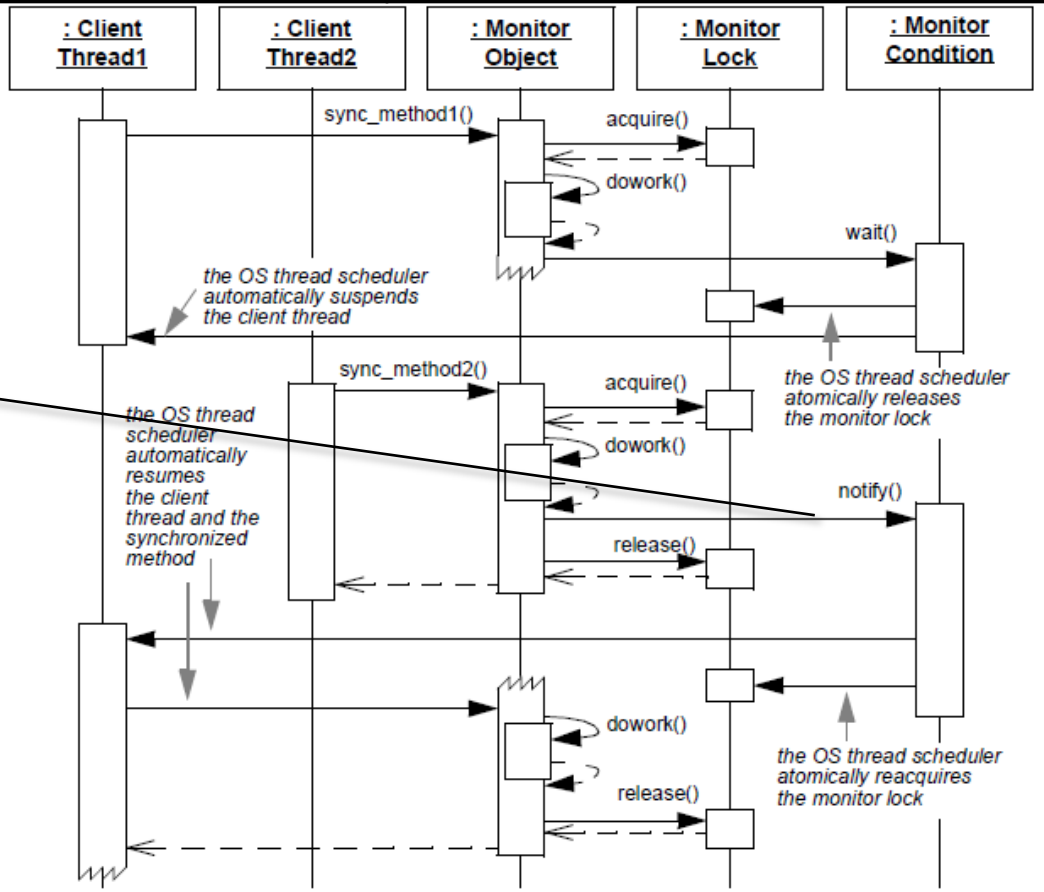


Implementing a Synchronized Request Queue

Context	Problem	Solution
• JAWS's **ACE_Task** for *Half-Sync/Half-Async* contains a request queue	• Naïve request queues incurs race conditions or "busy waiting" when multiple threads put/get requests	• Apply the ***Monitor Object*** pattern to synchronize the request queue efficiently & conveniently

Dynamics

Monitor condition notification

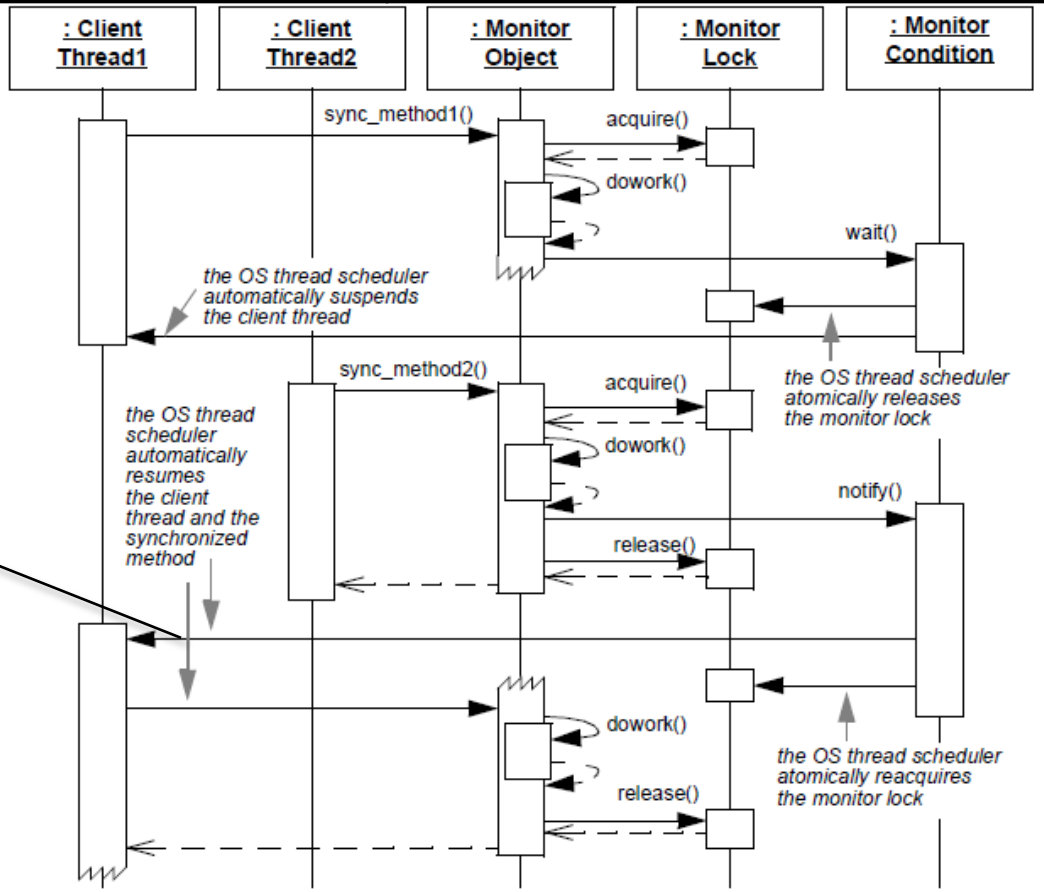


Implementing a Synchronized Request Queue

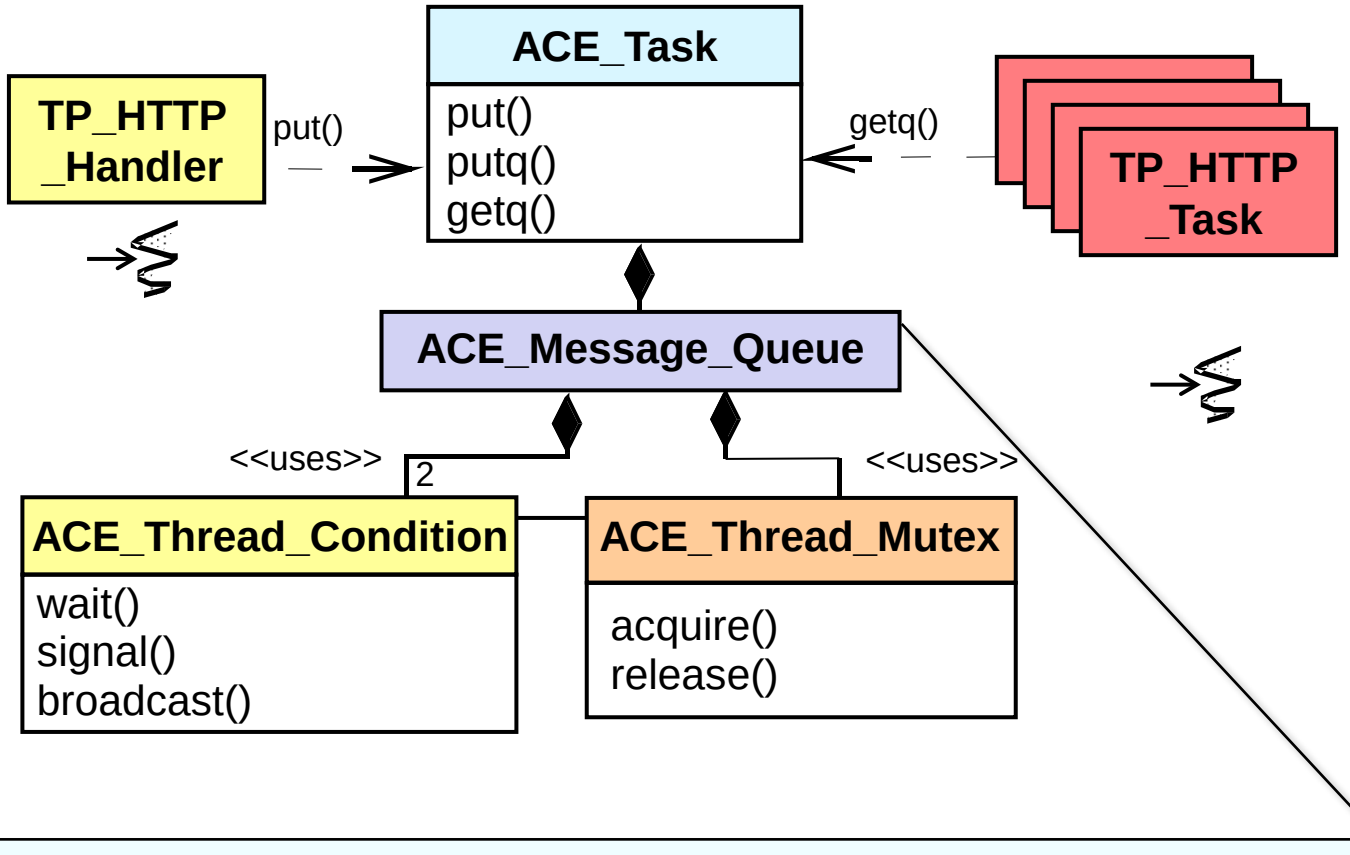
Context	Problem	Solution
• JAWS's **ACE_Task** for *Half-Sync/Half-Async* contains a request queue	• Naïve request queues incurs race conditions or "busy waiting" when multiple threads put/get requests	• Apply the ***Monitor Object*** pattern to synchronize the request queue efficiently & conveniently

Dynamics

Synchronized method thread resumption

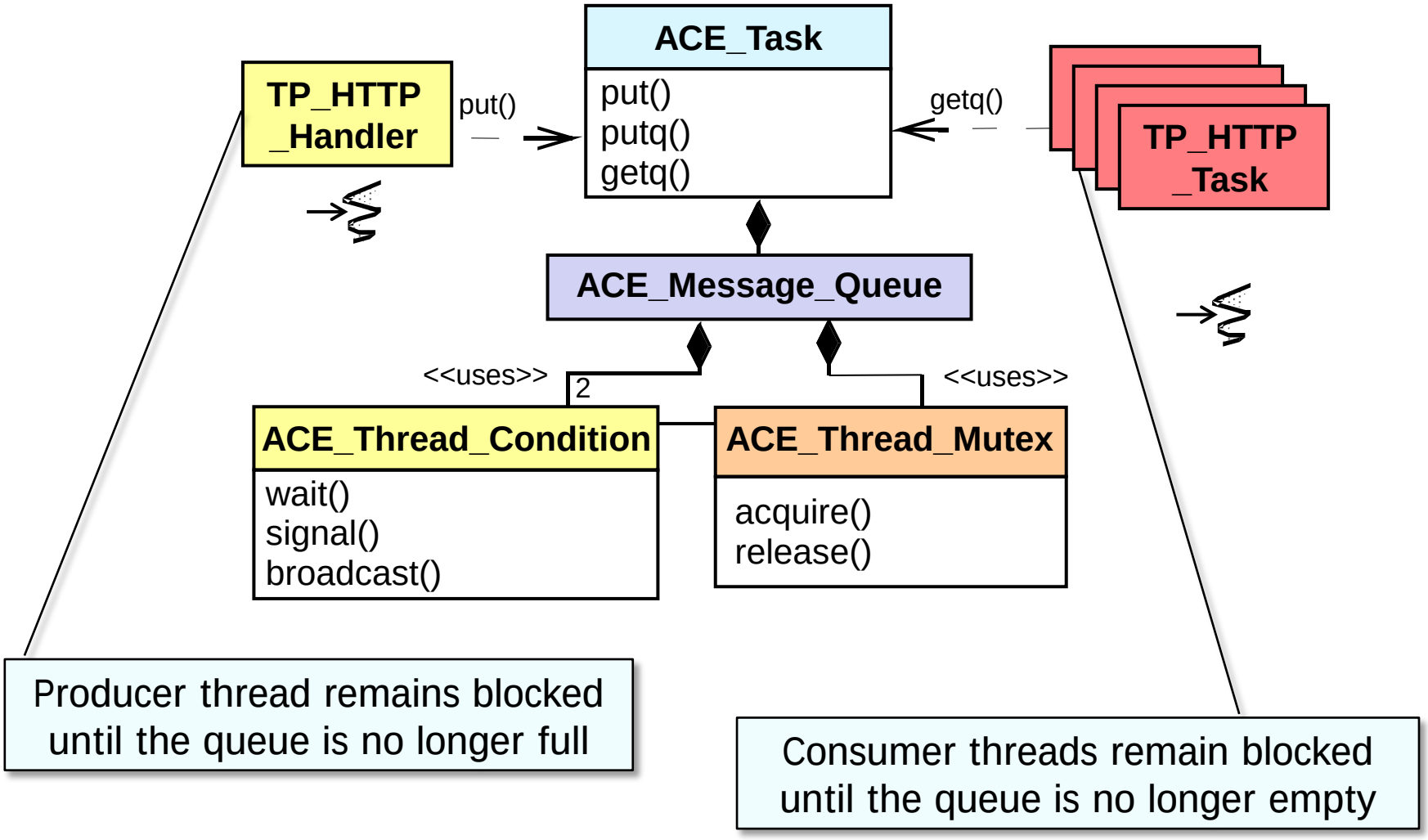


Applying Monitor Object Pattern in JAWS



Uses ***ACE_Message_Queue*** *& implements the queue's* ***not_empty_*** *&* ***not_full_*** *monitor conditions via ACE wrapper facades*

Applying Monitor Object Pattern in JAWS

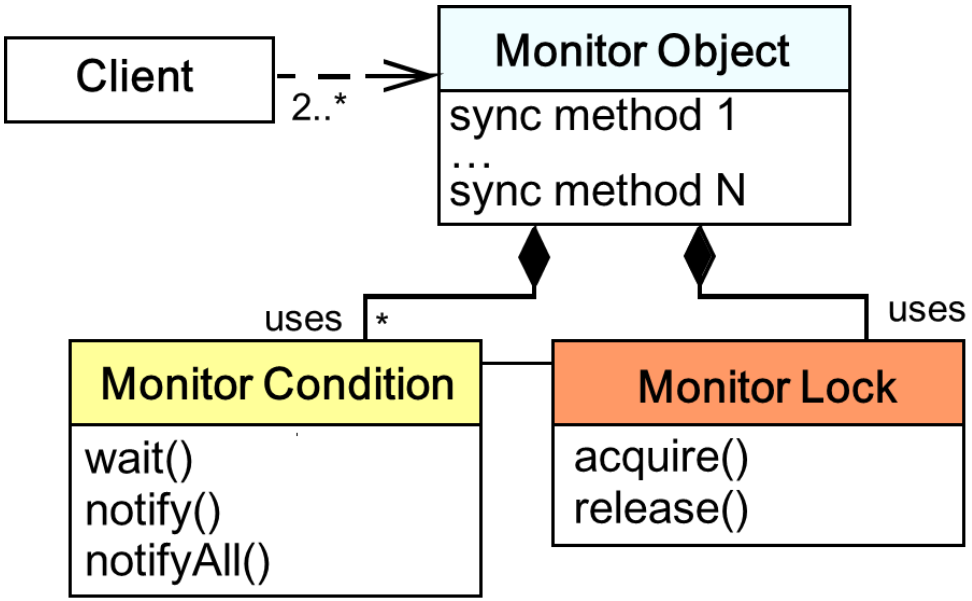


It's instructive to compare *Monitor Object* pattern with *Active Object* pattern

Benefits of the Monitor Object Pattern

Simplification of concurrency control

- This pattern presents a concise programming model for sharing an object among cooperating threads where object synchronization corresponds to method invocations



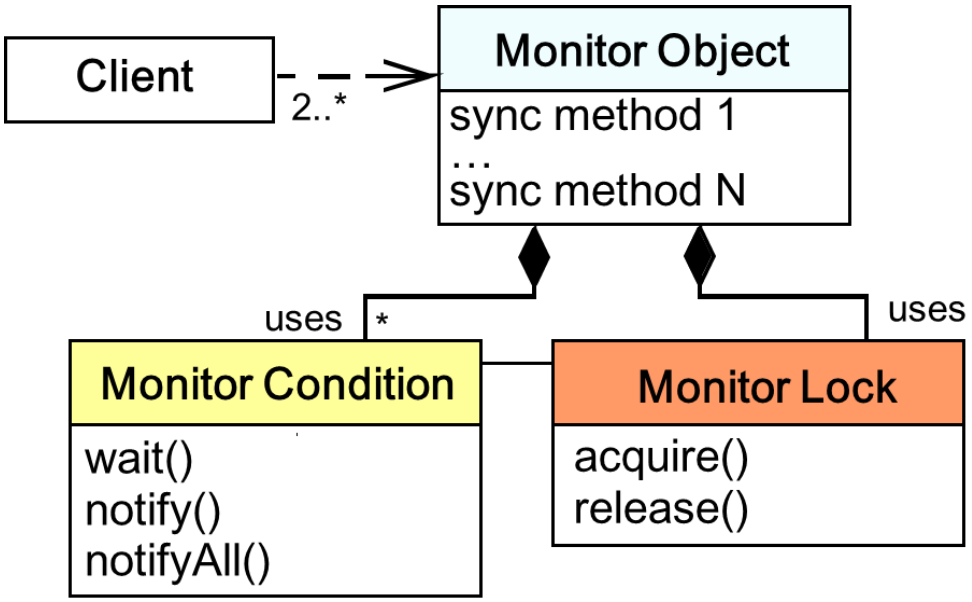
Benefits of the Monitor Object Pattern

Simplification of concurrency control

- This pattern presents a concise programming model for sharing an object among cooperating threads where object synchronization corresponds to method invocations

Simplification of scheduling method execution

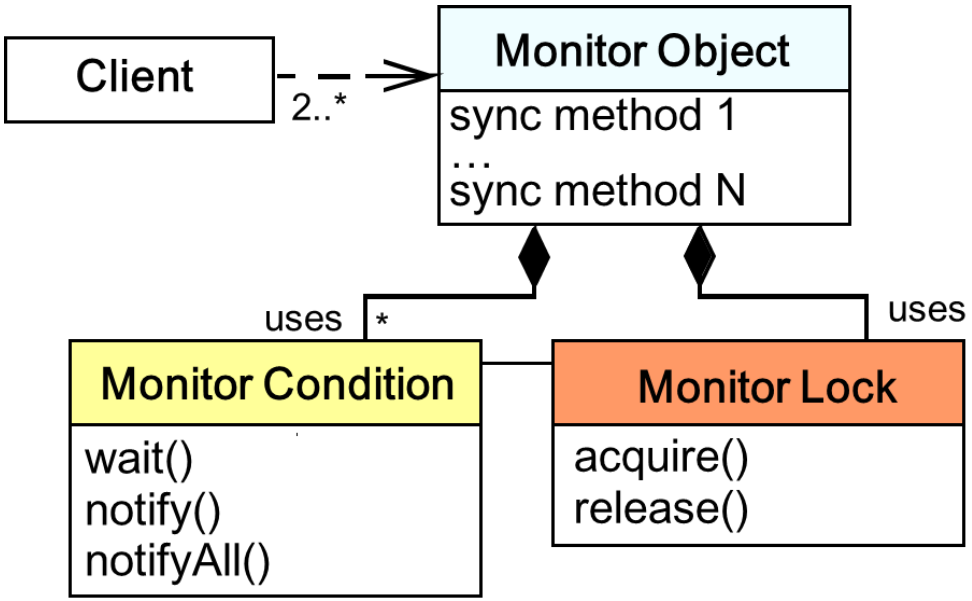
- Synchronized methods use their monitor conditions to determine the circumstances under which they should suspend or resume their execution & that of collaborating monitor objects



Limitations of the Monitor Object Pattern

Limited Scalability

- A single monitor lock can limit scalability due to increased contention when multiple threads serialize on a monitor object



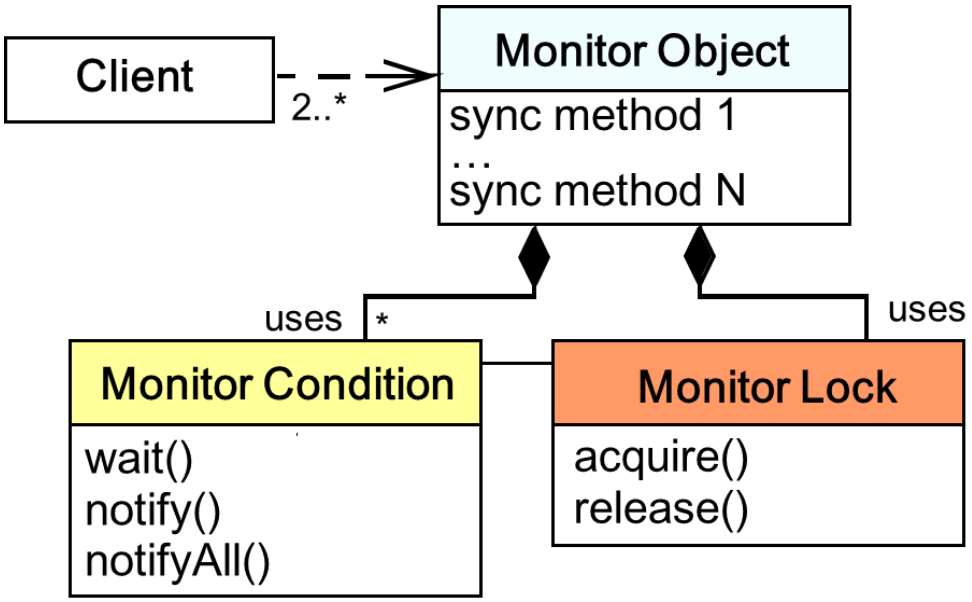
Limitations of the Monitor Object Pattern

Limited Scalability

- A single monitor lock can limit scalability due to increased contention when multiple threads serialize on a monitor object

Complicated extensibility semantics

- These result from the coupling between a monitor object's functionality & its synchronization mechanisms



Limitations of the Monitor Object Pattern

Limited Scalability

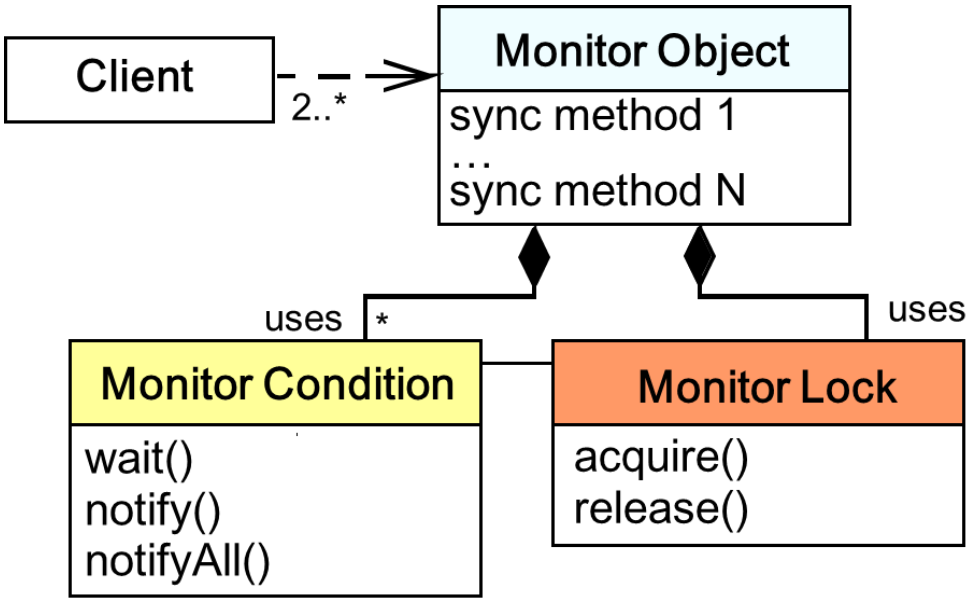
- A single monitor lock can limit scalability due to increased contention when multiple threads serialize on a monitor object

Complicated extensibility semantics

- These result from the coupling between a monitor object's functionality & its synchronization mechanisms

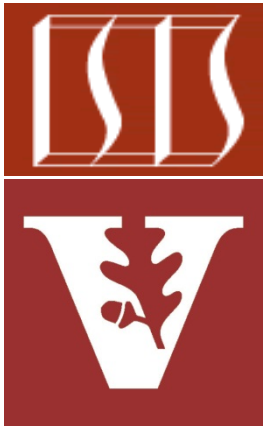
Nested monitor lockout

- This problem can occur when a monitor object is nested within another monitor object



Patterns & Frameworks for Concurrency & Synchronization: Part 7

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



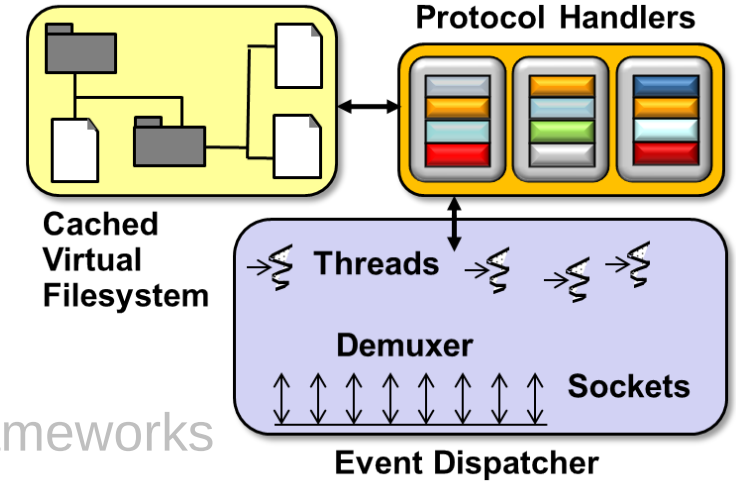
Professor of Computer Science

Institute for Software Integrated Systems

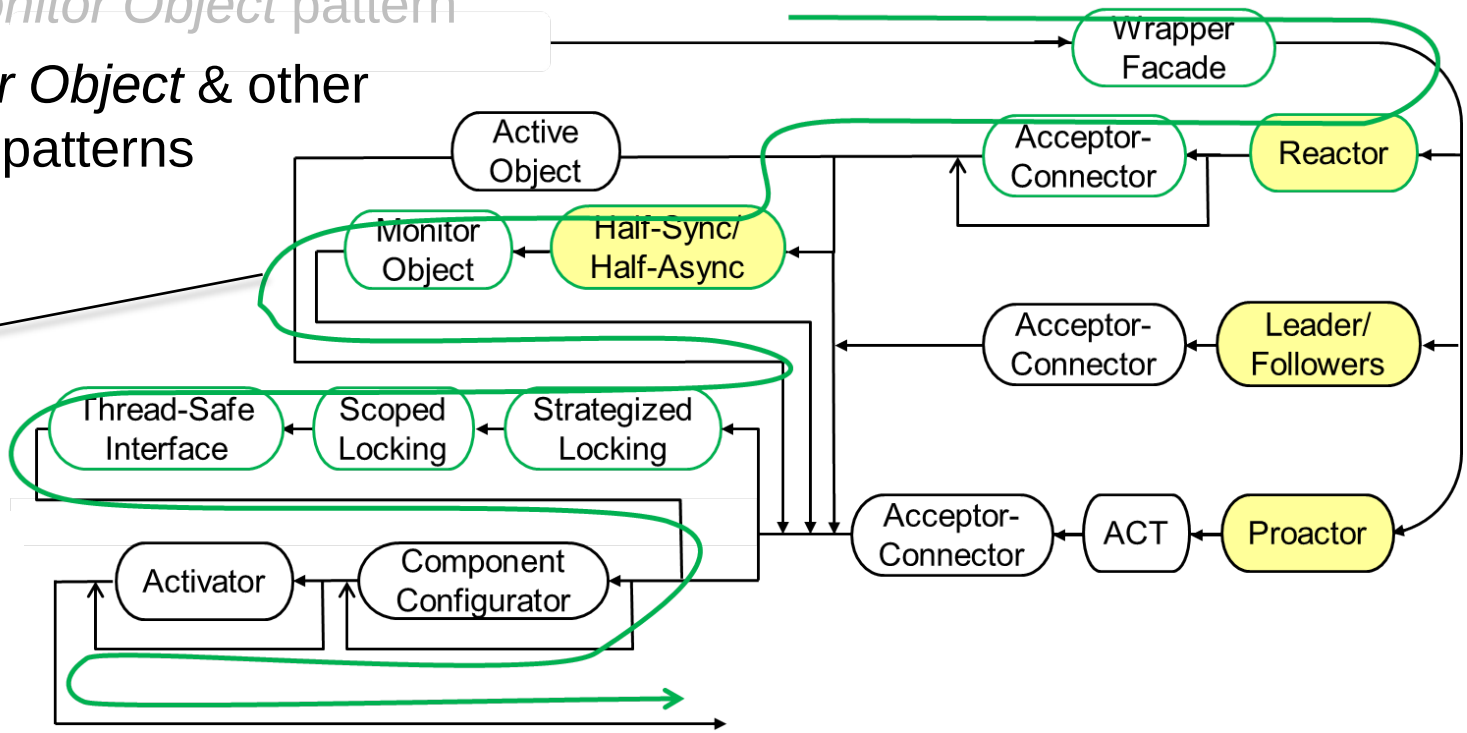
Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

- Describe the *Active Object* pattern
- Describe the ACE *Task* framework
- Apply ACE *Task* & *Acceptor-Connector* frameworks to JAWS
- Describe the *Half-Sync/Half-Async* pattern
- Implement *Half-Sync/Half-Async* using ACE frameworks
- Describe the *Monitor Object* pattern
- Applying *Monitor Object* & other synchronization patterns

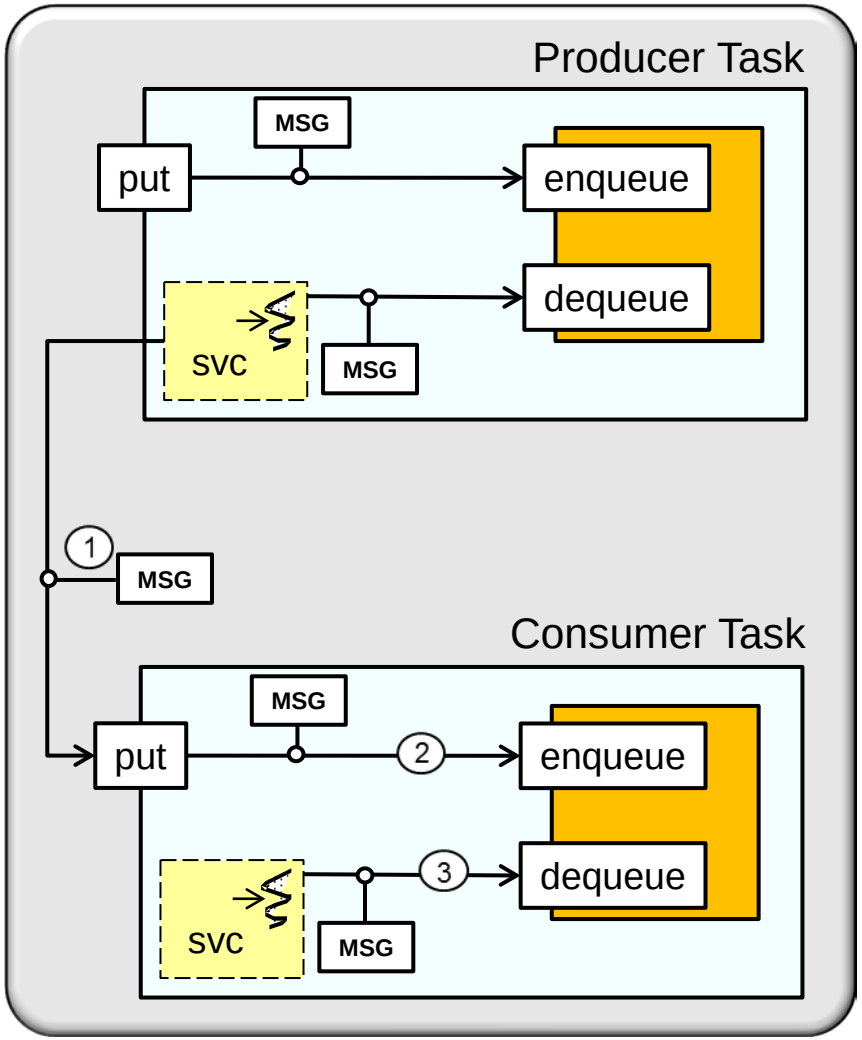


Another scalable multi-threaded path thru the JAWS pattern language



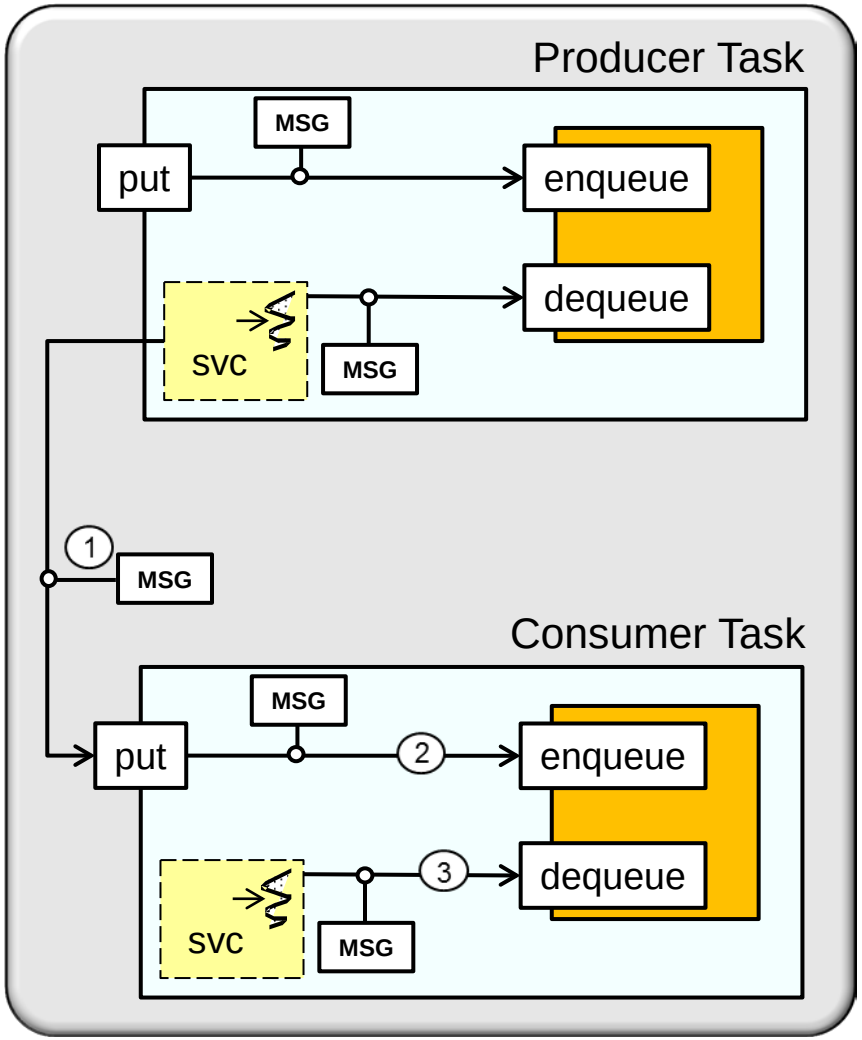
Motivation for the ACE_Message_Queue Class

- When producer & consumer tasks are collocated in the same process, tasks often exchange messages via intra-process message queues
  - Producer task(s) insert messages into a synchronized message queue serviced by consumer task(s) that remove & process the messages



Motivation for the ACE_Message_Queue Class

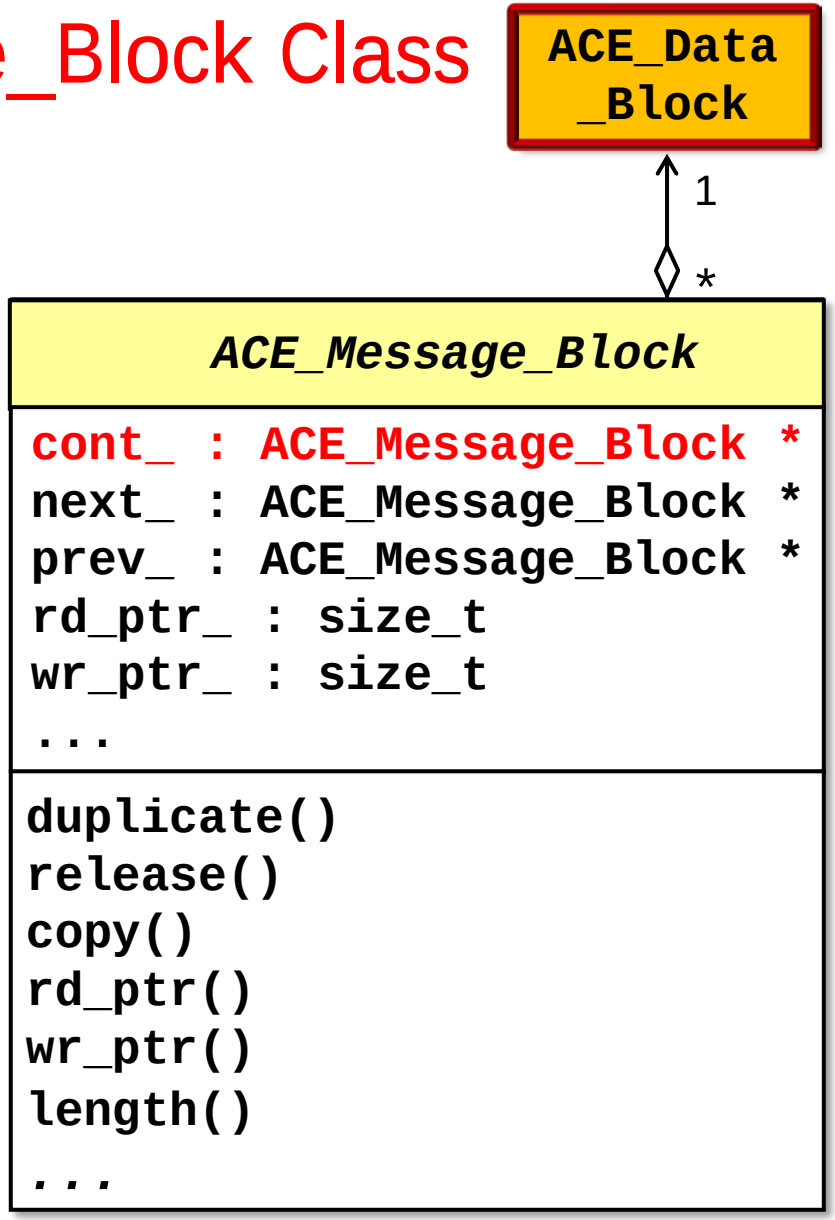
- When producer & consumer tasks are collocated in the same process, tasks often exchange messages via intra-process message queues
  - Producer task(s) insert messages into a synchronized message queue serviced by consumer task(s) that remove & process the messages
- If the queue is full, producers can either block or wait a bounded amount of time to insert their messages
  - Likewise, if the queue is empty, consumers can either block or wait a bounded amount of time to remove messages



The ACE_Message_Block Class

Provides means to manipulate messages efficiently via the following capabilities:

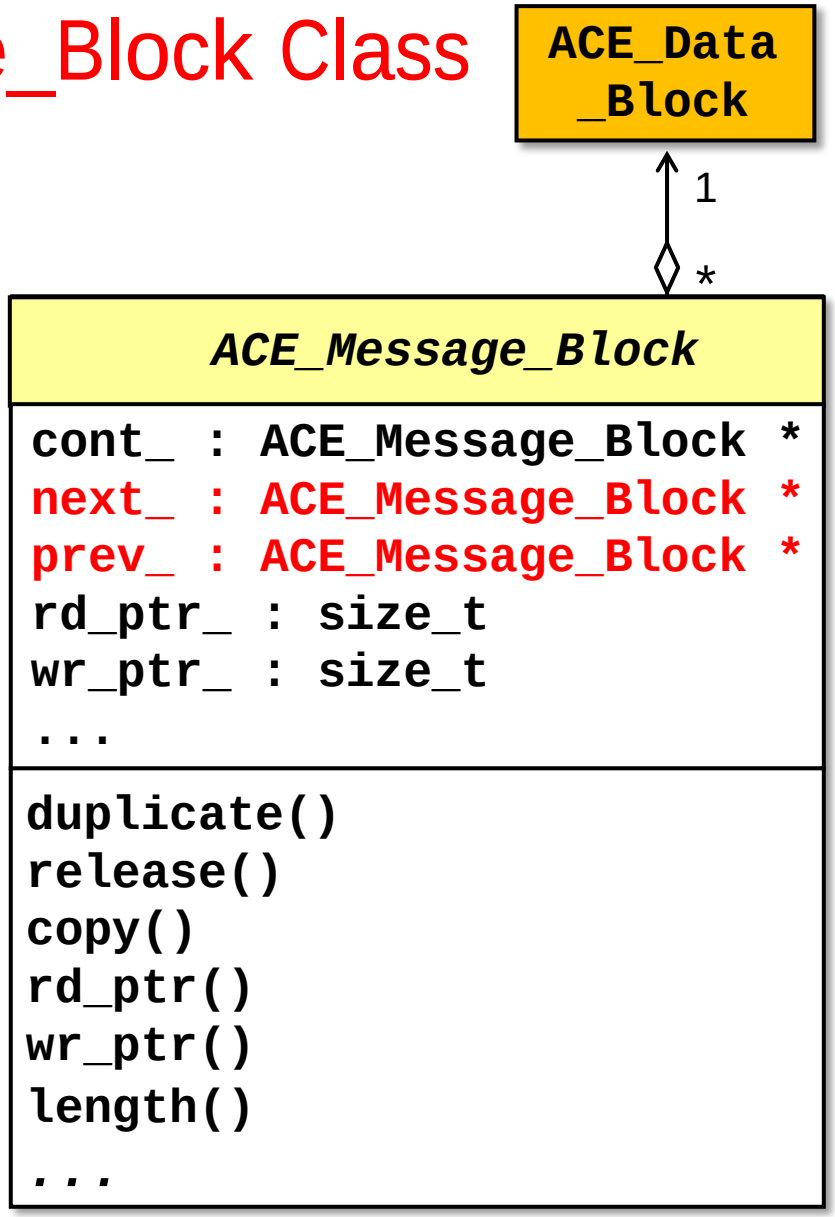
- Contains a pointer to a ref-counted **`ACE_Data_Block`** that points to the actual data associated with a message
  - Allows chaining of multiple messages together into a composite message



The ACE_Message_Block Class

Provides means to manipulate messages efficiently via the following capabilities:

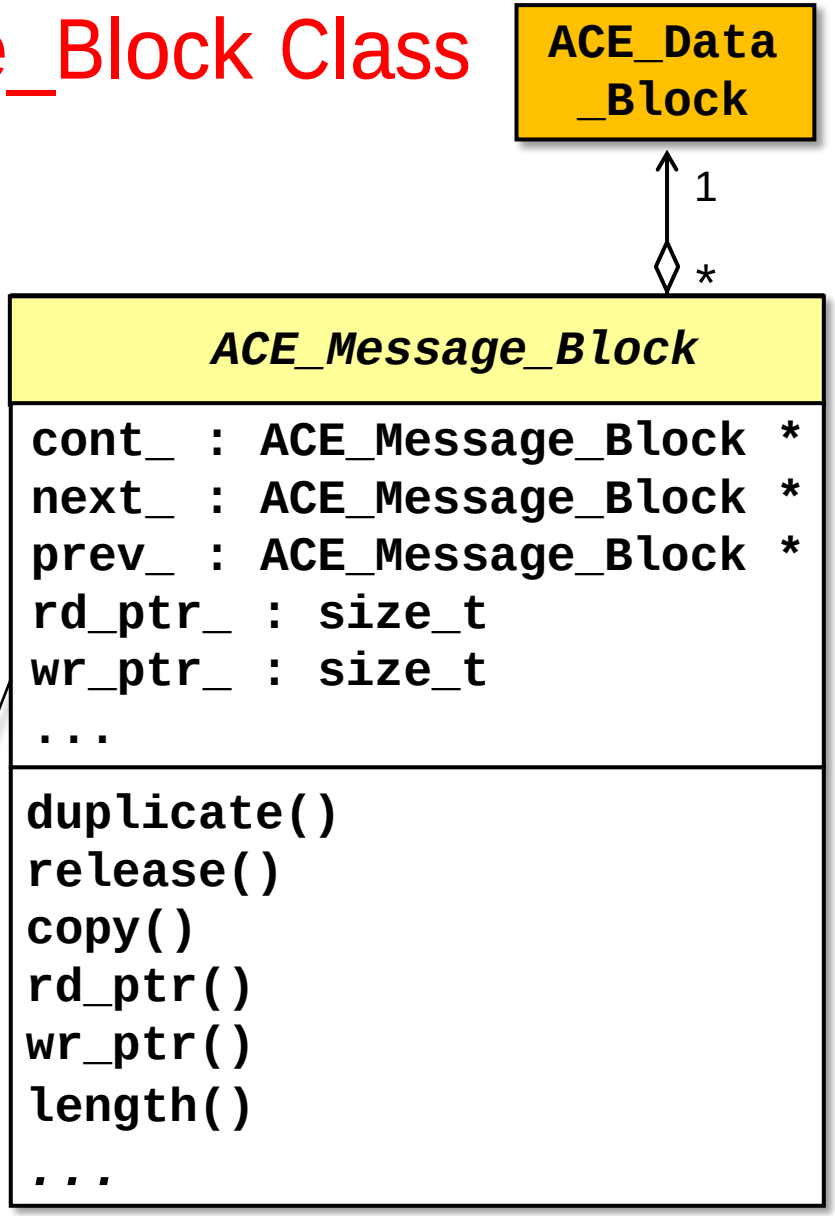
- Contains a pointer to a ref-counted **`ACE_Data_Block`** that points to the actual data associated with a message
  - Allows chaining of multiple messages together into a composite message
- Multiple messages can also be joined together to form doubly-linked list for **`ACE_Message_Queue`**



The ACE_Message_Block Class

Provides means to manipulate messages efficiently via the following capabilities:

- Contains a pointer to a ref-counted **`ACE_Data_Block`** that points to the actual data associated with a message
  - Allows chaining of multiple messages together into a composite message
- Multiple messages can also be joined together to form doubly-linked list for **`ACE_Message_Queue`**
- It treats synchronization & memory management properties as *aspects*



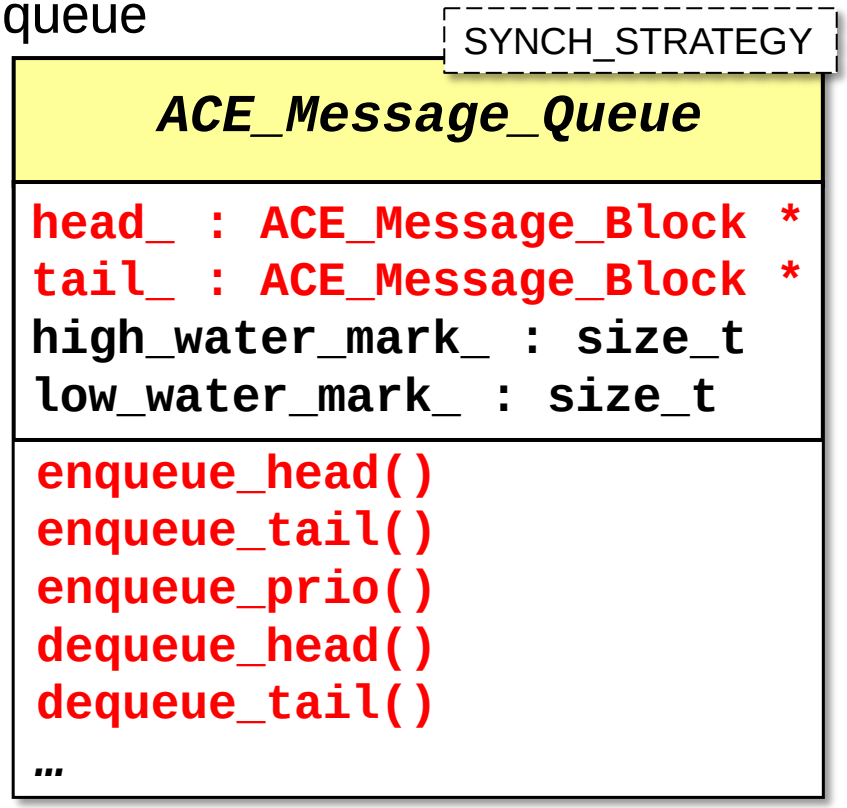
Composite structure minimizes dynamic memory allocation & data copying

Handles *variability* of message operations via *common* API

The ACE_Message_Queue Class

This class provides a portable intra-process messaging mechanism that provides the following capabilities:

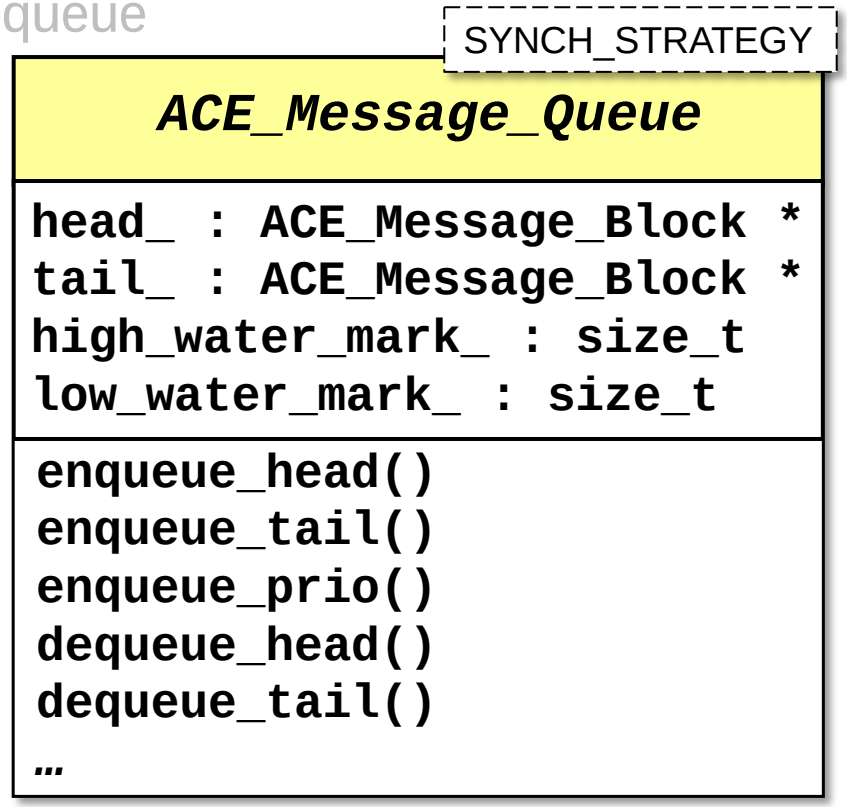
- Messages can be
  - Enqueued at front or rear of queue, or in priority order
  - Dequeued from the front or back of the queue



The ACE_Message_Queue Class

This class provides a portable intra-process messaging mechanism that provides the following capabilities:

- Messages can be
  - Enqueued at front or rear of queue, or in priority order
  - Dequeued from the front or back of the queue
- Timeouts can be used to avoid blocking

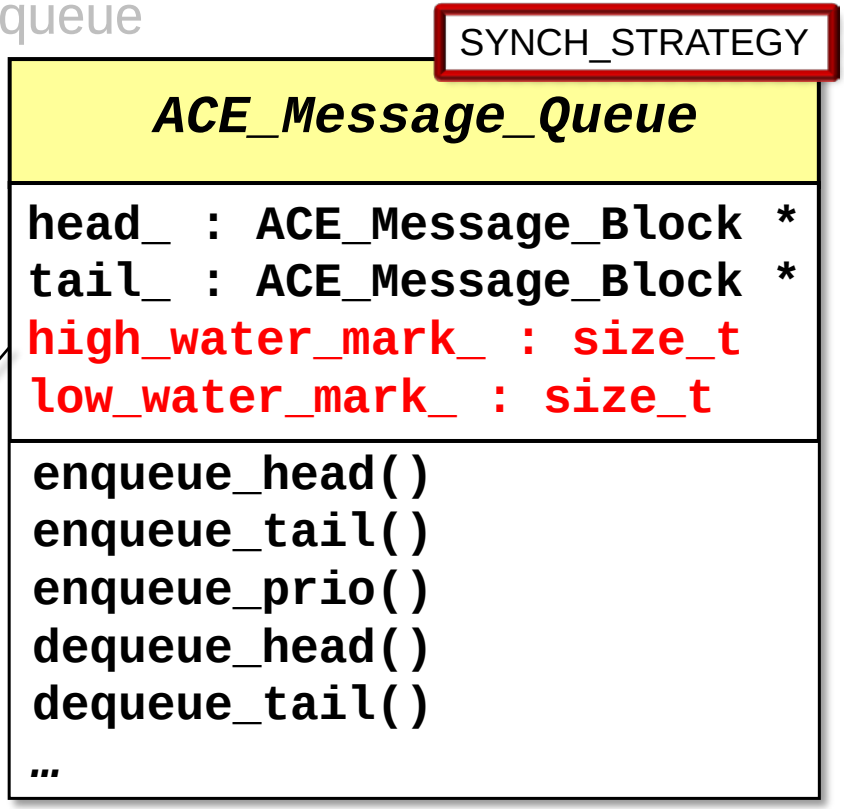


The ACE_Message_Queue Class

This class provides a portable intra-process messaging mechanism that provides the following capabilities:

- Messages can be
  - Enqueued at front or rear of queue, or in priority order
  - Dequeued from the front or back of the queue
- Timeouts can be used to avoid blocking
- It can run in either multi- or single-threaded configurations,
  - Multi-threaded configuration supports configurable flow control to prevent fast producers from swamping resources of slower consumers

Allocators can be strategized so memory used by messages can be obtained from various sources



Handles *variability* of queueing synchronization mechanisms via *common* API

Transparently Parameterizing Synchronization

Context	Problem
• The concurrency patterns described earlier impact **ACE_Message_Queue** synchronization strategies	• Hard-coding synchronization strategies into **ACE_Message_Queue** implementations is inflexible, but maintaining multiple versions manually is not scalable

```
class Nonsynch_Message_Queue {
  ...
protected:
  ACE_Message_Block *head_;
  ACE_Message_Block *tail_;
  ...
};
```

No synchronizers

Transparently Parameterizing Synchronization

Context	Problem
• The concurrency patterns described earlier impact **ACE_Message_Queue** synchronization strategies	• Hard-coding synchronization strategies into **ACE_Message_Queue** implementations is inflexible, but maintaining multiple versions manually is not scalable

```
class Nonsynch_Message_Queue {
  ...
protected:
  ACE_Message_Block *head_;
  ACE_Message_Block *tail_;
  ...
};
```

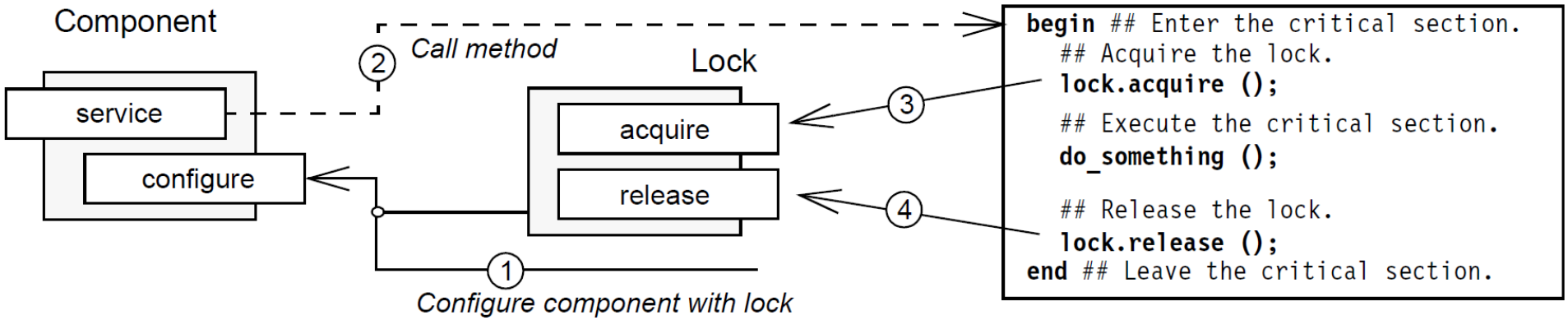
Synchronizers 

```
class Synch_Message_Queue {
  ...
protected:
  ACE_Message_Block *head_;
  ACE_Message_Block *tail_;

  ACE_Thread_Mutex lock_;
  ACE_Condition<ACE_Thread_Mutex>
    not_empty_;
  ACE_Condition<ACE_Thread_Mutex>
    not_full_;
  ...
};
```

Transparently Parameterizing Synchronization

Context	Problem	Solution
• The concurrency patterns described earlier impact **ACE_Message_Queue** synchronization strategies	• Hard-coding synchronization strategies into **ACE_Message_Queue** implementations is inflexible, but maintaining multiple versions manually is not scalable	• Apply *Strategized Locking* pattern to parameterize the synchronization mechanisms of **ACE_Message_Queue**

Strategized Locking parameterizes synchronization mechanisms that protect a component's critical sections from concurrent access



See www.dre.vanderbilt.edu/~schmidt/PDF/locking-patterns.pdf for more info

Applying Strategized Locking to ACE_Message_Queue

1. Strategize the locking mechanisms

```
template <typename SYNCH_STRATEGY>
class ACE_Message_Queue {
  ...
protected:
  ACE_Message_Block *head_;
  ACE_Message_Block *tail_;

  typename SYNCH_STRATEGY::MUTEX lock_;
  typename SYNCH_STRATEGY::CONDITION not_empty_;
  typename SYNCH_STRATEGY::CONDITION not_full_;
  ...
};
```

Parameterized locking strategy

Traits that coordinate concurrent access

Applying Strategized Locking to ACE_Message_Queue

2. Define a family of traits classes that convey information used by **`ACE_Message_Queue`** to determine policies at compile time

```
class ACE_NULL_SYNCH {
public:
  typedef ACE_Null_Mutex
          MUTEX;
  typedef ACE_Null_Condition
          CONDITION;
  typedef ACE_Null_Semaphore
          SEMAPHORE;
  ...
};
```

No-op synchronizers

```
class ACE_MT_SYNCH {
public:
  typedef ACE_Thread_Mutex
          MUTEX;
  typedef ACE_Condition
            <ACE_Thread_Mutex>
          CONDITION;
  typedef ACE_Thread_Semaphore
          SEMAPHORE;
  ...
};
```

Actual synchronizers

Applying Strategized Locking to ACE_Message_Queue

2. Define a family of traits classes that convey information used by **`ACE_Message_Queue`** to determine policies at compile time

Traits classes needn't derive from a common base class or use virtual methods!

```
class ACE_NULL_SYNCH {
public:
  typedef ACE_Null_Mutex
          MUTEX;
  typedef ACE_Null_Condition
          CONDITION;
  typedef ACE_Null_Semaphore
          SEMAPHORE;
  ...
};
```

```
class ACE_MT_SYNCH {
public:
  typedef ACE_Thread_Mutex
          MUTEX;
  typedef ACE_Condition
            <ACE_Thread_Mutex>
          CONDITION;
  typedef ACE_Thread_Semaphore
          SEMAPHORE;
  ...
};
```

Applying Strategized Locking to ACE_Message_Queue

3. Use the traits classes to instantiate **ACE_Message_Queue** with the desired type of synchronization strategy

No-op synchronizers

```
typedef ACE_Message_Queue
          <ACE_NULL_SYNCH>
        ST_MQ;

ST_MQ st_mq;
ACE_Message_Block *mb;
...

st_mq.dequeue_head (mb);
```

Does not block

Applying Strategized Locking to ACE_Message_Queue

3. Use the traits classes to instantiate **`ACE_Message_Queue`** with the desired type of synchronization strategy

```
typedef ACE_Message_Queue
          <ACE_NULL_SYNCH>
        ST_MQ;

ST_MQ st_mq;
ACE_Message_Block *mb;
...

st_mq.dequeue_head (mb);
```

Actual synchronizers

```
typedef ACE_Message_Queue
          <ACE_MT_SYNCH>
        MT_MQ;

MT_MQ mt_mq;
ACE_Message_Block *mb;
...

mt_mq.dequeue_head (mb);
```

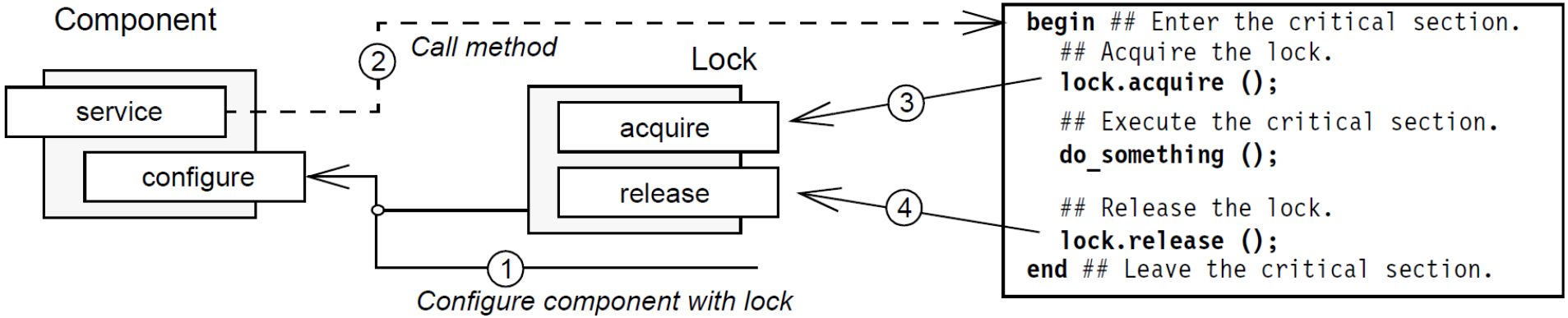
Does block

It's also common practice to use dynamic polymorphism to strategize locking

Benefits of the Strategized Locking Pattern

Enhanced flexibility & customization

- It is straightforward to configure & customize a component for designated concurrency models because the synchronization aspects of components are strategized



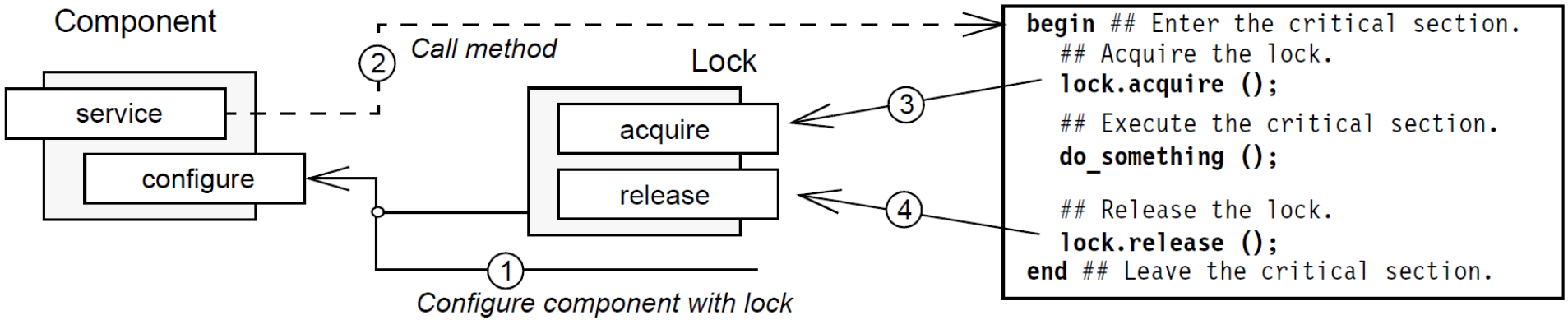
Benefits of the Strategized Locking Pattern

Enhanced flexibility & customization

- It is straightforward to configure & customize a component for designated concurrency models because the synchronization aspects of components are strategized

Decreased maintenance effort for components

- It is straightforward to add enhancements & bug fixes to a component because there is only one implementation, rather than a separate implementation for each concurrency model



Benefits of the Strategized Locking Pattern

Enhanced flexibility & customization

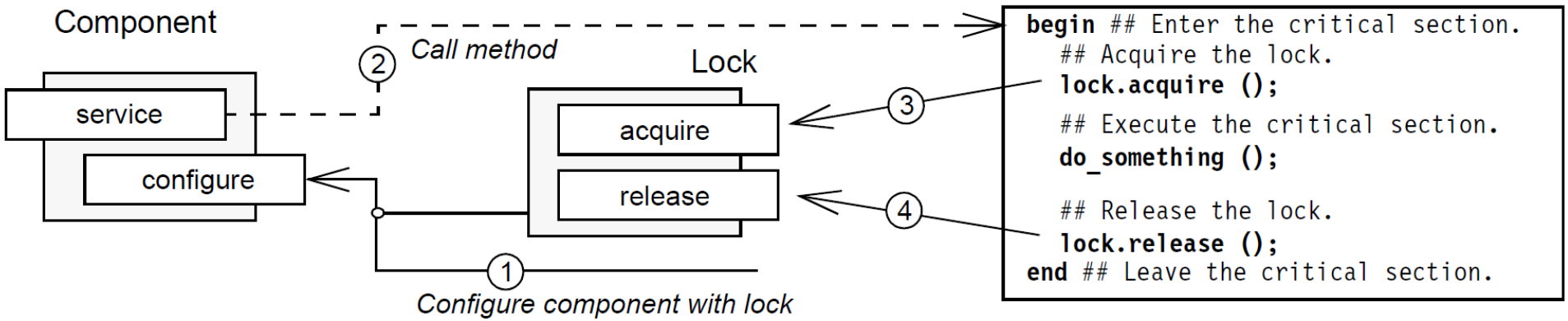
- It is straightforward to configure & customize a component for designated concurrency models because the synchronization aspects of components are strategized

Decreased maintenance effort for components

- It is straightforward to add enhancements & bug fixes to a component because there is only one implementation, rather than a separate implementation for each concurrency model

Improved reuse

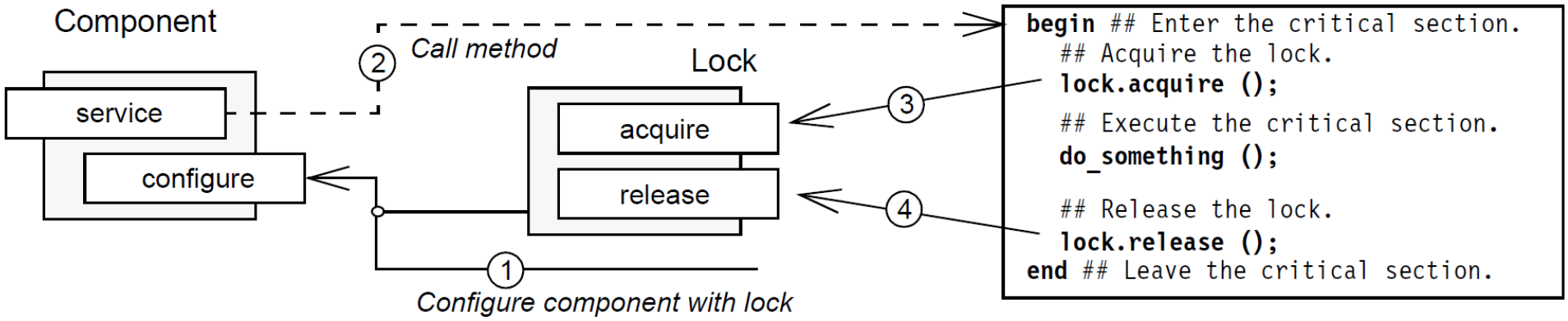
- Components implemented using this pattern are more reusable, because their locking strategies can be configured orthogonally to their behavior



Limitations of the Strategized Locking Pattern

Obtrusive locking

- If templates are used to parameterize locking aspects this will expose the locking strategies to application code



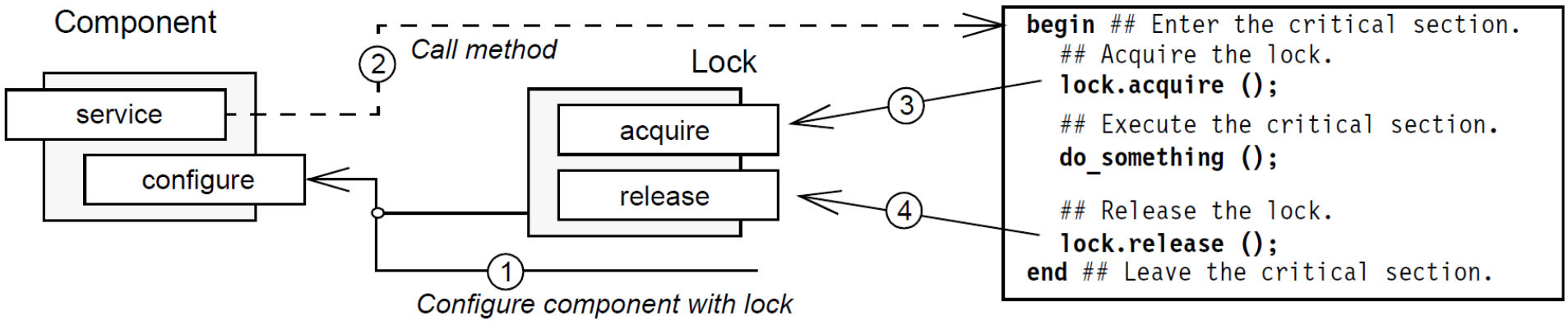
Limitations of the Strategized Locking Pattern

Obtrusive locking

- If templates are used to parameterize locking aspects this will expose the locking strategies to application code

Over-engineering

- Externalizing a locking mechanism by placing it in a component's interface may actually provide too much flexibility in certain situations
  - e.g., inexperienced developers may try to parameterize a component with the wrong type of lock, resulting in improper compile-time or runtime behavior



Ensuring Locks are Released Properly

Context	Problem
• A shared resource that is concurrently accessed & updated by multiple threads must be protected by a lock	• If programmers acquire & release locks explicitly, it's hard to ensure locks are released in all code paths

```
template <typename SYNCH_STRATEGY>
class ACE_Message_Queue {
public:
...
  int dequeue_head (ACE_Message_Block &*mb, ACE_Time_Value *to) {
    lock_.acquire ();
    ...
    lock_.release ();
  }
...
```

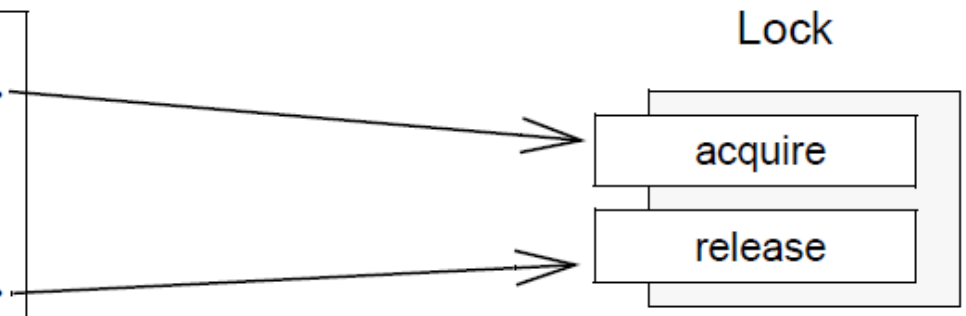
Control can leave scope in C++ prematurely due to return, break, continue, goto, or unhandled exception

Ensuring Locks are Released Properly

Context	Problem	Solution
• A shared resource that is concurrently accessed & updated by multiple threads must be protected by a lock	• If programmers acquire & release locks explicitly, it's hard to ensure locks are released in all code paths	• Apply the *Scoped Locking* pattern to automatically acquire a lock when control enters a scope & automatically release the lock when control leaves the scope

Scoped Locking ensures that entering a critical section becomes thread-safe & leaving the critical section releases all acquired locks

```
begin ## Enter the critical section.
   ## Acquire the lock automatically.

   ## Execute the critical section.
   do_something ();

   ## Release the lock automatically.
end ## Leave the critical section.
```



See www.dre.vanderbilt.edu/~schmidt/PDF/locking-patterns.pdf for more info

Applying Scoped Locking to ACE_Message_Queue

1. Define a guard class that acquires & releases a particular type of lock in its constructor & destructor, respectively

```
template <typename LOCK>
class ACE_Guard {
public:
  ACE_Guard (LOCK &lock): lock_ (&lock) { lock_->acquire (); }

 ~ACE_Guard () { lock_->release (); }
  ...

private:
  LOCK *lock_;
}
```

Store a pointer to lock & acquire lock in constructor

Destructor releases lock when guard goes out of scope

Pointer to the lock that's being managed

ACE_Guard illustrates another benefit of applying the *Wrapper Façade* pattern

Applying Scoped Locking to ACE_Message_Queue

2. Let critical sections correspond to the scope & lifetime of a guard object

```
template <typename SYNCH_STRATEGY>
class ACE_Message_Queue {
public:
...
  int dequeue_head (ACE_Message_Block *&mb, ACE_Timeout *to) {

    ACE_Guard<typename SYNCH_STRATEGY::MUTEX> guard (lock_);
    ...
  }
...
```

Use *Scoped Locking* to acquire & release `lock_` automatically

`lock_` released automatically when `guard` goes out of scope

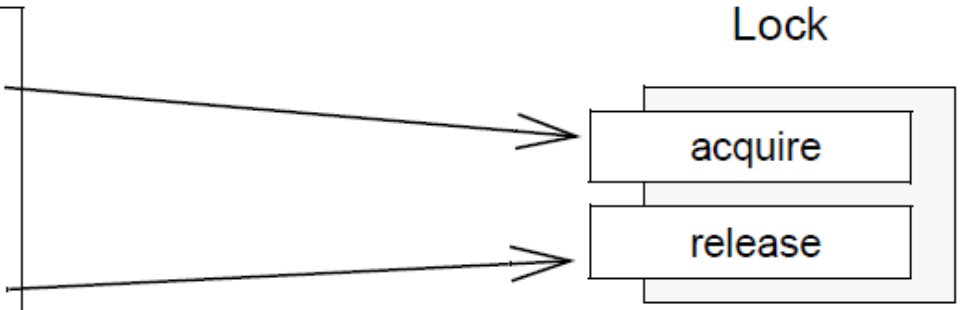
Instances of **`ACE_Guard`** can be allocated on the run-time stack to acquire & release locks in method or block scopes that define critical sections

Benefit of the Scoped Locking Pattern

Increased robustness

- This pattern increases the robustness of concurrent applications by eliminating common programming errors related to synchronization & multi-threading
  - By applying *Scoped Locking*, locks are acquired & released automatically when control enters & leaves critical sections defined by C++ method & block scopes

```
begin ## Enter the critical section.
   ## Acquire the lock automatically.

   ## Execute the critical section.
   do_something ();

   ## Release the lock automatically.
end ## Leave the critical section.
```



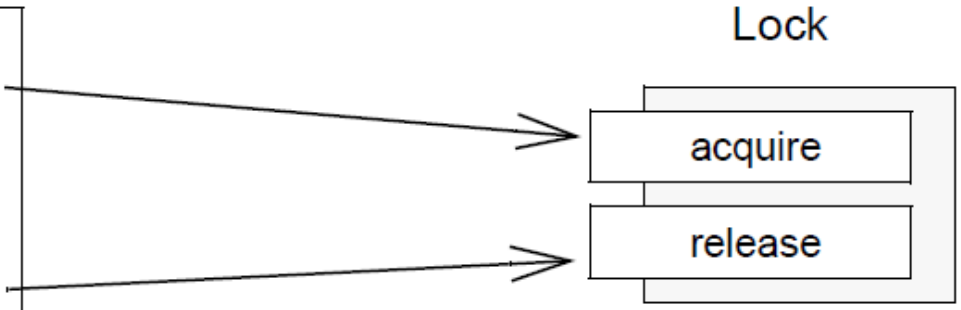
See en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization for C++ idiom

Limitations of the Scoped Locking Pattern

Potential for deadlock when used recursively

- If a method that uses *Scoped Locking* calls itself recursively, 'self-deadlock' will occur if the lock is not a recursive mutex

```
begin ## Enter the critical section.
  ## Acquire the lock automatically.

  ## Execute the critical section.
  do_something ();

  ## Release the lock automatically.
end ## Leave the critical section.
```



Limitations of the Scoped Locking Pattern

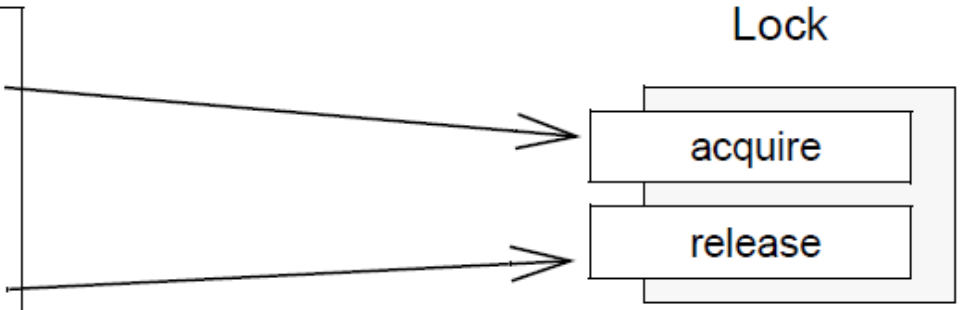
Potential for deadlock when used recursively

- If a method that uses *Scoped Locking* calls itself recursively, 'self-deadlock' will occur if the lock is not a recursive mutex

Limitations with language-specific semantics

- *Scoped Locking* is based on programming language features & thus isn't integrated with OS-specific system calls
  - Locks may therefore not be released automatically when threads abort or exit inside a guarded critical section, e.g., via **`longjmp()`**

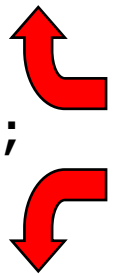
```
begin ## Enter the critical section.
  ## Acquire the lock automatically.

  ## Execute the critical section.
  do_something ();

  ## Release the lock automatically.
end ## Leave the critical section.
```



Minimizing Unnecessary Locking

Context	Problem
• Components in multi-threaded applications that (1) contain intra-component method calls & (2) have applied *Strategized Locking*	• Thread-safe components should not lock unnecessarily • Thread-safe components should be designed to avoid "self-deadlock"

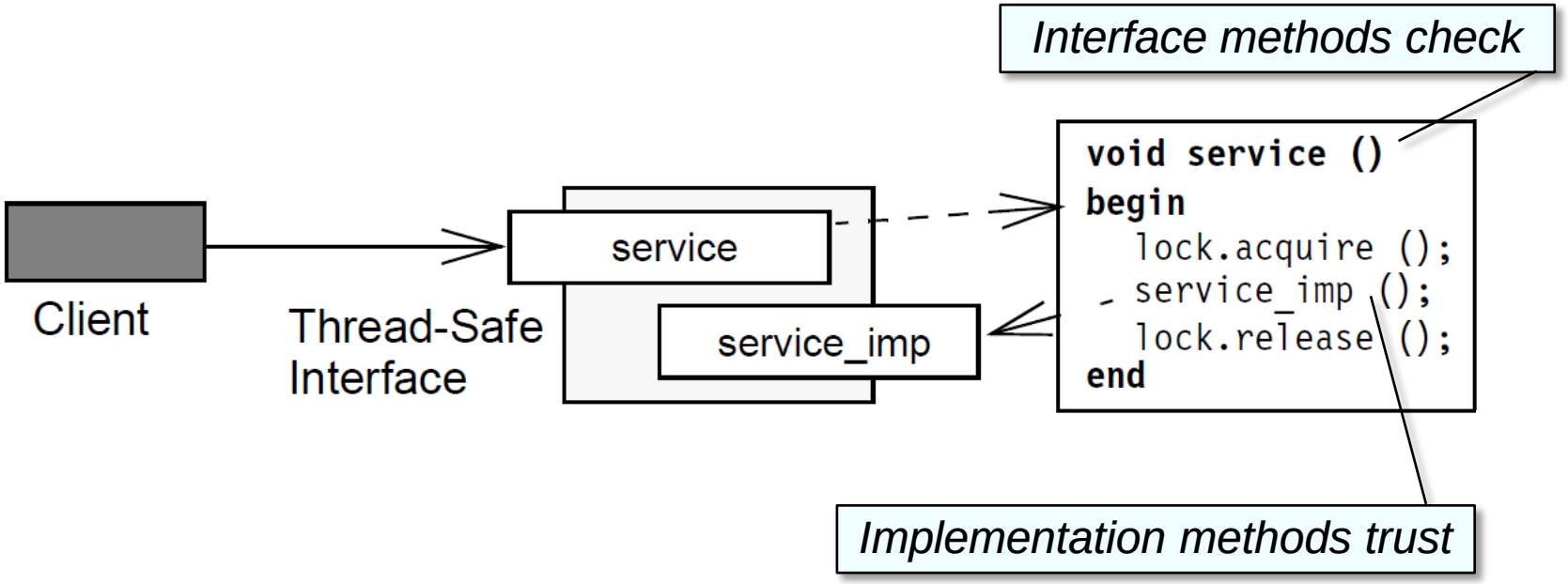
```
template <typename SYNCH_STRATEGY>
class ACE_Message_Queue {
public:
  int dequeue_head (ACE_Message_Block *&mb, ACE_Timeout *to) {
    ACE_Guard<typename SYNCH_STRATEGY::MUTEX> guard (lock_);
    ...
    while (is_empty())
      not_empty_cond_.wait ();
      ...
  int is_empty () {
    ACE_Guard<typename SYNCH_STRATEGY::MUTEX> guard (lock_);
    return cur_bytes_ == 0 && cur_count_ == 0;
  ...
```



Errors and/or overhead may occur depending on type of locking strategy

Minimizing Unnecessary Locking

Context	Problem	Solution
• Components in multi-threaded applications that (1) contain intra-component method calls & (2) have applied *Strategized Locking*	• Thread-safe components should not lock unnecessarily • Thread-safe components should be designed to avoid "self-deadlock"	• Apply the *Thread-safe Interface* pattern to minimize locking overhead & ensure that intra-component method calls don't incur 'self-deadlock'



See www.dre.vanderbilt.edu/~schmidt/PDF/locking-patterns.pdf for more info

Applying Thread-safe Interface to ACE_Message_Queue

Split a component's methods into publicly accessible interface methods & corresponding private implementation methods

```
template <typename SYNCH_STRATEGY>
class ACE_Message_Queue {
public:
  int dequeue_head (ACE_Message_Block *&mb, ACE_Timeout *to) {
    ACE_Guard<typename SYNCH_STRATEGY::MUTEX> guard (lock_);
    ...
    while (is_empty_i ())
      not_empty_cond_.wait ()
      ...

private:
  int is_empty_i () {
    return cur_bytes_ == 0 && cur_count_ == 0;
  ...
```

Interface method checks lock is acquired, calls associated implementation method, & automatically releases lock

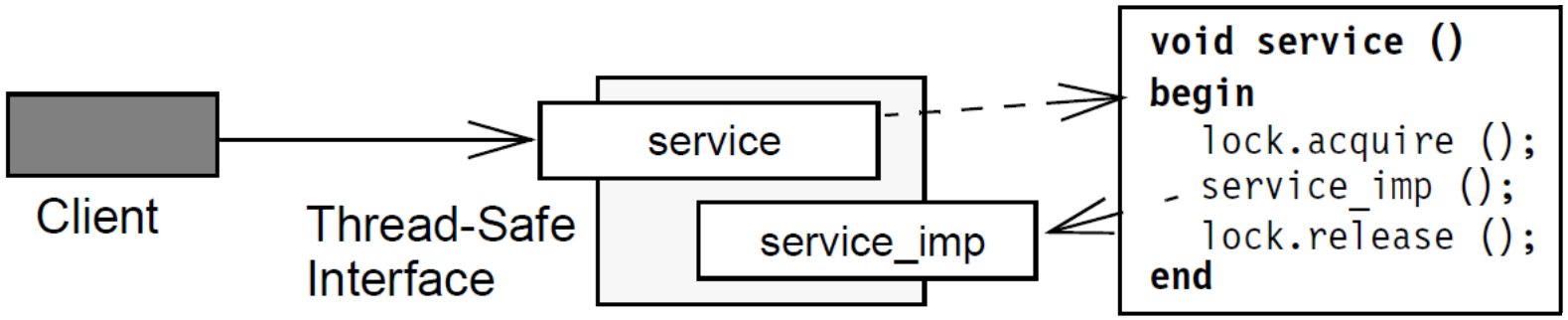
Implementation method trusts lock is held, does its work, & only invokes other implementation methods

This pattern allows a broader range of locking strategies

Benefits of the Thread-safe Interface Pattern

Increased robustness

- This pattern ensures that self-deadlock does not occur due to intra-component method calls



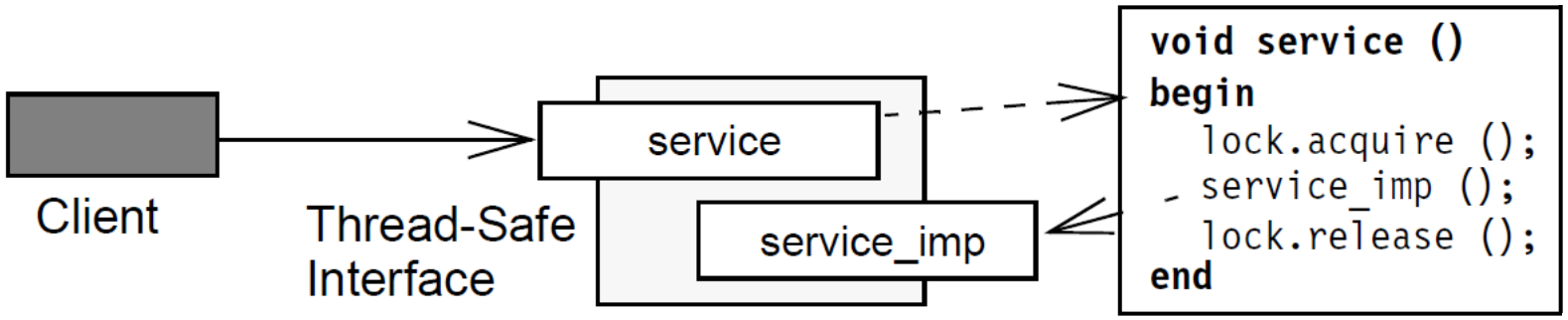
Benefits of the Thread-safe Interface Pattern

Increased robustness

- This pattern ensures that self-deadlock does not occur due to intra-component method calls

Enhanced performance

- This pattern ensures that locks are not acquired or released unnecessarily



Benefits of the Thread-safe Interface Pattern

Increased robustness

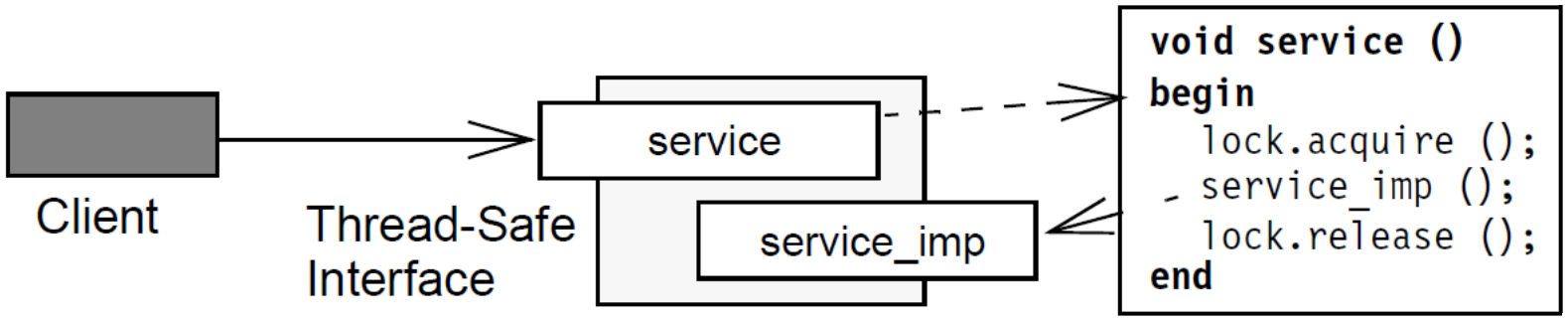
- This pattern ensures that self-deadlock does not occur due to intra-component method calls

Enhanced performance

- This pattern ensures that locks are not acquired or released unnecessarily

Simplification of software

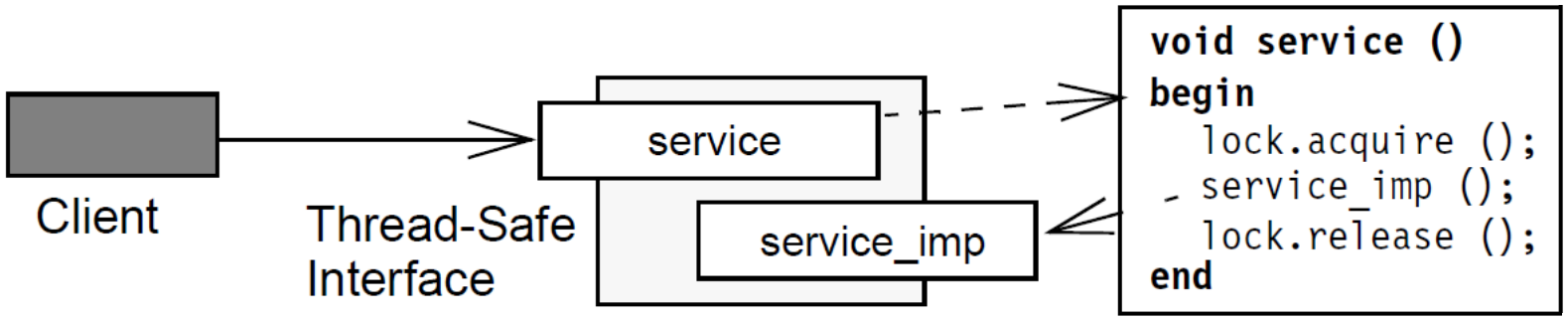
- Separating the locking & functionality concerns can help to simplify both aspects



Limitations of the Thread-safe Interface Pattern

Additional indirection & extra methods

- Each interface method requires at least one implementation method
  - Increases memory footprint & may also add an extra level of indirection



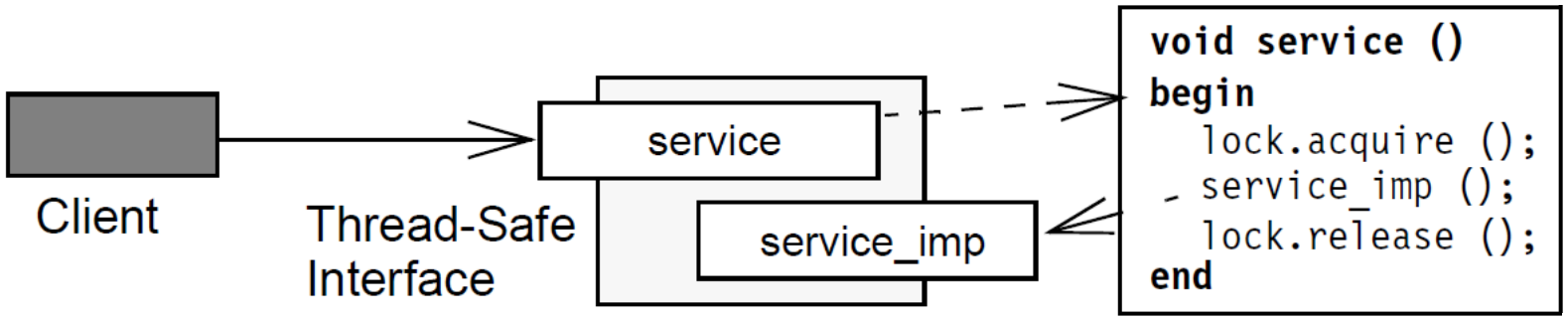
Limitations of the Thread-safe Interface Pattern

Additional indirection & extra methods

- Each interface method requires at least one implementation method
  - Increases memory footprint & may also add an extra level of indirection

Potential for misuse

- OO languages support class-level rather than object-level access control
  - An object can thus bypass the public interface to call a private method on another object of same class, thereby bypassing that object's lock



Limitations of the Thread-safe Interface Pattern

Additional indirection & extra methods

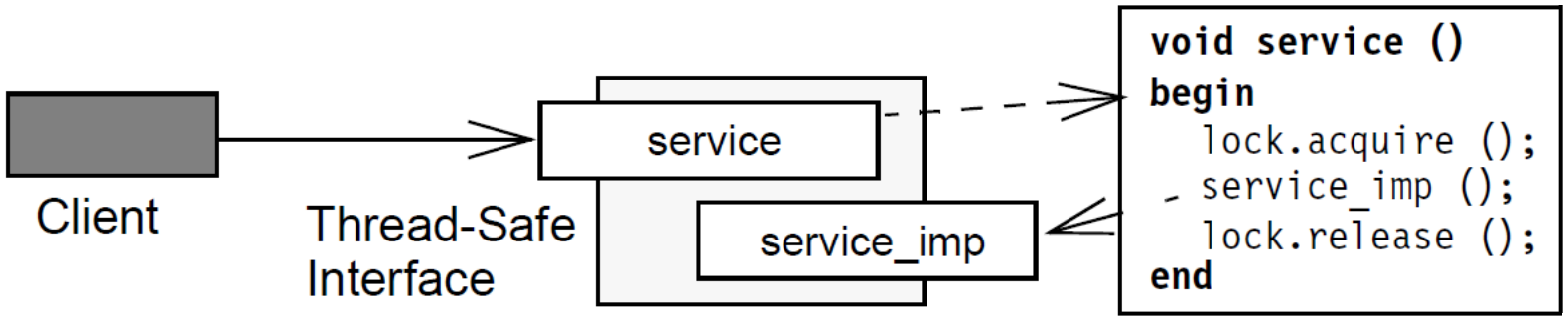
- Each interface method requires at least one implementation method
  - Increases memory footprint & may also add an extra level of indirection

Potential for misuse

- OO languages support class-level rather than object-level access control
  - An object can thus bypass the public interface to call a private method on another object of same class, thereby bypassing that object's lock

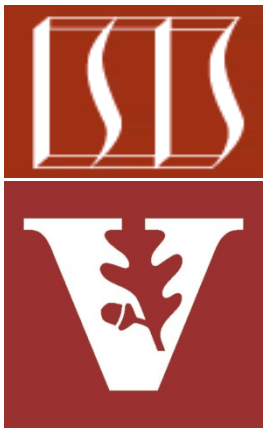
Potential overhead

- This pattern prevents multiple components from sharing the same lock & prevents locking at a finer granularity than the component, which can increase lock contention



Patterns & Frameworks for Concurrency & Synchronization: Part 8

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



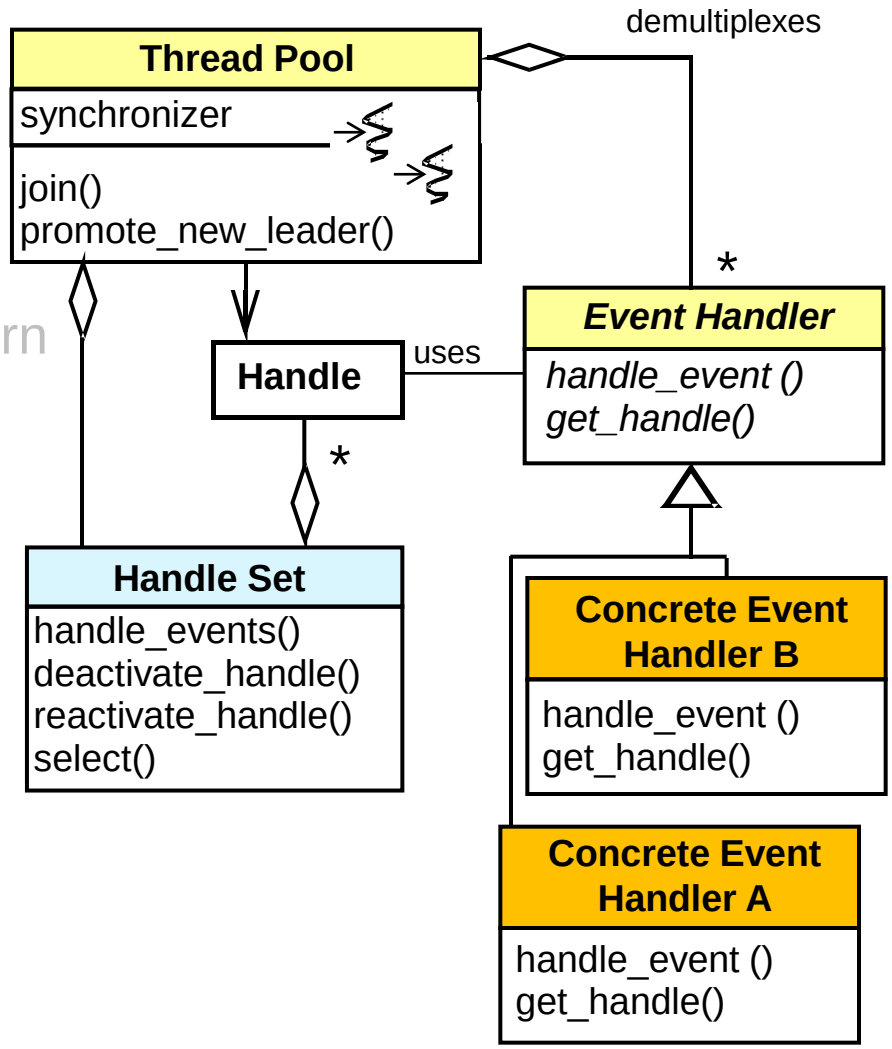
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

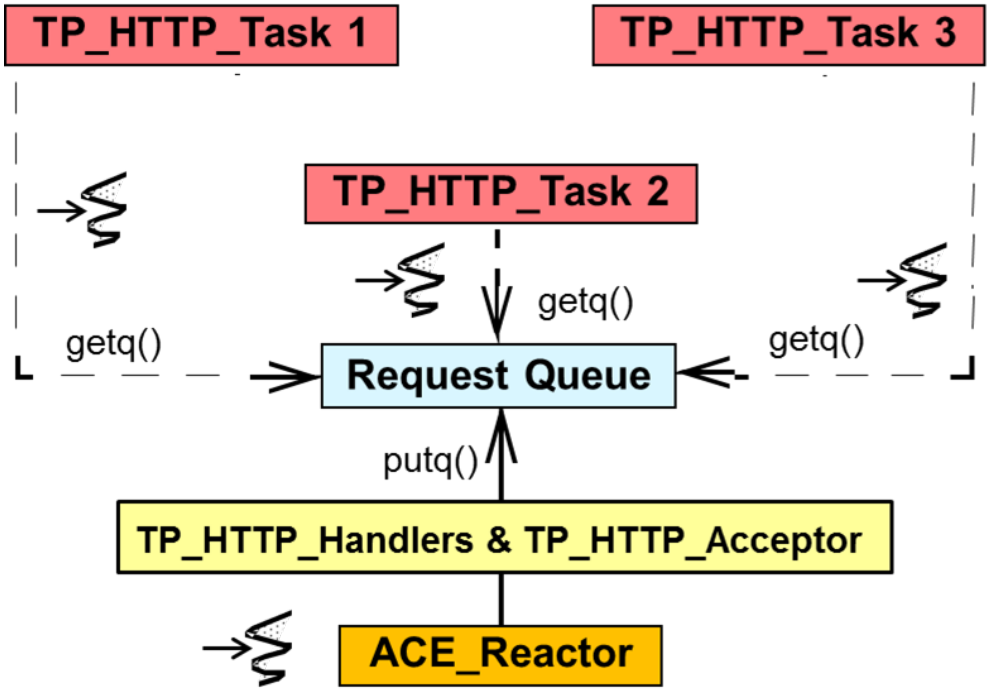
Topics Covered in this Part of the Module

- Describe the *Active Object* pattern
- Describe the ACE *Task* framework
- Apply ACE *Task* & *Acceptor-Connector* frameworks to JAWS
- Describe the *Half-Sync/Half-Async* pattern
- Implement *Half-Sync/Half-Async* using ACE frameworks
- Describe the *Monitor Object* pattern
- Applying *Monitor Object* & other synchronization patterns
- Describe the *Leader/ Followers* pattern



Enhancing Predictability with Leader/Followers

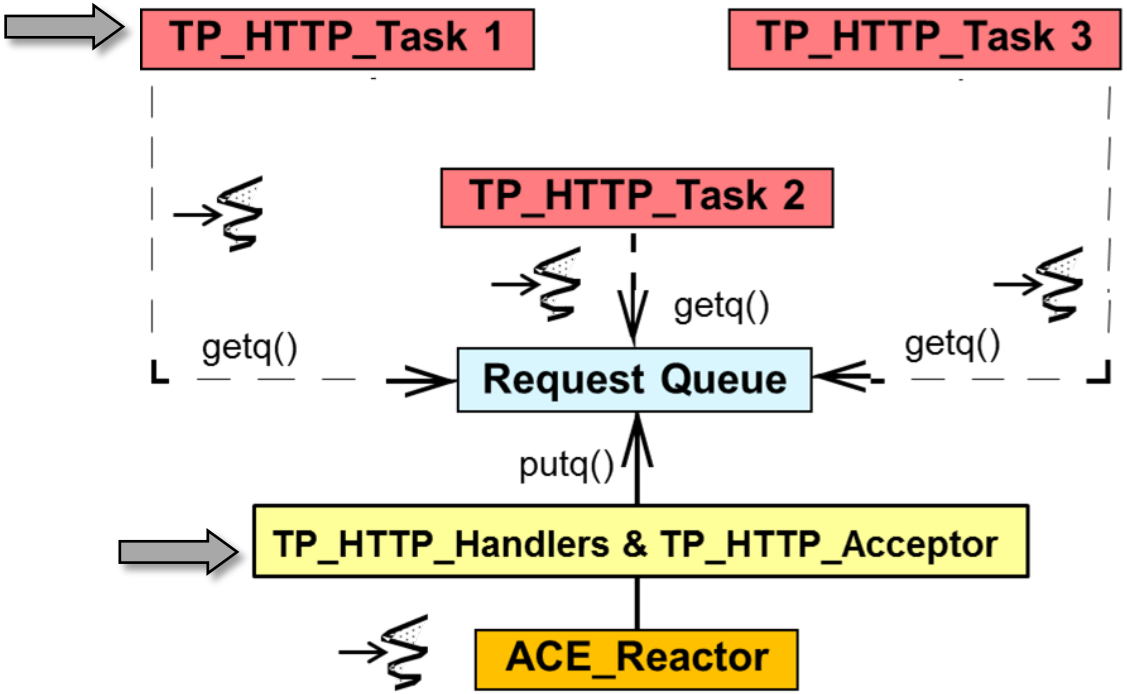
Context	Problem
• Web servers that run on multi-core & -CPU platforms	• *Half-Sync/Half-Async* can incur excessive memory management, synchronization, context switching, & data movement overhead in some settings



Enhancing Predictability with Leader/Followers

Context	Problem
• Web servers that run on multi-core & -CPU platforms	• *Half-Sync/Half-Async* can incur excessive memory management, synchronization, context switching, & data movement overhead in some settings

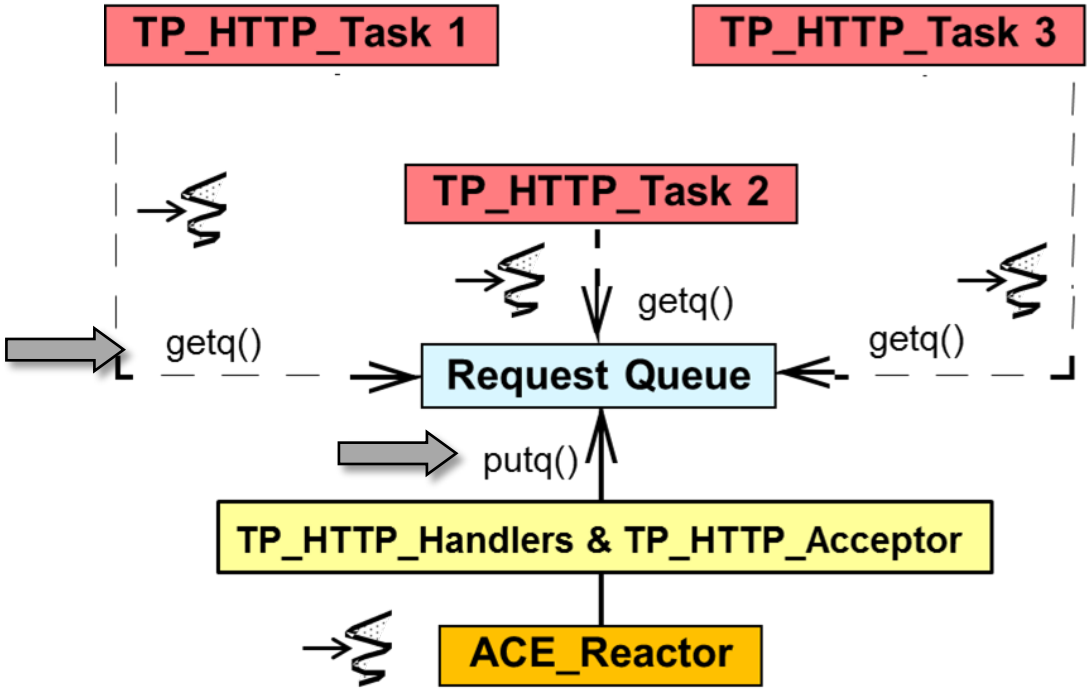
- Dynamic memory (de)allocation



Enhancing Predictability with Leader/Followers

Context	Problem
• Web servers that run on multi-core & -CPU platforms	• *Half-Sync/Half-Async* can incur excessive memory management, synchronization, context switching, & data movement overhead in some settings

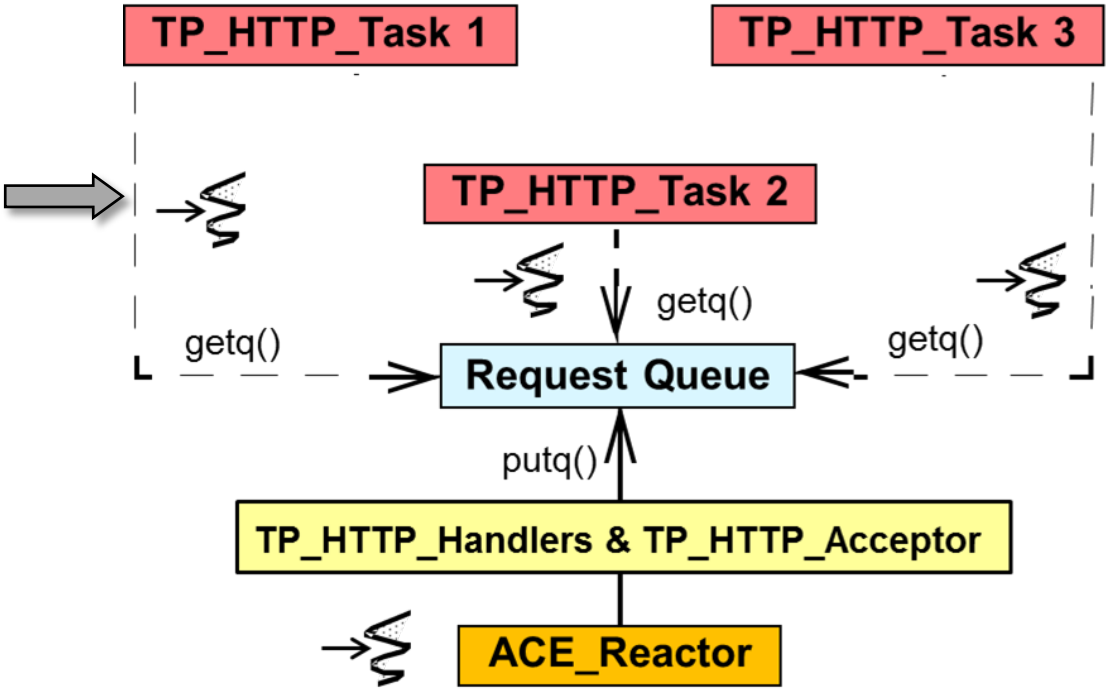
- Dynamic memory (de)allocation
- Synchronization operations



Enhancing Predictability with Leader/Followers

Context	Problem
• Web servers that run on multi-core & -CPU platforms	• *Half-Sync/Half-Async* can incur excessive memory management, synchronization, context switching, & data movement overhead in some settings

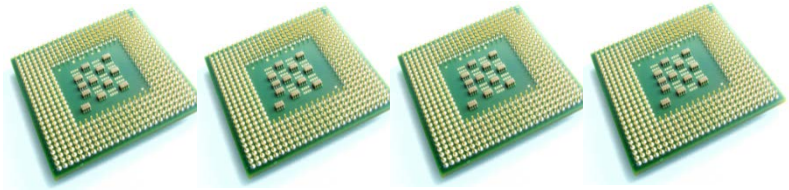
- Dynamic memory (de)allocation
- Synchronization operations
- Context switches

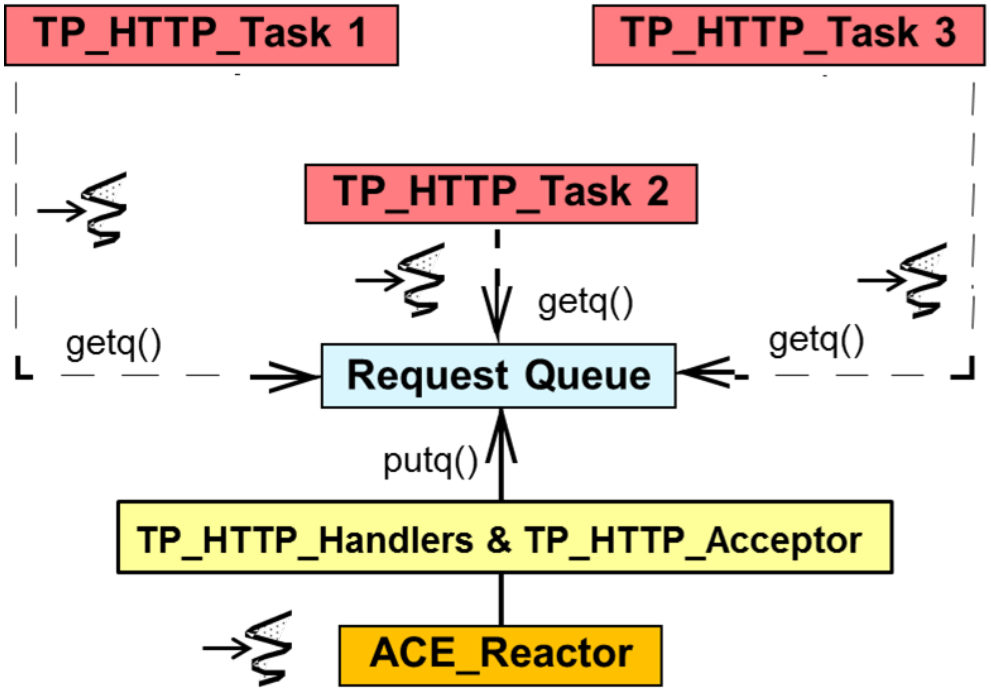


Enhancing Predictability with Leader/Followers

Context	Problem
• Web servers that run on multi-core & -CPU platforms	• *Half-Sync/Half-Async* can incur excessive memory management, synchronization, context switching, & data movement overhead in some settings

- Dynamic memory (de)allocation
- Synchronization operations
- Context switches
- CPU cache updates



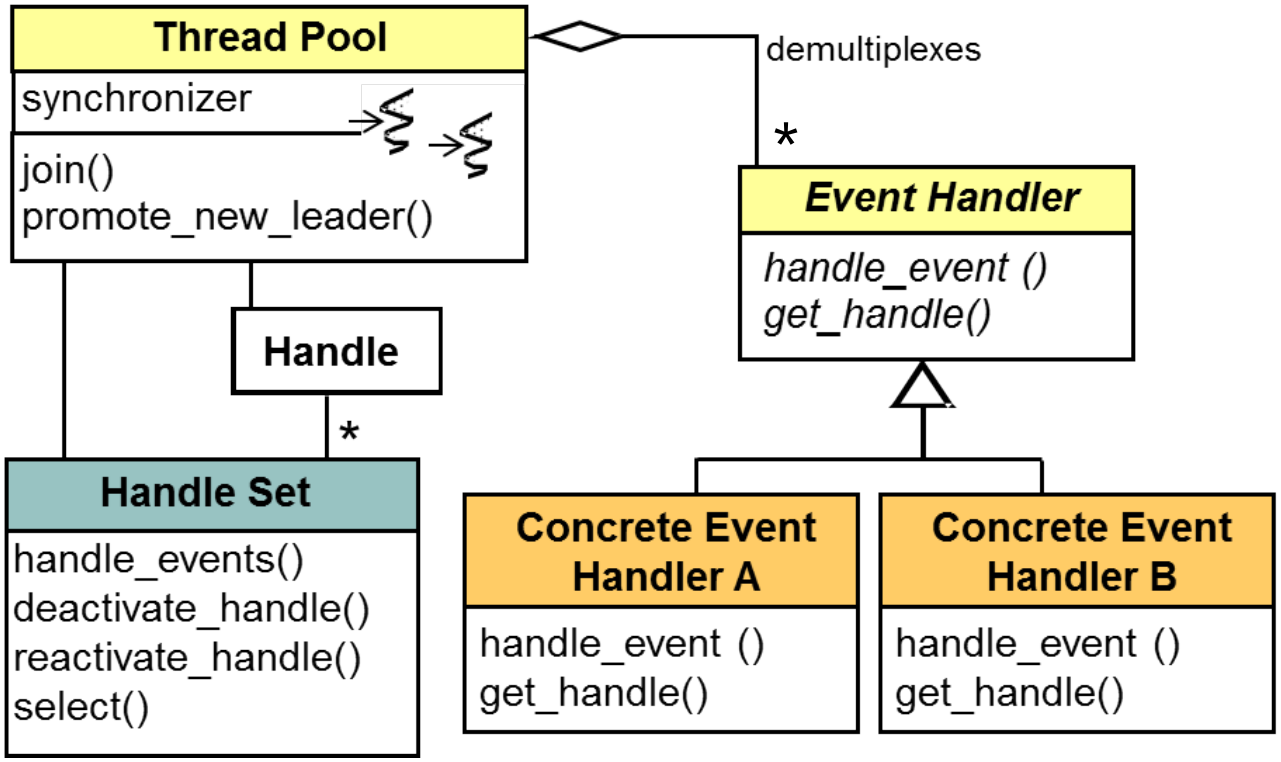


Enhancing Predictability with Leader/Followers

Context	Problem	Solution
• Web servers that run on multi-core & -CPU platforms	• *Half-Sync/Half-Async* can incur excessive memory management, synchronization, context switching, & data movement overhead in some settings	• Apply the ***Leader/ Followers*** pattern to reduce thread pool overhead & jitter

Structure

Leader/Follower provides an efficient & predictable concurrency model where multiple threads take turns sharing event sources to process service requests that occur on the event sources

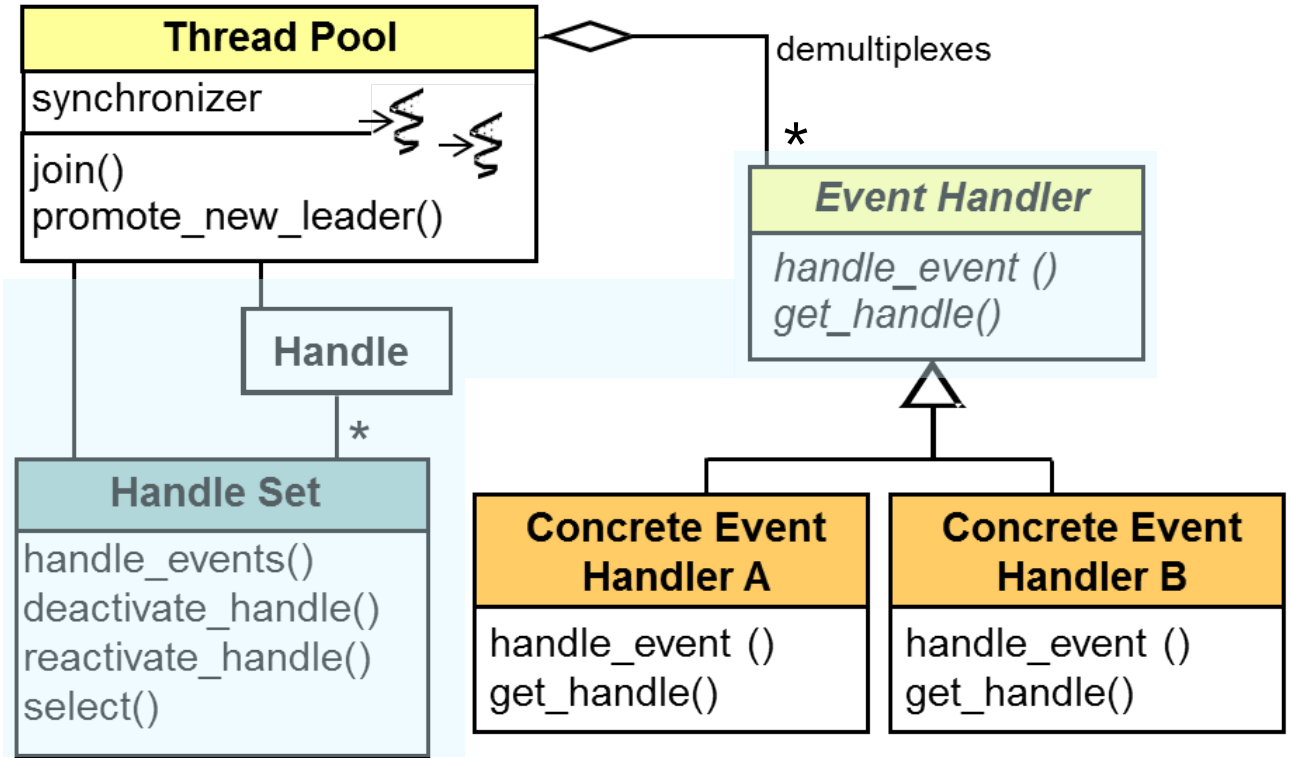


Enhancing Predictability with Leader/Followers

Context	Problem	Solution
• Web servers that run on multi-core & -CPU platforms	• *Half-Sync/Half-Async* can incur excessive memory management, synchronization, context switching, & data movement overhead in some settings	• Apply the ***Leader/ Followers*** pattern to reduce thread pool overhead & jitter

Structure

Leader/Follower provides an efficient & predictable concurrency model where multiple threads take turns sharing event sources to process service requests that occur on the event sources

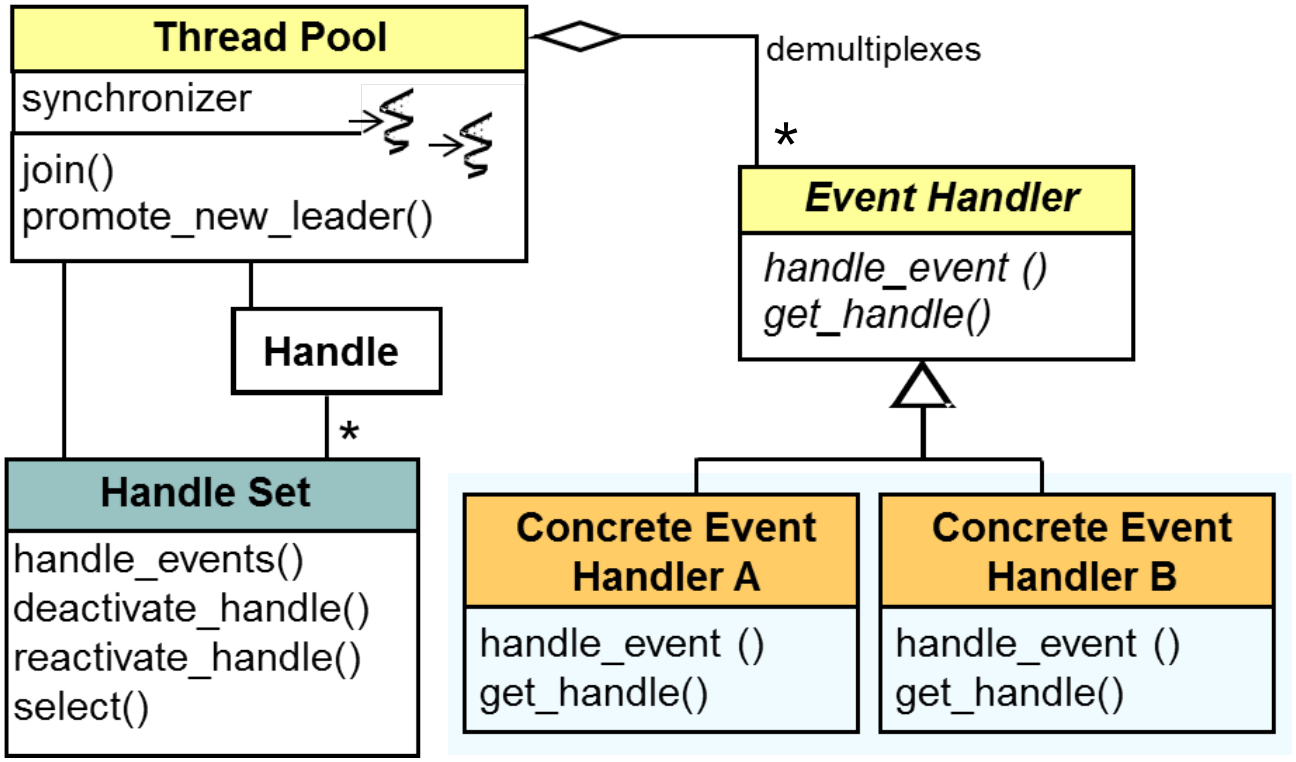


Enhancing Predictability with Leader/Followers

Context	Problem	Solution
• Web servers that run on multi-core & -CPU platforms	• *Half-Sync/Half-Async* can incur excessive memory management, synchronization, context switching, & data movement overhead in some settings	• Apply the ***Leader/ Followers*** pattern to reduce thread pool overhead & jitter

Structure

Leader/Follower provides an efficient & predictable concurrency model where multiple threads take turns sharing event sources to process service requests that occur on the event sources

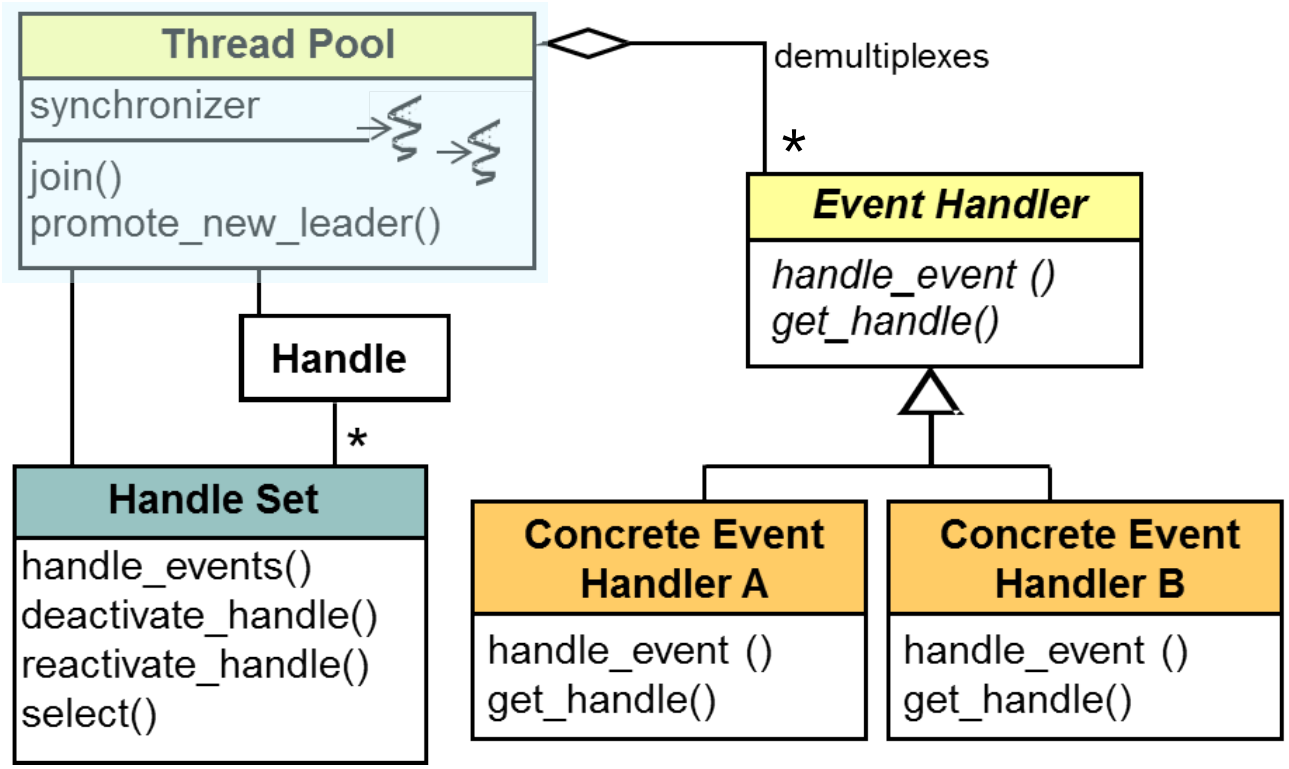


Enhancing Predictability with Leader/Followers

Context	Problem	Solution
• Web servers that run on multi-core & -CPU platforms	• *Half-Sync/Half-Async* can incur excessive memory management, synchronization, context switching, & data movement overhead in some settings	• Apply the ***Leader/ Followers*** pattern to reduce thread pool overhead & jitter

Structure

Leader/Follower provides an efficient & predictable concurrency model where multiple threads take turns sharing event sources to process service requests that occur on the event sources

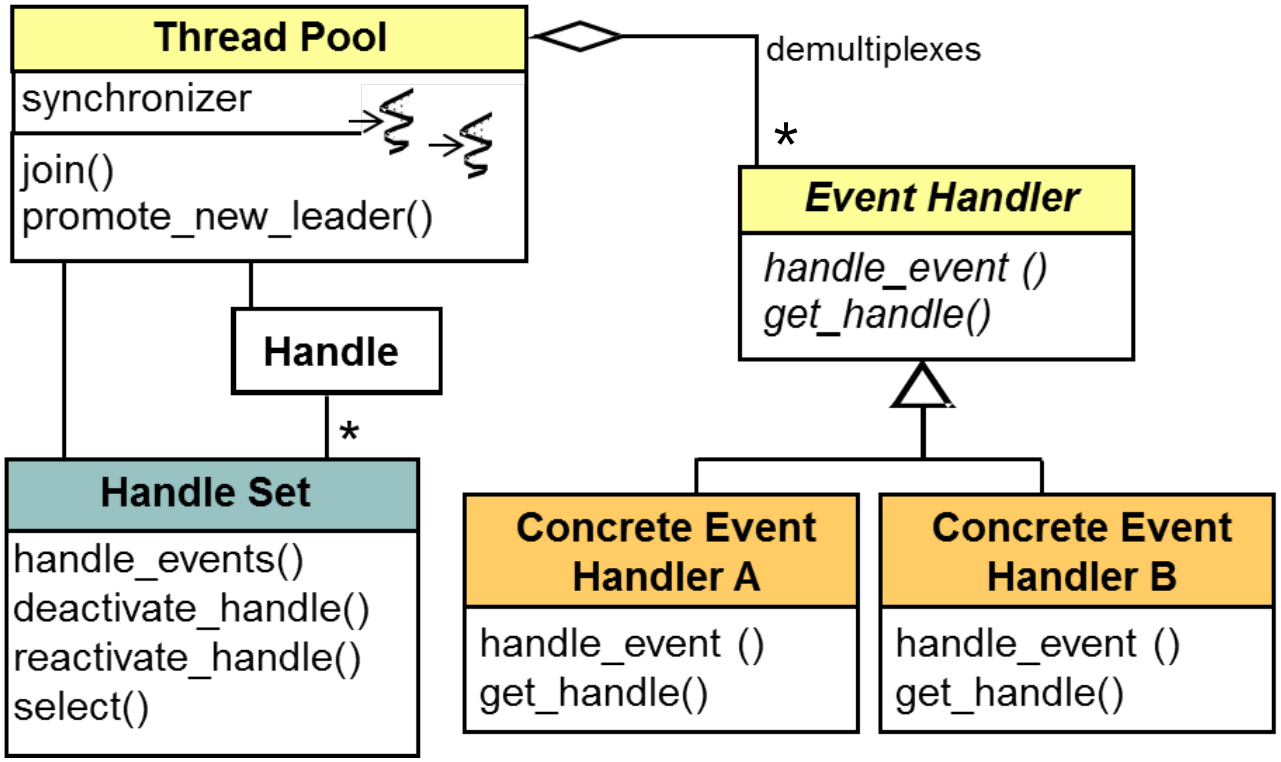


Enhancing Predictability with Leader/Followers

Context	Problem	Solution
• Web servers that run on multi-core & -CPU platforms	• *Half-Sync/Half-Async* can incur excessive memory management, synchronization, context switching, & data movement overhead in some settings	• Apply the ***Leader/ Followers*** pattern to reduce thread pool overhead & jitter

Structure

Leader/Follower provides an efficient & predictable concurrency model where multiple threads take turns sharing event sources to process service requests that occur on the event sources

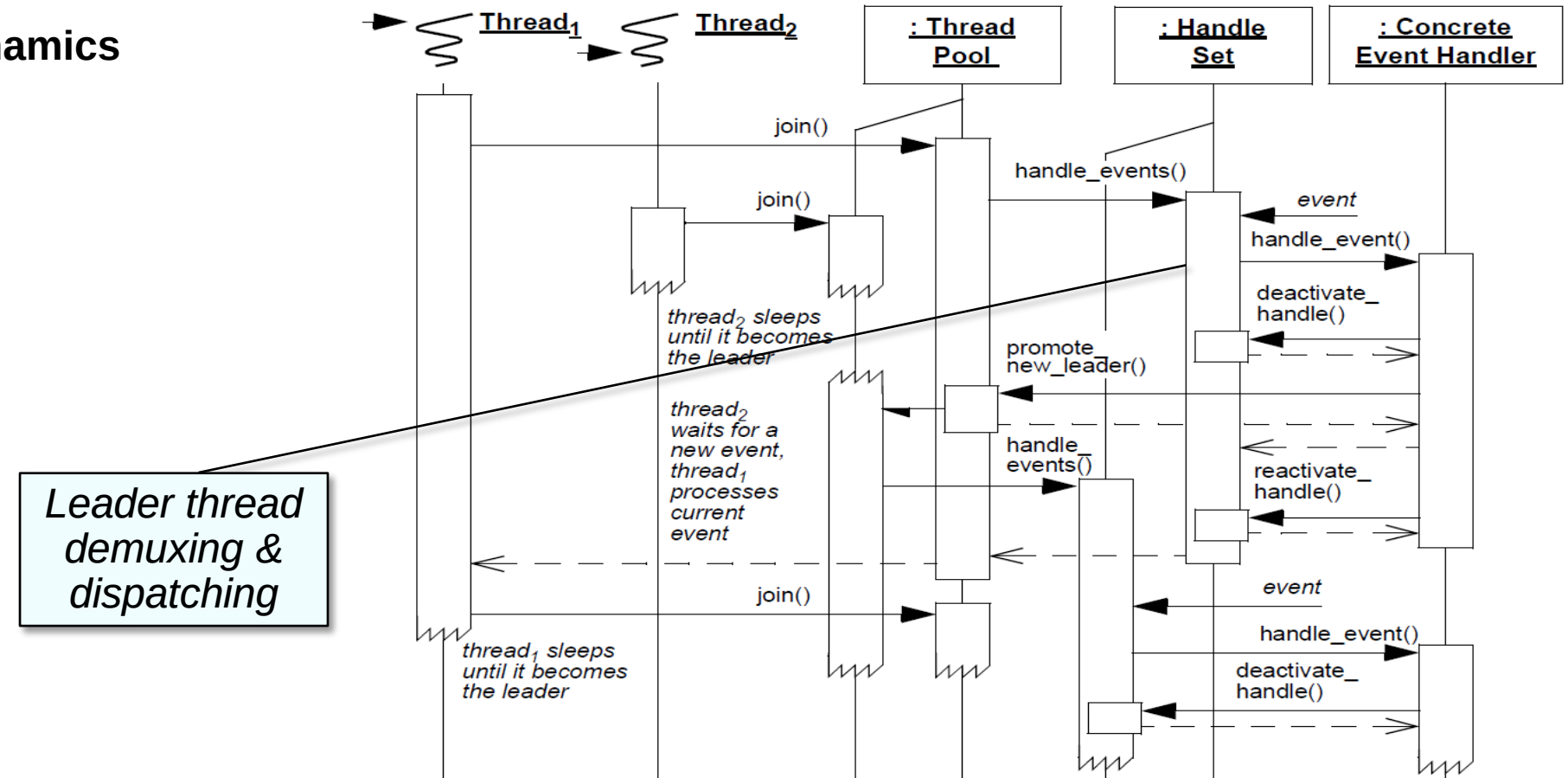


See www.dre.vanderbilt.edu/~schmidt/PDF/LF.pdf for more info on this pattern

Enhancing Predictability with Leader/Followers

Context	Problem	Solution
• Web servers that run on multi-core & -CPU platforms	• *Half-Sync/Half-Async* can incur excessive memory management, synchronization, context switching, & data movement overhead in some settings	• Apply the ***Leader/ Followers*** pattern to reduce thread pool overhead & jitter

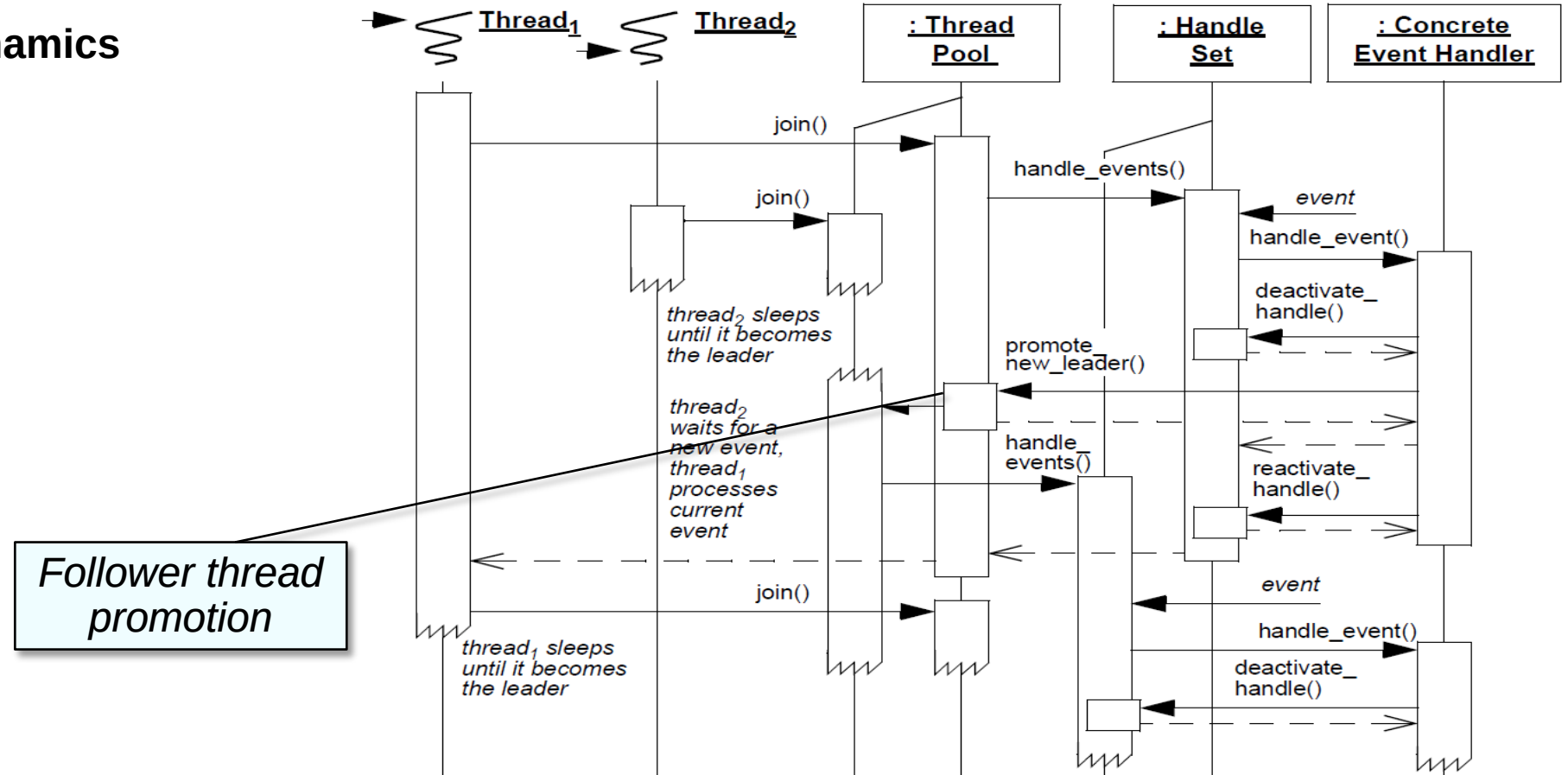
Dynamics



Enhancing Predictability with Leader/Followers

Context	Problem	Solution
• Web servers that run on multi-core & -CPU platforms	• *Half-Sync/Half-Async* can incur excessive memory management, synchronization, context switching, & data movement overhead in some settings	• Apply the ***Leader/ Followers*** pattern to reduce thread pool overhead & jitter

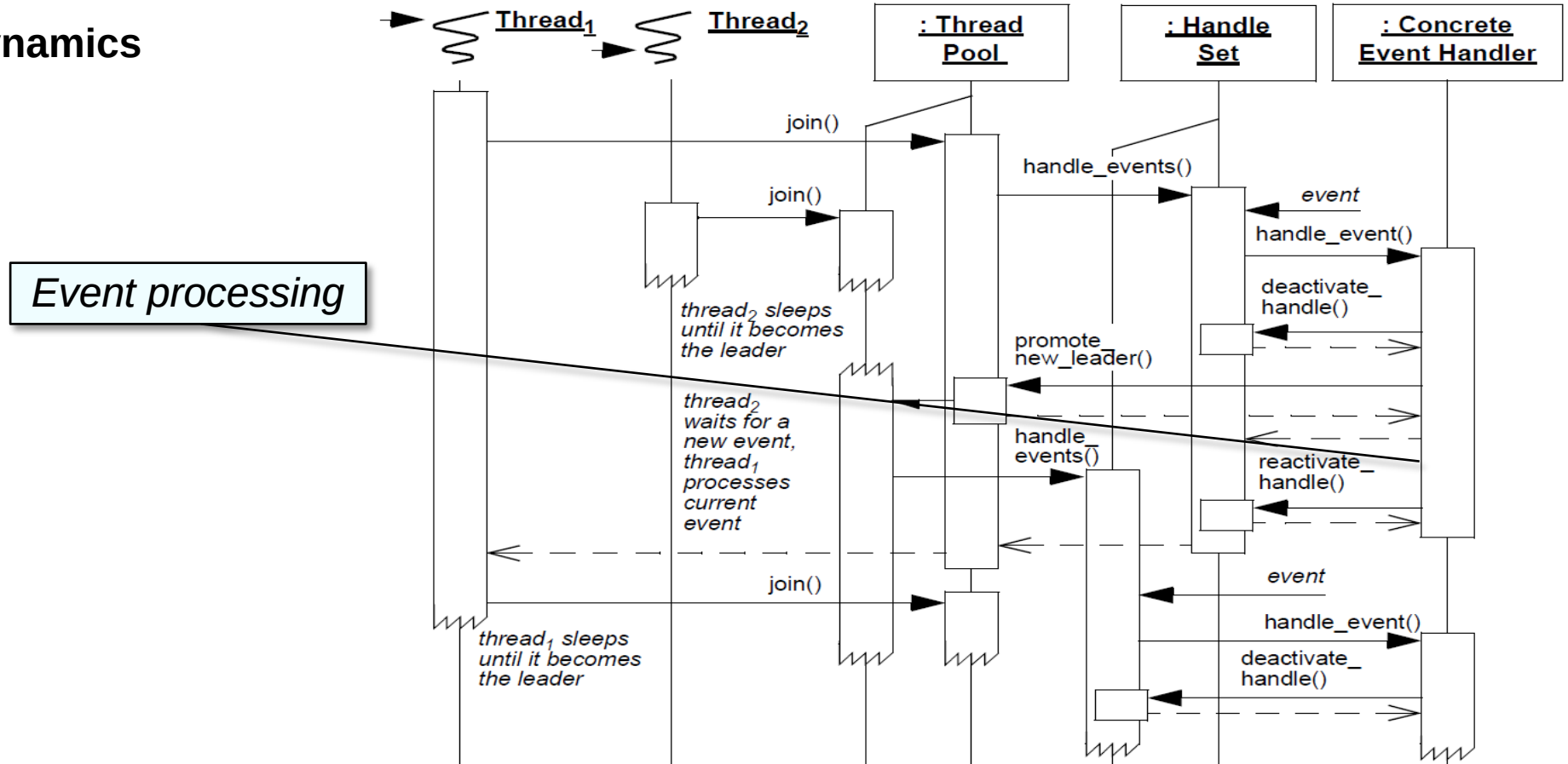
Dynamics



Enhancing Predictability with Leader/Followers

Context	Problem	Solution
• Web servers that run on multi-core & -CPU platforms	• *Half-Sync/Half-Async* can incur excessive memory management, synchronization, context switching, & data movement overhead in some settings	• Apply the ***Leader/ Followers*** pattern to reduce thread pool overhead & jitter

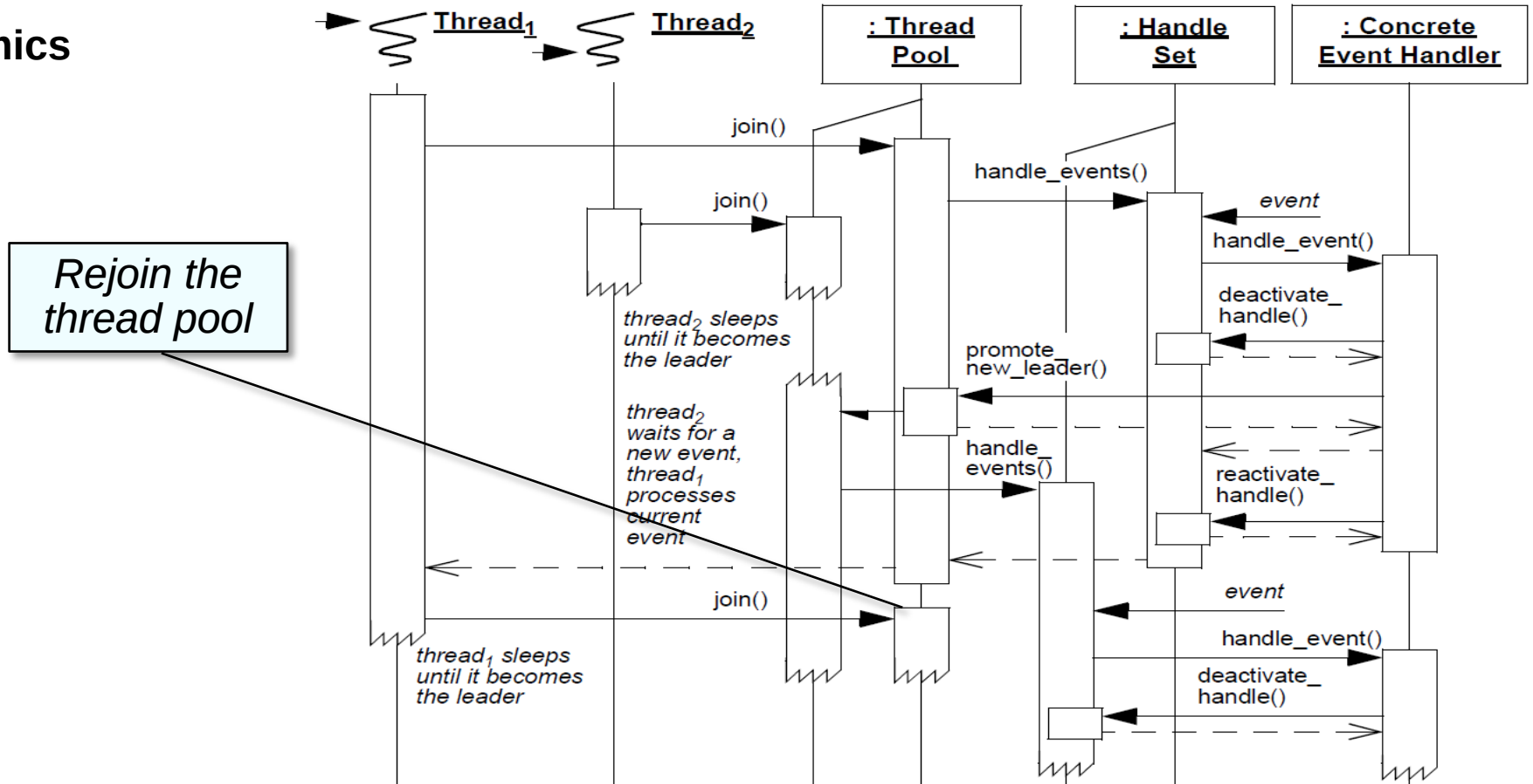
Dynamics



Enhancing Predictability with Leader/Followers

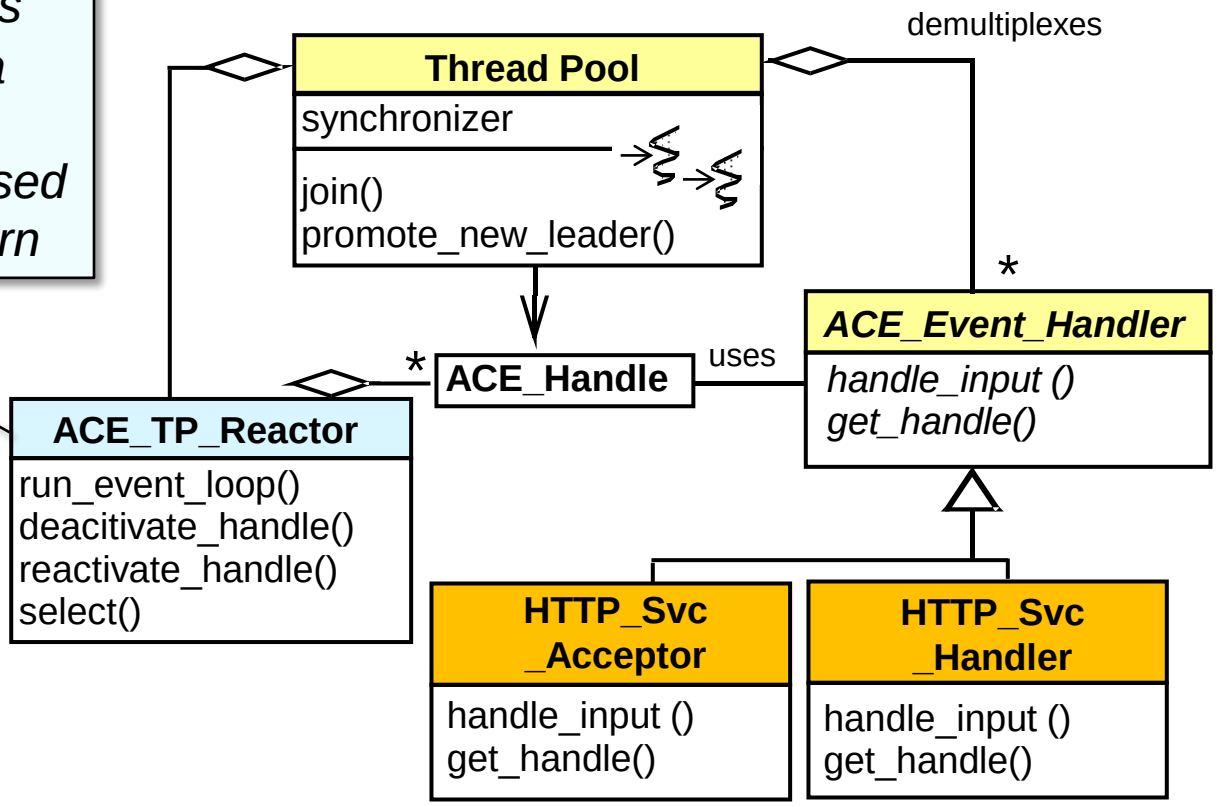
Context	Problem	Solution
• Web servers that run on multi-core & -CPU platforms	• *Half-Sync/Half-Async* can incur excessive memory management, synchronization, context switching, & data movement overhead in some settings	• Apply the ***Leader/ Followers*** pattern to reduce thread pool overhead & jitter

Dynamics



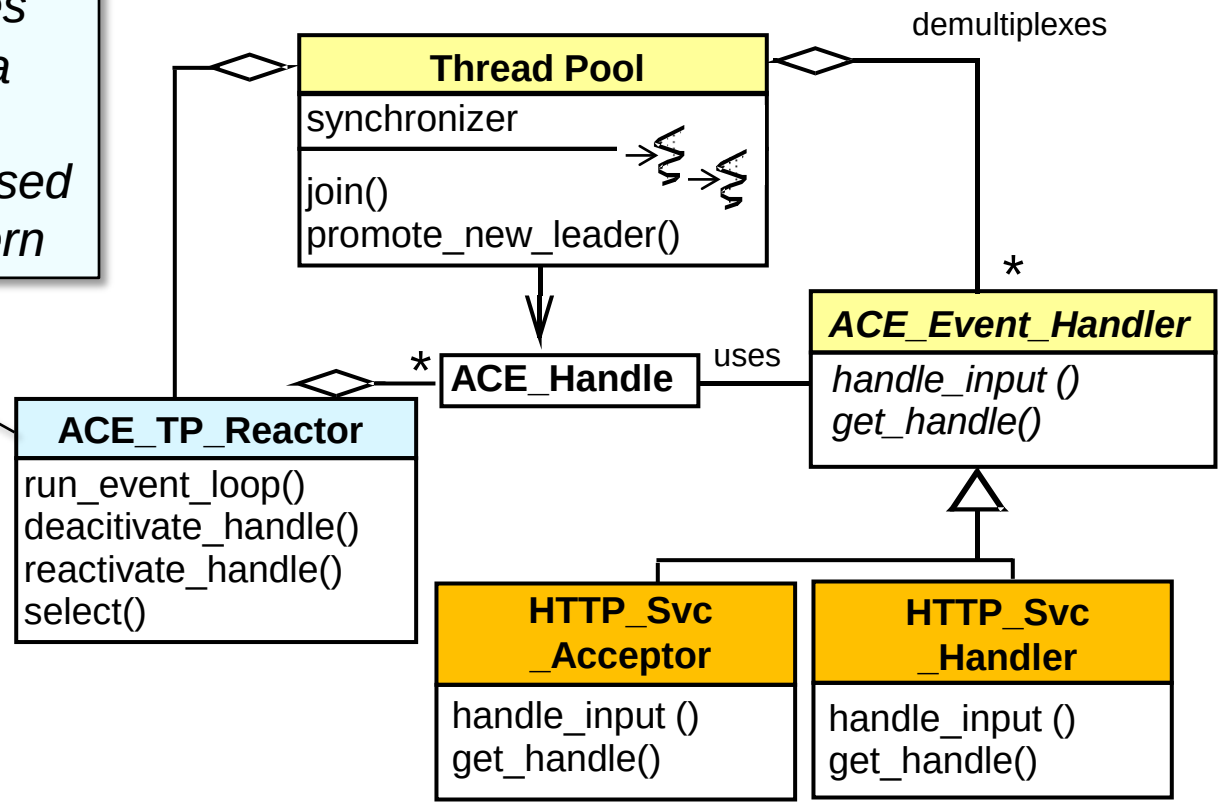
Applying Leader/Followers Pattern in JAWS

ACE_TP_Reactor *eliminates need for—& overhead of—a separate Reactor thread & synchronized request queue used in Half-Sync/Half-Async pattern*



Applying Leader/Followers Pattern in JAWS

ACE_TP_Reactor *eliminates need for—& overhead of—a separate Reactor thread & synchronized request queue used in Half-Sync/Half-Async pattern*



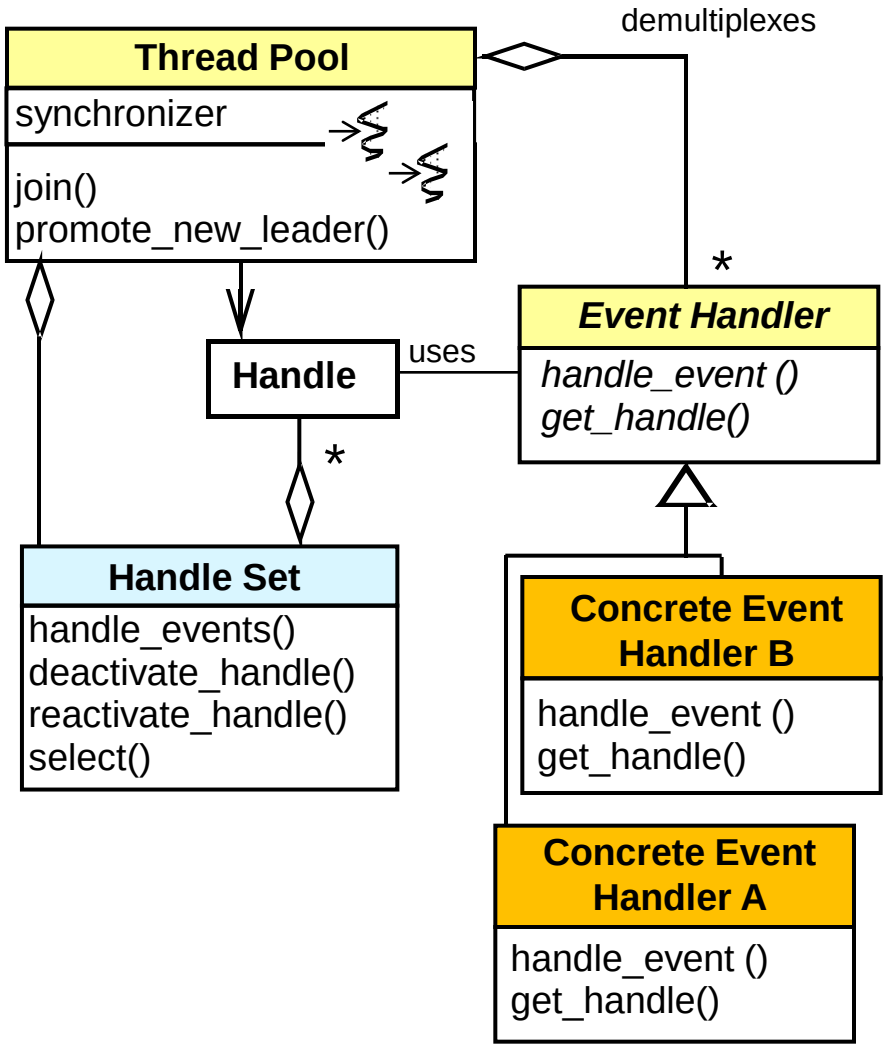
Half-Sync/Half-Async design may still be more appropriate for certain types of servers, however, because

- It can reorder & prioritize client requests more flexibly via its request queue
- It may be more scalable since it queues requests in virtual memory

Benefits of the Leader/Followers Pattern

Performance enhancements

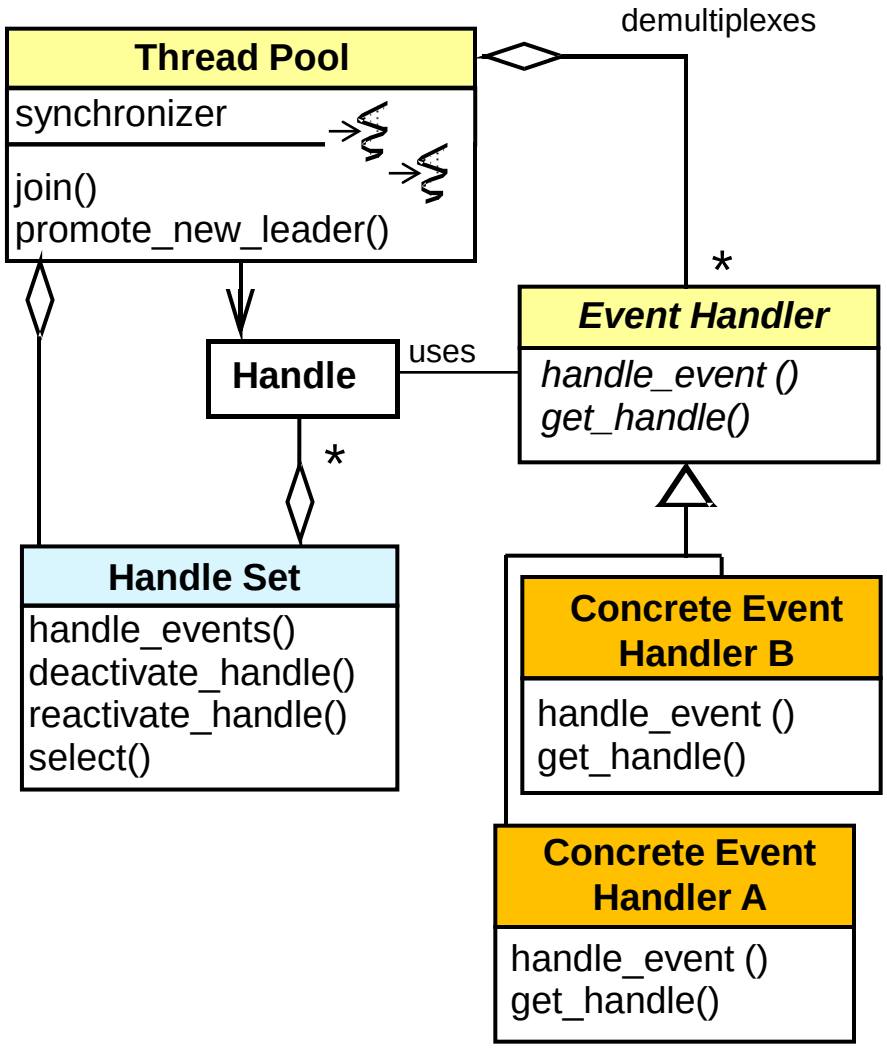
- Improves CPU cache affinity & removes need for dynamic memory allocation & data buffer sharing between threads



Benefits of the Leader/Followers Pattern

Performance enhancements

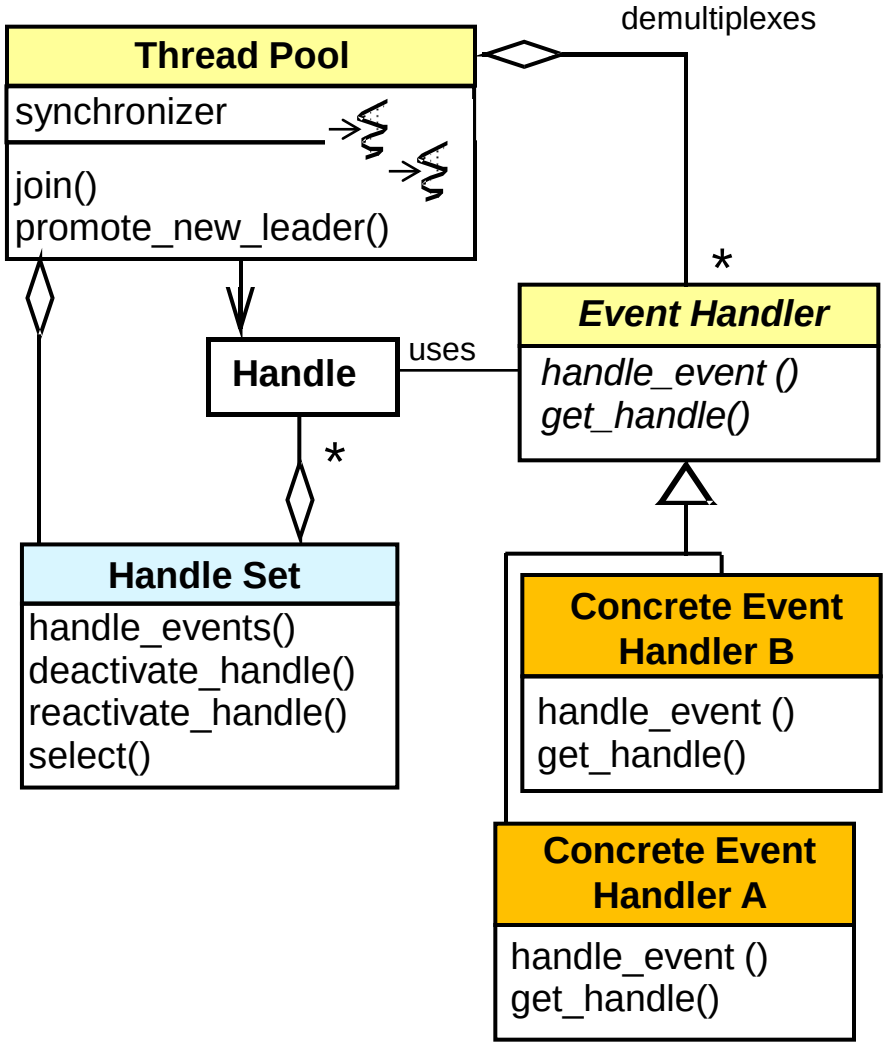
- Improves CPU cache affinity & removes need for dynamic memory allocation & data buffer sharing between threads
- Minimizes locking overhead by not exchanging data between threads, thus reducing thread synchronization



Benefits of the Leader/Followers Pattern

Performance enhancements

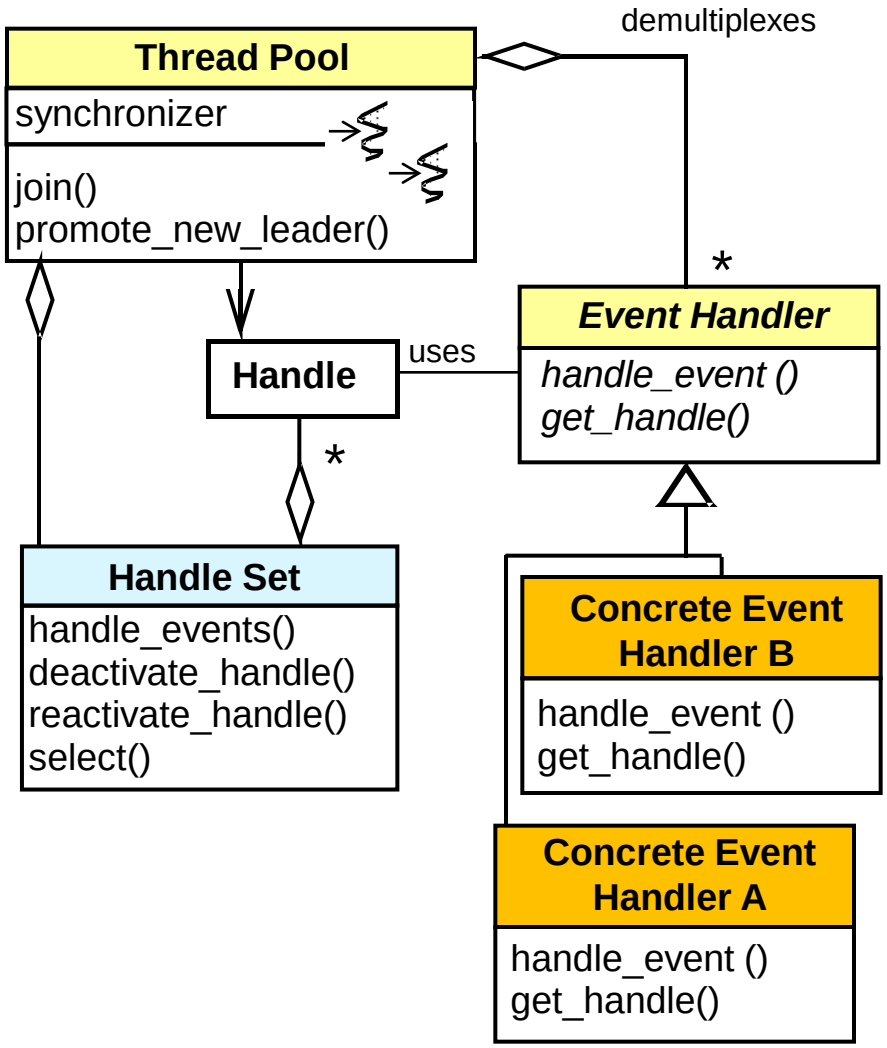
- Improves CPU cache affinity & removes need for dynamic memory allocation & data buffer sharing between threads
- Minimizes locking overhead by not exchanging data between threads, thus reducing thread synchronization
- Minimizes priority inversion because no extra queueing is used in the server



Benefits of the Leader/Followers Pattern

Performance enhancements

- Improves CPU cache affinity & removes need for dynamic memory allocation & data buffer sharing between threads
- Minimizes locking overhead by not exchanging data between threads, thus reducing thread synchronization
- Minimizes priority inversion because no extra queueing is used in the server
- Doesn't require a context switch to handle each event



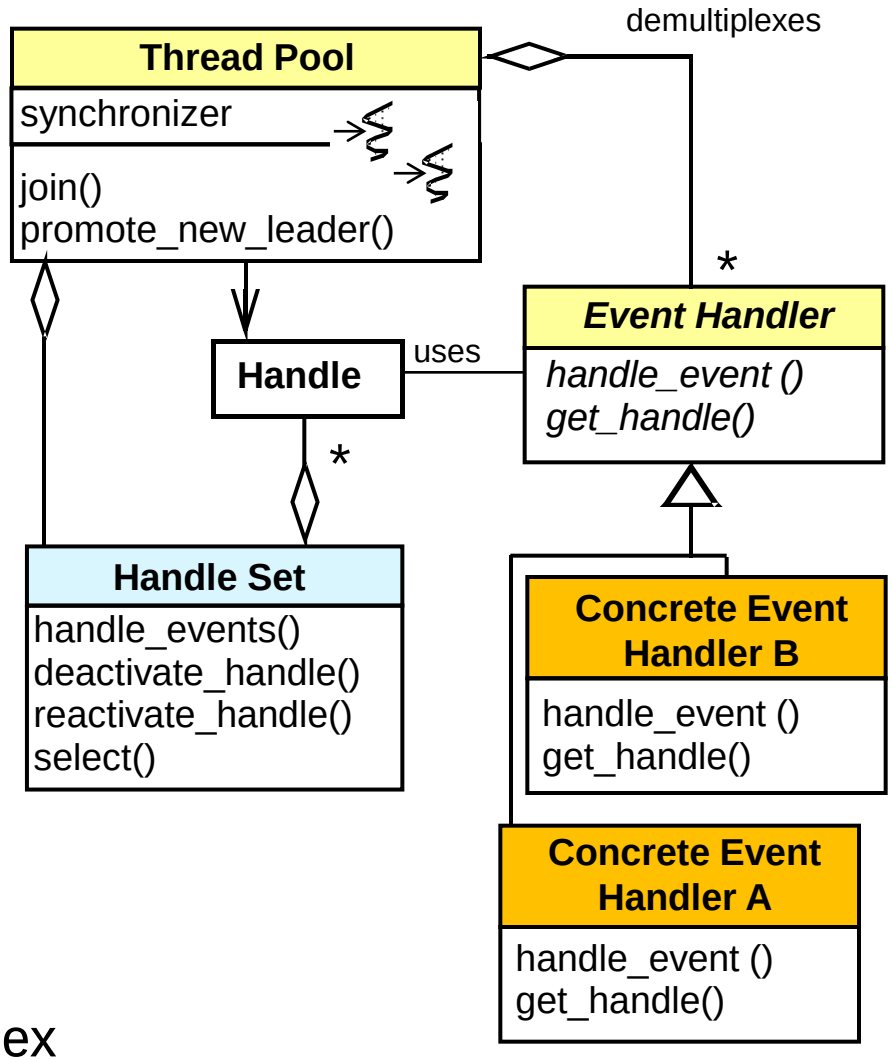
Benefits of the Leader/Followers Pattern

Performance enhancements

- Improves CPU cache affinity & removes need for dynamic memory allocation & data buffer sharing between threads
- Minimizes locking overhead by not exchanging data between threads, thus reducing thread synchronization
- Minimizes priority inversion because no extra queueing is used in the server
- Doesn't require a context switch to handle each event

Programming simplicity

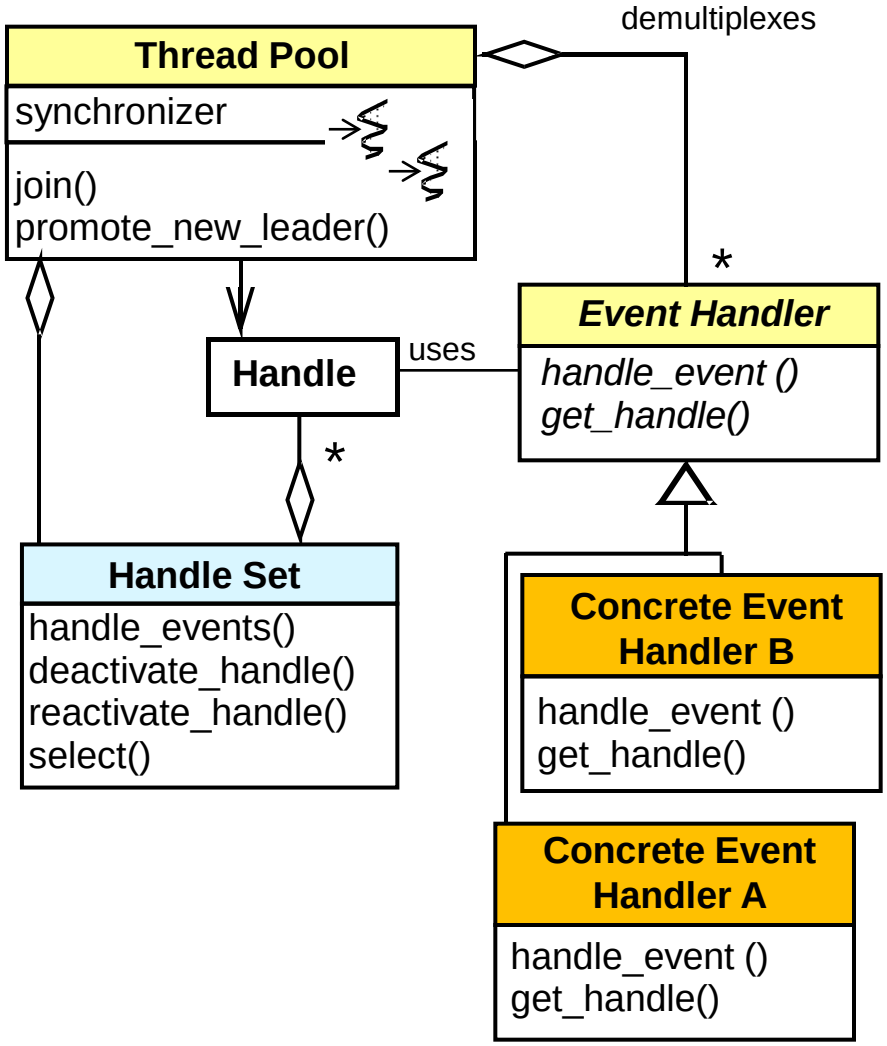
- Simplifies programming of concurrency models where multiple threads receive requests, process responses, & demultiplex connections using a shared handle set



Limitations of the Leader/Followers Pattern

Implementation complexity

- The advanced variants of the pattern are hard to implement



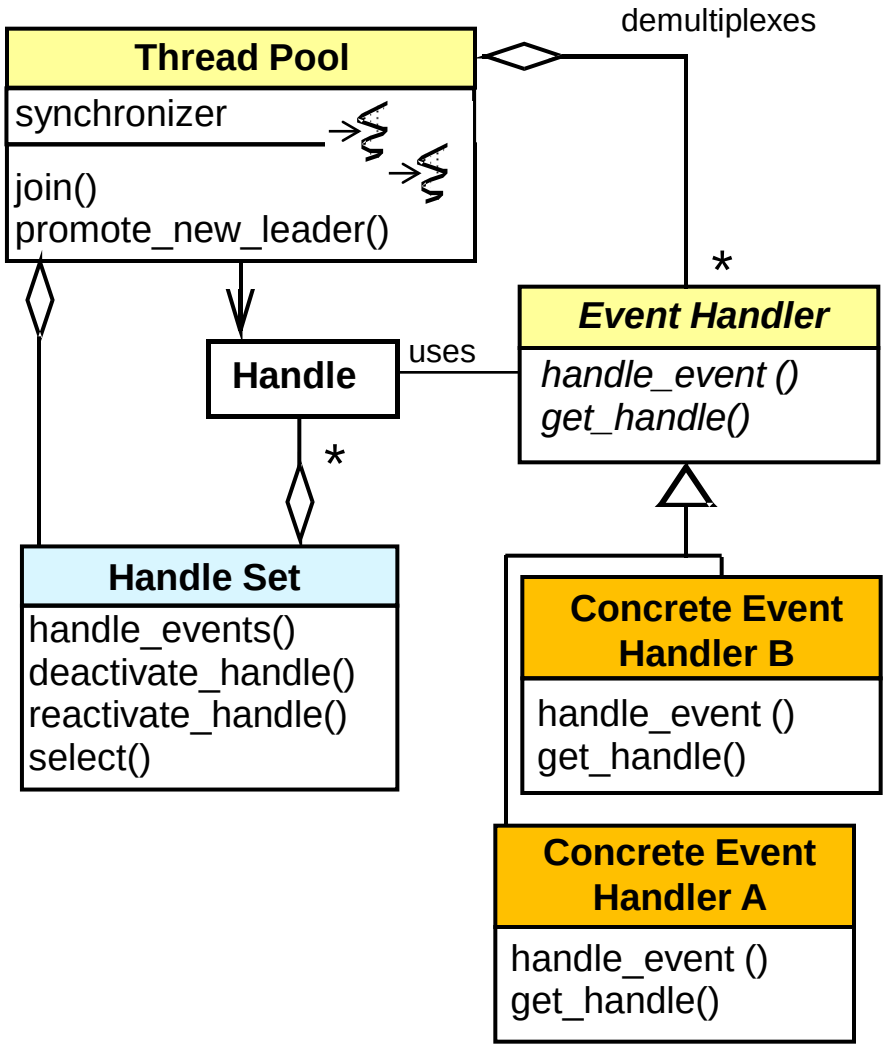
Limitations of the Leader/Followers Pattern

Implementation complexity

- The advanced variants of the pattern are hard to implement

Lack of flexibility

- It is hard to discard or reorder events because there is no explicit queue



Limitations of the Leader/Followers Pattern

Implementation complexity

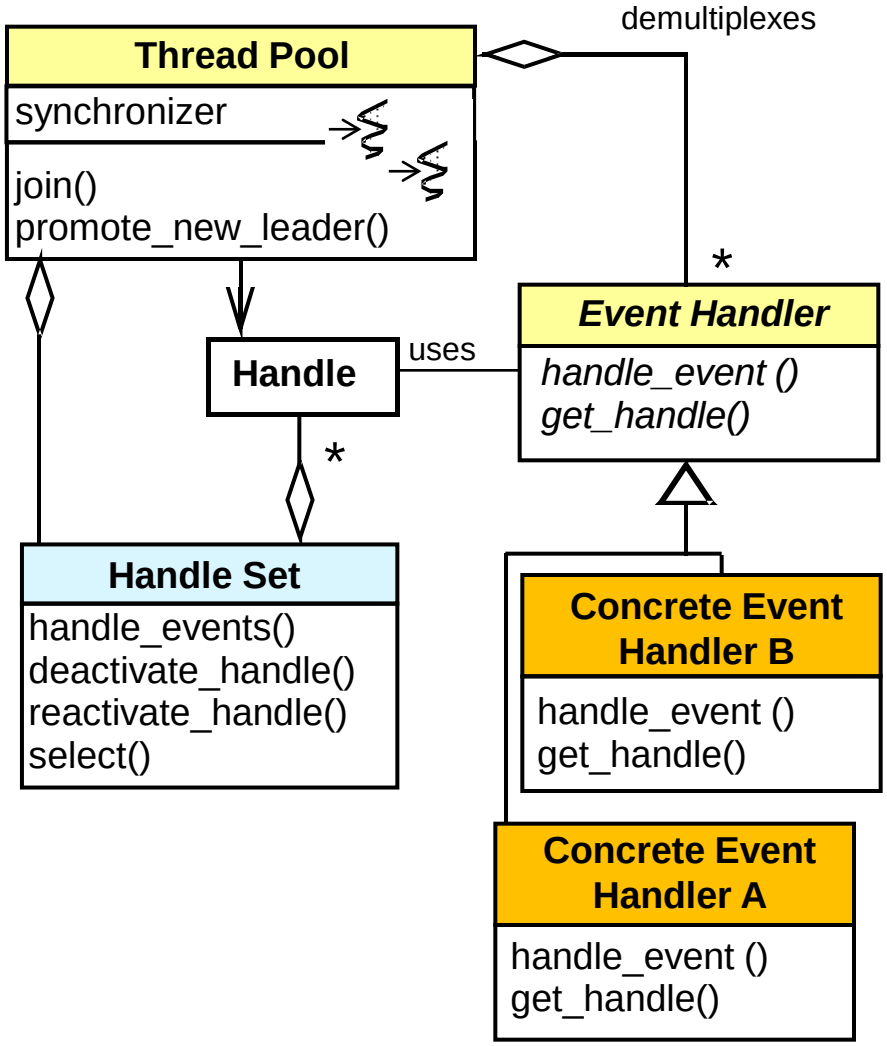
- The advanced variants of the pattern are hard to implement

Lack of flexibility

- It is hard to discard or reorder events because there is no explicit queue

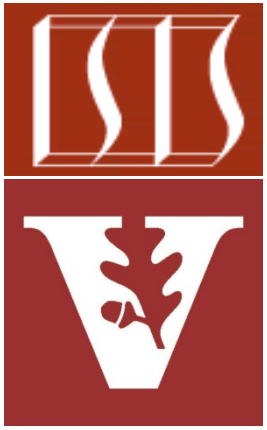
Network I/O bottlenecks

- Serializes processing by allowing only a single thread at a time to wait on the handle set
  - Could become a bottleneck because only one thread at a time (de)muxes I/O events



Patterns & Frameworks for Concurrency & Synchronization: Part 9

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



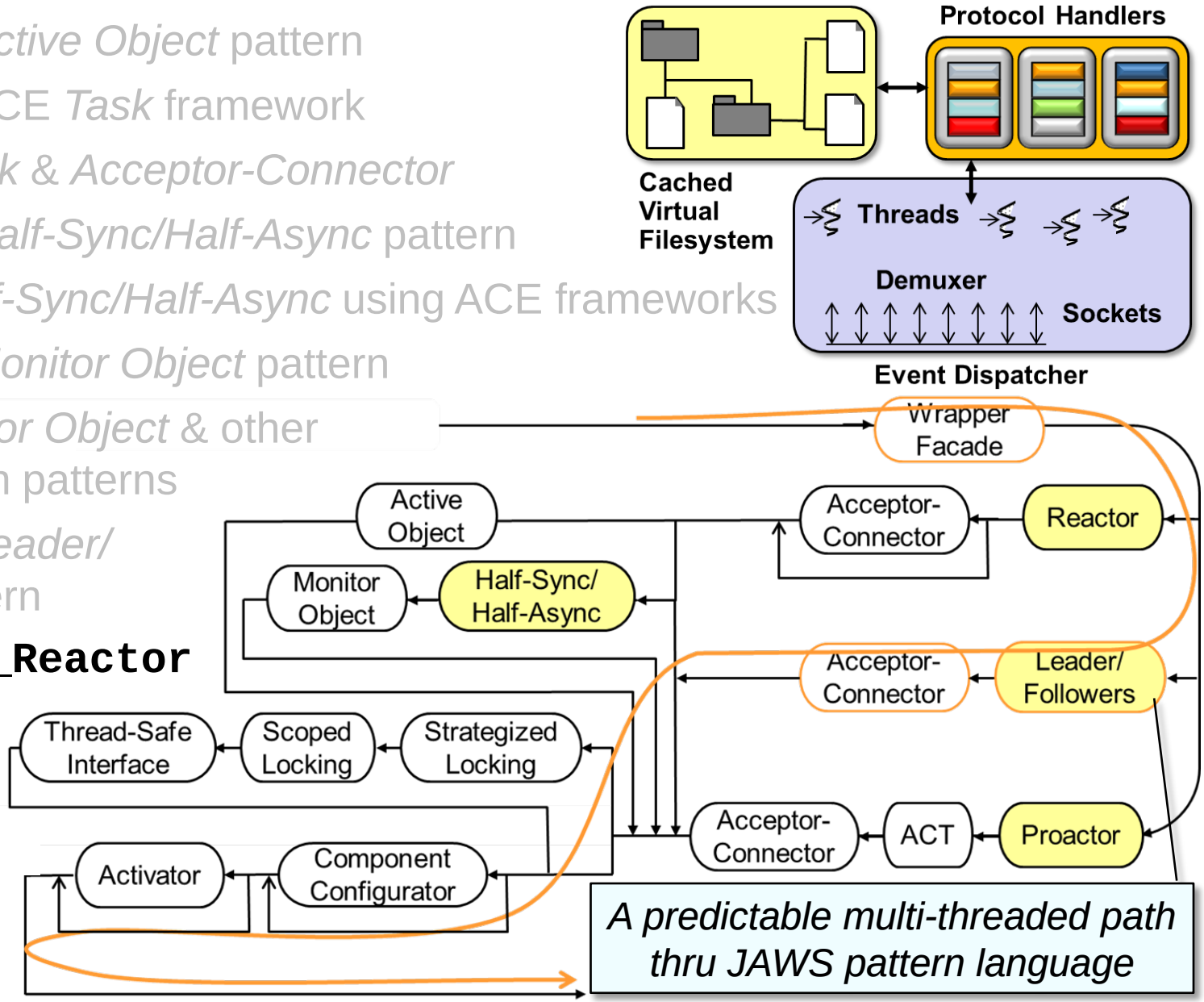
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

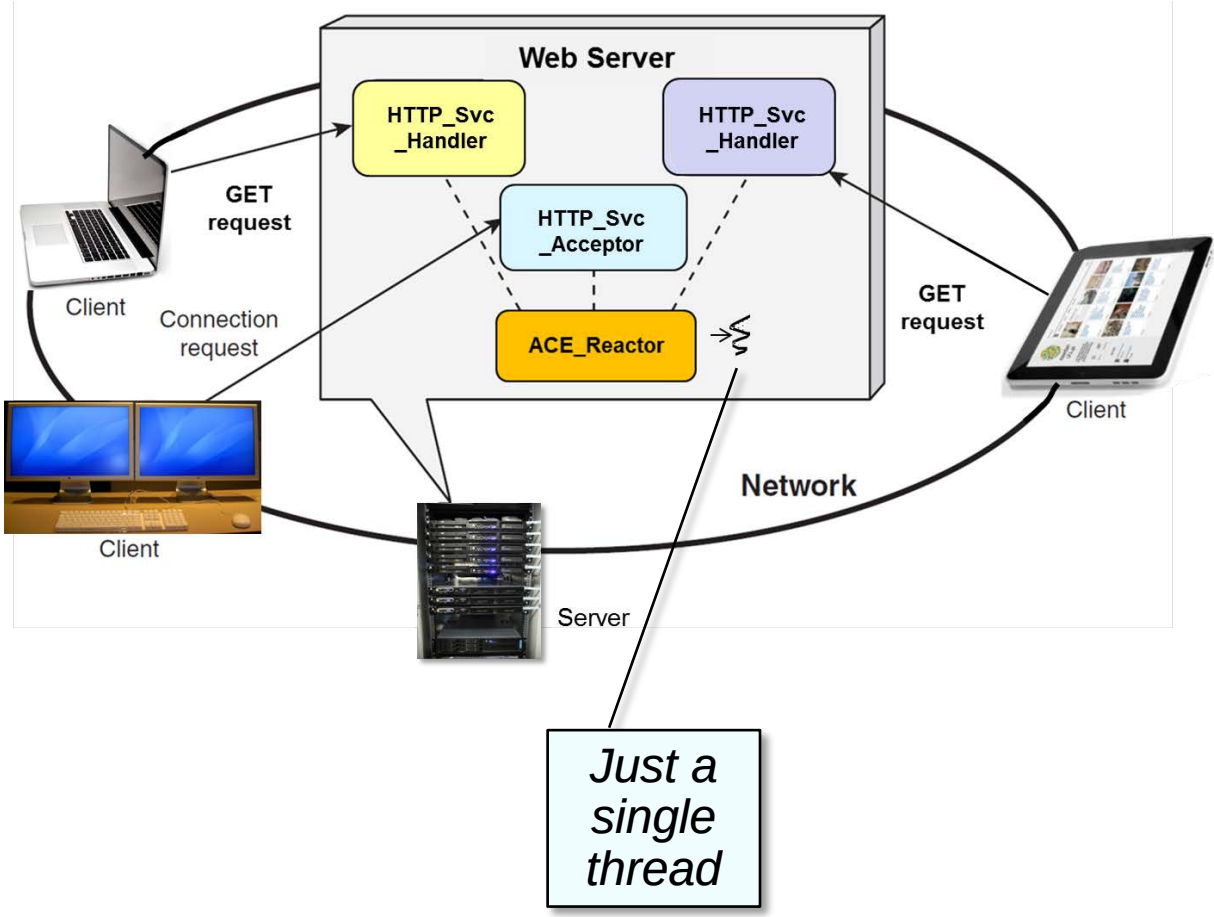
Topics Covered in this Part of the Module

- Describe the *Active Object* pattern
- Describe the ACE *Task* framework
- Apply ACE *Task* & *Acceptor-Connector*
- Describe the *Half-Sync/Half-Async* pattern
- Implement *Half-Sync/Half-Async* using ACE frameworks
- Describe the *Monitor Object* pattern
- Applying *Monitor Object* & other synchronization patterns
- Describe the *Leader/Followers* pattern
- Apply **`ACE_TP_Reactor`** part of *Reactor* framework to JAWS



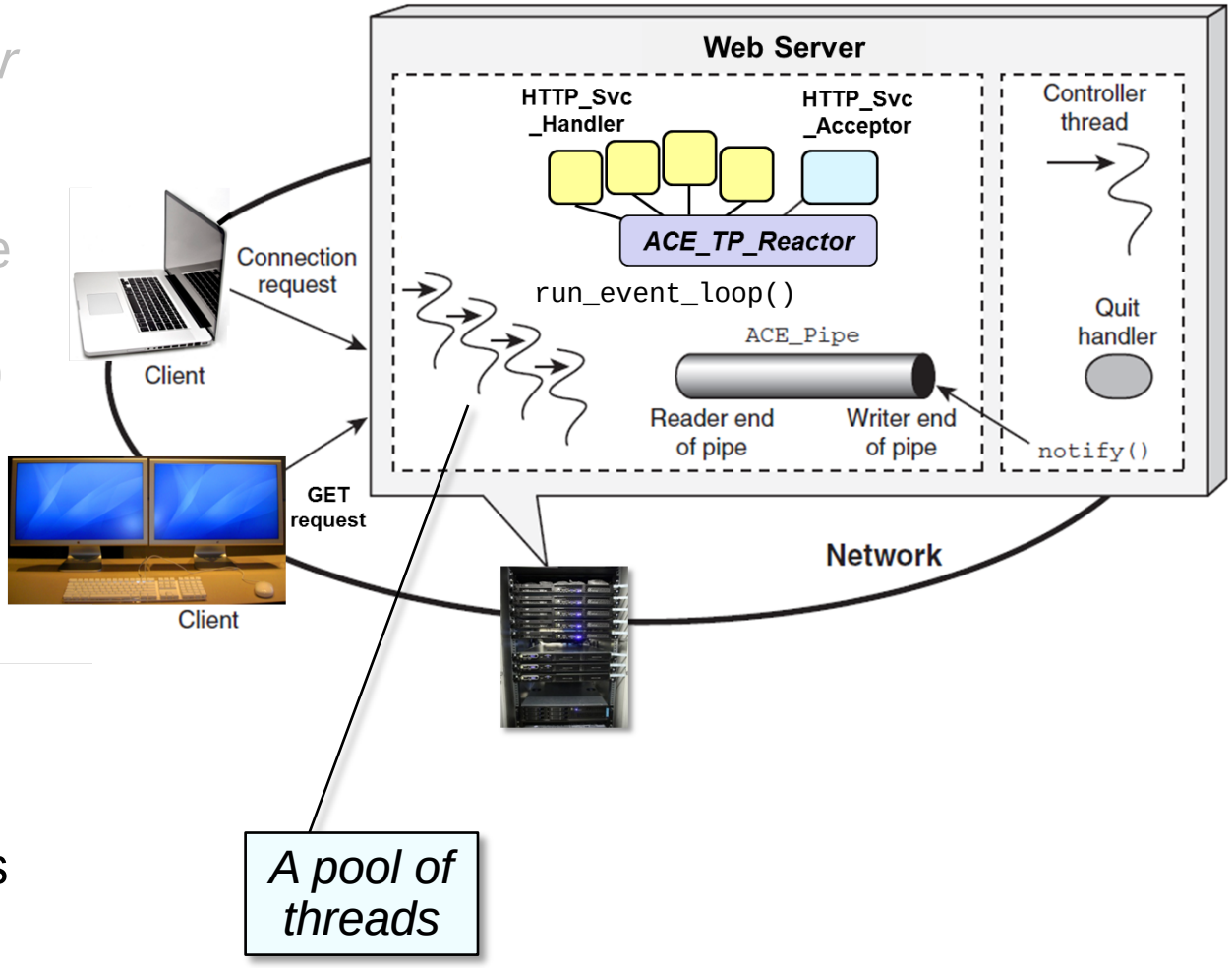
Motivation for the ACE_TP_Reactor Class

- Although the ACE *Reactor* **`ACE_Select_Reactor`** is flexible, it's limited for multi-threaded apps since only its owner thread can call **`run_event_loop()`**



Motivation for the ACE_TP_Reactor Class

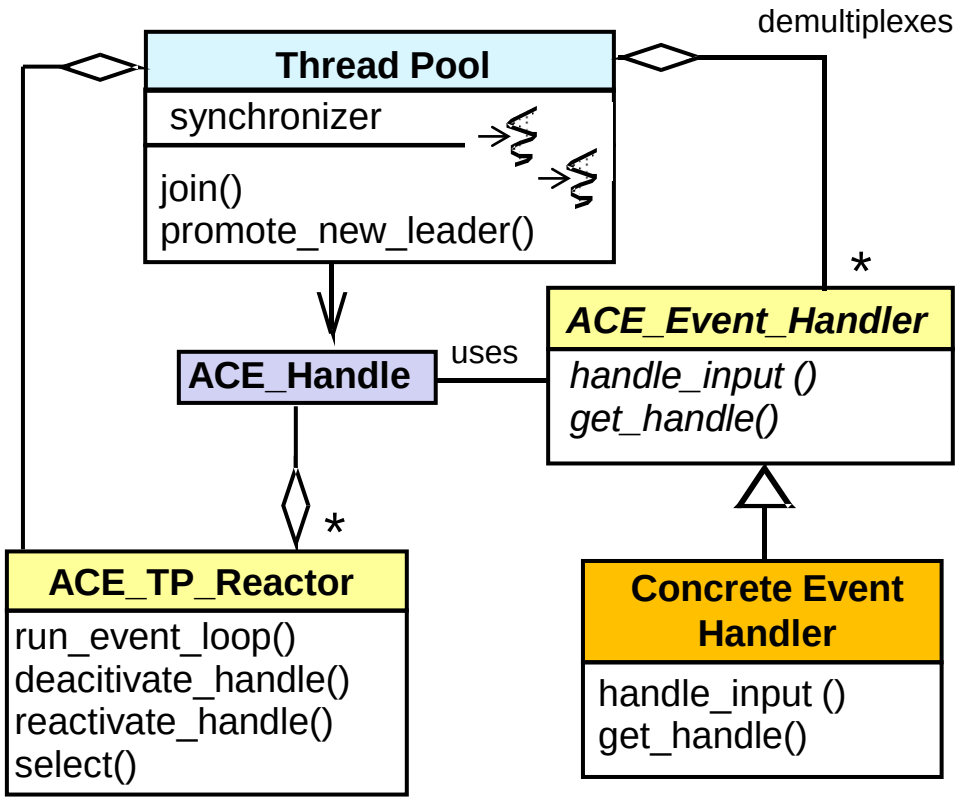
- Although the ACE *Reactor* **`ACE_Select_Reactor`** is flexible, it's limited for multi-threaded apps since only its owner thread can call **`run_event_loop()`**
- The **`ACE_TP_Reactor`** implements the *Leader/Followers* pattern
  - Enables the use of ACE *Reactor* framework in more scalable multi-threaded configurations



The ACE_TP_Reactor Class

Inherits from **`ACE_Select_Reactor`**, implements **`ACE_Reactor`** interface, & applies the *Leader/Followers* pattern to provide the following capabilities:

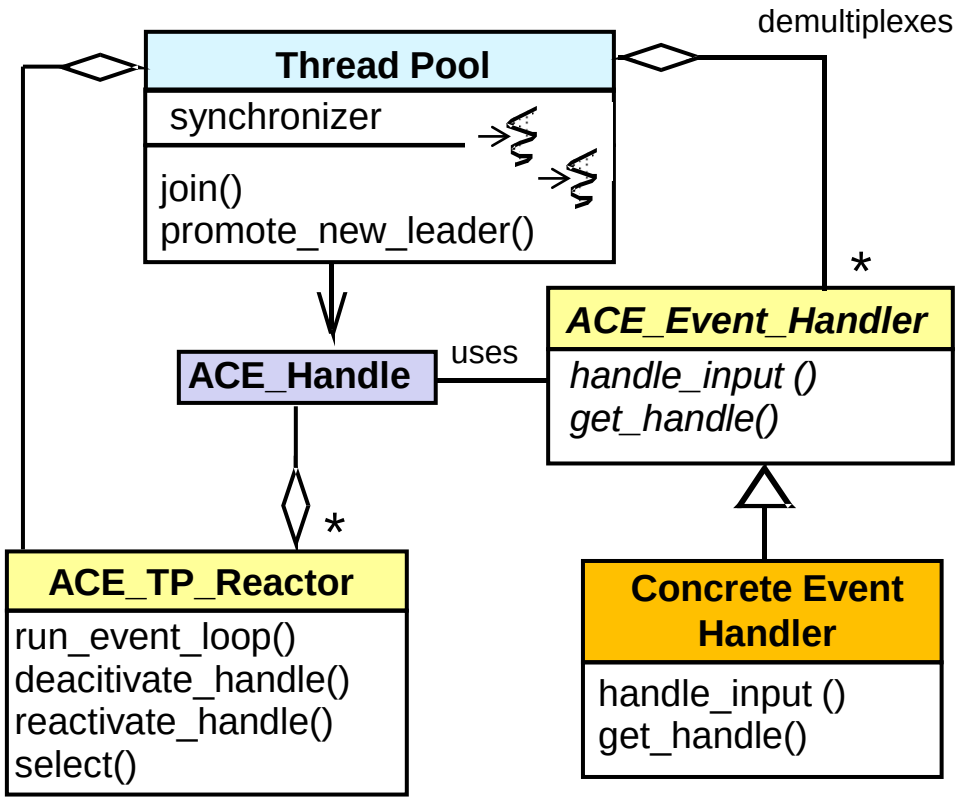
- A pool of threads can call its **`run_event_loop()`** method
  - Improves scalability by handling events on multiple handles concurrently



The ACE_TP_Reactor Class

Inherits from **`ACE_Select_Reactor`**, implements **`ACE_Reactor`** interface, & applies the *Leader/Followers* pattern to provide the following capabilities:

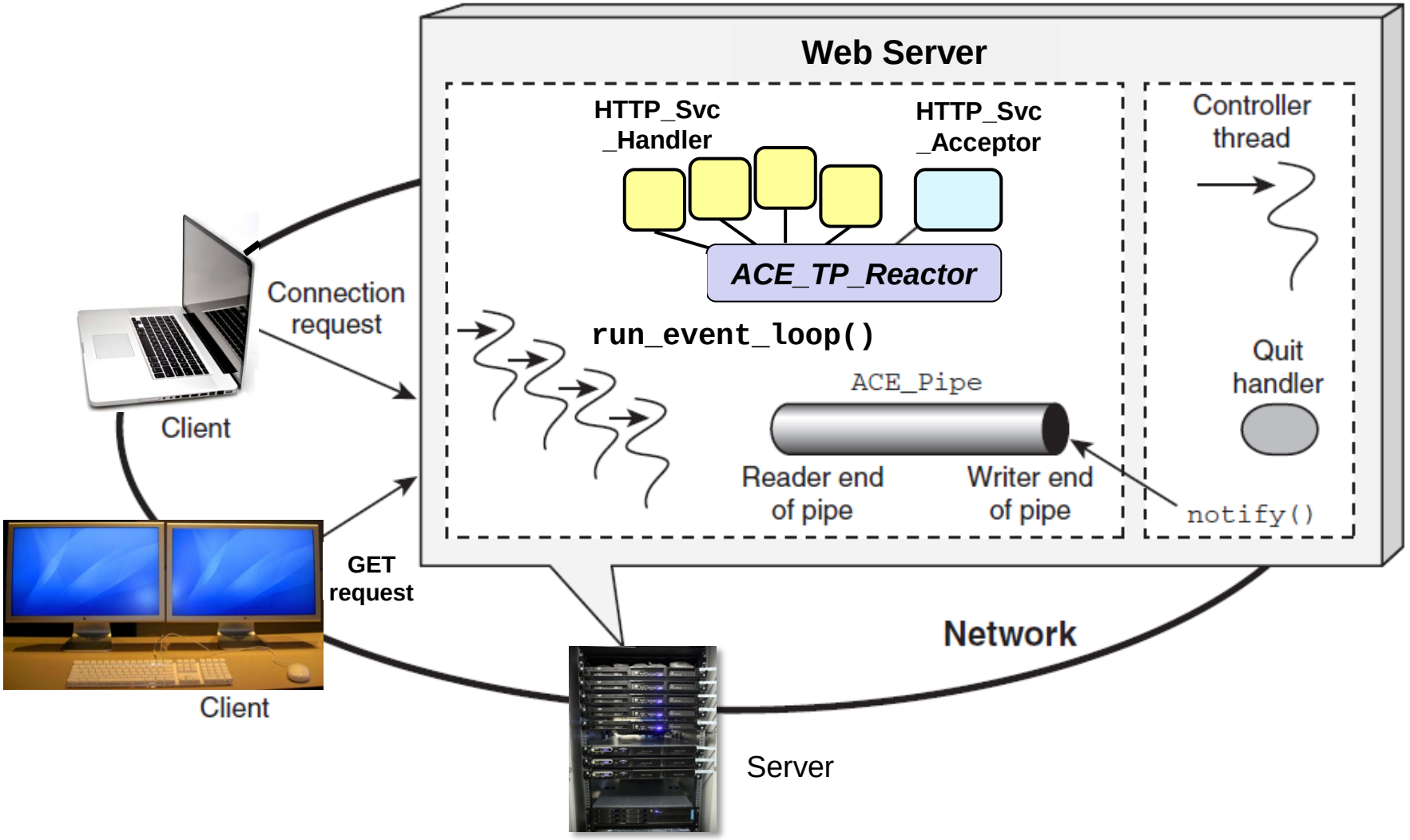
- A pool of threads can call its **`run_event_loop()`** method
  - Improves scalability by handling events on multiple handles concurrently
- Prevents multiple I/O events from being dispatched to same event handler in different threads
  - Preserves **`ACE_Select_Reactor`**'s I/O dispatching behavior, so there's no need to add synchronization locks



Handles *variability* of multi-threaded access to **`select()`** via a *common* API

Using the ACE_TP_Reactor Class with JAWS

Extend the ACE *Acceptor-Connector* JAWS web server example to spawn a pool of threads that share I/O handles



Using the ACE_TP_Reactor Class with JAWS

```
 1 #include <string>
 2 #include <ace/streams.h>
 3 #include <ace/Reactor.h>
 4 #include <ace/TP_Reactor.h>
 5 #include <ace/Thread_Manager.h>
 6 #include "Reactor_HTTP_Server.h"

 7 void *controller (void *);
 8 void *event_loop (void *);

 9 typedef Reactor_HTTP_Server<HTTP_Svc_Acceptor>
10         HTTP_Server_Daemon;
```

Forward declarations

Note reuse of earlier JAWS web server classes

Using the ACE_TP_Reactor Class

```
13 int main (int argc, char *argv[]) {
14   const size_t THREAD_POOL_SIZE = 8;
15    ACE_TP_Reactor tp_reactor;
16    ACE_Reactor reactor (&tp_reactor);

17   new HTTP_Server_Daemon (argc, argv, &reactor);

18   ACE_Thread_Manager::instance ()->spawn_n
19     (THREAD_POOL_SIZE, event_loop, &reactor);
20   ACE_Thread_Manager::instance ()->spawn
21     (controller, &reactor);

22   return ACE_Thread_Manager::instance ()->wait ();
23 }
```

Create an ACE_TP_Reactor & associate it with ACE_Reactor

Dynamic allocation ensures proper deletion semantics

Spawn threads to run reactor loop & controller

Barrier synchronization

Using the ACE_TP_Reactor Class

```
1 static void *event_loop (void *arg) {
2   ACE_Reactor *reactor = static_cast<ACE_Reactor *> (arg);
3   reactor->run_reactor_event_loop ();
4   return 0;
5 }
```

Run in multiple threads

Using the ACE_TP_Reactor Class

```
1 static void *event_loop (void *arg) {
2   ACE_Reactor *reactor = static_cast<ACE_Reactor *> (arg);
3   reactor->run_reactor_event_loop ();
4   return 0;
5 }
```

Run in a separate thread

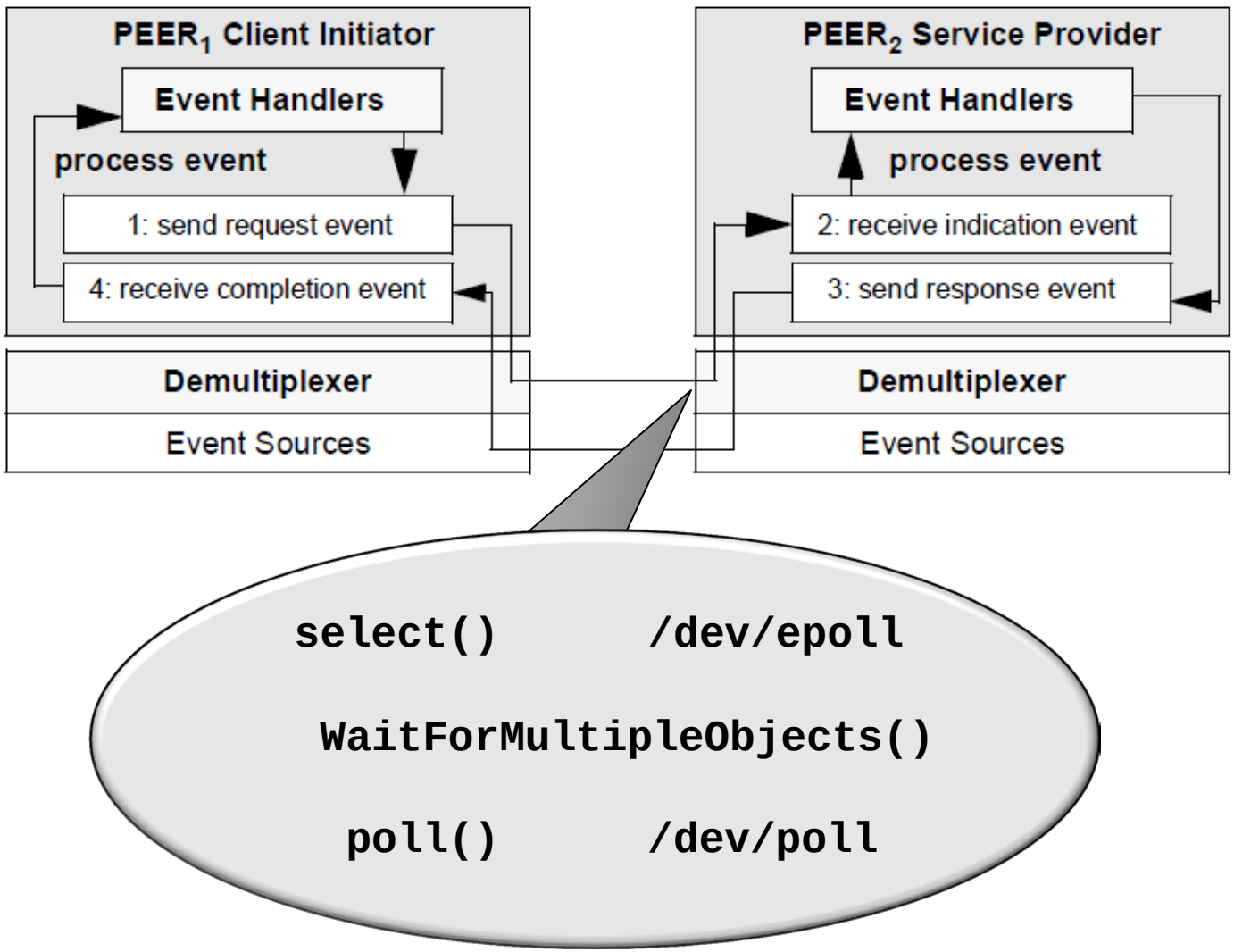
```
1 static void *controller (void *arg) {
2   ACE_Reactor *reactor = static_cast<ACE_Reactor *> (arg);
3   Quit_Handler *quit_handler = new Quit_Handler (reactor);
4   for (;;) {
5     std::string user_input;
6     std::getline (cin, user_input, '\n');
7     if (user_input == "quit") {
8       reactor->notify (quit_handler);
9       break;
10    }
11  }
    ...
```

Use the Reactor's notify pipe to wakeup the reactor & inform it to shut down by calling `handle_exception()`

The single-threaded & multi-threaded *Reactor* web servers are quite similar

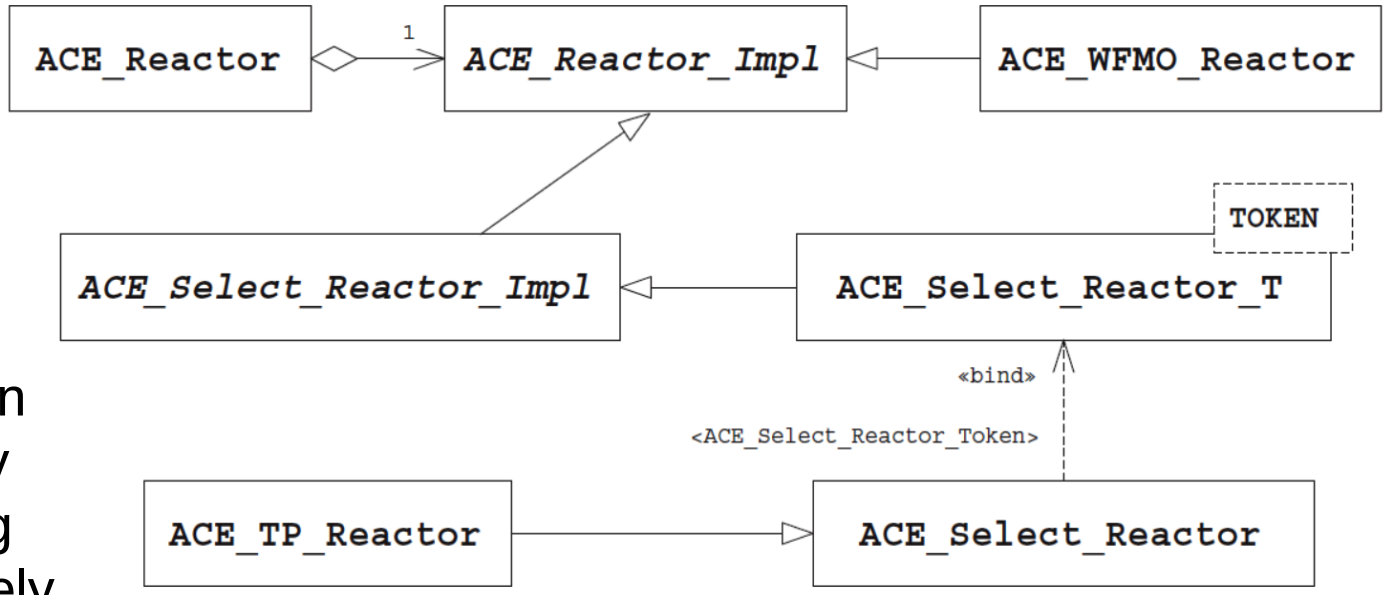
Summary

- Different OS concurrency & event (de)muxing mechanisms can present challenges & opportunities



Summary

- Different OS concurrency & event (de)muxing mechanisms can present challenges & opportunities
- ACE frameworks can use OS concurrency & event (de)muxing capabilities effectively
  - Complex design issues are isolated in the ACE *Reactor* framework, rather than in app code



Summary

- Different OS concurrency & event (de)muxing mechanisms can present challenges & opportunities
- ACE frameworks can use OS concurrency & event (de)muxing capabilities effectively
  - Complex design issues are isolated in the ACE *Reactor* framework, rather than in app code
- **`ACE_WFMO_Reactor`** provides similar thread pool capabilities for Windows platforms