AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

Architecture & Design of Embedded Real-Time Systems (TI-AREM)

Concurrency Patterns 2: (POSA2 Concurrency Pattern)

Version: 19.02.2015

Abstract

1. Active Object Pattern (POSA2)
2. Half-Sync/Half-Async Pattern (POSA2)
3. Leader/Follower Pattern (POSA2)

1. Active Object Pattern Abstract

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

Context

- Clients that access objects running in separate threads of control

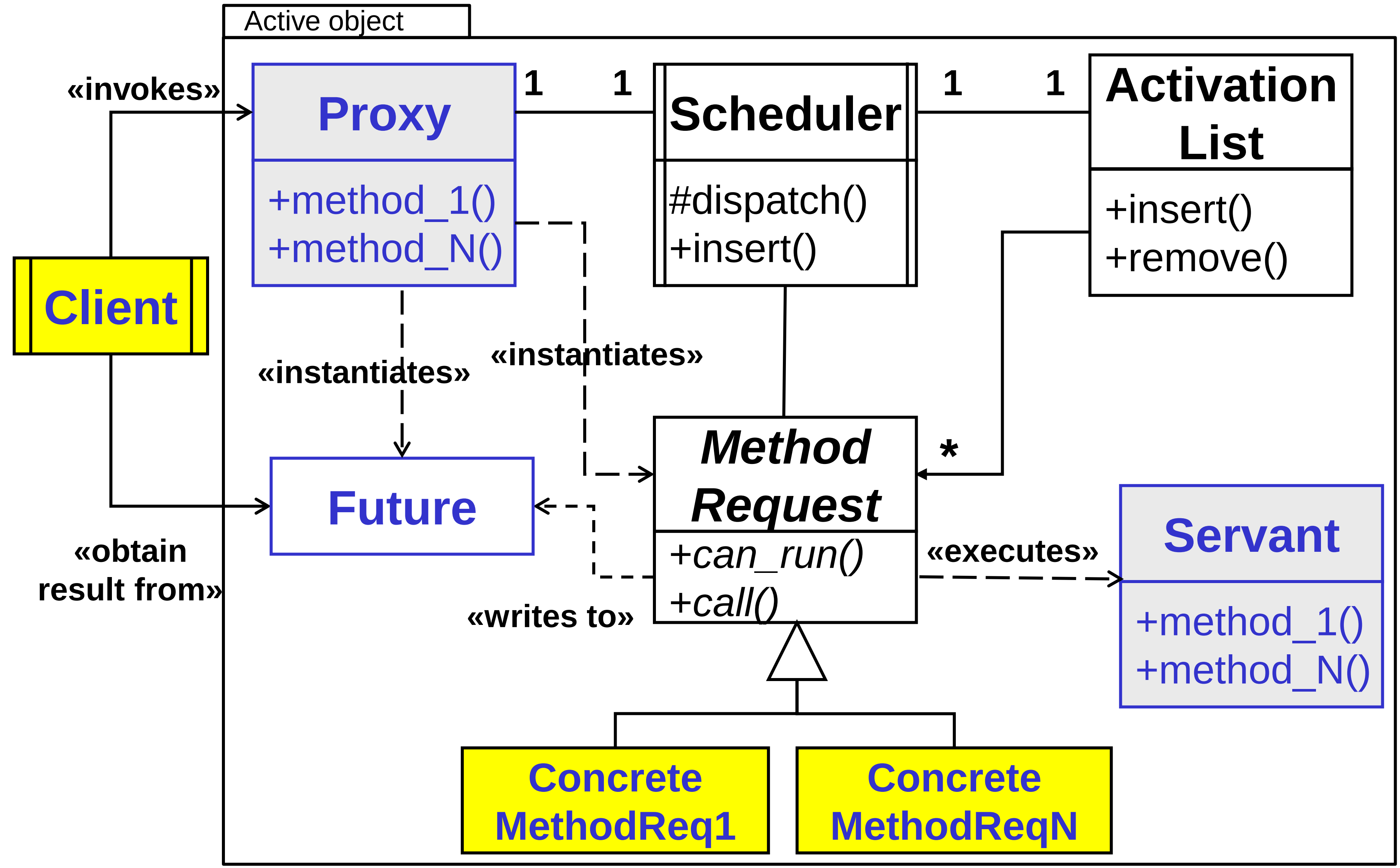
Problem

- Many applications benefits from using concurrent objects to improve their quality of service
- A concurrent object resides in its own thread of control
- If such an object is shared and modified by several client threads – we have to synchronize access to its methods and data

Solution

- **Decouple** method **invocation** on the object from method **execution**
- **Method invocation** should occur in the clients thread
- **Method execution** should occur in a separate thread
- Design the decoupling so the client thread appears to invoke an ordinary method

Active Object Structure



Active Object Dynamics

1. Method request and scheduling

2. Method execution

3. Completion

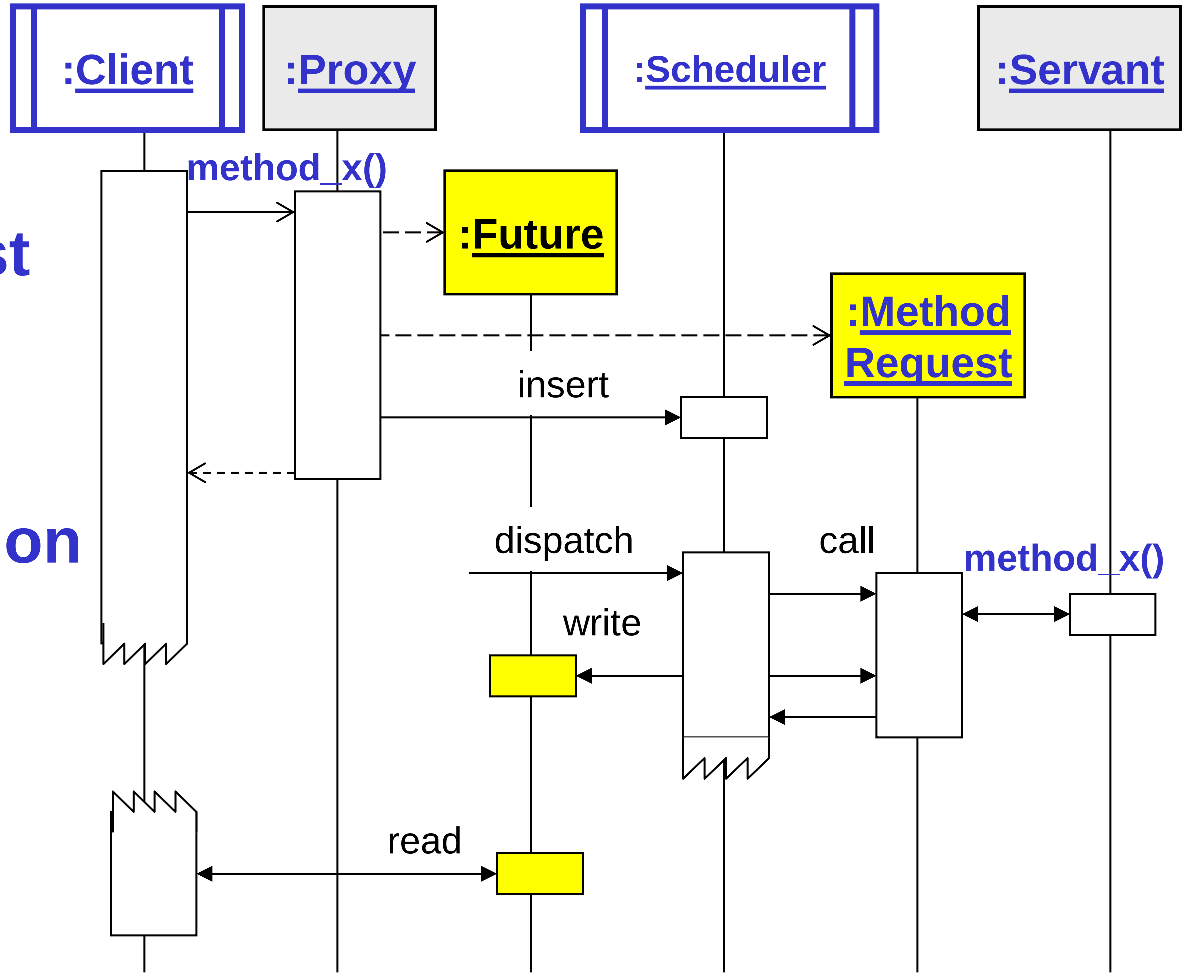


Image Acquisition Example

- OO developers generally prefer ***method-oriented*** request/response semantics to ***message-oriented*** semantics
- The Active Object pattern supports this preference via **strongly-typed async method** APIs:
  - Several types of parameters can be passed:
    - Requests contain in/inout arguments
    - Results carry out/inout arguments & results
  - Callback object or poller object can be used to retrieve results

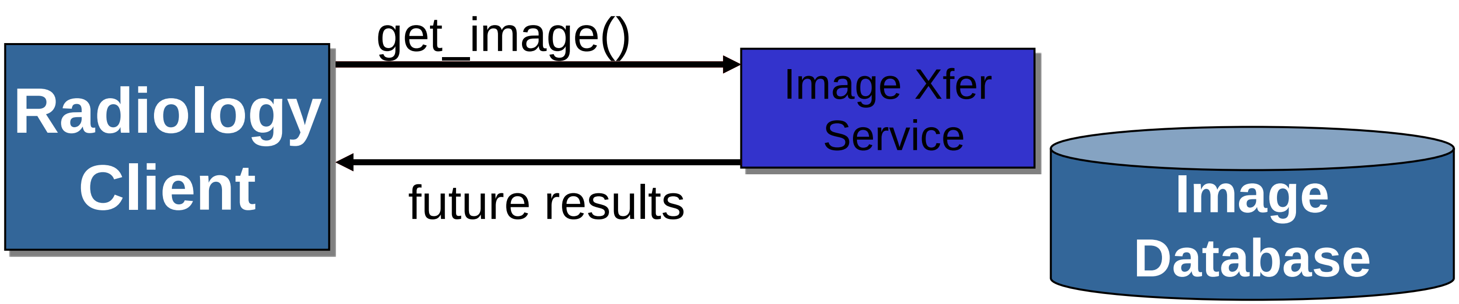
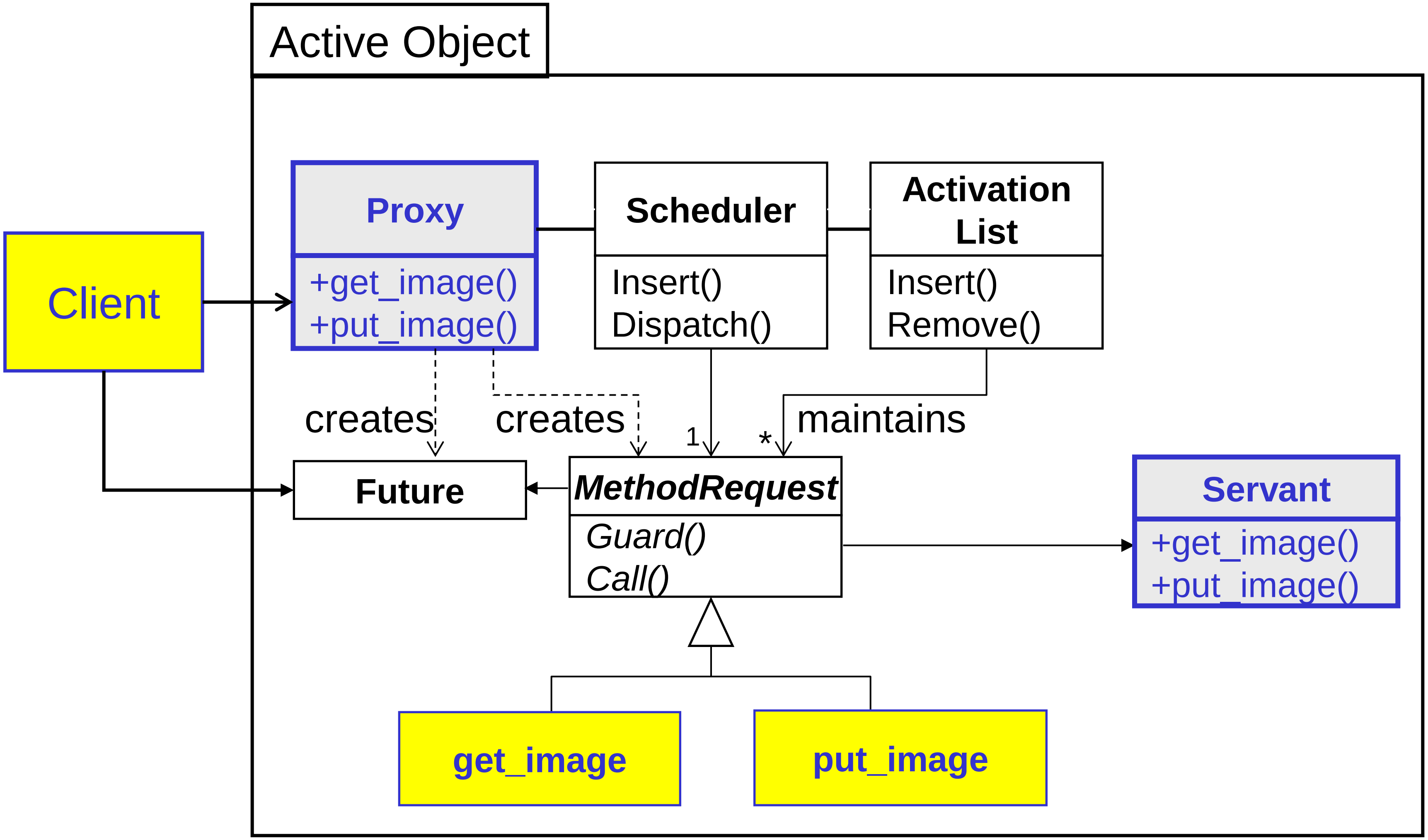


Image Acquisition Example



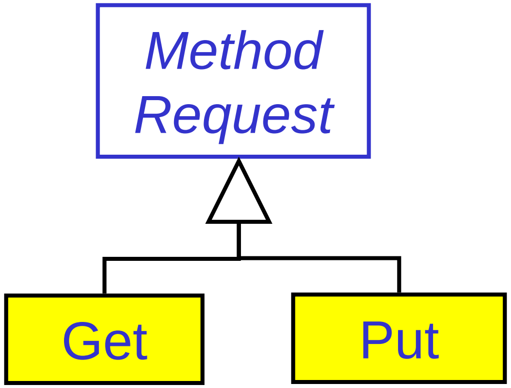
Implementation Steps

1. Implement the servant
2. Implement the invocation infrastructure
3. Implement the activation list
4. Implement the active object’s scheduler
5. Determine rendezvous and return value policy

2.1 Implement the Proxy (MQ_Proxy)

```
class MQ_Proxy {
public:
   MQ_Proxy(size_t size = MQ_MAX_SIZE):
          scheduler_(size), servant_(size) { }
   void put(const Message &msg) {
      Method_Request *mr= new Put(servant_,msg);  // request object
      scheduler_.insert(mr);
   }
   Message_Future get() {
      Message_Future result;       // counted pointer implementation
      Method_Request *mr= new Get(servant_, result); // request object
      scheduler_.insert(mr);
      return result;                    // returns a copy
   }
private:
   MQ_Servant servant_;            // implements the active object
   MQ_Scheduler scheduler_;
};
```

2.2 Implement the Method Request




```
class Method_Request {
public:
   // Evaluate the synchronization constraint
   virtual bool can_run() const = 0;

   // Execute the method
   virtual void call() = 0;
};
```

2.2 Get class

```
class Get : public Method_Request {
public:
   Get(MQ_Servant *rep, const Message_Future &f) :
            servant_(rep), result_(f) { }
   virtual bool can_run() const {
      // Synchronization constraint:
      //          cannot call <get> until queue is not empty
      return !servant_->empty();
   }
   virtual void call() {
      result_= servant_->get();
   }
private:
   MQ_Servant *servant_;
   Message_Future result_;
};
```

Class Message_Future

```
class Message_Future {
public:
   Message_Future();   // creates a <Msg.Future_Imp>
   Message_Future(const Message_Future &f);
   Message_Future(const Message &Message);

   void operator= (const Message_Future &f);

   // Block upto <timeout> time waiting to obtain result
   Message result(Time_Value *timeout =0) const;
private:
   // uses the Counted Pointer idiom
   Message_Future_Implementation *future_impl_;
};
```

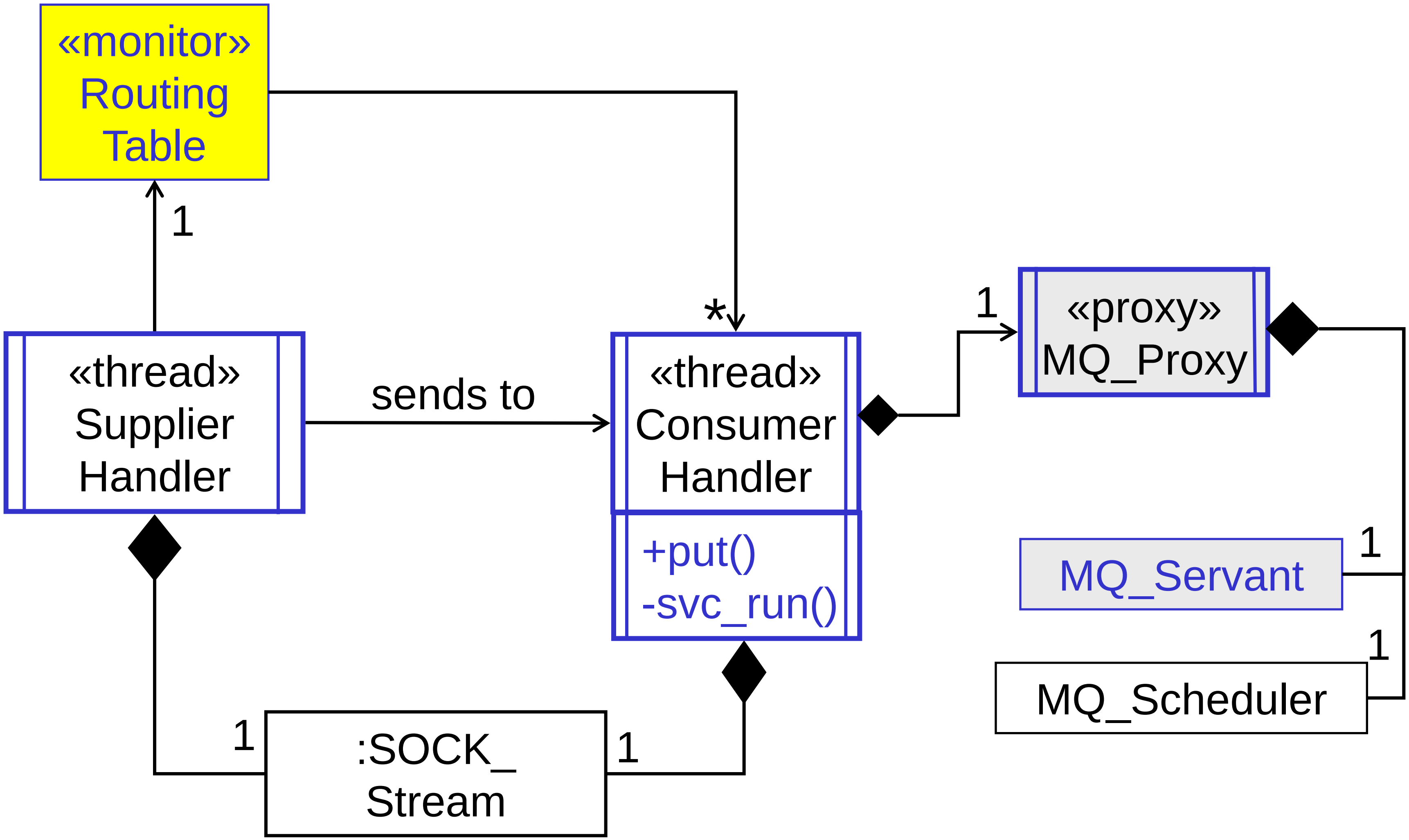
Class MQ_Scheduler

```
class MQ_Scheduler {
public:
   MQ_Scheduler(size_t high_water_mark) : act_list_(high_water_mark)
  {
      Tread_Manager::instance()->spawn(&svc_run, this);
   }

   void insert(Method_Request *mr) { act_list_.insert(mr); }
protected:
   virtual void dispatch();
private:
   Activation_List act_list_;

   static void *svc_run(void *args) {
      MQ_Scheduler *this_obj = static_cast<MQ_Scheduler *> (args);
      this_obj->dispatch();          // equal to a thread run method
   }
};
```

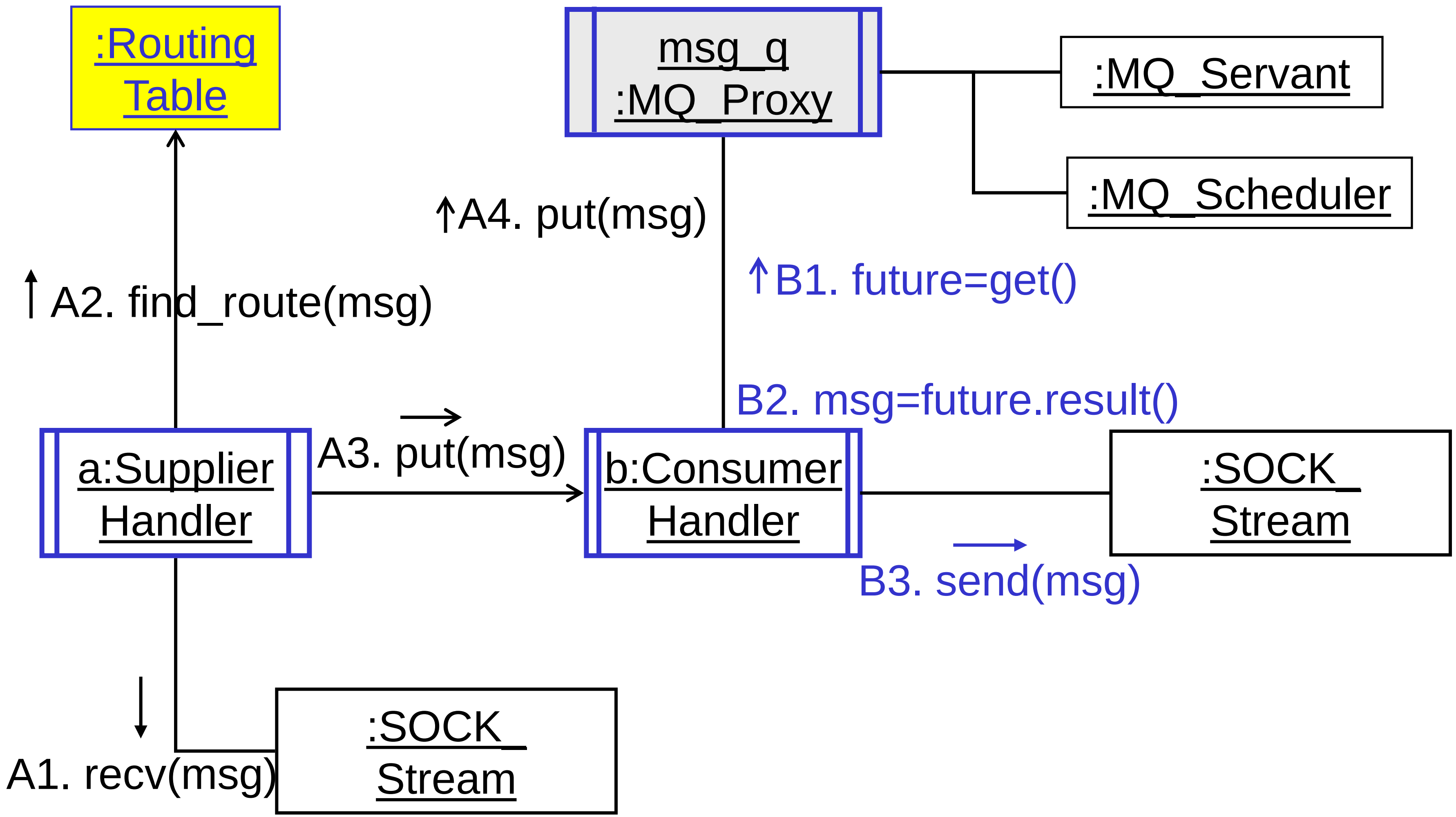
MQ_Scheduler::dispatch()

```
void MQ_Scheduler::dispatch() {
    for (; ;) {        // forever
        Activation_List::iterator request;

        for (request= act_list_.begin(); request != act_list_.end(); ++request)
        {
            if  ( (*request).can_run() )
            {
                act_list_.remove(*request);
                (*request).call();
                delete *request;                // NB! deletes MethodRequest obj.
            }
        }
    }
}
```

Gateway Example – Class Diagram



Gateway Example



Supplier_Handler

```
void Supplier_Handler::route_message(const Message &msg)
{
   // Locate the appropriate consumer based on the address info in msg
   Consumer_Handler *consumer_handler_=
         routing_table_.find_route(msg.address());

   // Put the Message into the Consumer Handler's queue
   consumer_handler_->put(msg);
}
```

Class Consumer_Handler

```
class Consumer_Handler {
public:
   Consumer_Handler() { Thread_Manager::instance().spawn(&svc_run, this); }
   void put(const Message &msg) { msg_q_.put(msg); }

private:
   MQ_Proxy msg_q_;                      // proxy to active object
   SOCK_Stream connection_;

   static void *svc_run(void *args) {
      Consumer_Handler *this_obj=
         static_cast<Consumer_Handler *> (args);
      for (; ;) {
         Message_Future future= this_obj->msq_q_.get();
         Message msg= future.result();    // blocking read
         this_obj->connection_.send(msg, msg.length());
      }
   }
};
```

Variants: Integrated Scheduler

```
class MQ_Scheduler {
public:
   MQ_Scheduler(size_t size) : servant_(size), act_list_(size) { }

   void put(cons Message m) {
      Method_Request *mr = new Put(&servant_, m);
      act_list_.insert(mr);
   }

   Message_Future get() {
      Message_Future result;          // Counted pointer
      Method_Request *mr = new Get(&servant, result);
      act_list_.insert(mr);
      return result;
   }
   // other methods
private:
   MQ_Servant servant_;
   Activation_List act_list_;
};
```

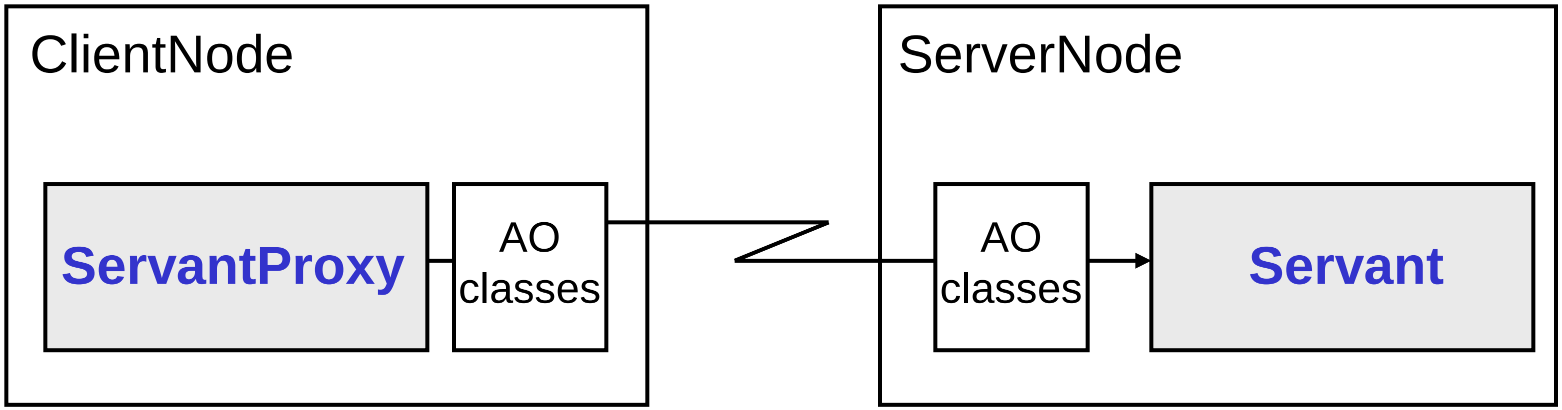
Future as a Template Class

```
template <class TYPE>
class Future {
public:
    Future();
    Future(const Future<TYPE> &r);
    ~Future();
    void operator = (const Future<TYPE> &r);
     void cancel();
     // block upto <timeout> time waiting to obtain result
     TYPE result(Time_Value *timeout =0) const;
private:
     //
};
```

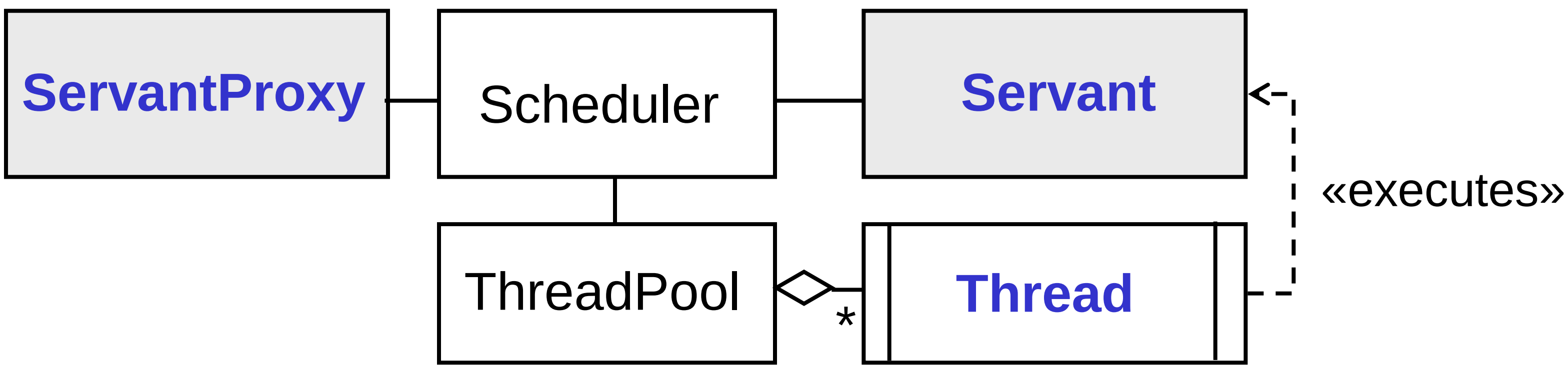
AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

More Variants

Distributed active object



Thread Pool variant



Active Object Benefits

- Enhanced type-safety
  - Compared with async message passing
- Enhances concurrency & simplifies synchronized complexity
  - Concurrency is enhanced by allowing client threads & asynchronous method executions to run simultaneously
  - Synchronization complexity is simplified by using a scheduler that evaluates synchronization constraints to guarantee serialized 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 are executed according to the synchronization constraints defined by their guards & by scheduling policies

Active Object Liabilities

- Performance overhead
  - Depending on how an active object's scheduler is implemented:
    - context switching, synchronization, & data movement overhead may occur when scheduling & executing active object invocations
- Complicated debugging
  - It is hard to debug programs that use the Active Object pattern due to the concurrency & non-determinism of the various active object schedulers & the underlying OS thread scheduler

Known Uses

- ACE Framework
- Siemens Syngo
- Siemens FlexRouting – automatic call distribution
- Java – JDK1.3

2. Half-Sync/Half-Async Pattern Abstract

- The ***Half-Sync/Half-Async* architectural pattern** decouples asynchronous and synchronous service processing in concurrent systems, to simplify programming without unduly reducing performance

- The pattern introduces two intercommunicating layers, one for **asynchronous** and one for **synchronous** service processing

Context

- A **concurrent** system that performs both **asynchronous** and **synchronous** processing services that must **inter-communicate**

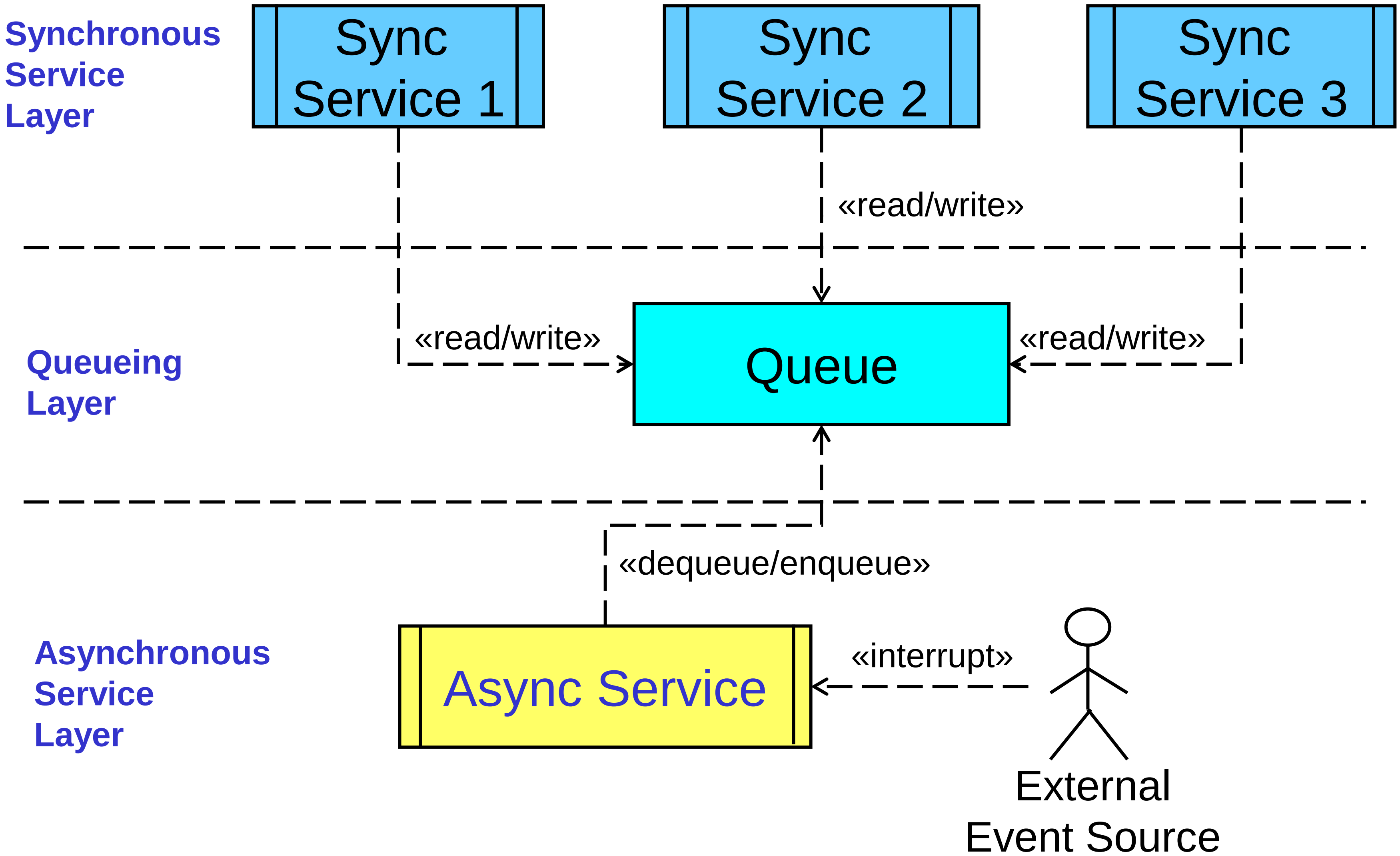
Problem

- Concurrent systems often contains a mixture of asynchronous and synchronous processing
- **System programmers** want to use:
  - asynchrony to improve performance
  - services are mapped to HW interrupt handlers or SW signal handlers
- **Application programmers** want to use:
  - synchronous processing to simplify their programs

Solution

- Decompose the **services** in the system into two layers:
  - A synchronous layer
  - An asynchronous layer
- Add a queuing layer between

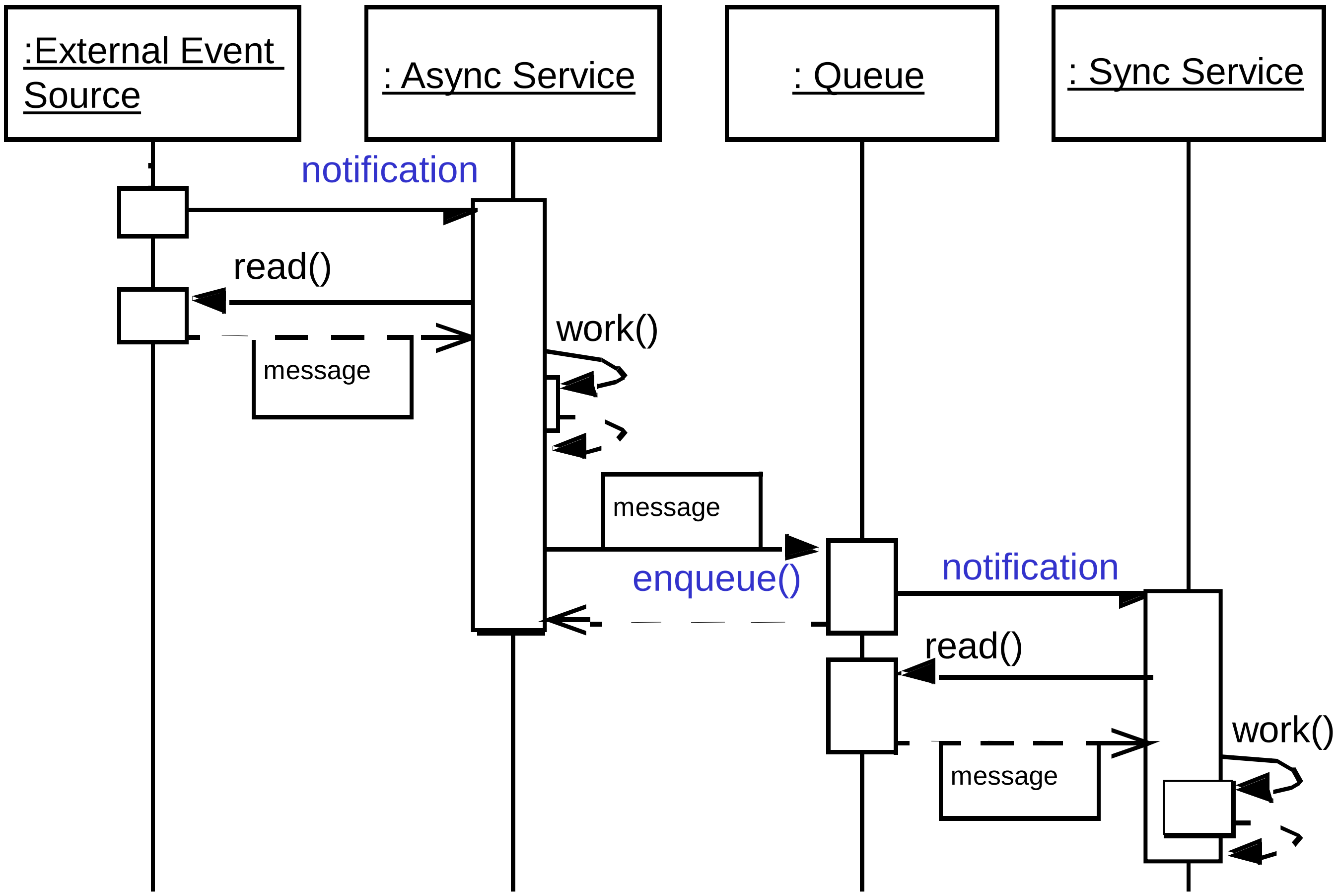
Half-Sync/Half-Async Structure



Two Service Layers

- Process higher-layer services, e.g. long-duration database queries or file transfers, **synchronously** in separate **threads** to simplify programming
- Process lower-layer services, e.g. short-lived protocol handlers driven by interrupts, **asynchronously** to enhance performance

Dynamics


:External Event Source
: Async Service
: Queue
: Sync Service
notification
read()
message
work()
message
enqueue()
notification
read()
message
work()

Implementation Steps

1. Decompose the overall system into three layers
2. Implement the services in the **synchronous layer**
3. Implement the services in the **asynchronous layer**
4. Implement the **queuing layer**

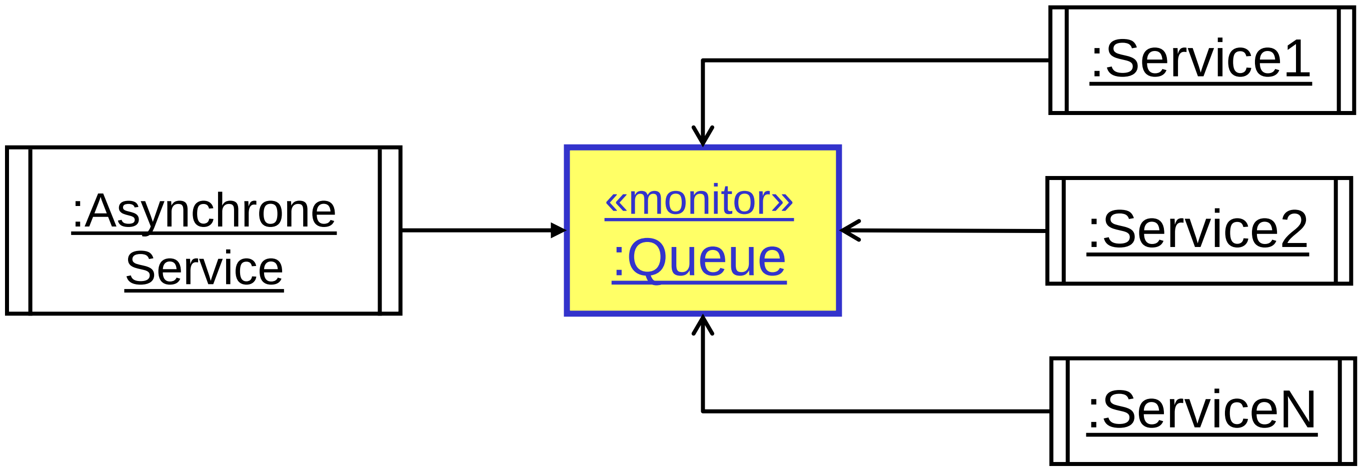
4.1 Implement the Buffering Strategy

- Implement
  - the ordering strategy
    - FIFO or priority order
  - the serialization strategy
  - the notification strategy
  - the flow-control strategy

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

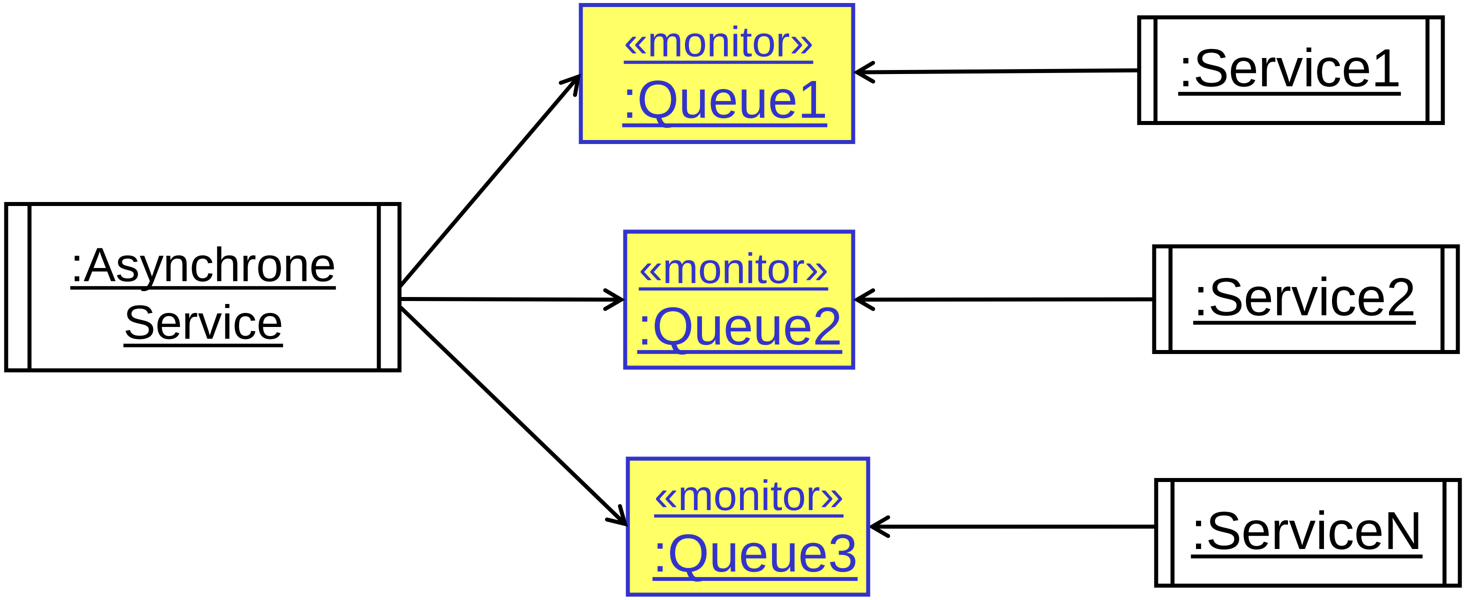
4.2 Implement the (de)multiplexing strategy(1)

- One shared queue for all services (a Singleton queue)
  - Could be a Monitor object implemented as a Singleton
  - Must allow multiple wait in the monitor

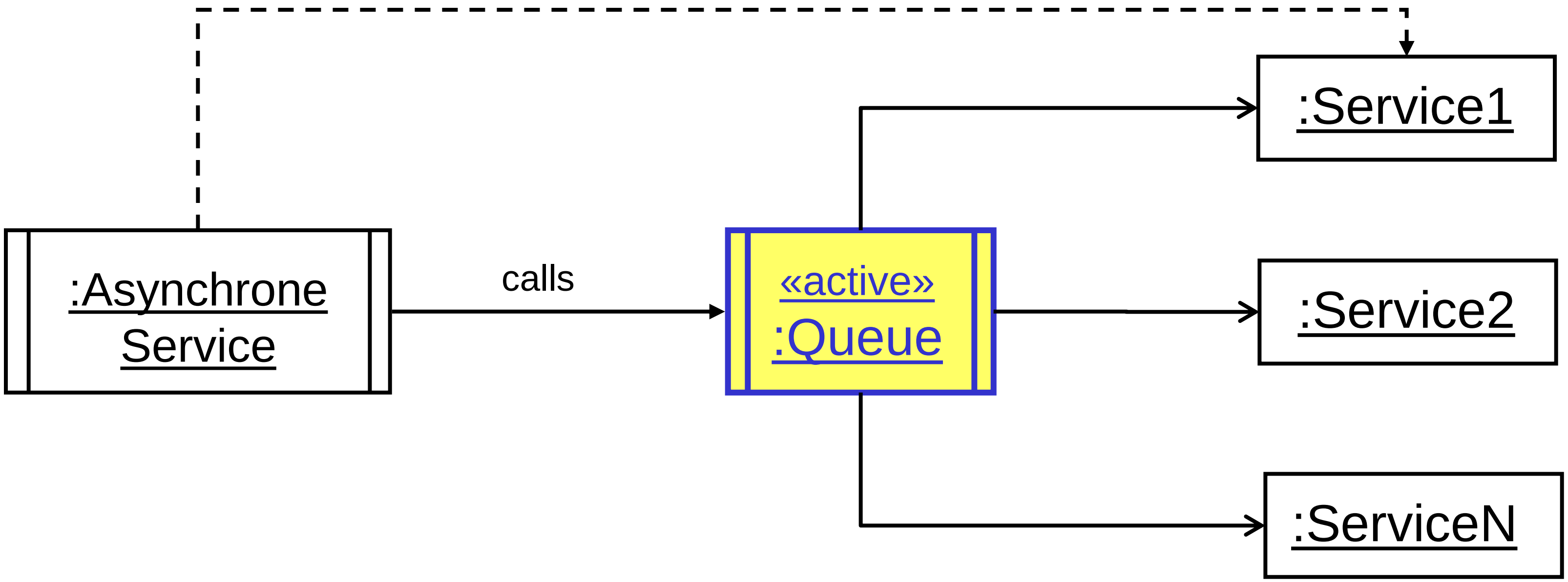


4.2 Implement the (de)multiplexing strategy(2)

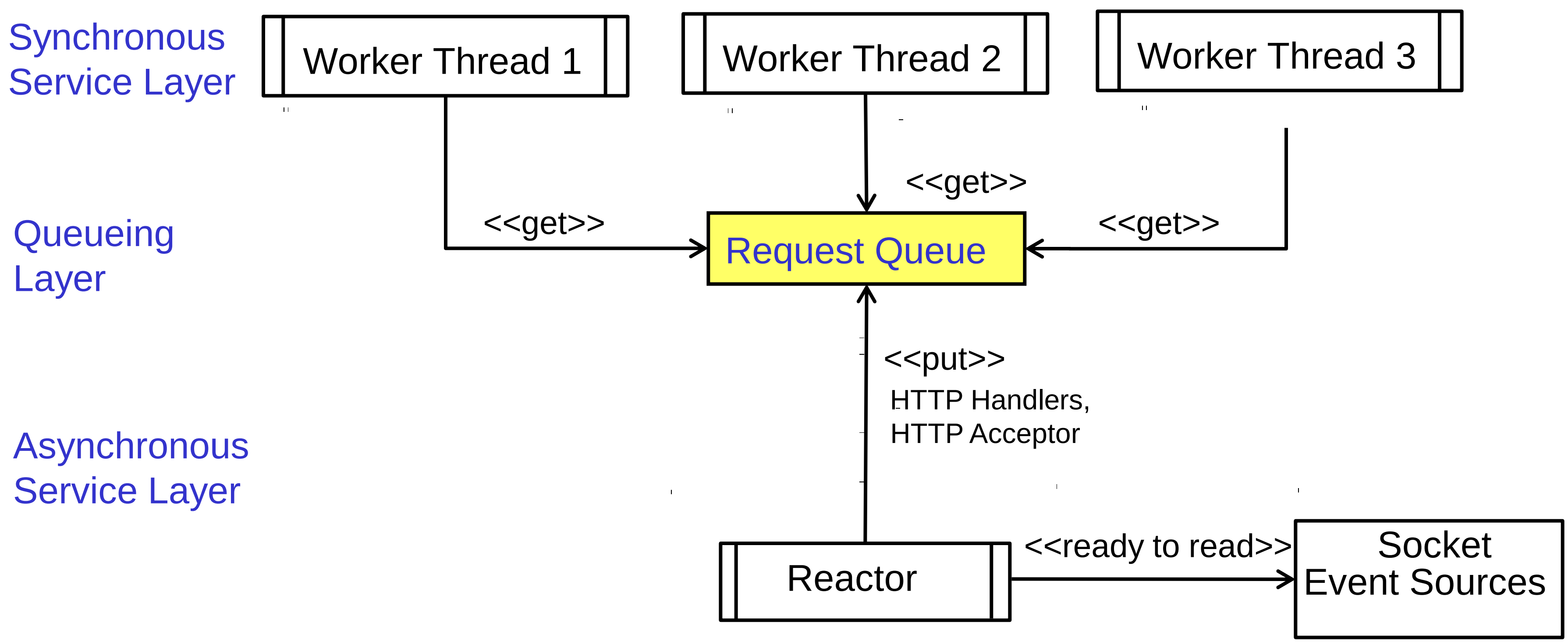
- Multiple queues – one queue per service
  - requires a demultiplexing strategy



Active object – as a queue object



Applying the Half-Sync/Half-Async Pattern in JAWS Web Server



Half-Sync/Half-Async Benefits

- 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 control 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

Half-Sync/Half-Async Liabilities

- A boundary-crossing penalty may be incurred
  - This overhead arises from context switching, synchronization, & data copying overhead when data is transferred between the sync & async service layers via the queueing layer
- Higher-level application services may not benefit from the efficiency of async I/O
  - Depending on the design of operating system or application framework interfaces, it may not be possible for higher-level services to use low-level async I/O devices effectively
- Complexity of debugging & testing
  - Applications written with this pattern can be hard to debug due its concurrent execution

3. Leader/Follower Pattern Abstract

The **Leader/Followers architectural pattern** provides an efficient concurrency model where multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process service requests that occur on the event sources

Context

- An event-driven application where multiple service request arriving on a set of event sources must be processed **efficiently** by multiple threads that share the event sources

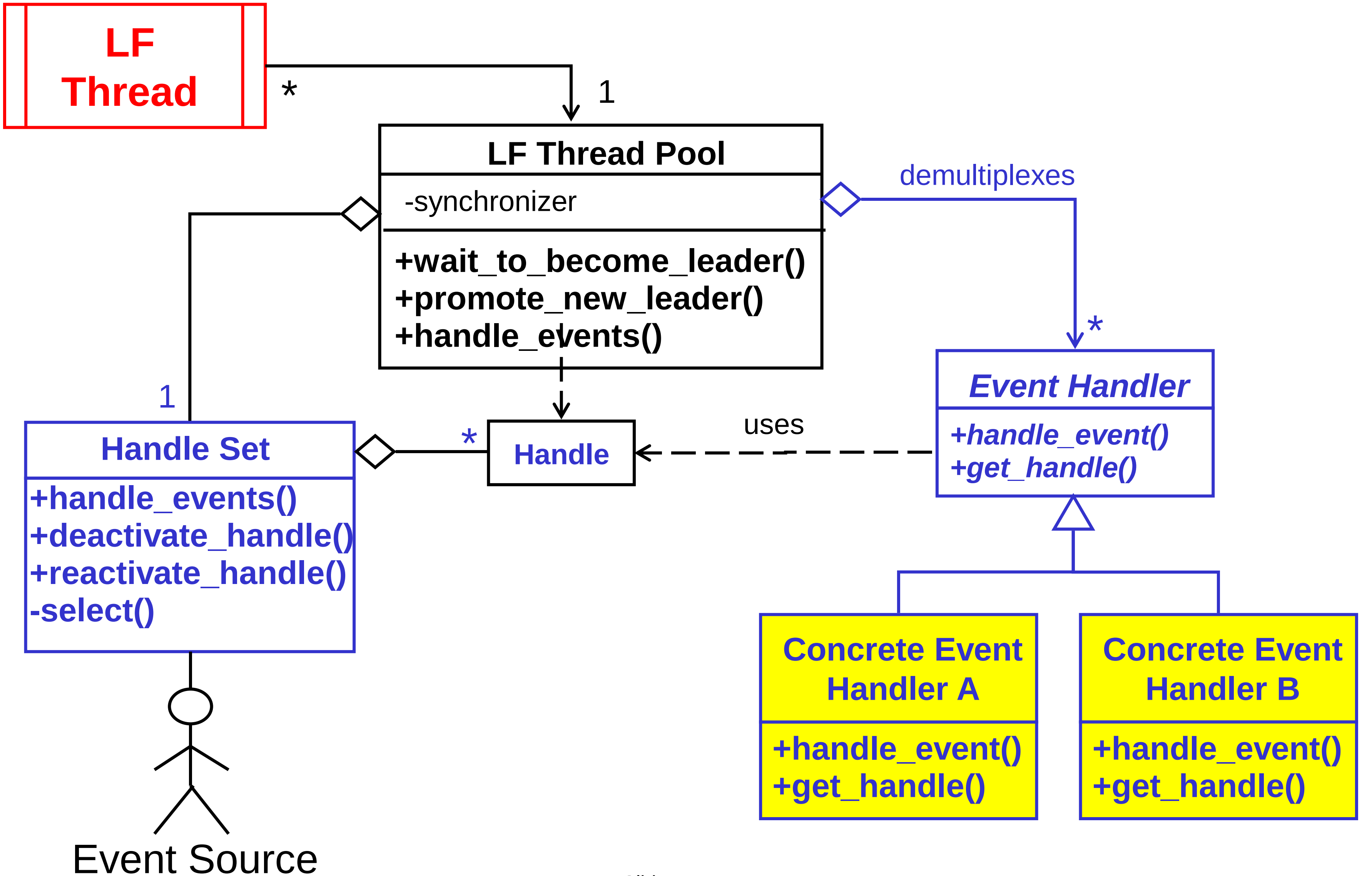
Problem

- It is hard to implement **high-performance** multi-threaded server applications
- These applications process high volumes of multiple event types, such as CONNECT, READ and WRITE
- Context switching overhead
- Dynamic memory allocation
- Multiple threads calling ***select***

Solution

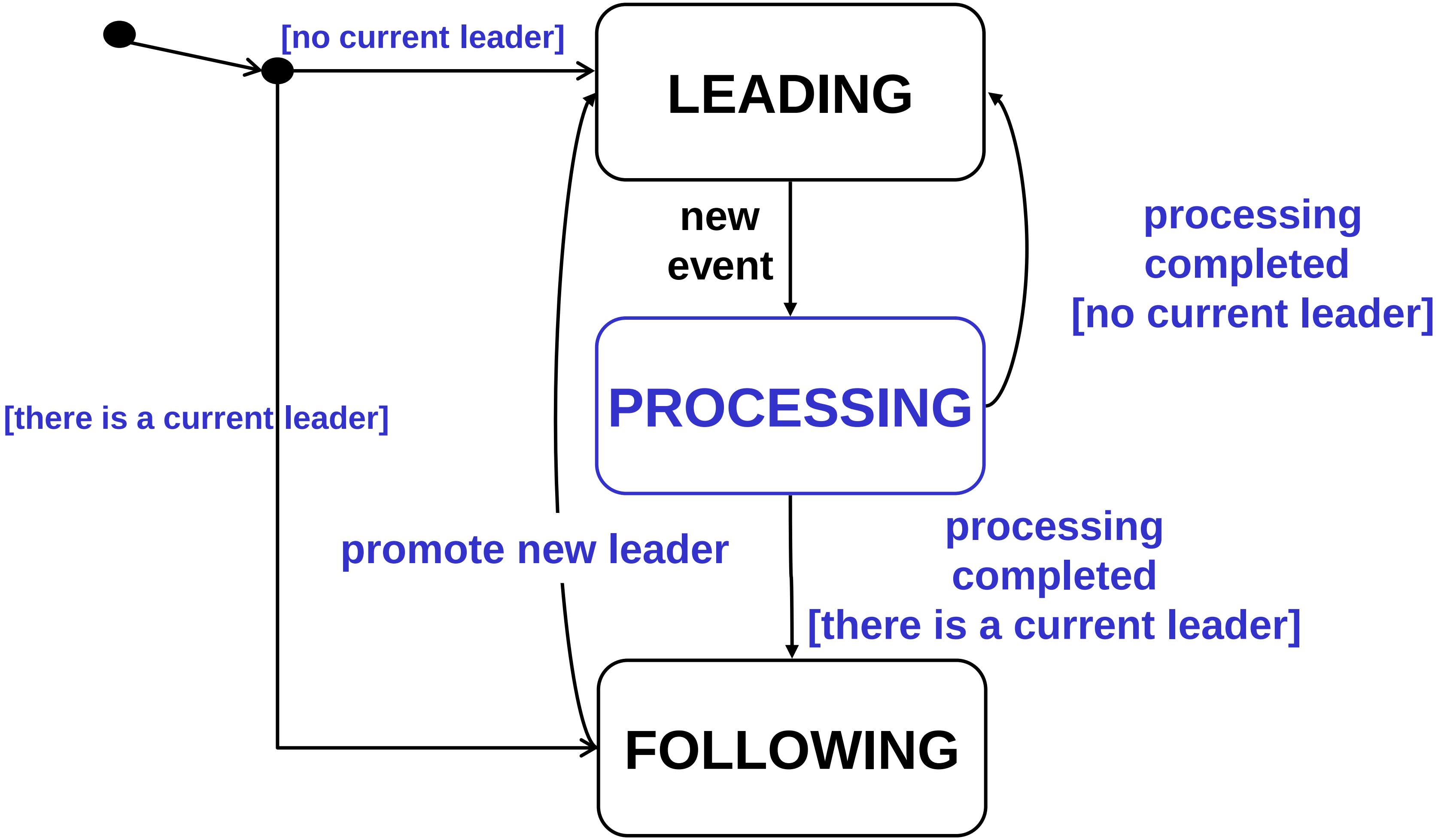
- Structure a pool of threads to share a set of event sources efficiently by ***taking turns*** demultiplexing events and synchronously dispatching the events to application services that process them
  - allow one ***leader*** thread to wait for an event
  - other ***follower*** threads can queue up waiting for their turn
  - after detecting an event – the leader promotes a follower to leader. It then plays the role of a ***processing*** thread

Leader/Followers Structure

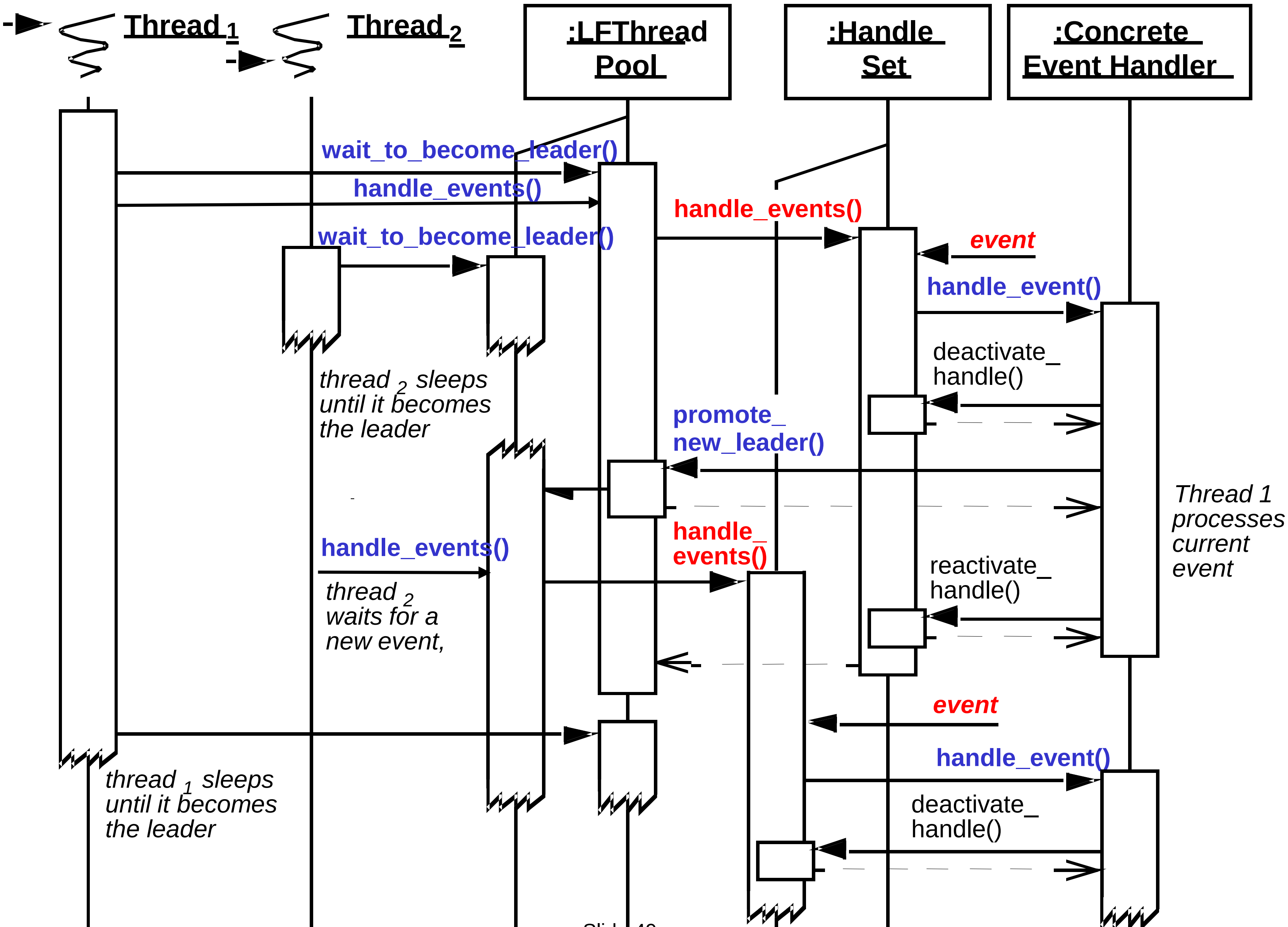


AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Thread State Diagram



Leader/Followers Pattern Dynamics



Implementation Steps

1. Choose the handle and handle set mechanisms
2. Implement a protocol for temporarily (de)activating handles in a handle set
3. Implement a thread pool
4. Implement a method to become a leader
5. Implement the follower promotion protocol
6. Implement the event handlers

2. Handle deactivating/reactivating

```
class Reactor {
public:
    // temporarily deactivate the <HANDLE> from the handle set
    void deactivate_handle(HANDLE, Event_Type et);
    void reactivate_handle(HANDLE, Event_Type et);

//
};
```

Deactivating the handle from the handle set avoids ***race conditions***

- that can occur between the time when a new leader is selected and the event is processed

3. Implement the Thread Pool

```
class LF_Thread_Pool {
public:
    LF_Thread_Pool(Reactor *r) : reactor_(r) {  }

    // Promote a follower thread to become the new leader
    void promote_new_leader();
    void wait_to_become_leader();
    void handle_events();
    void deactivate_handle(HANDLE, Event_Type et);
    void reactivate_handle(HANDLE, Event_Type et);
private:
    Reactor *reactor_;
    Thread_Semaphore leader_semaphore_;  // free
};
```

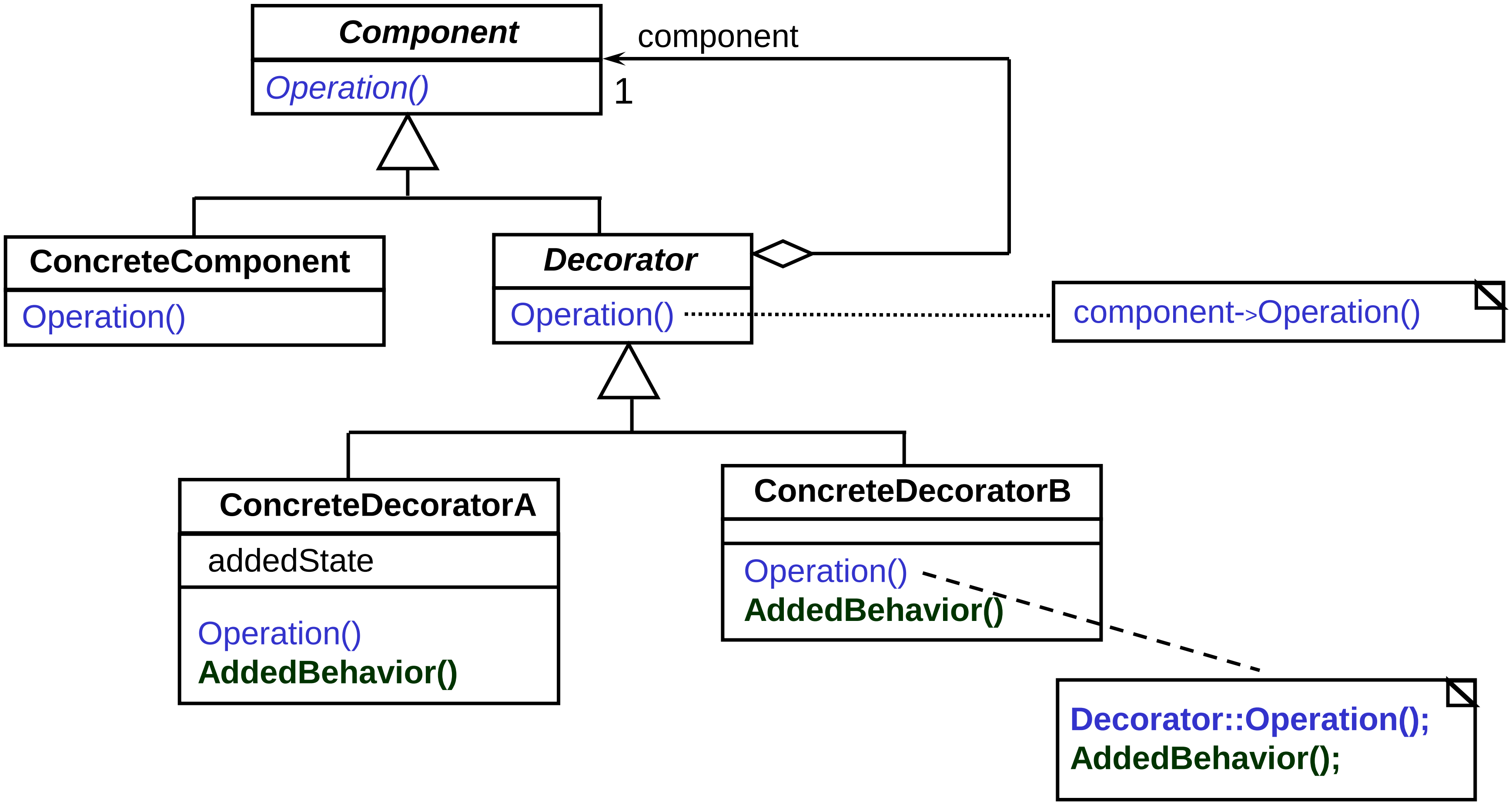
4. + 5. Implement leader management

```
void LF_Thread_Pool::wait_to_become_leader()
{
    leader_semaphore_.wait();  // other threads in pools waits here
    // now a leader
}

class LF_Thread_Pool::promote_new_leader()
{
    leader_semaphore_.signal();  // give up leadership to next waiting
                                 // thread
}

class LF_Thread_Pool::handle_events()
{
    reactor->handle_events();
}
```

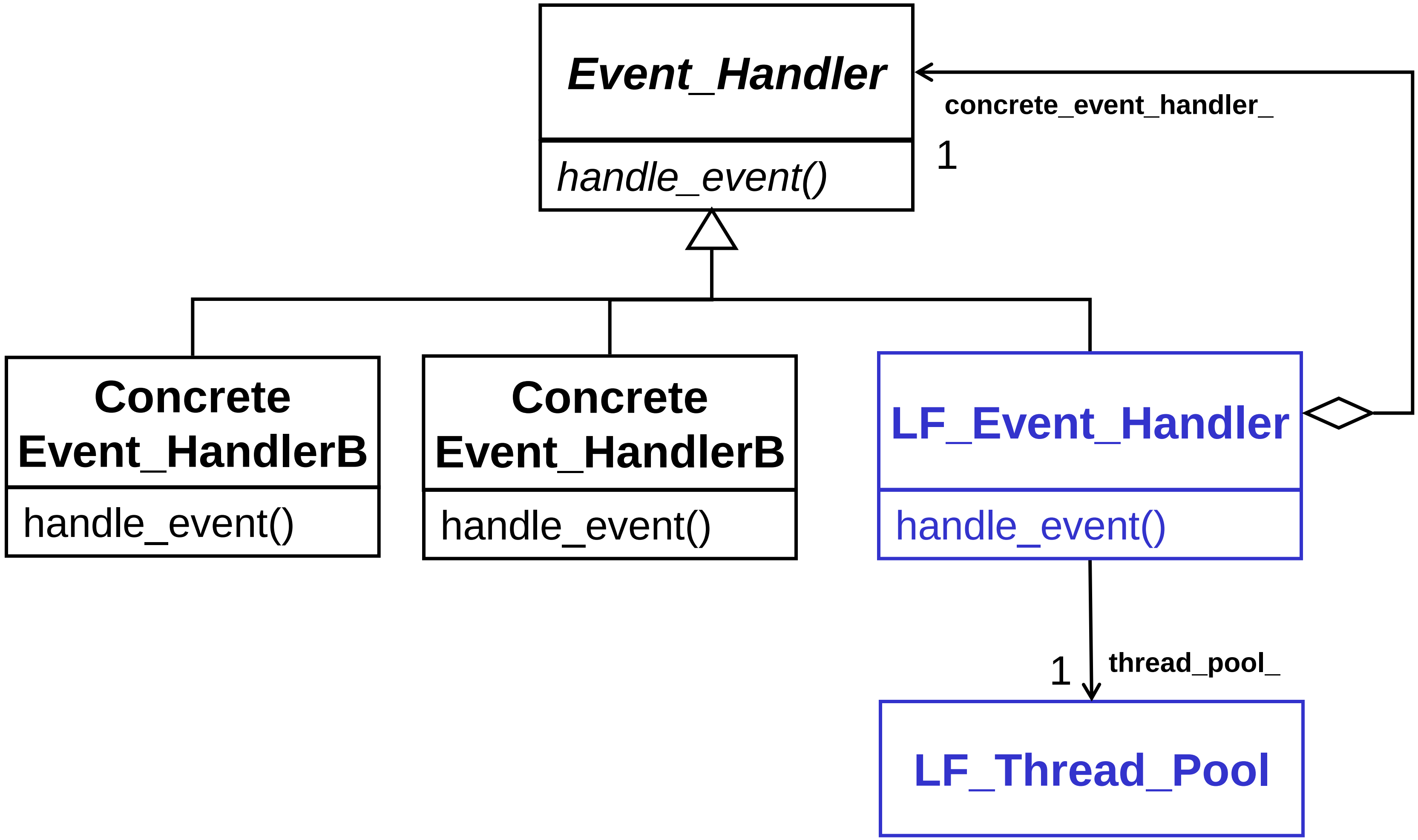
LF_Thread Class

```
class LF_Thread : public Thread {
public:
   LF_Thread(LF_Thread_Pool *tp) : tread_pool_(tp) { }

   void run()
   {
        while (1)
        {
            thread_pool_->wait_to_become_leader();
            thread_pool->handle_events();
        }
    }
}
private:
     LF_Thread_Pool *thread_pool_;
};
```

Decorator – GoF Structure Pattern



LF_Event_Handler Decorator



LF_Event_Handler Class

```
class LF_Event_Handler : public Event_Handler {
public:
   LF_Event_Handler(Event_Handler *eh, LF_Thread_Pool *tp) :
         concrete_event_handler_(eh), tread_pool_(tp) { }

   virtual void handle_event(HANDLE h, Event_Type et)
   {
      thread_pool_->deactivate_handle(h, et);
      thread_pool_->promote_new_leader();
      // Dispatch application-specific event processing code
      concrete_event_handler_->handle_event(h,et);
      thread_pool_->reactivate_handle(h, et);
   }
private:
   Event_Handler *concrete_event_handler_;
   LF_Thread_Pool *thread_pool_;
};
```

Main Program

```
const int MAX_THREADS = 20;

void *worker_threads(void *);    // Forward declaration

int main() {
   LF_Thread_Pool  thread_pool(Reactor::instance());
   LF_Threads *active_threads[MAX_THREADS];

   // code to set up a passive acceptor omitted
   //
   // create Threads with link to thread_pool
   for (int i=0; i < MAX_THREADS-1; i++)
          active_threads[i]= new LF_Thread(&thread_pool);
   for (int i=0; i < MAX_THREADS-1; i++)
          active_threads[i]->start();     // start thread
   while (1)
       ;   // main thread idles
};
```

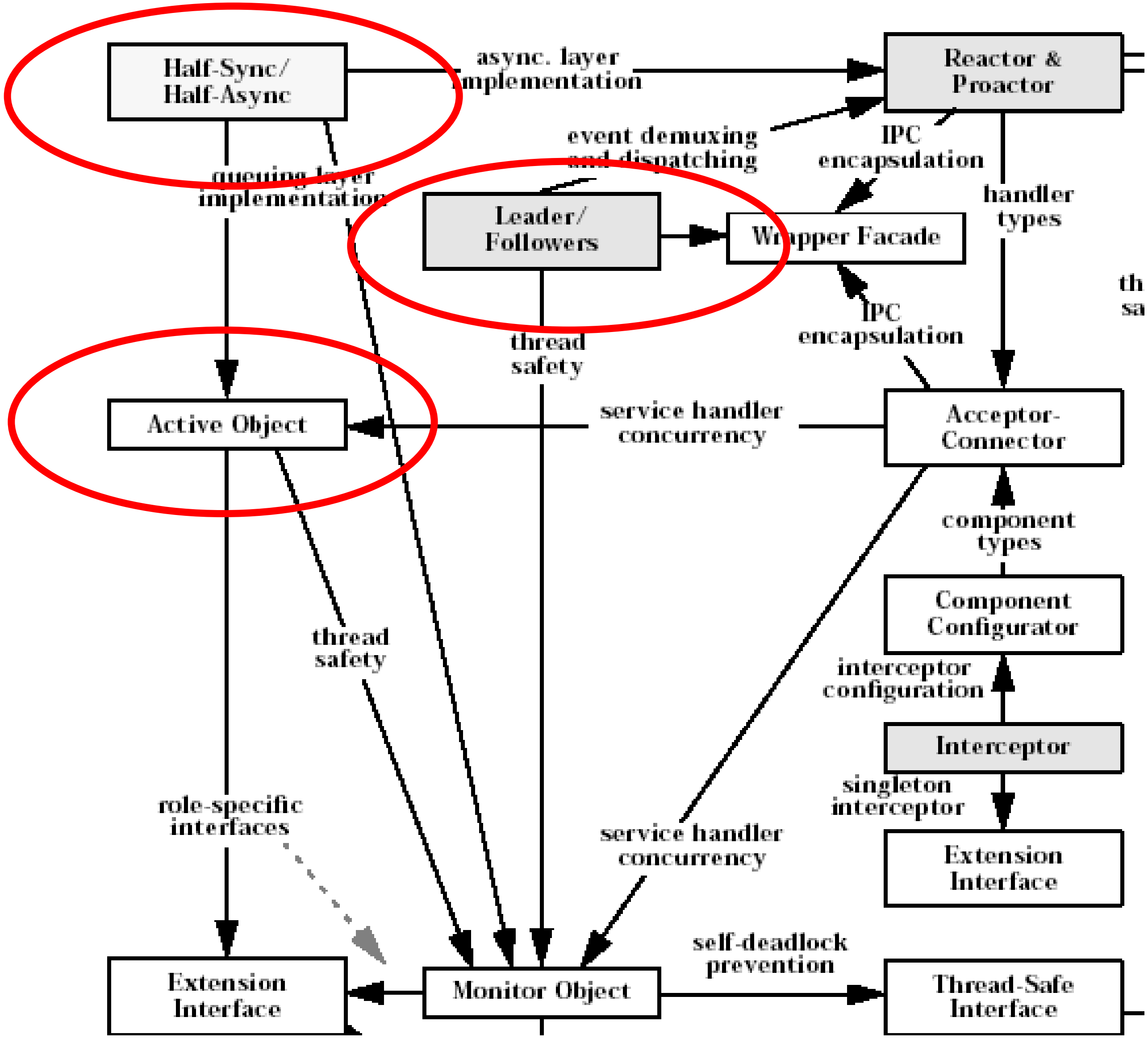
Leader/Followers Benefits

- Performance enhancements
  - It enhances CPU cache affinity and eliminates the need for dynamic memory allocation & data buffer sharing between threads
  - It minimizes locking overhead by not exchanging data between threads, thereby reducing thread synchronization
  - It can minimize priority inversion because no extra queuing is introduced in the server
  - It doesn’t require a context switch to handle each event, reducing dispatching latency
- Programming simplicity
  - The Leader/Follower pattern simplifies the programming of concurrency models where multiple threads can receive requests, process responses & demultiplex connections using a shared handle set

Leader/Followers Liabilities

- Implementation complexity
  - The advanced variants of the Leader/Followers pattern are hard to implement
- Lack of flexibility
  - In the Leader/Followers model it is hard to discard or reorder events because there is no explicit queue
- Network I/O bottlenecks
  - The Leader/Followers pattern serializes processing by allowing only a single thread at a time to wait on the handle set, which could become a bottleneck because only one thread at a time can demultiplex I/O events

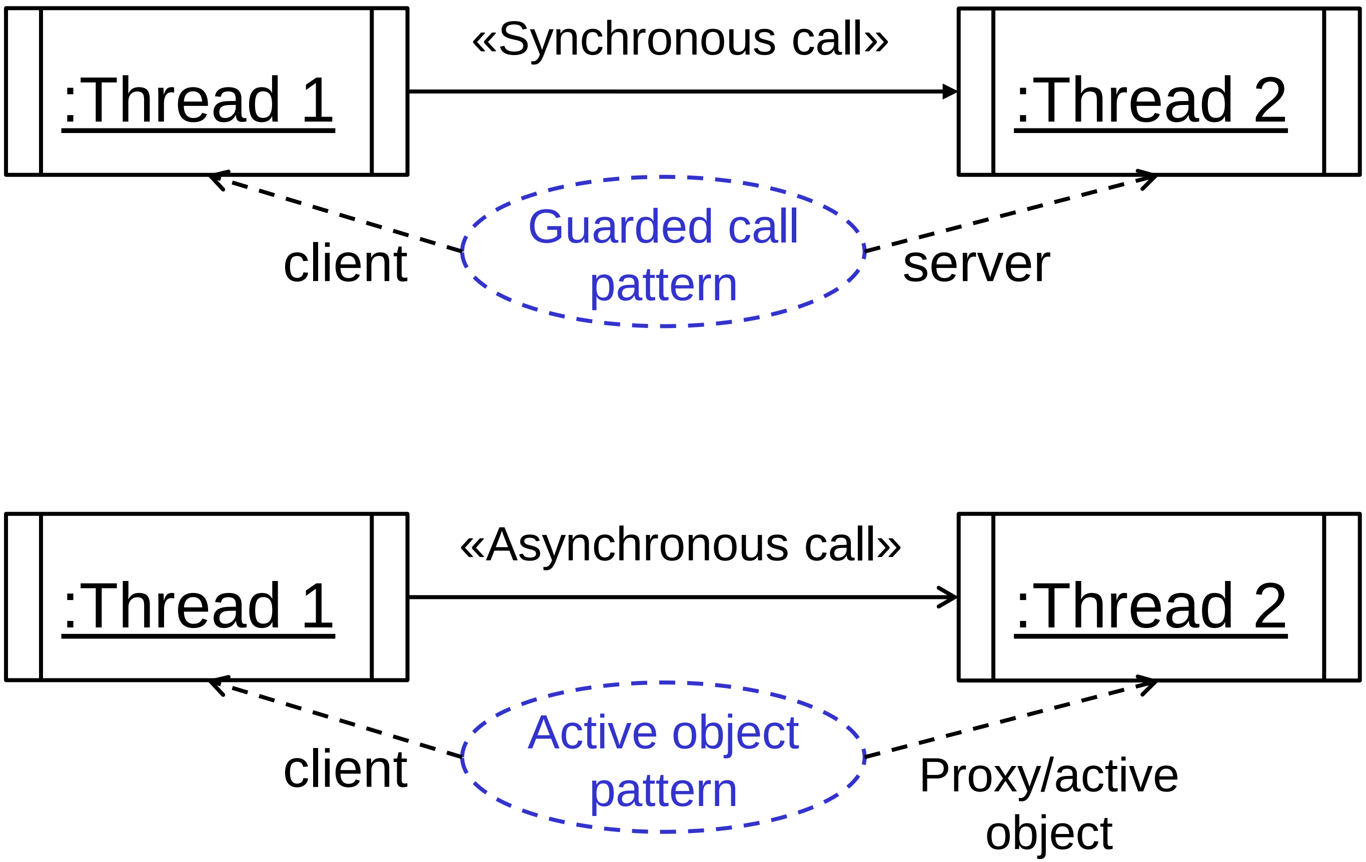
Relations to other POSA2 patterns



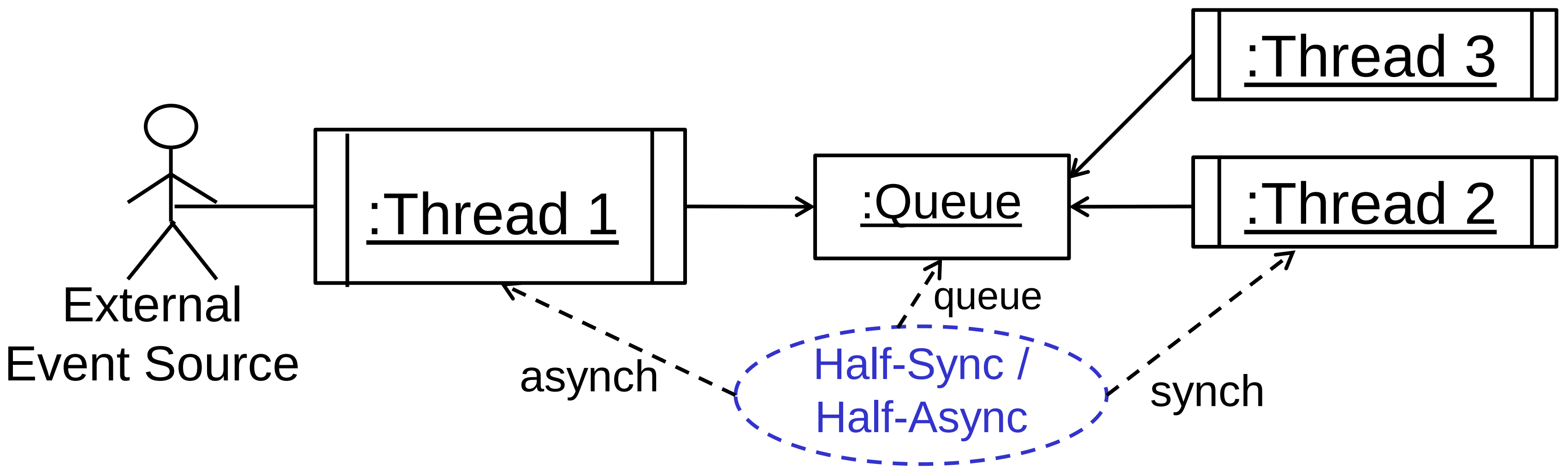
Summary

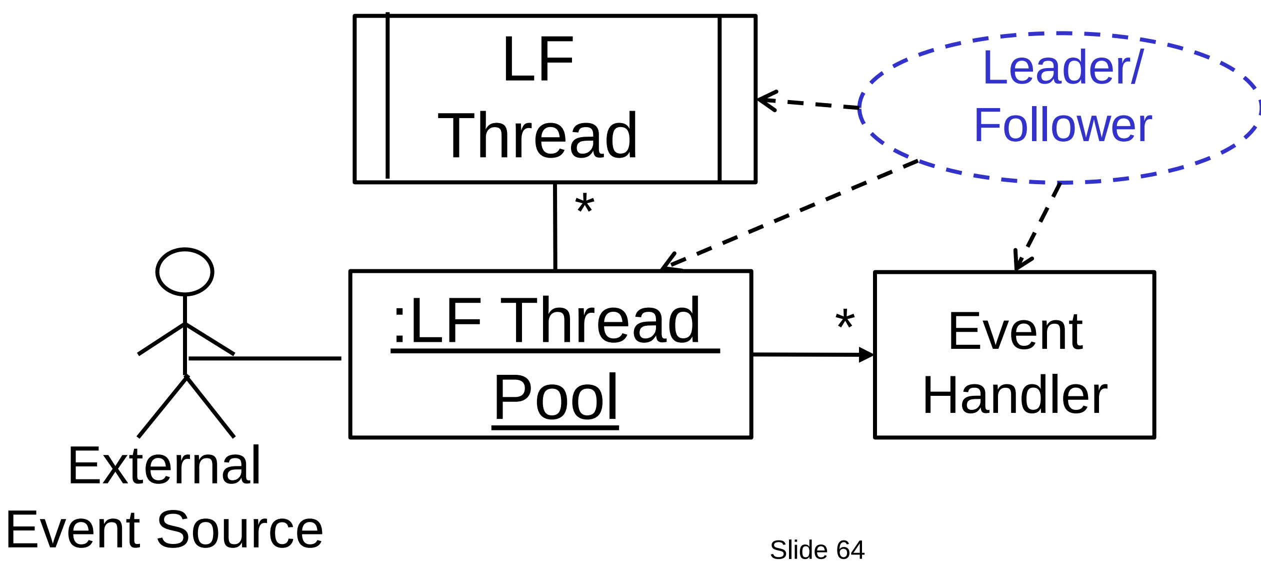
- Active Object POSA2 pattern
- Half-sync/Half-async POSA2 pattern
- Leader/Follower POSA2 pattern

Wrapping up on Concurrency (1)



Wrapping up on Concurrency (2)





Wrapping up on Concurrency (3)