AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

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

POSA2: *Active Object Design Pattern*

Version: 17-02-2015

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

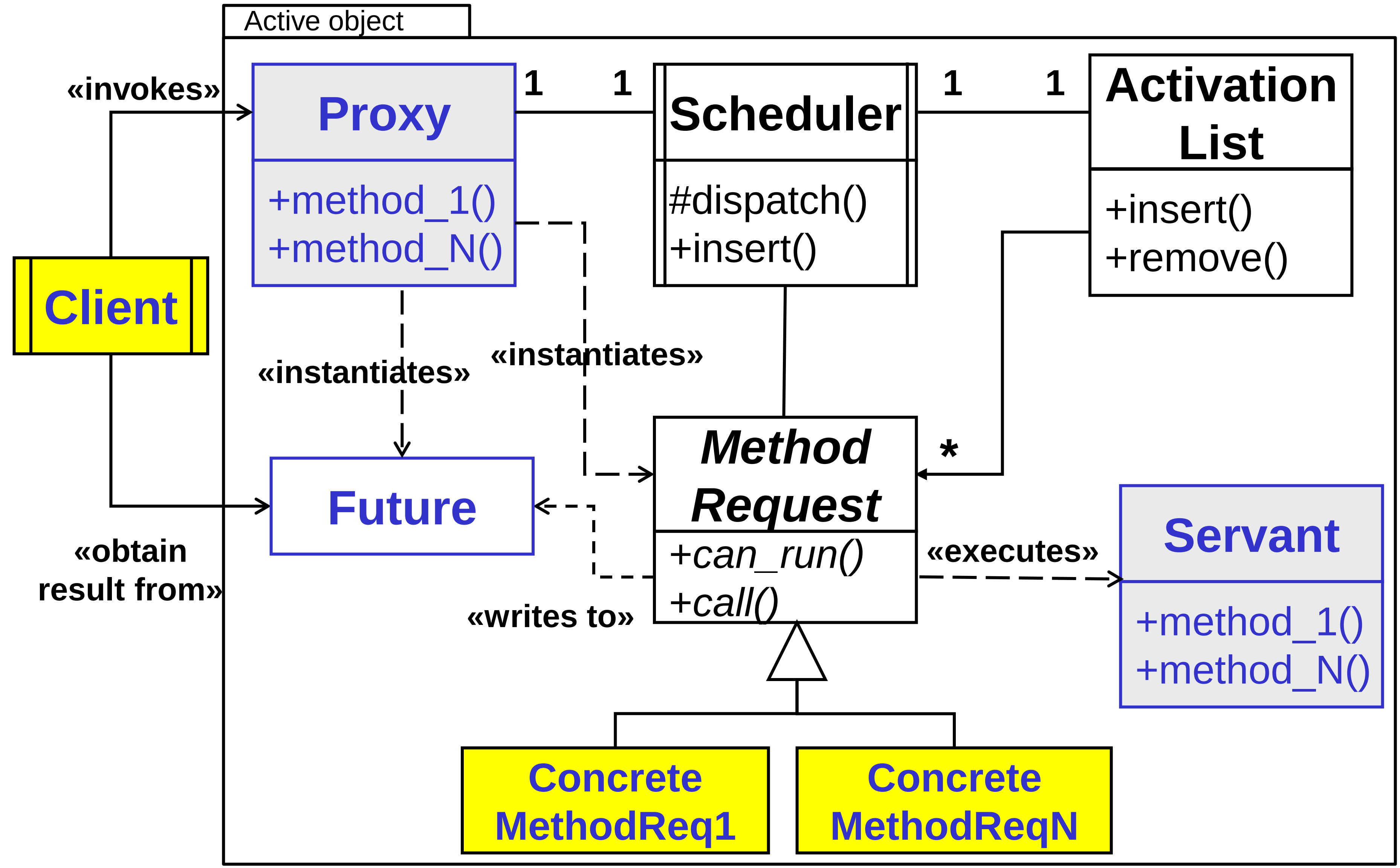
Problem

- 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



AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Active Object Dynamics

1. Method request and scheduling

2. Method execution

3. Completion

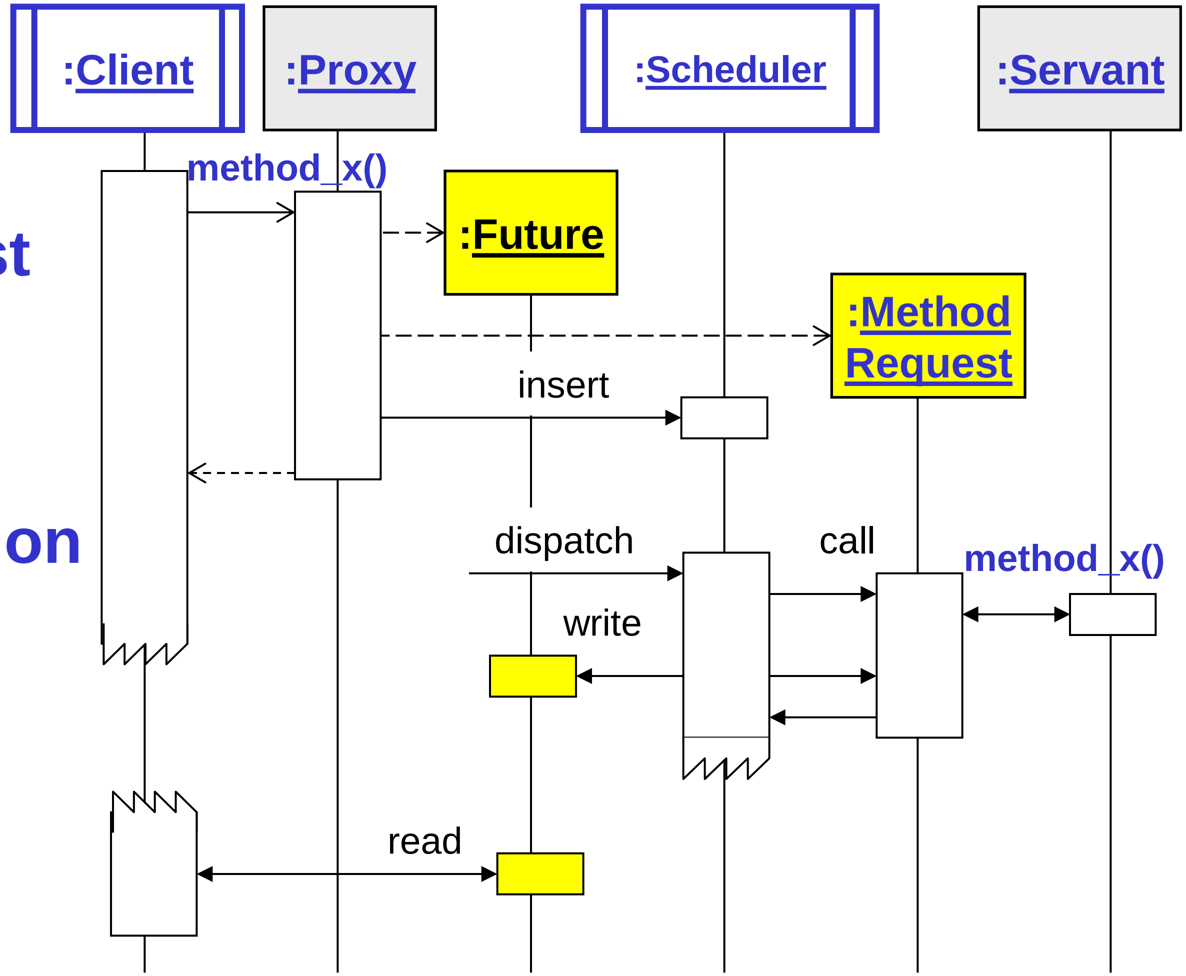


Image Acquisition Example

- 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

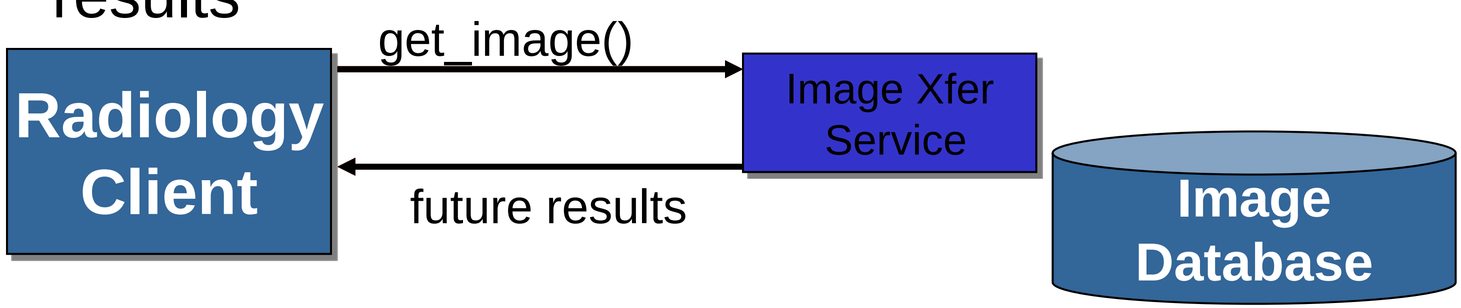
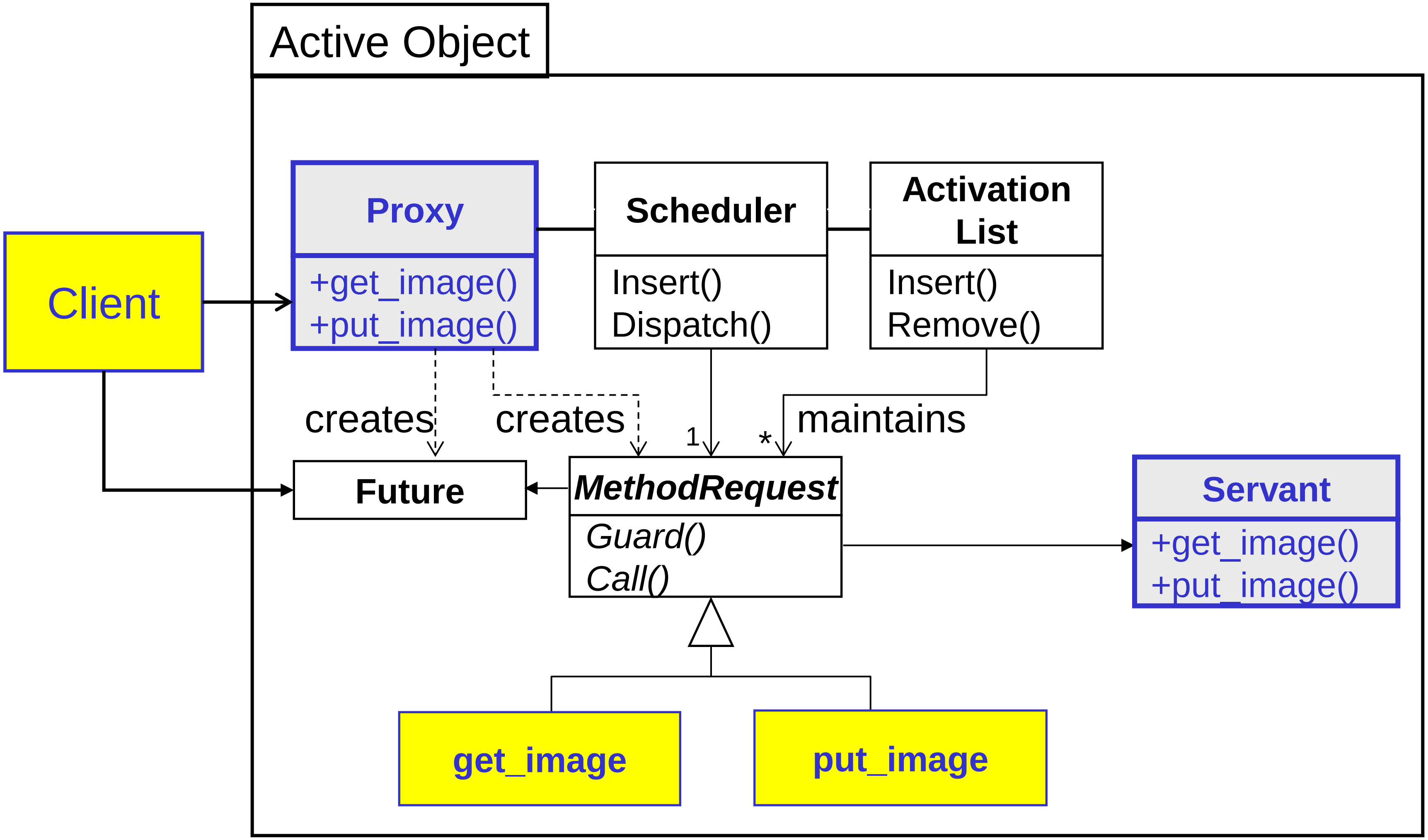


Image Acquisition Example



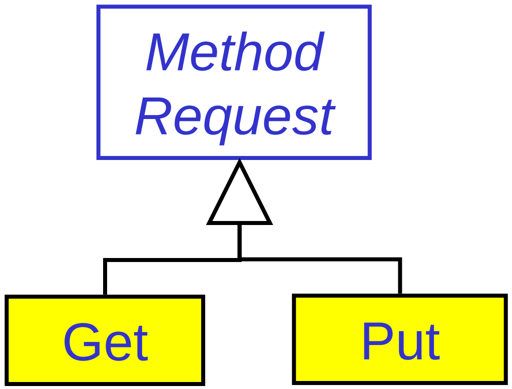
Implementation Steps

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_;
};
```

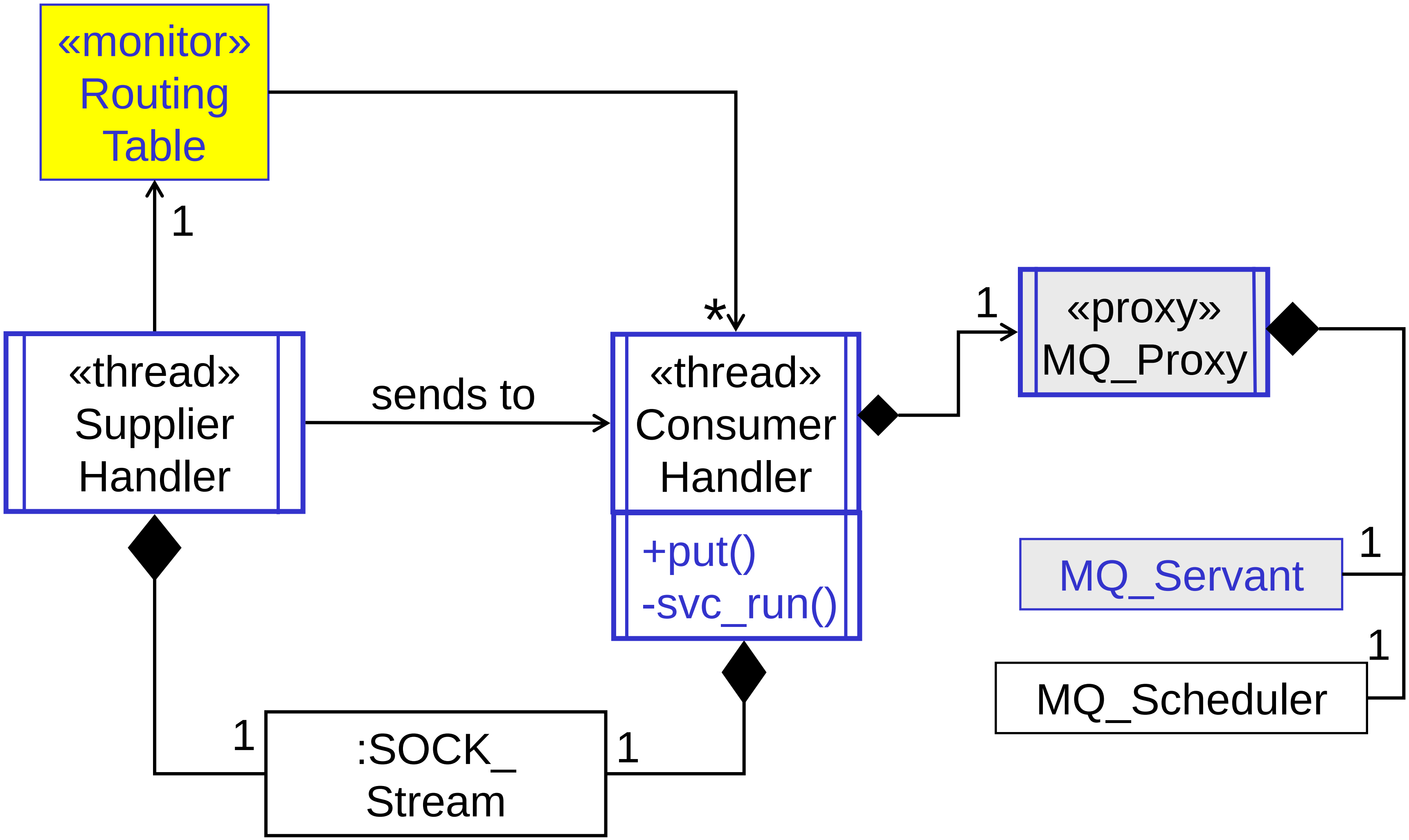
Class MQ_Scheduler

```
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
   }
};
```

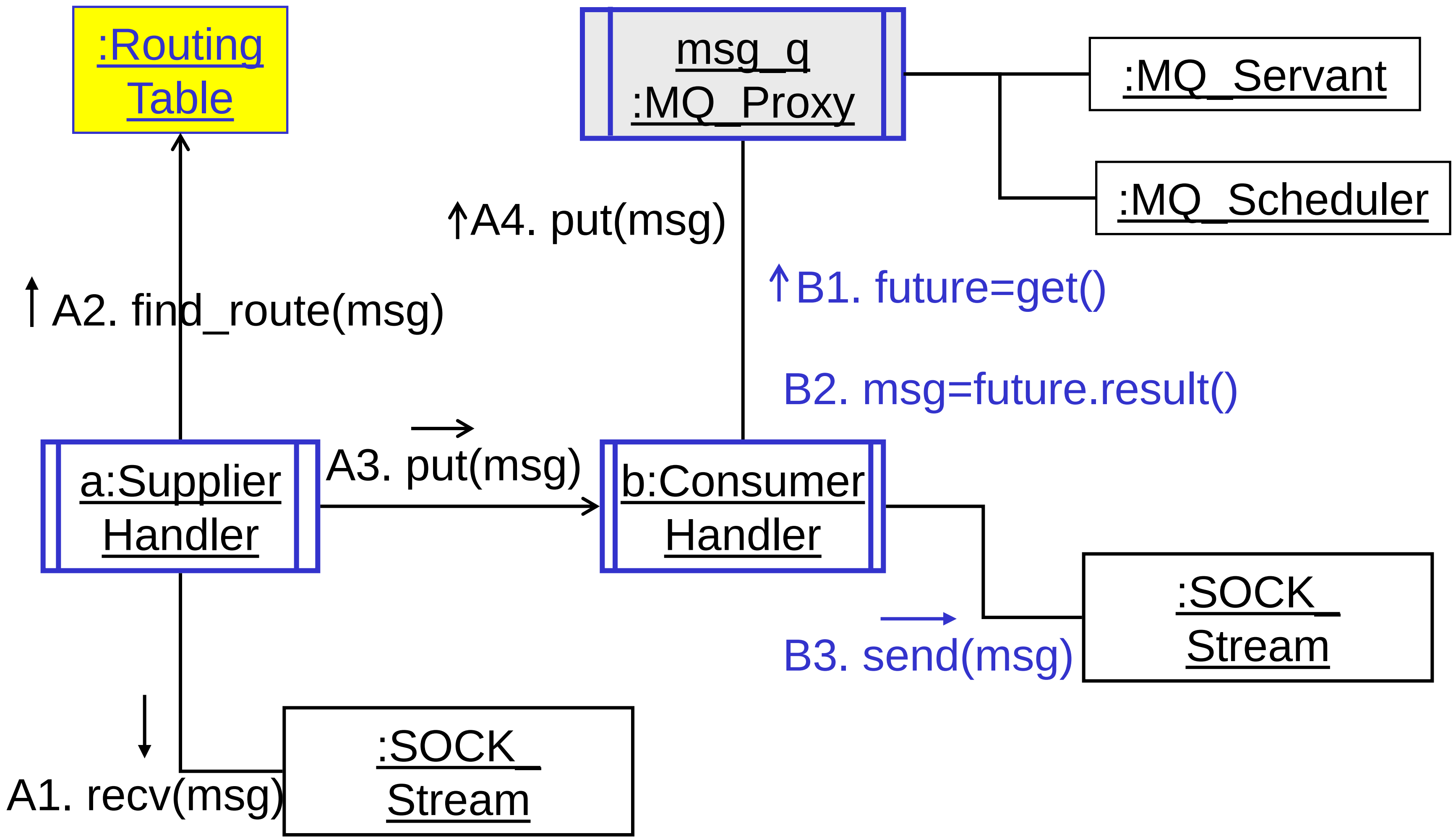
MQ_Scheduler::dispatch()

```
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);
}
```

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

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

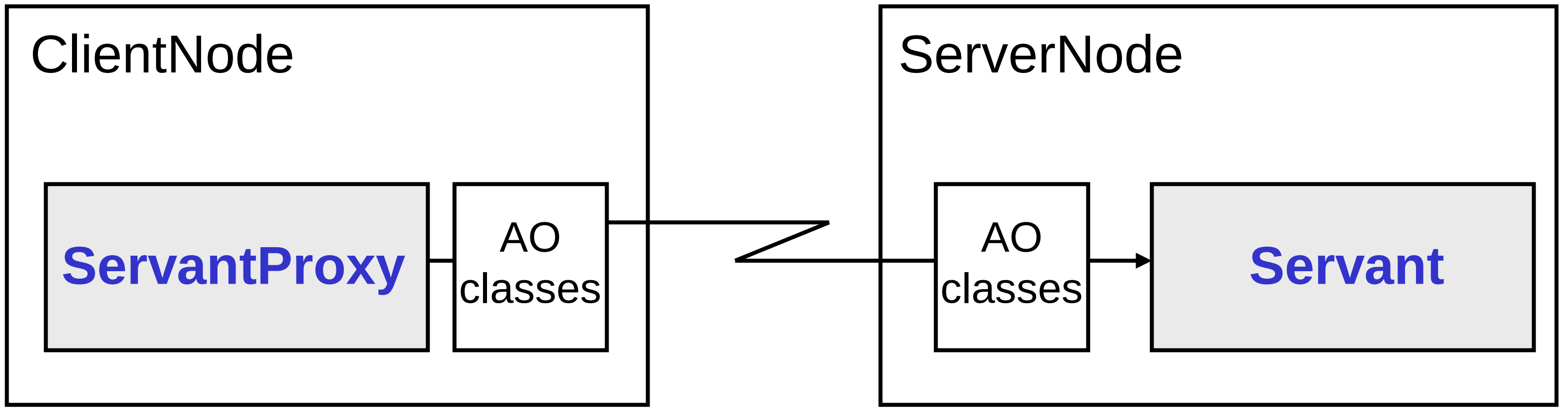
```
class MQ_Scheduler {
public:
   MQ_Scheduler(size_t size) : servant_(size), act_list_(size) { }

   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_;
};
```

Polymorhic **Future** Template Class

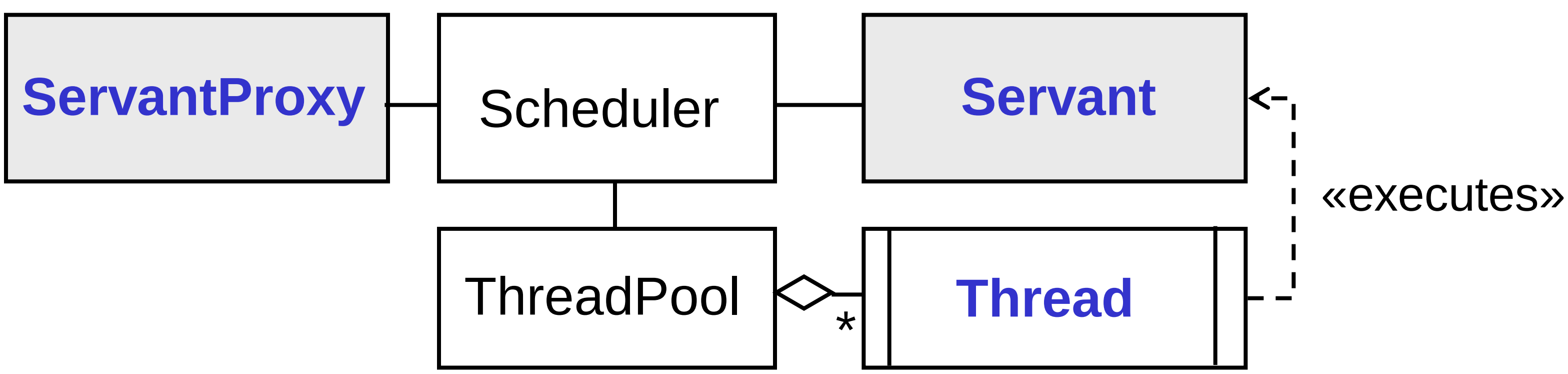
```
template <class TYPE>
class Future {
public:
    Future();
    Future(const Future<TYPE> &r);
    ~Future();
    void operator = (const Future<TYPE> &r);
     void cancel();
     // block upto <timeout> time waiting to obtain result
     TYPE result(Time_Value *timeout =0) const;
private:
     //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

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

Relation to other POSA2 patterns