Android Concurrency: The Active Object Pattern

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

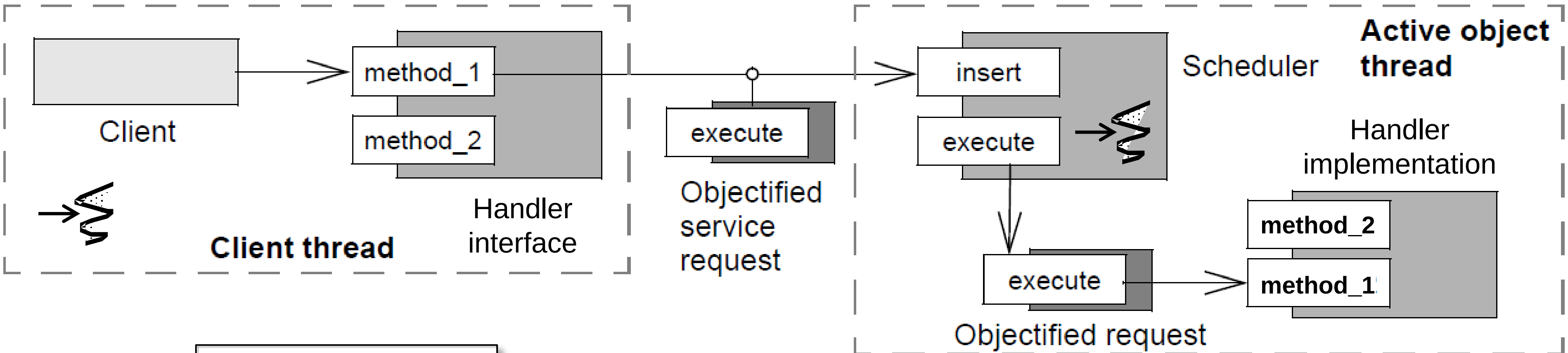
Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

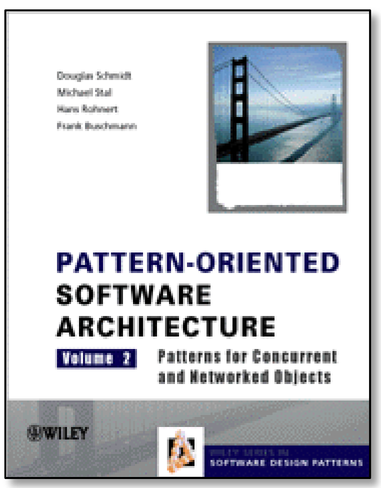
CS 282 Principles of Operating Systems II

Systems Programming for Android

Learning Objectives in this Part of the Module

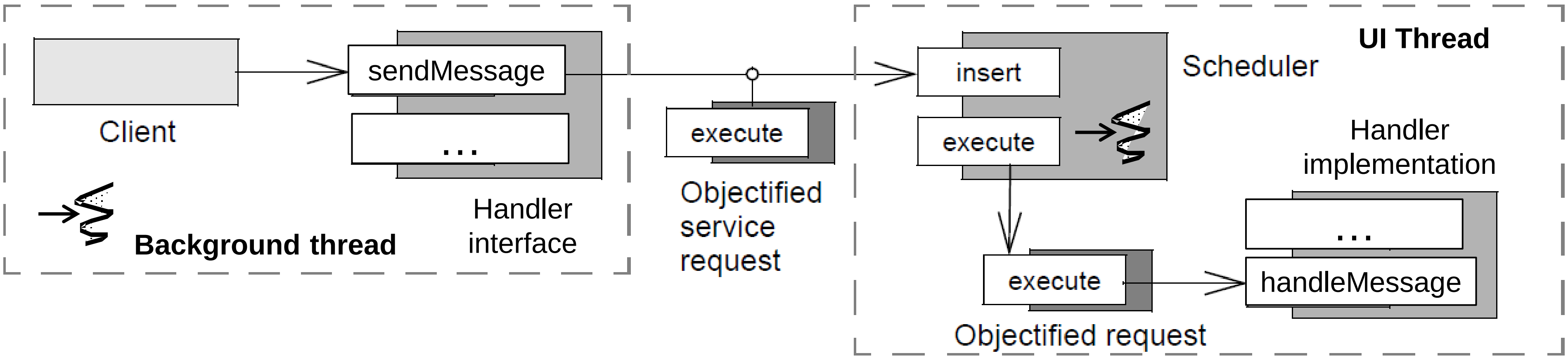
- Understand the *Active Object* pattern





Learning Objectives in this Part of the Module

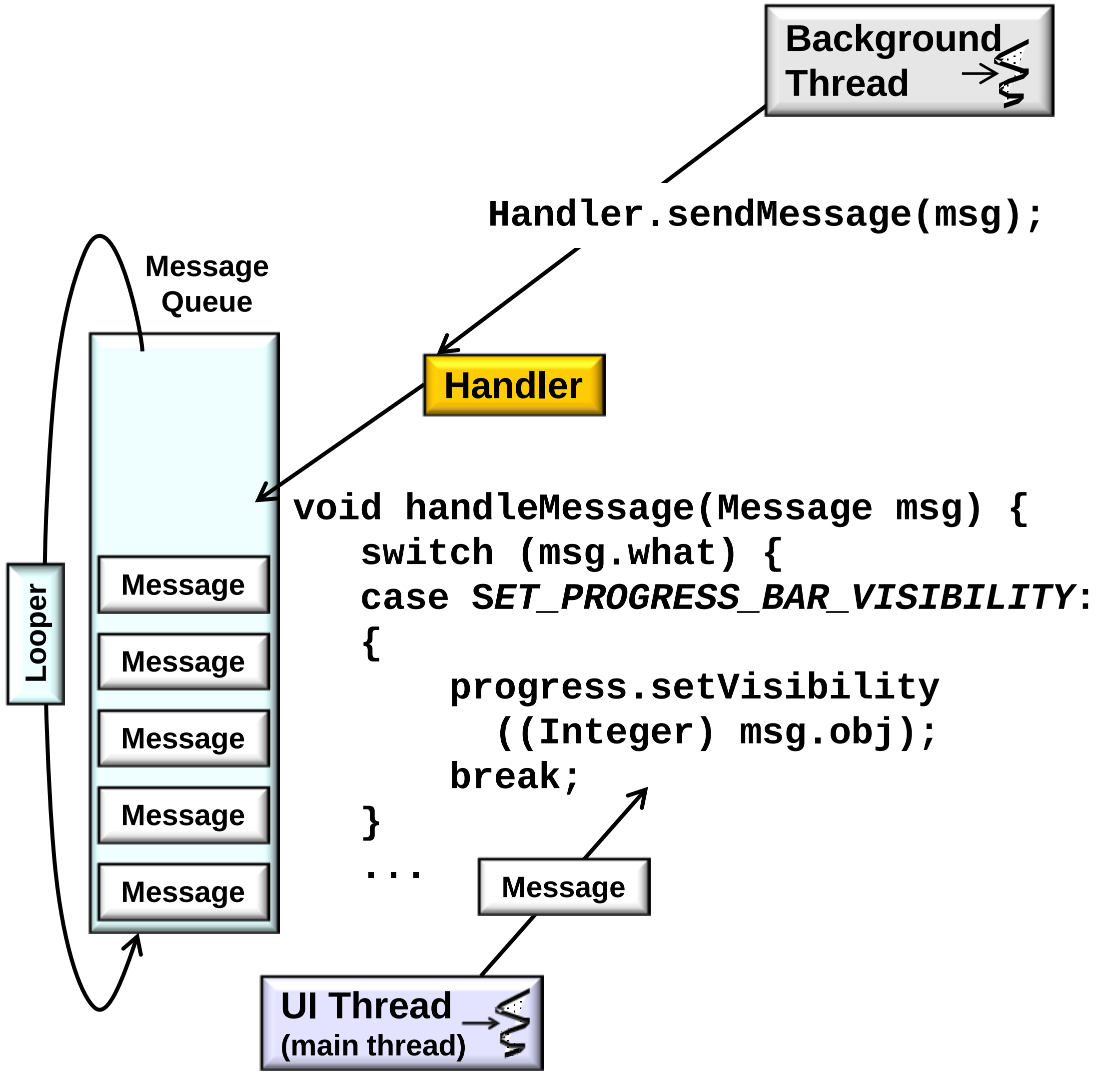
- Understand the *Active Object* pattern & how it's applied in Android



Challenge: Invoking Methods in Another Thread

Context

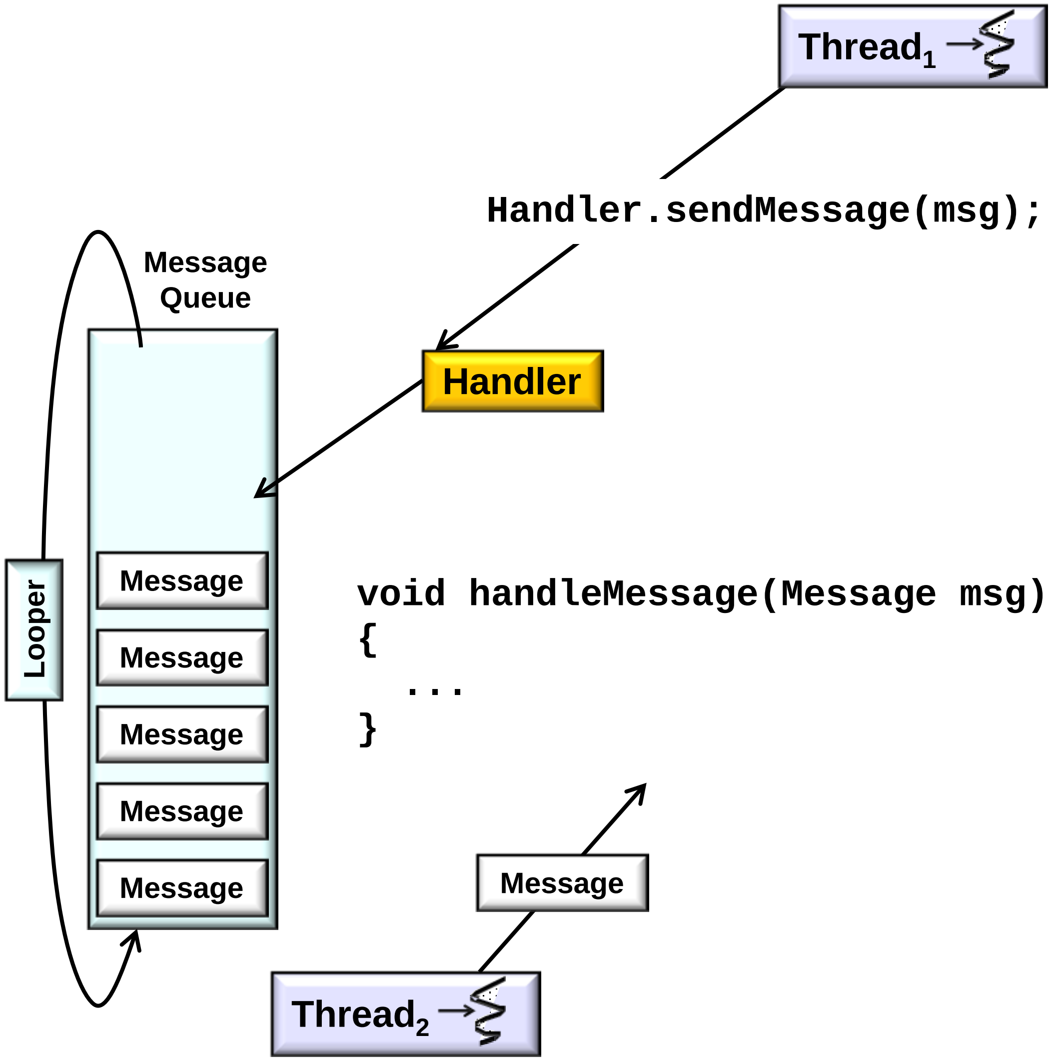
- Android clients that access objects running in separate threads of control
  - A “client” is any Android code that invokes a object’s method, e.g.,
    - A background Thread invoking sendMessage() on a Handler associated with the UI Thread



Challenge: Invoking Methods in Another Thread

Context

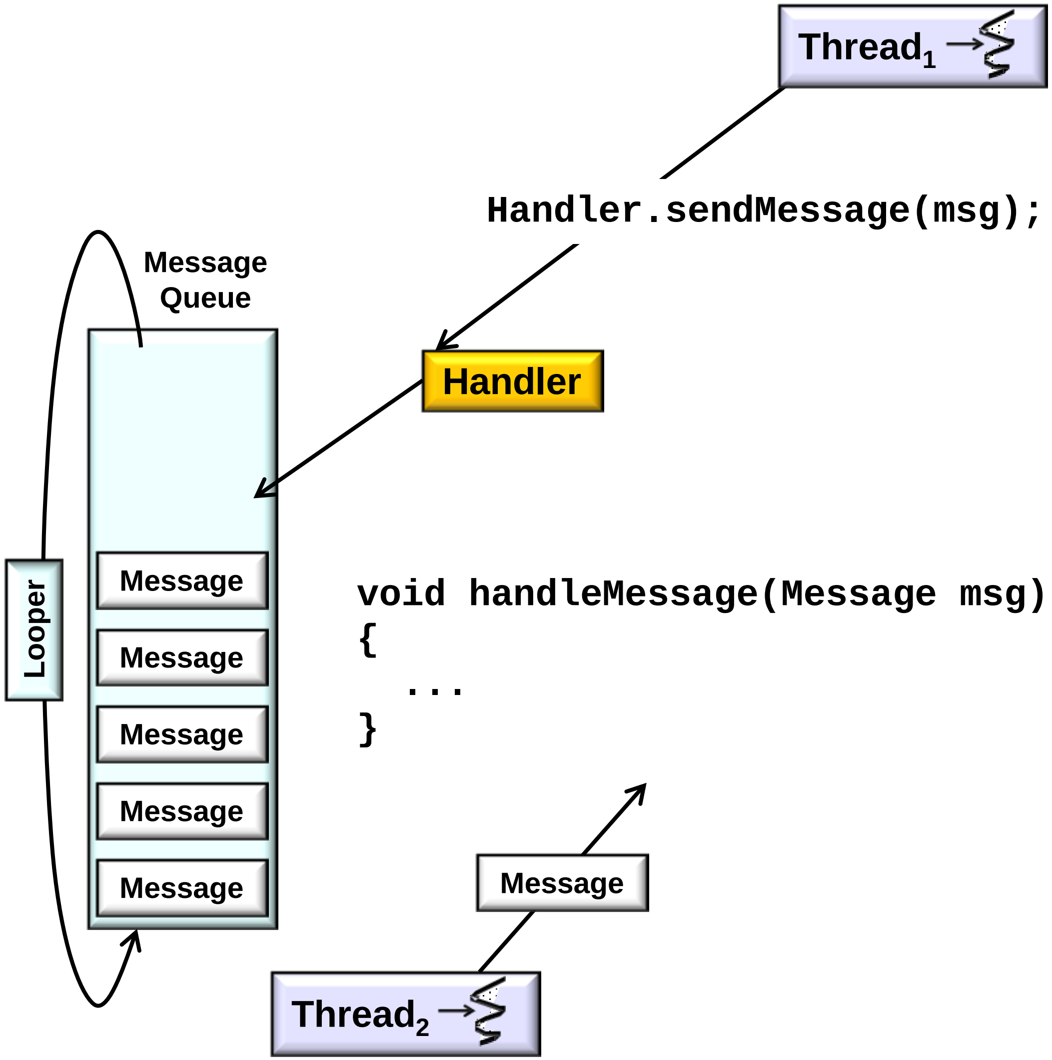
- Android clients that access objects running in separate threads of control
  - A "client" is any Android code that invokes a object's method, e.g.,
    - A background Thread invoking sendMessage() on a Handler associated with the UI Thread
    - More generally, any Threads that interact via Handlers/Messages



Challenge: Invoking Methods in Another Thread

Problems

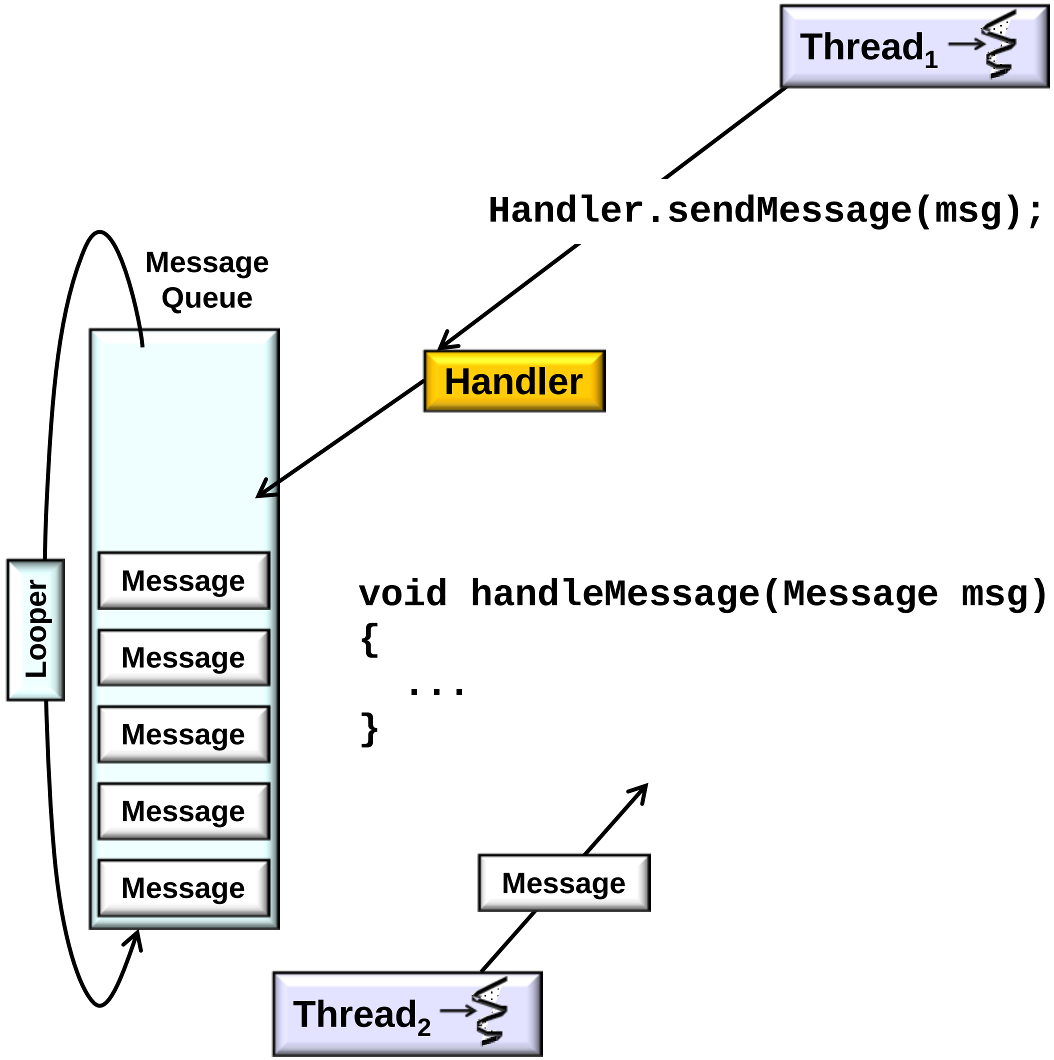
- Leveraging the parallelism available on a hardware/software platform (relatively) transparently



Challenge: Invoking Methods in Another Thread

Problems

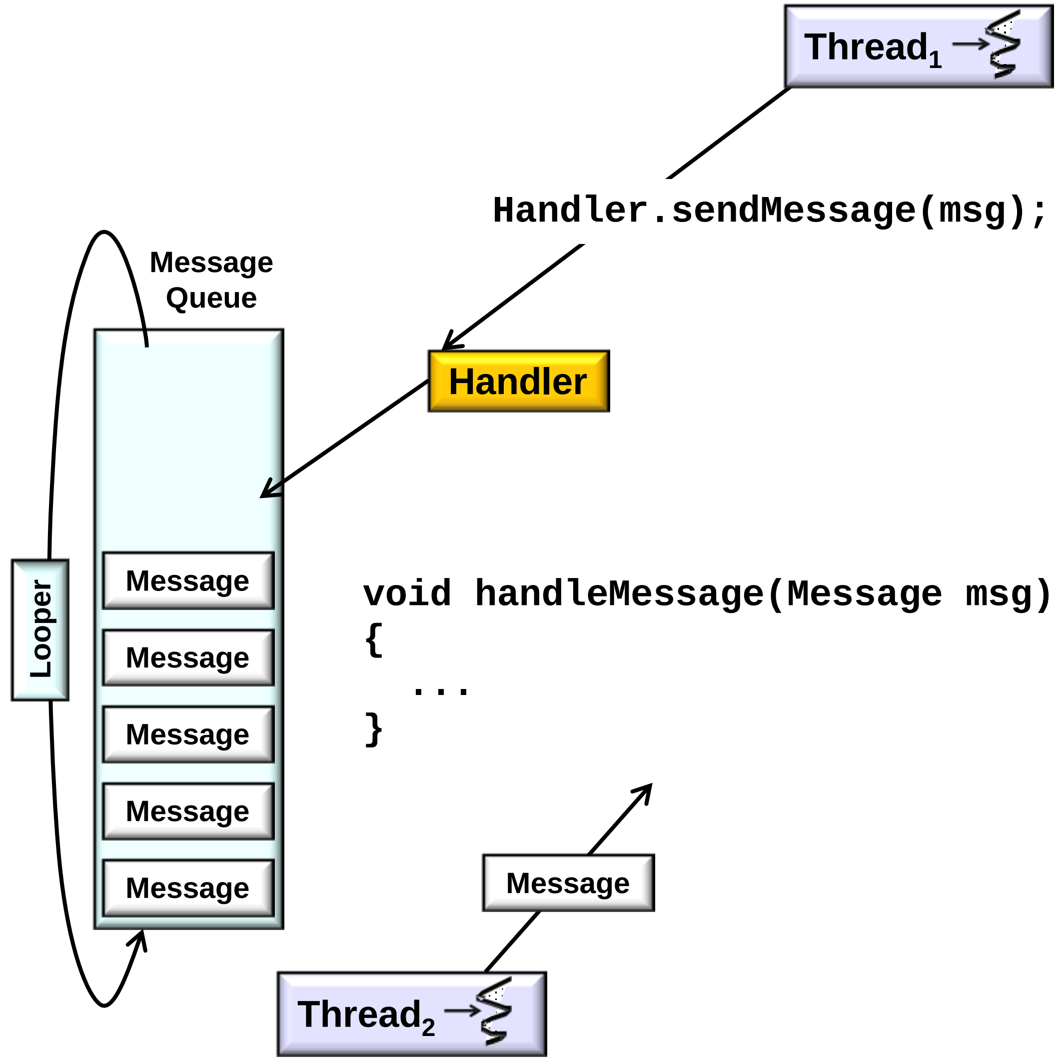
- Leveraging the parallelism available on a hardware/software platform (relatively) transparently
- Ensuring that processing-intensive methods invoked on an object concurrently do not block the entire process



Challenge: Invoking Methods in Another Thread

Problems

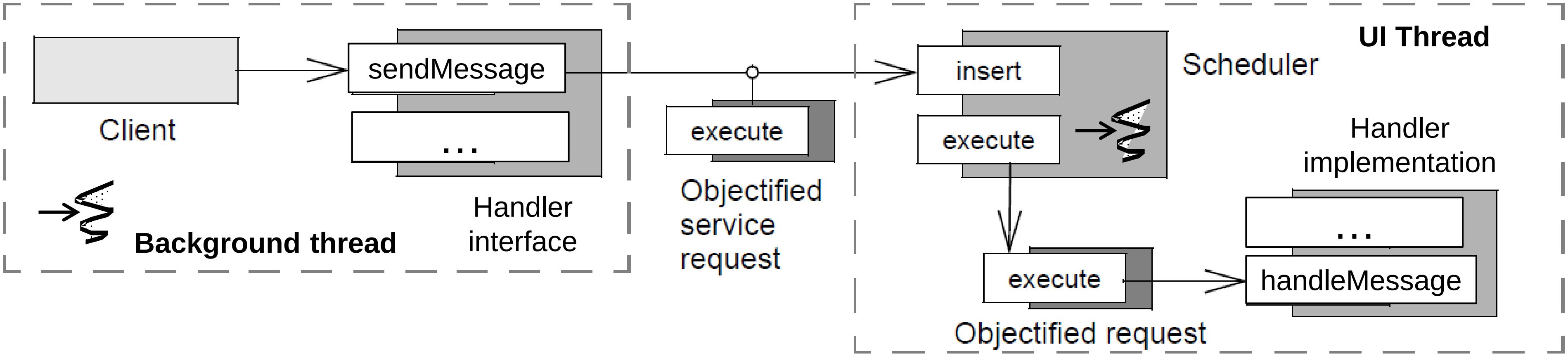
- Leveraging the parallelism available on a hardware/software platform (relatively) transparently
- Ensuring that processing-intensive methods invoked on an object concurrently do not block the entire process
- Making synchronized access to shared objects easy & intuitive to program



Challenge: Invoking Methods in Another Thread

Solution

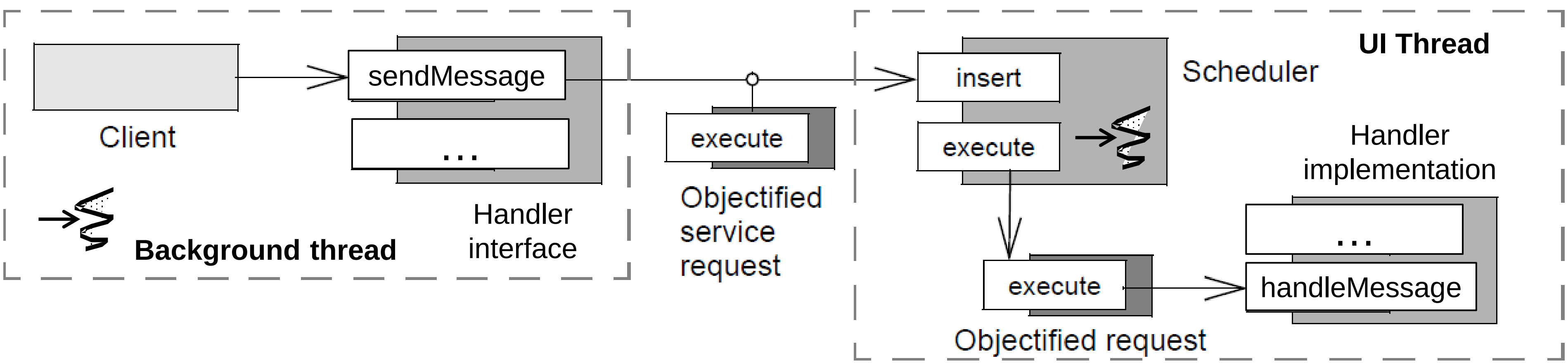
- Apply the *Active Object* pattern to decouple method invocation on the object from method execution
  - Method invocation should occur in the client's thread of control, whereas method execution should occur in a separate thread



Challenge: Invoking Methods in Another Thread

Solution

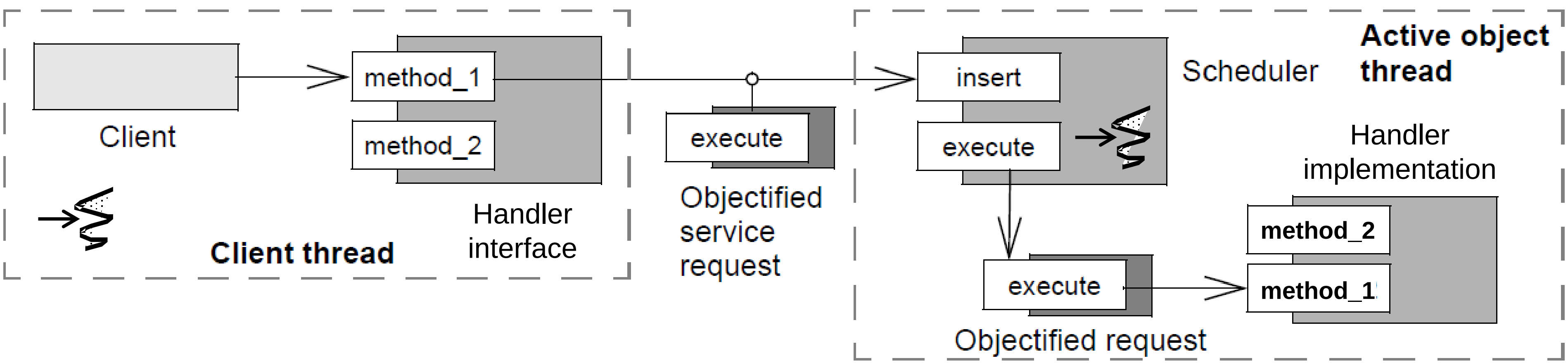
- Apply the *Active Object* pattern to decouple method invocation on the object from method execution
  - Method invocation should occur in the client's thread of control, whereas method execution should occur in a separate thread
  - The client should appear to invoke an ordinary method
    - i.e., the client shouldn't manipulate synchronization mechanisms explicitly



Active Object POSA2 Concurrency

Intent

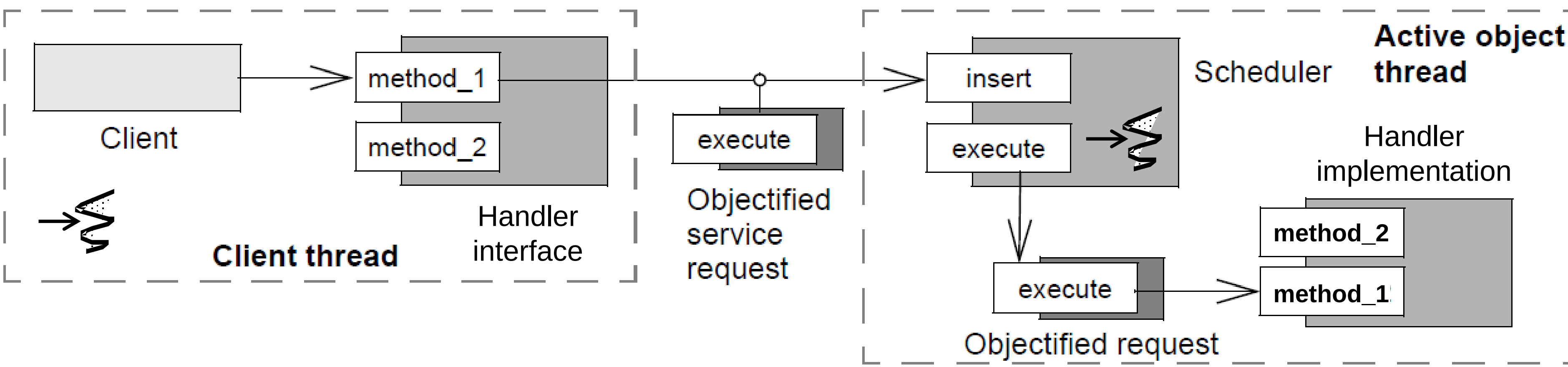
- Define service requests on components as the units of concurrency & run service requests on a component in different thread(s) from the requesting client thread
- Enable the client & component to interact asynchronously to produce & consume service results



Active Object POSA2 Concurrency

Applicability

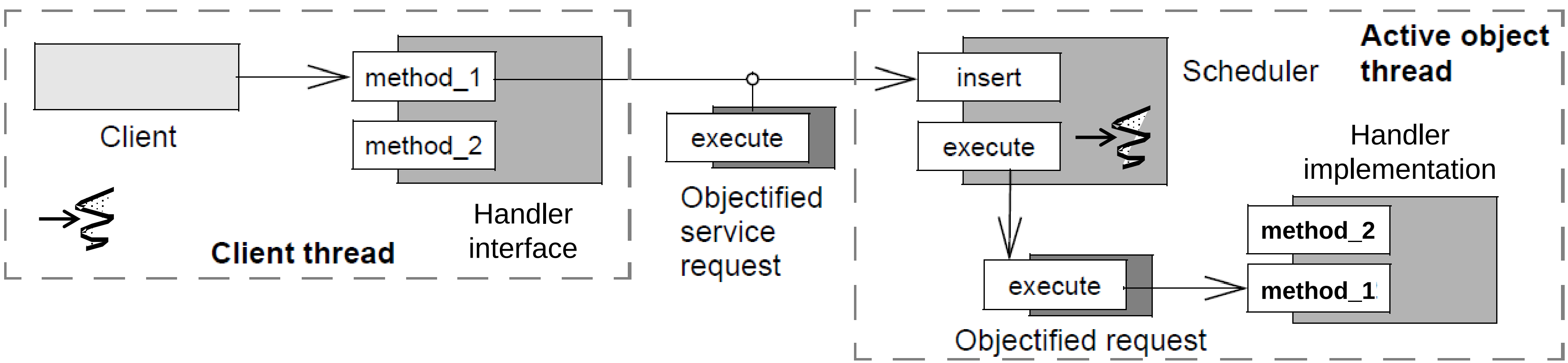
- When an object's interface methods should define its concurrency boundaries



Active Object POSA2 Concurrency

Applicability

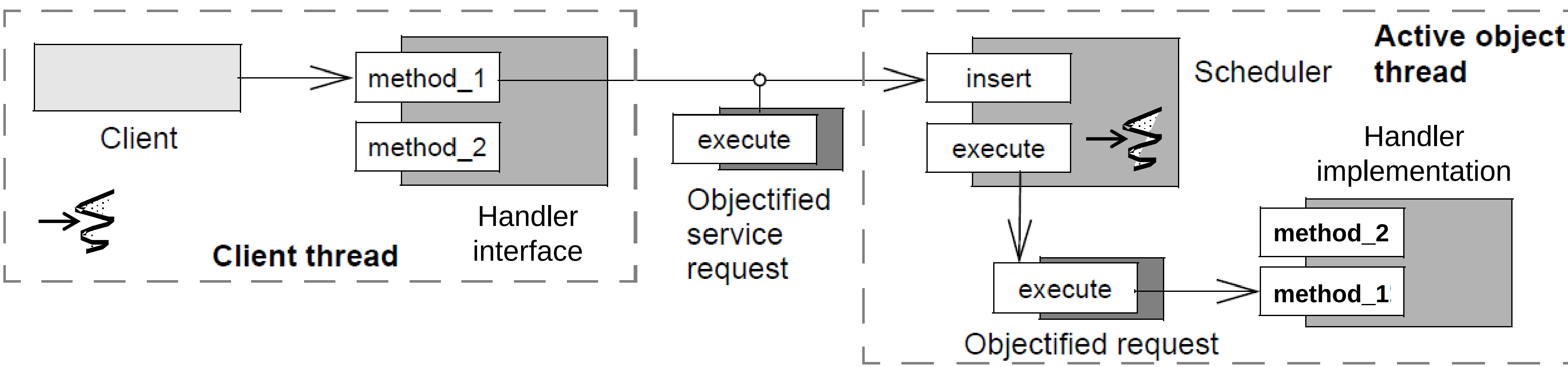
- When an object's interface methods should define its concurrency boundaries
- When objects should be responsible for method synchronization & scheduling transparently, without requiring explicit client intervention



Active Object POSA2 Concurrency

Applicability

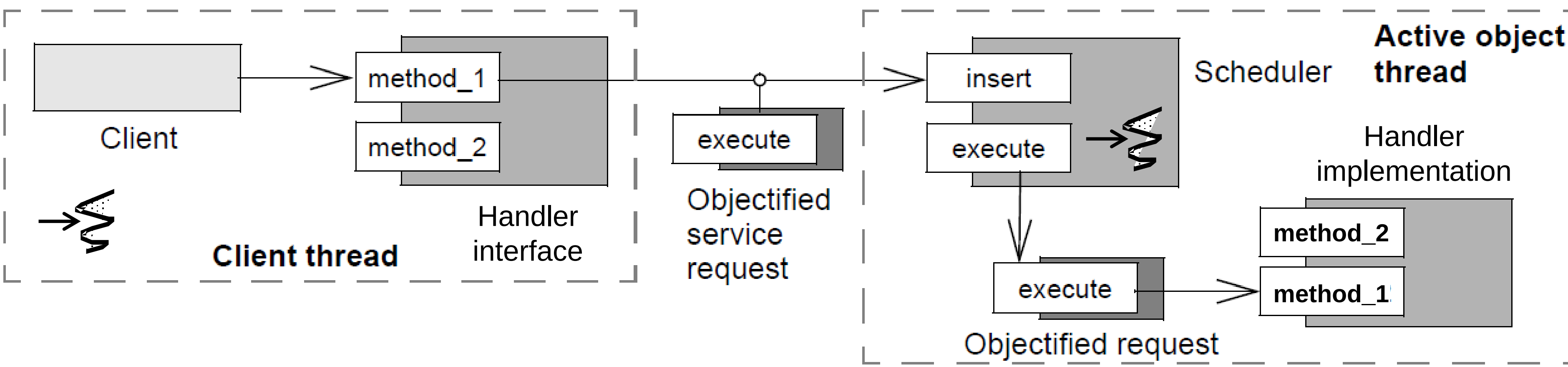
- When an object's interface methods should define its concurrency boundaries
- When objects should be responsible for method synchronization & scheduling transparently, without requiring explicit client intervention
- When an object's methods may block during their execution



Active Object POSA2 Concurrency

Applicability

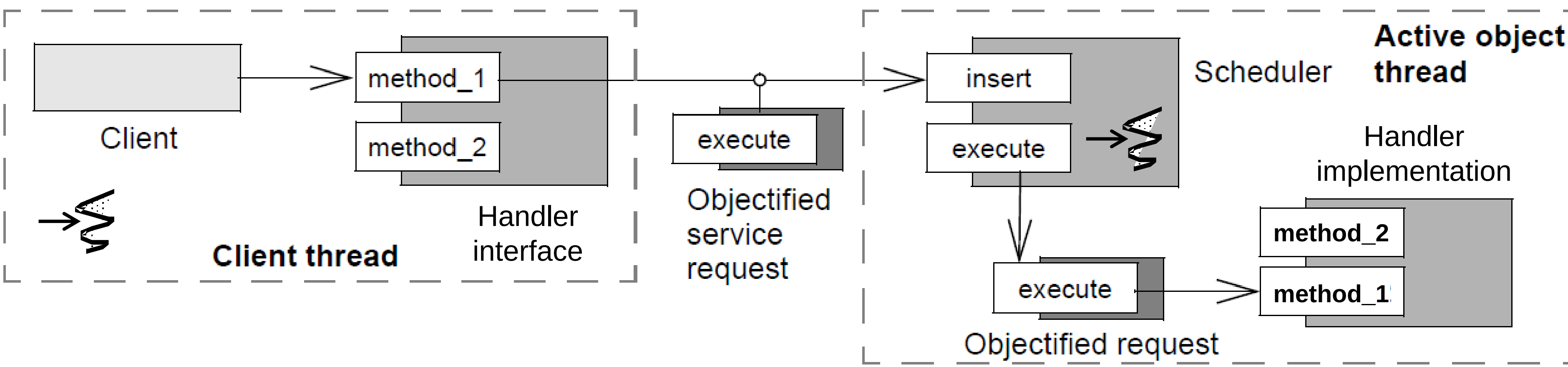
- When an object's interface methods should define its concurrency boundaries
- When objects should be responsible for method synchronization & scheduling transparently, without requiring explicit client intervention
- When an object's methods may block during their execution
- When multiple client method requests can run concurrently on an object



Active Object POSA2 Concurrency

Applicability

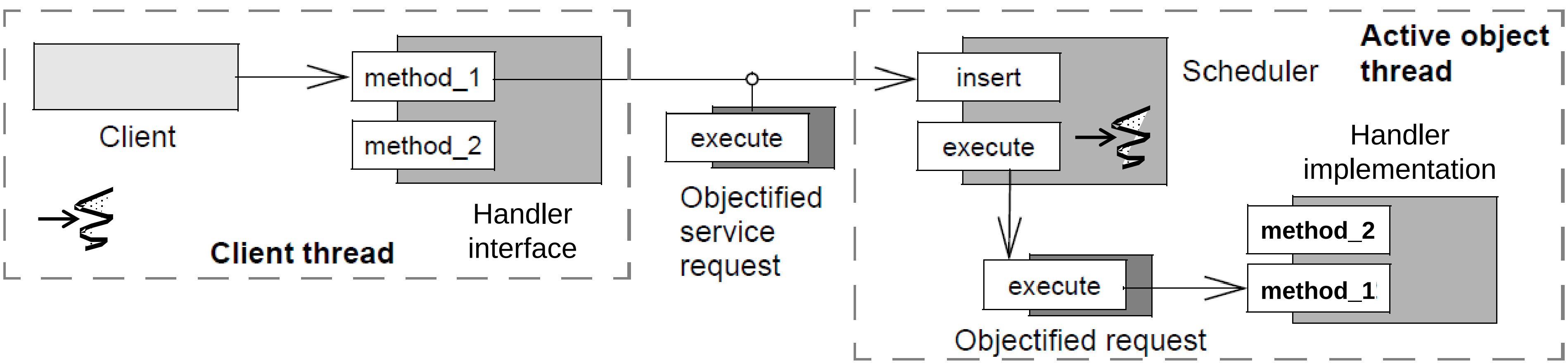
- When an object's interface methods should define its concurrency boundaries
- When objects should be responsible for method synchronization & scheduling transparently, without requiring explicit client intervention
- When an object's methods may block during their execution
- When multiple client method requests can run concurrently on an object
- When method invocation order might not match method execution order



Active Object POSA2 Concurrency

Applicability

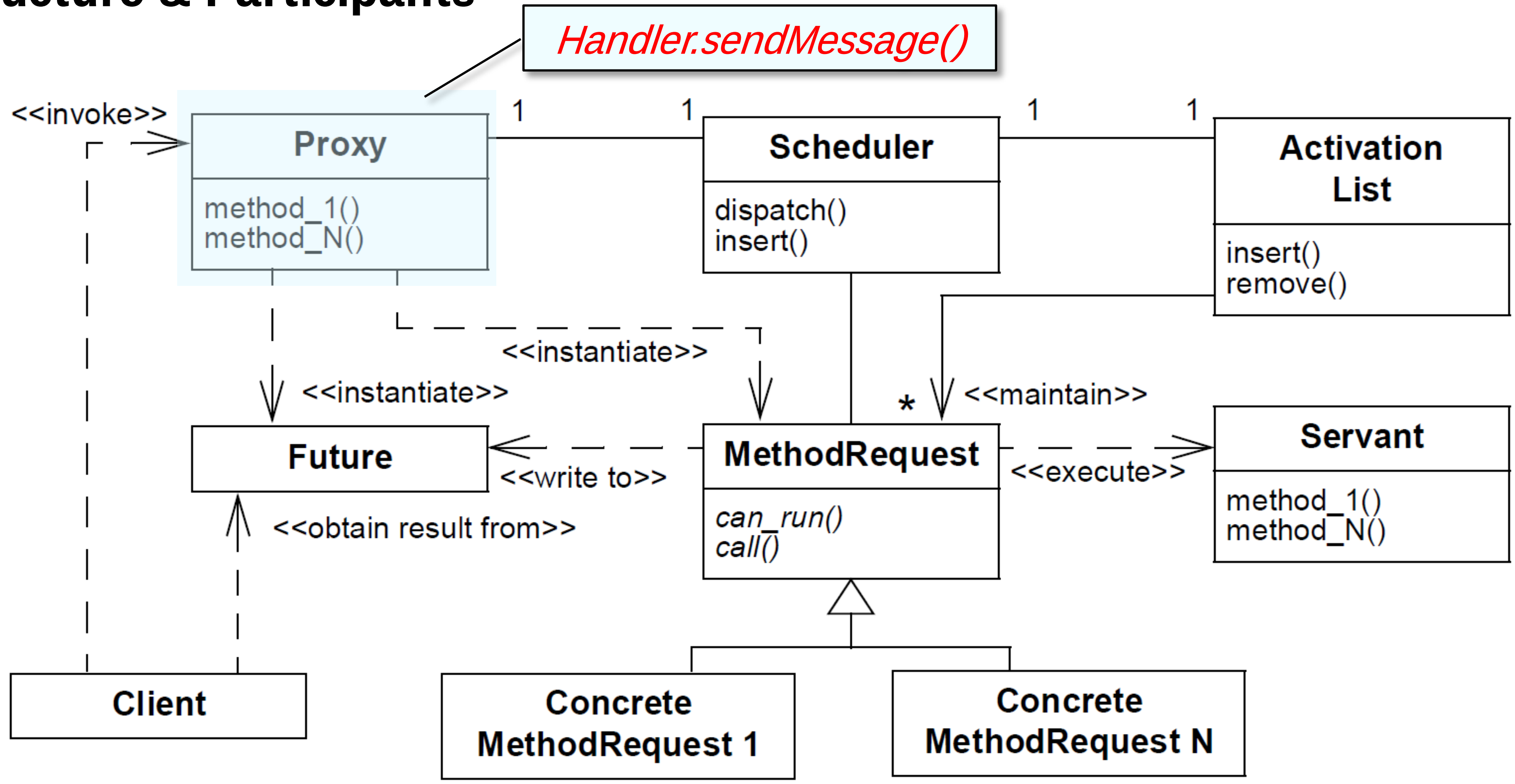
- When an object's interface methods should define its concurrency boundaries
- When objects should be responsible for method synchronization & scheduling transparently, without requiring explicit client intervention
- When an object's methods may block during their execution
- When multiple client method requests can run concurrently on an object
- When method invocation order might not match method execution order



Note the similarities between Active Object & Monitor Object wrt applicability

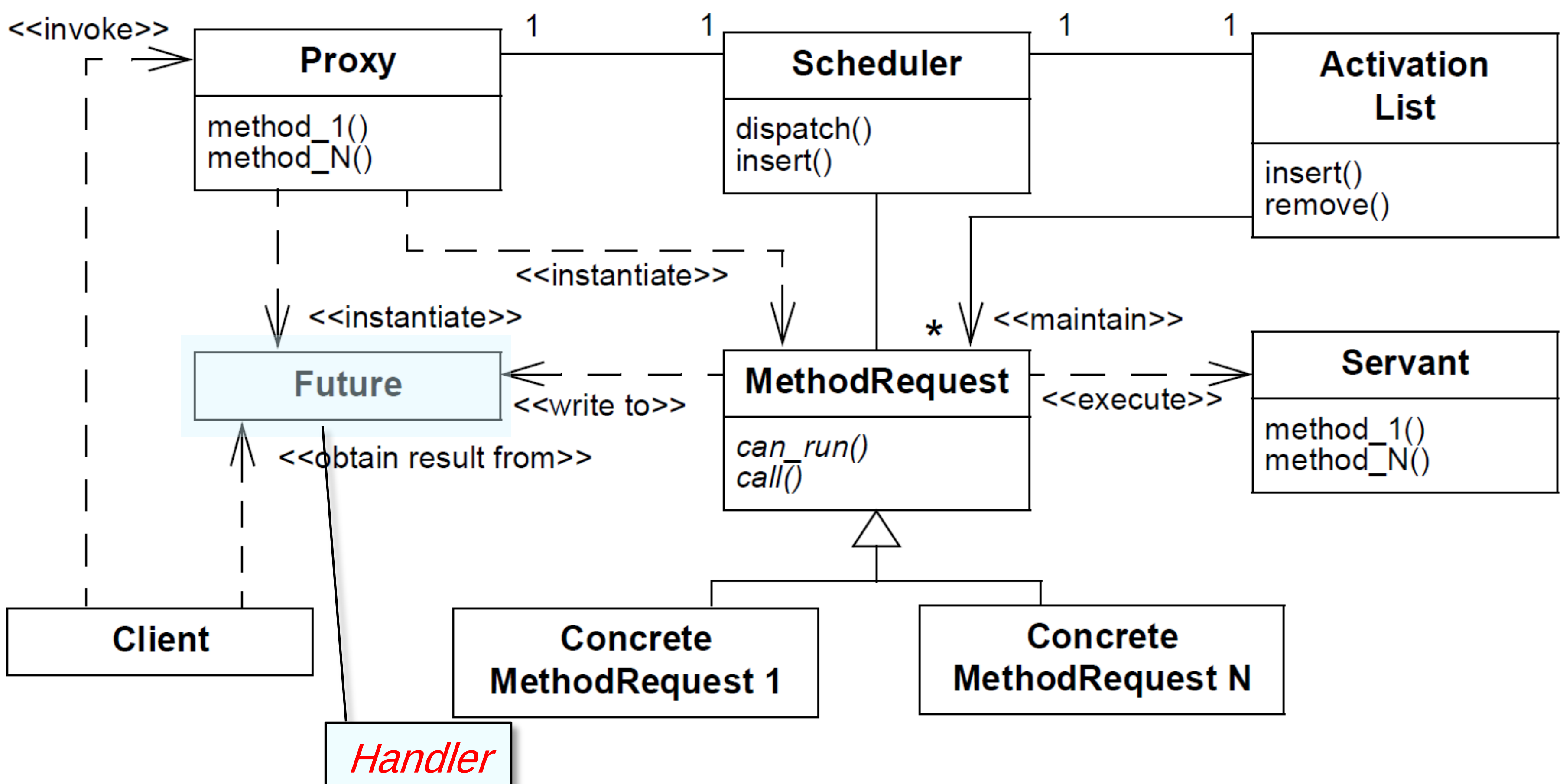
Active Object POSA2 Concurrency

Structure & Participants



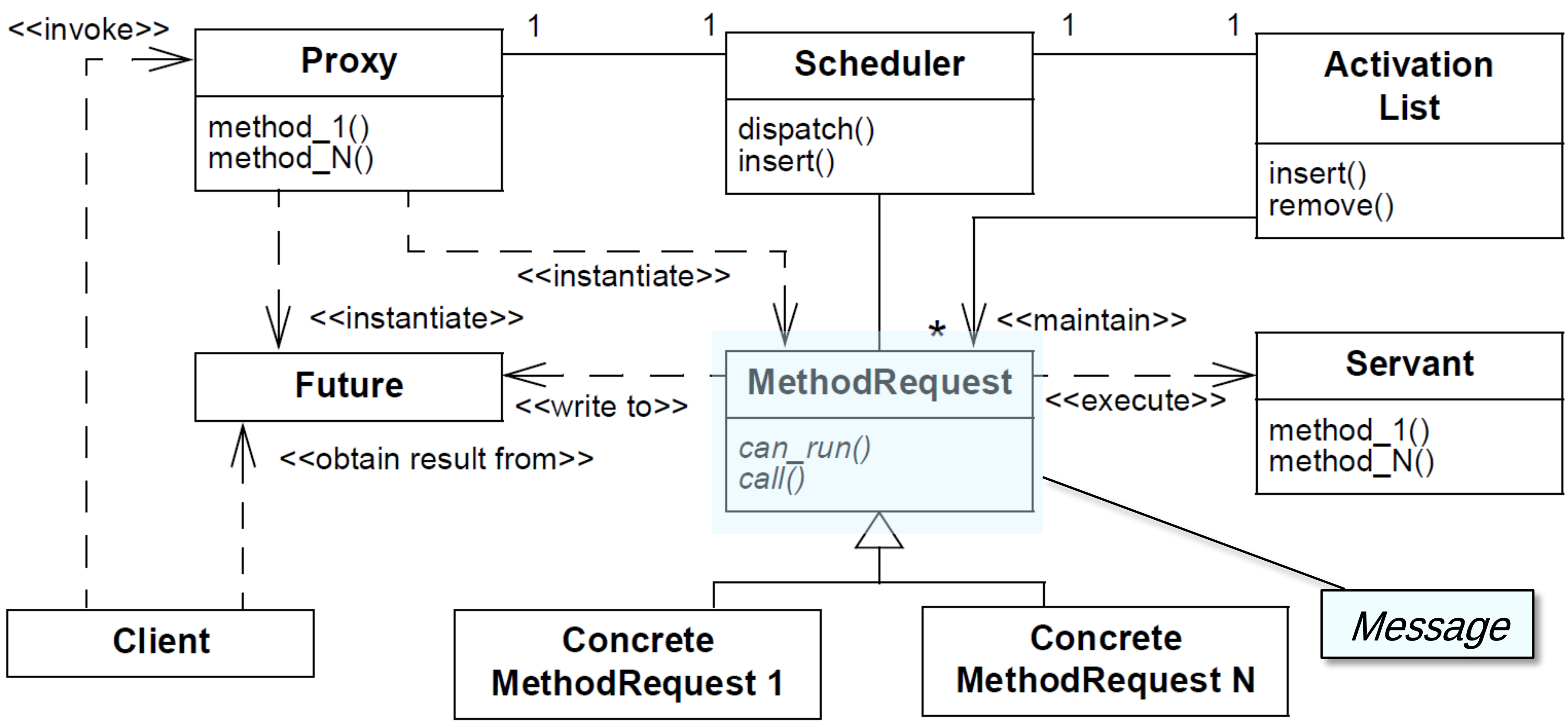
Active Object POSA2 Concurrency

Structure & Participants



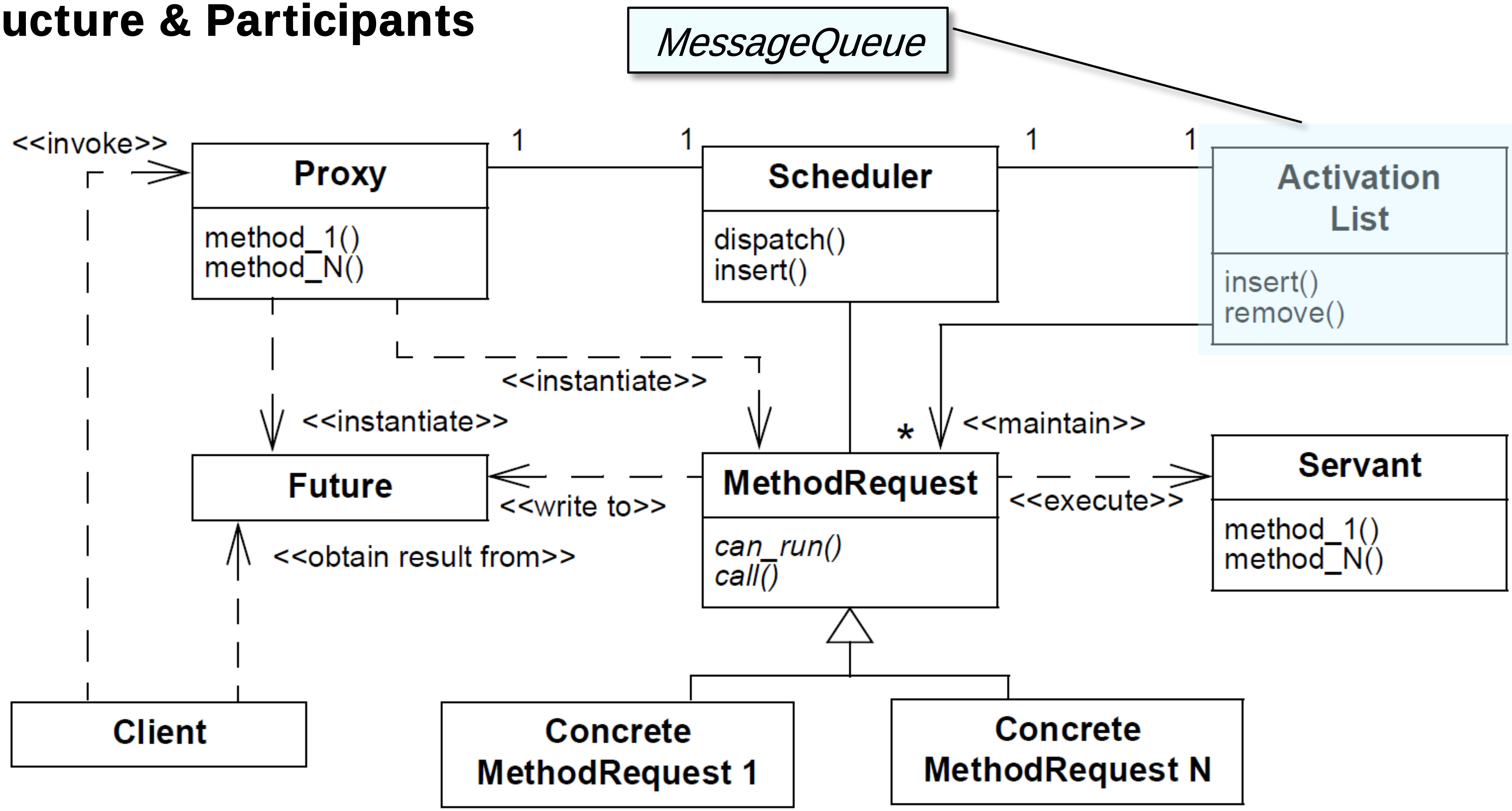
Active Object POSA2 Concurrency

Structure & Participants



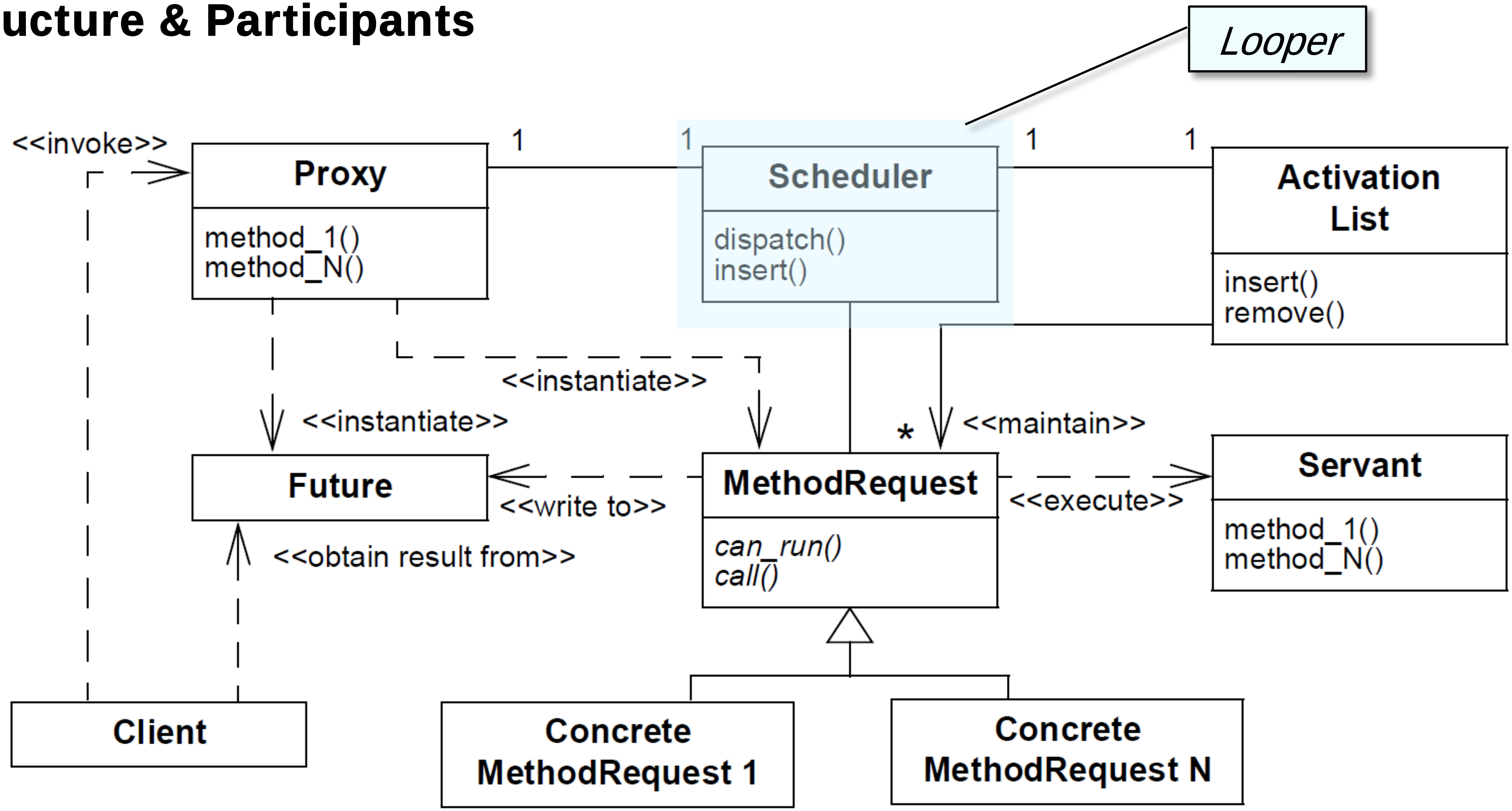
Active Object POSA2 Concurrency

Structure & Participants



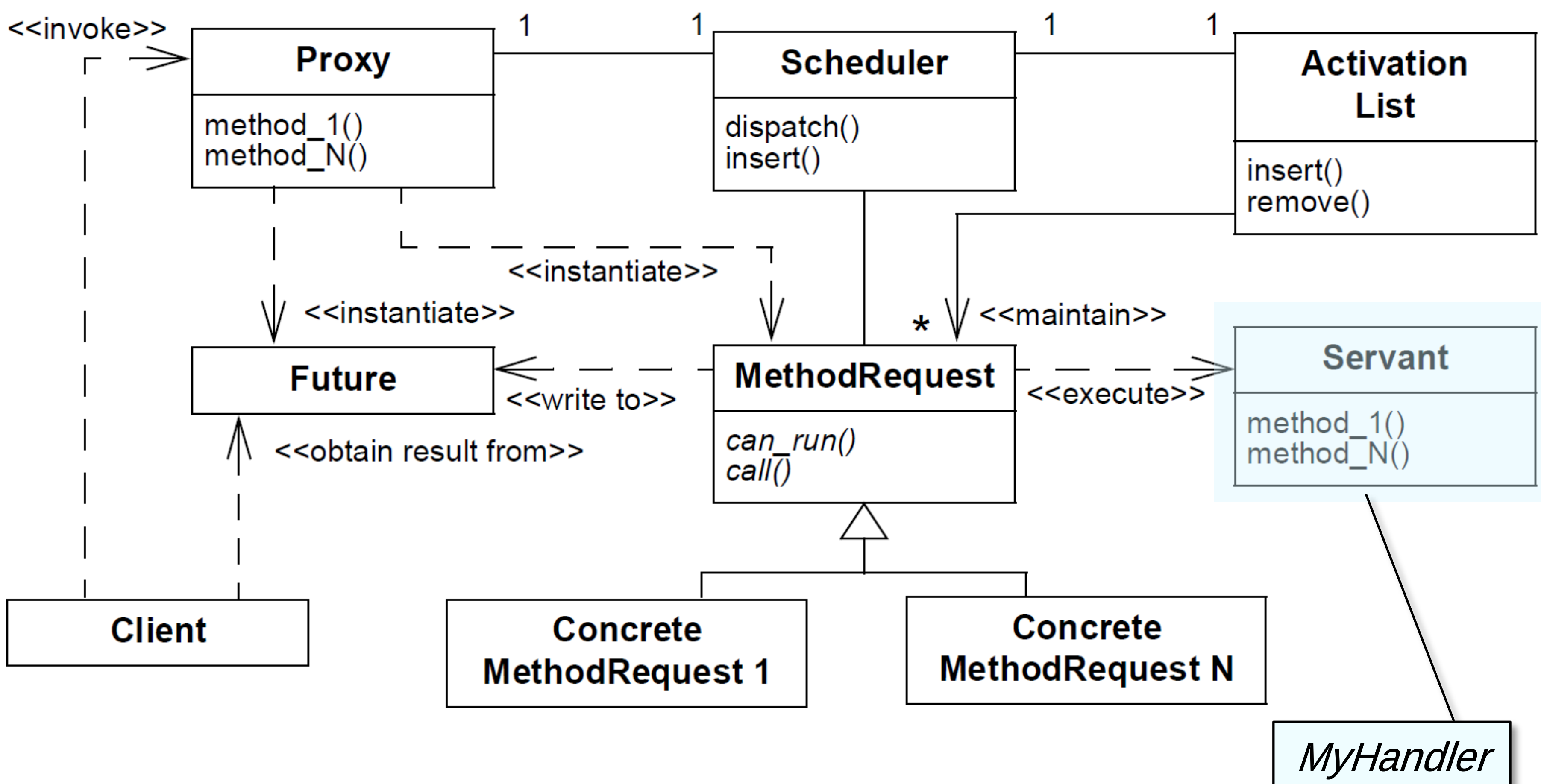
Active Object POSA2 Concurrency

Structure & Participants



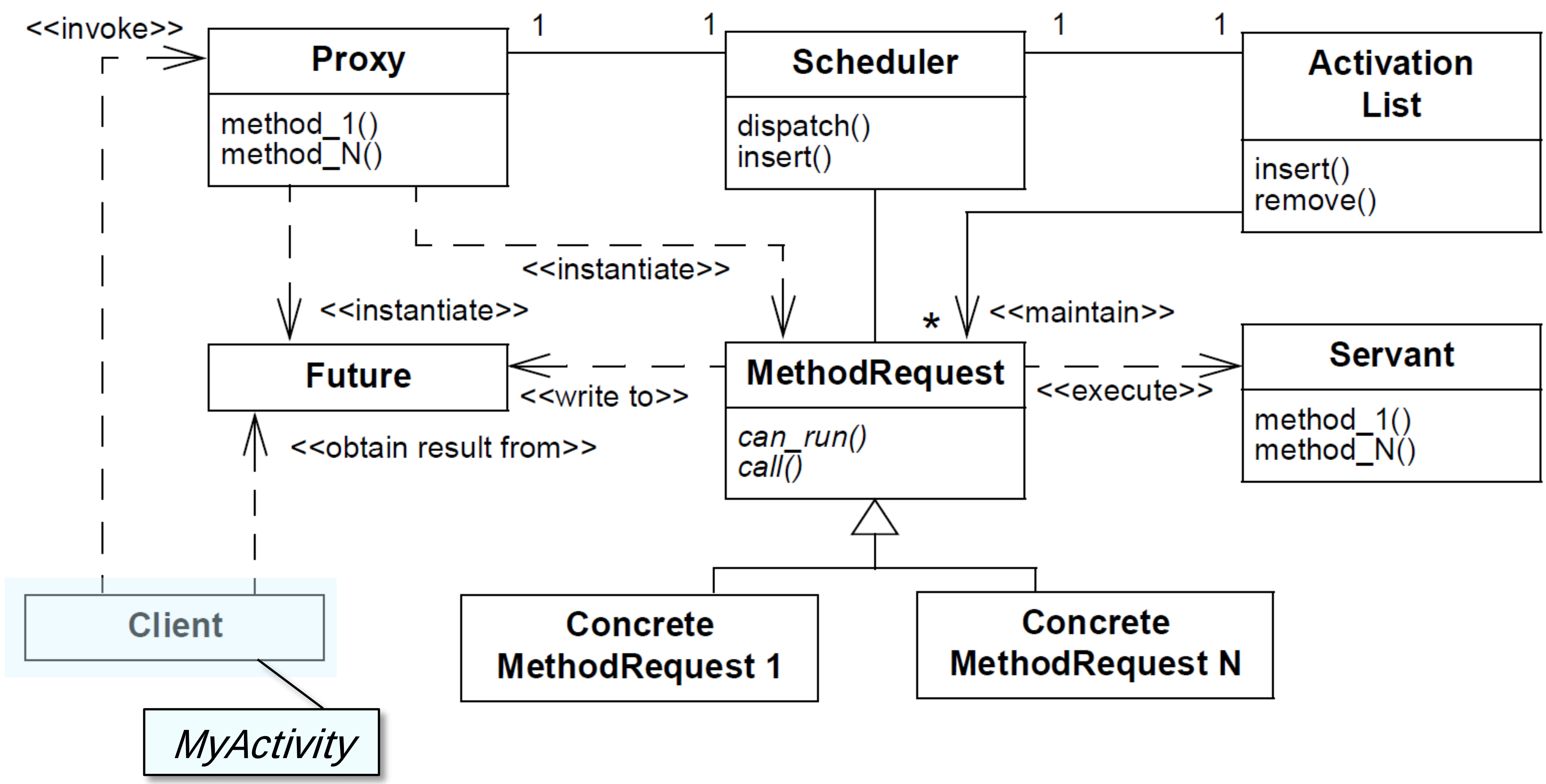
Active Object POSA2 Concurrency

Structure & Participants



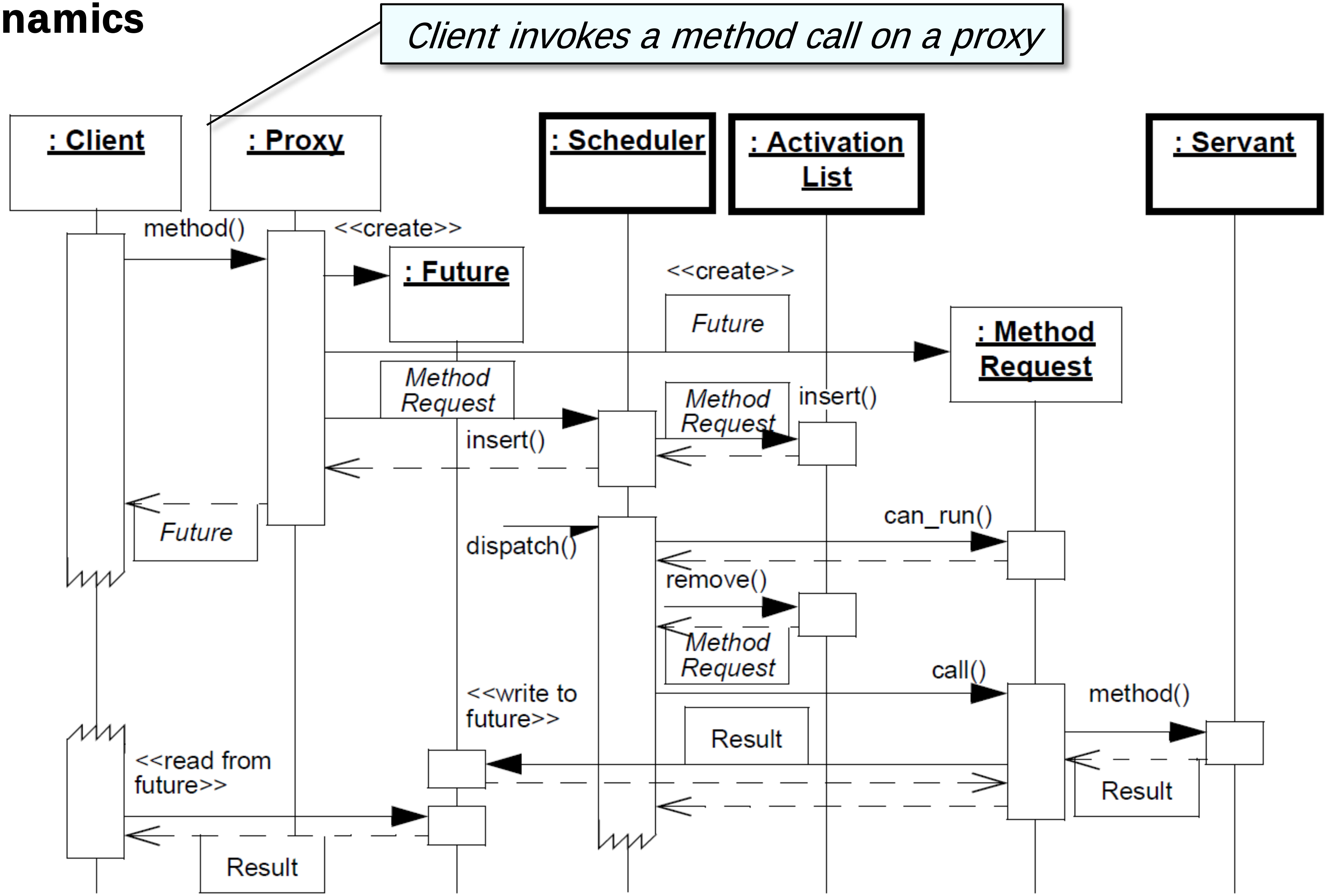
Active Object POSA2 Concurrency

Structure & Participants



Active Object POSA2 Concurrency

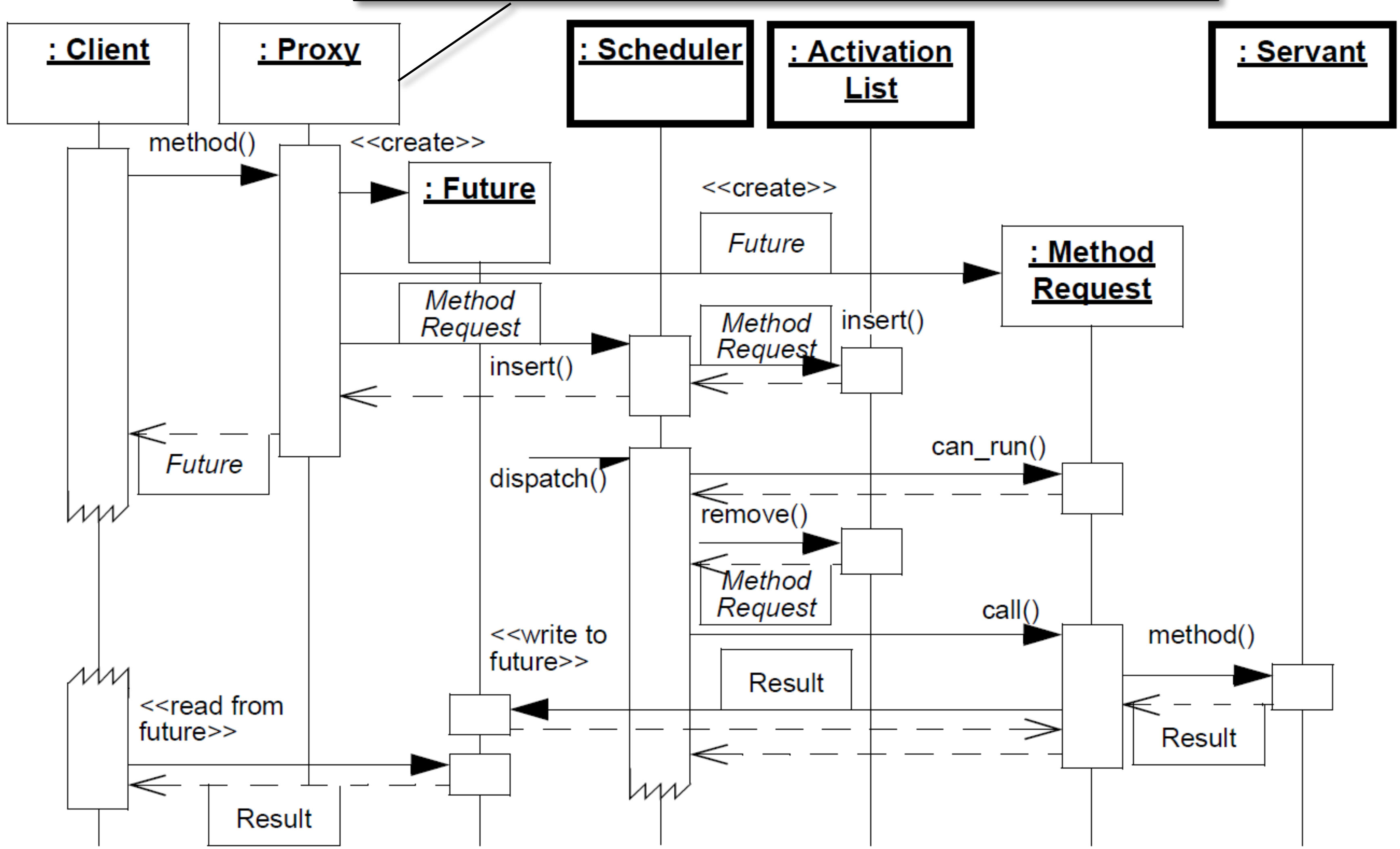
Dynamics



Active Object POSA2 Concurrency

Dynamics

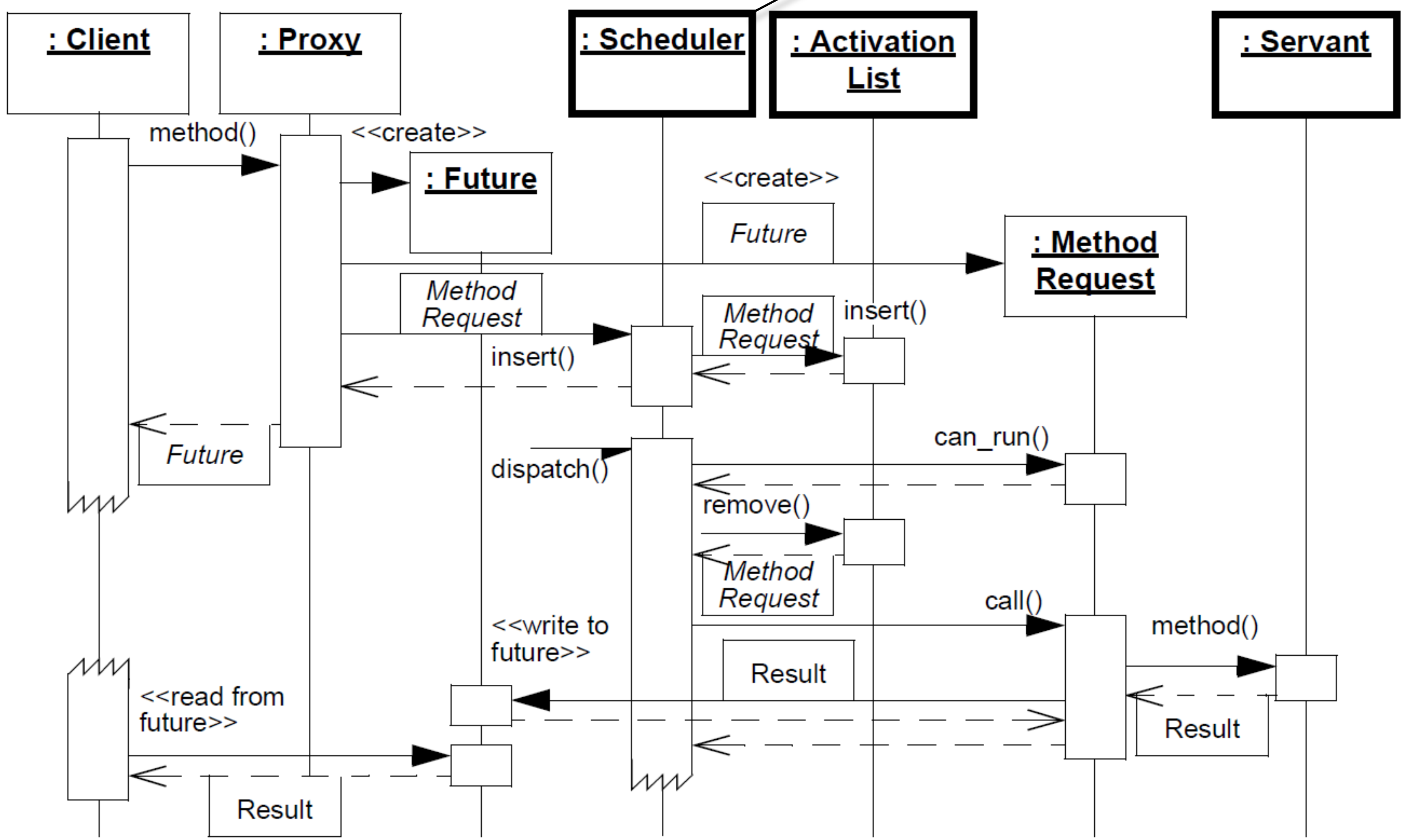
Proxy converts method call into method request, passes to scheduler, & returns future to client



Active Object POSA2 Concurrency

Dynamics

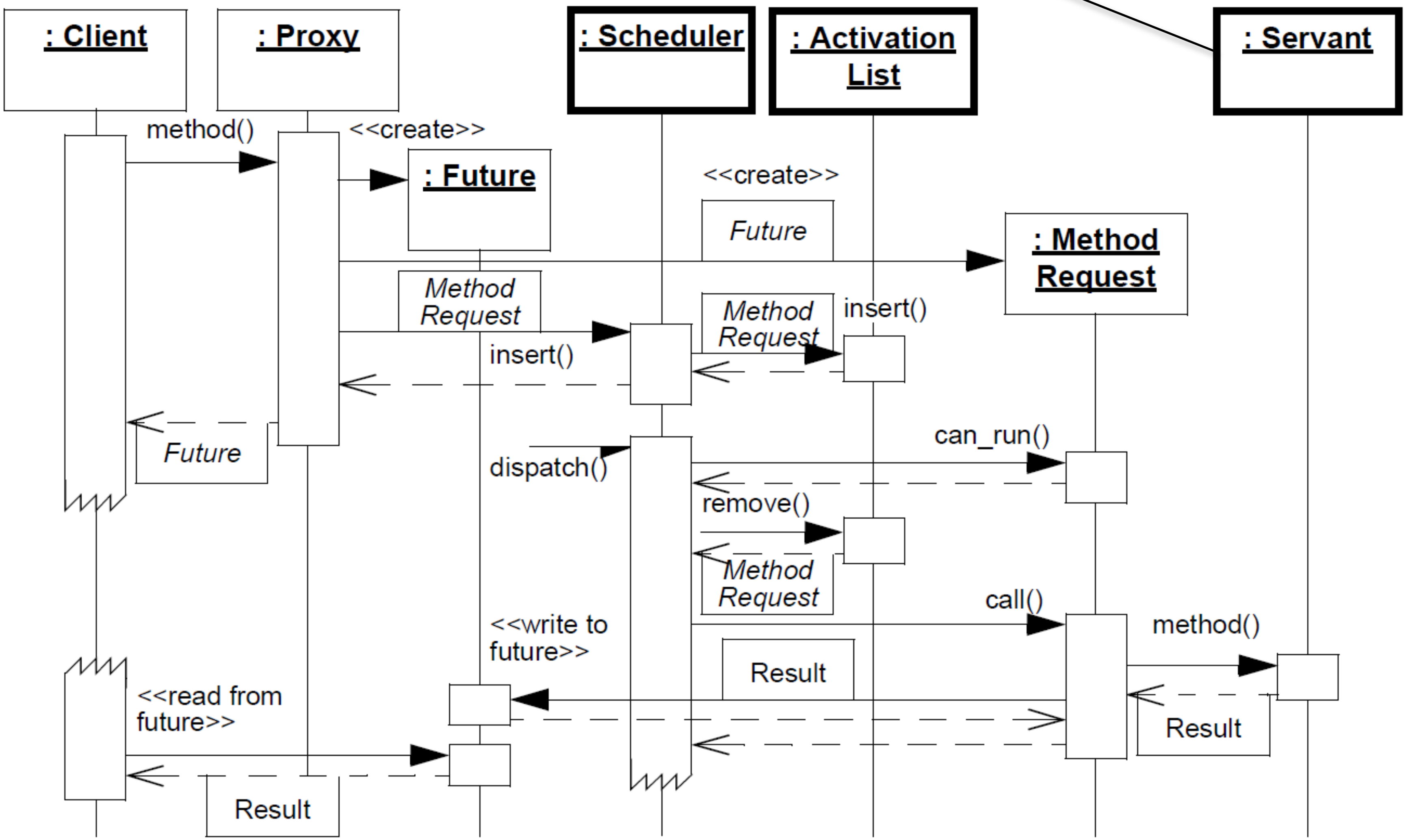
Scheduler enqueues method request



Active Object POSA2 Concurrency

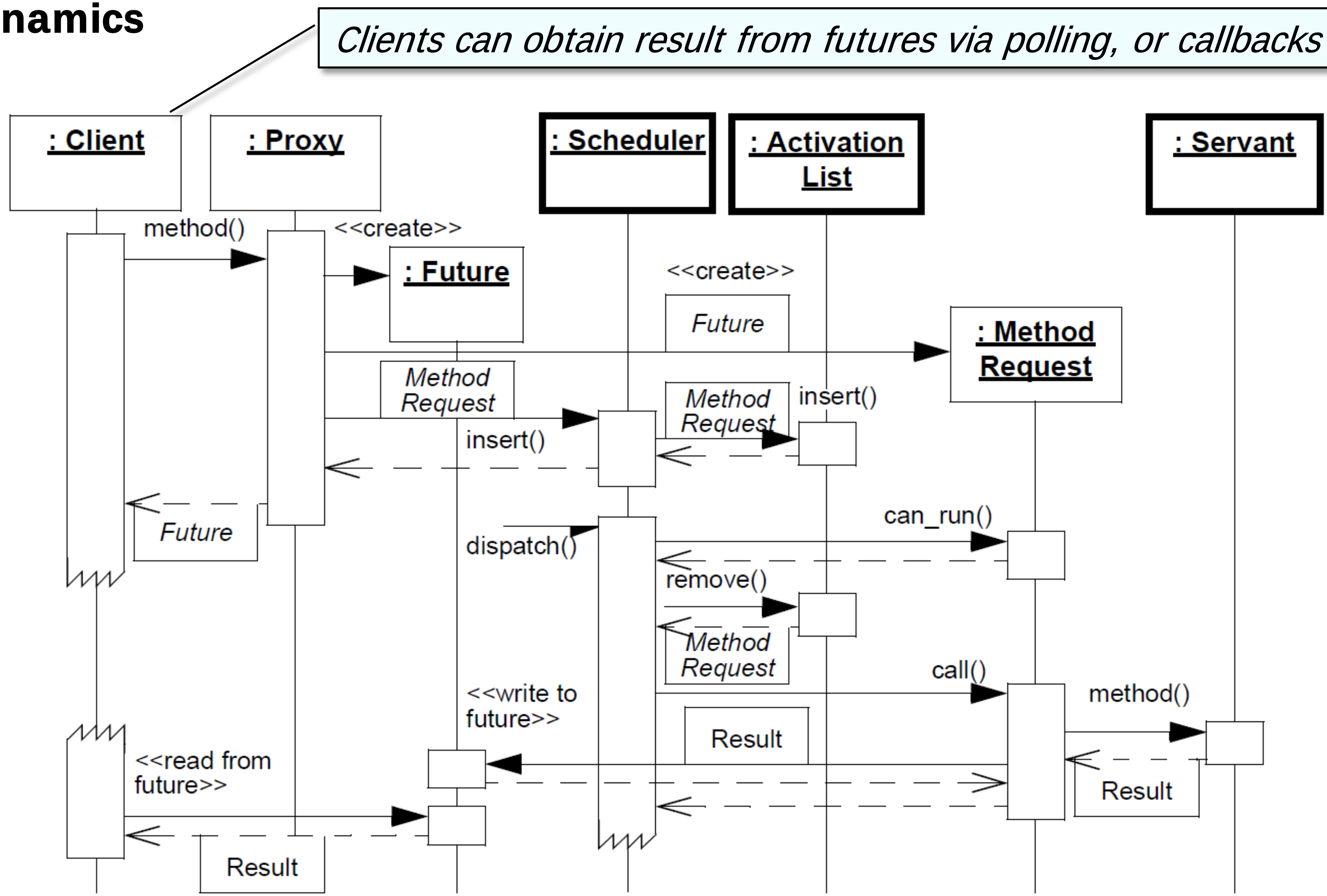
Dynamics

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



Active Object POSA2 Concurrency

Dynamics



Active Object POSA2 Concurrency

Implementation

- Implement the invocation infrastructure
  - Implement the proxy
    - Creates a concrete method request for each method invocation by a client

```
class Handler {
  boolean sendMessage (Message msg) {
    return sendMessageDelayed(msg, 0);
  }

  boolean sendMessageDelayed
     (Message msg, long delayMillis){
    return sendMessageAtTime(msg,
       SystemClock.uptimeMillis() +
       delayMillis);
  }

  boolean sendMessageAtTime
    (Message msg, long uptimeMillis) {
    MessageQueue queue = mQueue;
    queue.enqueueMessage
              (msg, uptimeMillis);
    ...
```

en.wikipedia.org/wiki/Proxy_pattern has more on the *Proxy* pattern

Active Object POSA2 Concurrency

Implementation

- Implement the invocation infrastructure
  - Implement the proxy
    - Creates a concrete method request for each method invocation by a client

```
class Handler {
  boolean sendMessage (Message msg) {
    return sendMessageDelayed(msg, 0);
  }

  boolean sendMessageDelayed
     (Message msg, long delayMillis){
    return sendMessageAtTime(msg,
       SystemClock.uptimeMillis() +
       delayMillis);
  }

  boolean sendMessageAtTime
    (Message msg, long uptimeMillis) {
    MessageQueue queue = mQueue;
    queue.enqueueMessage
              (msg, uptimeMillis);
    ...
```

frameworks/base/core/java/android/os/Handler.java has the source code

Active Object POSA2 Concurrency

Implementation

- Implement the invocation infrastructure
  - Implement the proxy
  - Implement the method requests
    - Method requests can be considered as command objects

```
class Handler {
  boolean sendMessage (Message msg) {
    return sendMessageDelayed(msg, 0);
  }

  boolean sendMessageDelayed
     (Message msg, long delayMillis){
    return sendMessageAtTime(msg,
       SystemClock.uptimeMillis() +
       delayMillis);
  }

  boolean sendMessageAtTime
    (Message msg, long uptimeMillis) {
    MessageQueue queue = mQueue;
    queue.enqueueMessage
              (msg, uptimeMillis);
    ...
```

Android Handler proxy & method requests are simpler than POSA active objects

Active Object POSA2 Concurrency

Implementation

- Implement the invocation infrastructure
- Implement the activation list
  - Used to insert & remove a method request
    - This list can be implemented as a synchronized bounded buffer shared between the client threads & the thread in which the active object's scheduler & servant run

```
public class MessageQueue {
  ...
  final boolean enqueueMessage
    (Message msg, long when) {
    ...
  }

  final Message next() {
    ...
  }
```

frameworks/base/core/java/android/os/MessageQueue.java has source code

Active Object POSA2 Concurrency

Implementation

- Implement the invocation infrastructure
- Implement the activation list
- Implement the scheduler
  - A scheduler is a command processor that manages the activation list & executes pending method requests whose synchronization constraints have been met

```
public class Looper {
  ...
  final MessageQueue mQueue;

  public static void loop() {
    ...

    for (;;) {
      Message msg =
        queue.next();
      ...

      msg.target.
        dispatchMessage(msg);
      ...
  }
  ...
```

wiki.hsr.ch/APF/files/CommandProcessor.pdf has more on *Command Processor*

Active Object POSA2 Concurrency

Implementation

- Implement the invocation infrastructure
- Implement the activation list
- Implement the scheduler
  - A scheduler is a command processor that manages the activation list & executes pending method requests whose synchronization constraints have been met

```
public class Looper {
  ...
  final MessageQueue mQueue;

  public static void loop() {
    ...

    for (;;) {
      Message msg =
        queue.next();
      ...

      msg.target.
        dispatchMessage(msg);
      ...
  }
  ...
```

frameworks/base/core/java/android/os/Looper.java has source code

Active Object POSA2 Concurrency

Implementation

- Implement the invocation infrastructure
- Implement the activation list
- Implement the scheduler
- Implement the servant
  - A servant defines the behavior & state being modeled as an active object

```
class WorkerHandler extends Handler {
    public void handleMessage
                     (Message msg) {
      switch (msg.what) {
      case SET_PROGRESS_BAR_VISIBILITY:
      {
        mAct.progress.setVisibility
          ((Integer) msg.obj);
        break;
      }
      case PROGRESS_UPDATE:
      {
        mAct.progress.setProgress
           ((Integer) msg.obj);
        break;
      }
     ...
```

Active Object POSA2 Concurrency

Implementation

- Implement the invocation infrastructure
- Implement the activation list
- Implement the scheduler
- Implement the servant
- Determine rendezvous & return value policy
  - The rendezvous policy determines how clients obtain return values from methods invoked on active objects

```
class WorkerHandler extends Handler {
    public void handleMessage
                         (Message msg) {
      ...
      WorkerArgs args =
        (WorkerArgs) msg.obj;

      Message reply =
        args.handler.obtainMessage();
      reply.obj = args;
      reply.arg1 = msg.arg1;

      reply.sendToTarget();
      ...
```

A common idiom is to pass the original Handler via a Message to a Worker Thread, which can then pass a response back to the original Handler

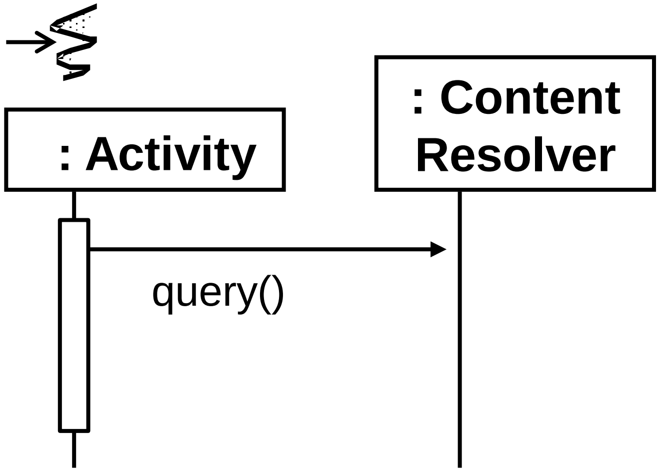
developer.android.com/reference/android/os/Message.html#sendToTarget()

Active Object POSA2 Concurrency

Applying Active Object in Android

- AsyncQueryHandler is a helper class that invokes ContentResolver calls asynchronously to avoid blocking the UI Thread
  - ContentResolver provides apps access to an underlying ContentProvider

Synchronous Query

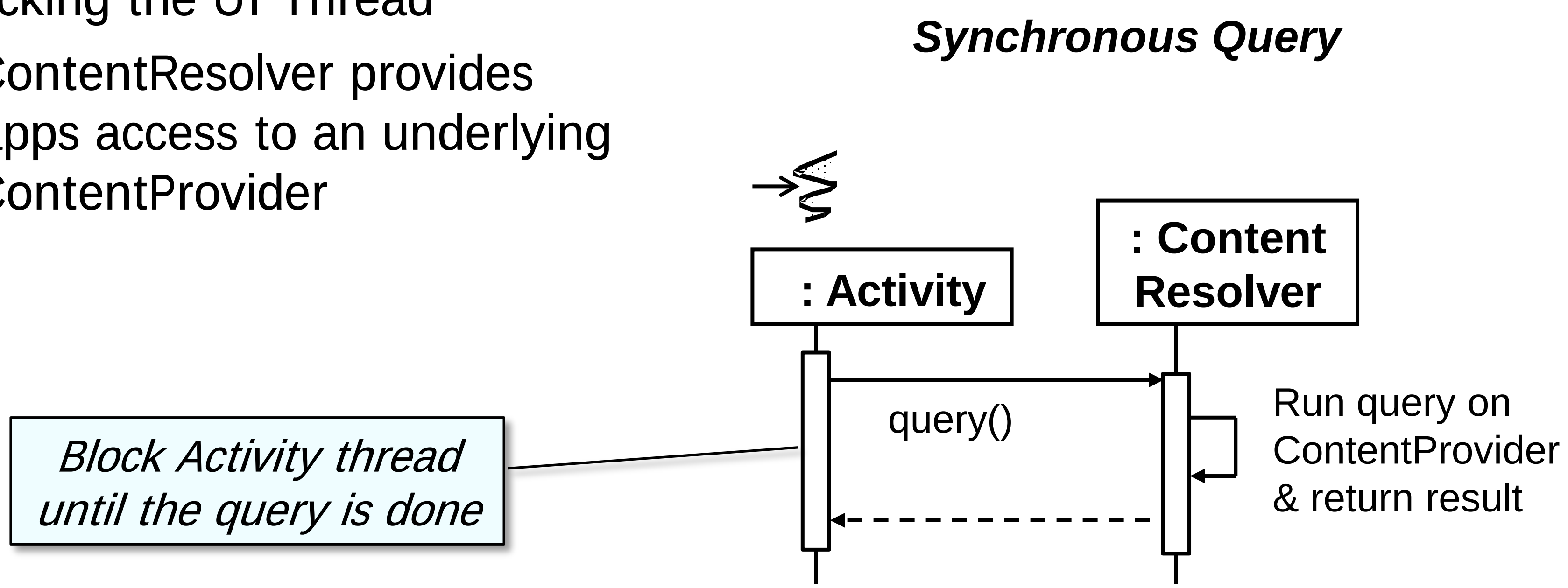


See developer.android.com/reference/android/content/ContentResolver.html

Active Object POSA2 Concurrency

Applying Active Object in Android

- AsyncQueryHandler is a helper class that invokes ContentResolver calls asynchronously to avoid blocking the UI Thread
  - ContentResolver provides apps access to an underlying ContentProvider



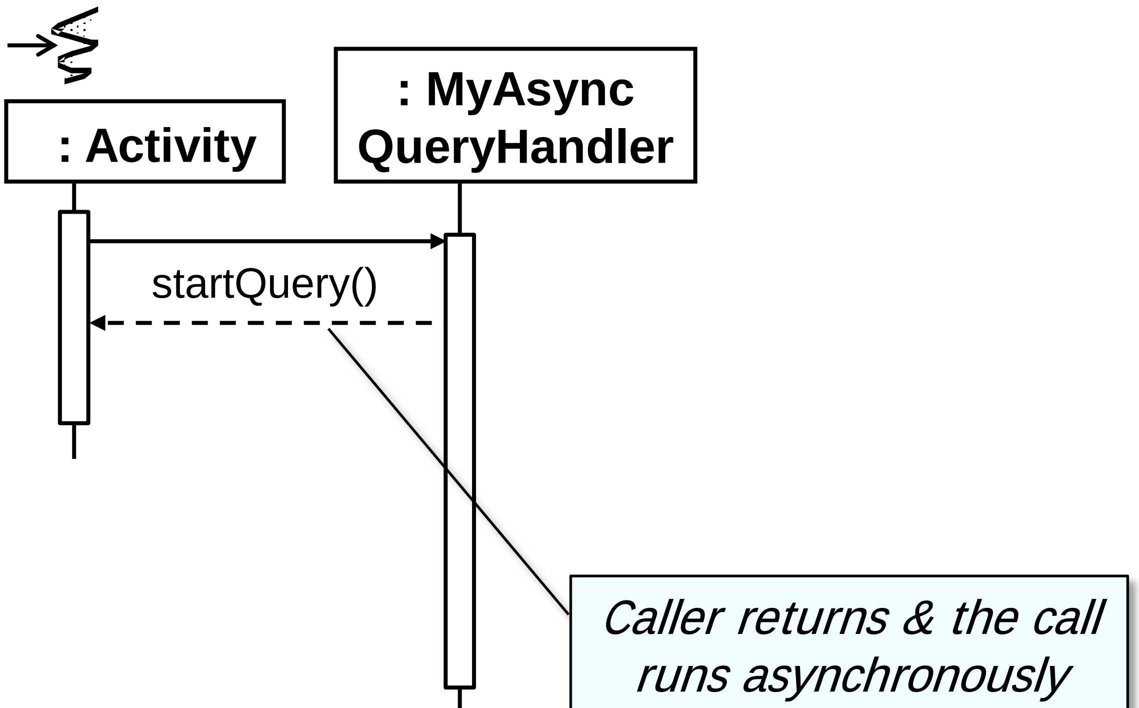
See developer.android.com/reference/android/content/ContentResolver.html

Active Object POSA2 Concurrency

Applying Active Object in Android

- AsyncQueryHandler is a helper class that invokes ContentResolver calls asynchronously to avoid blocking the UI Thread

Asynchronous Query



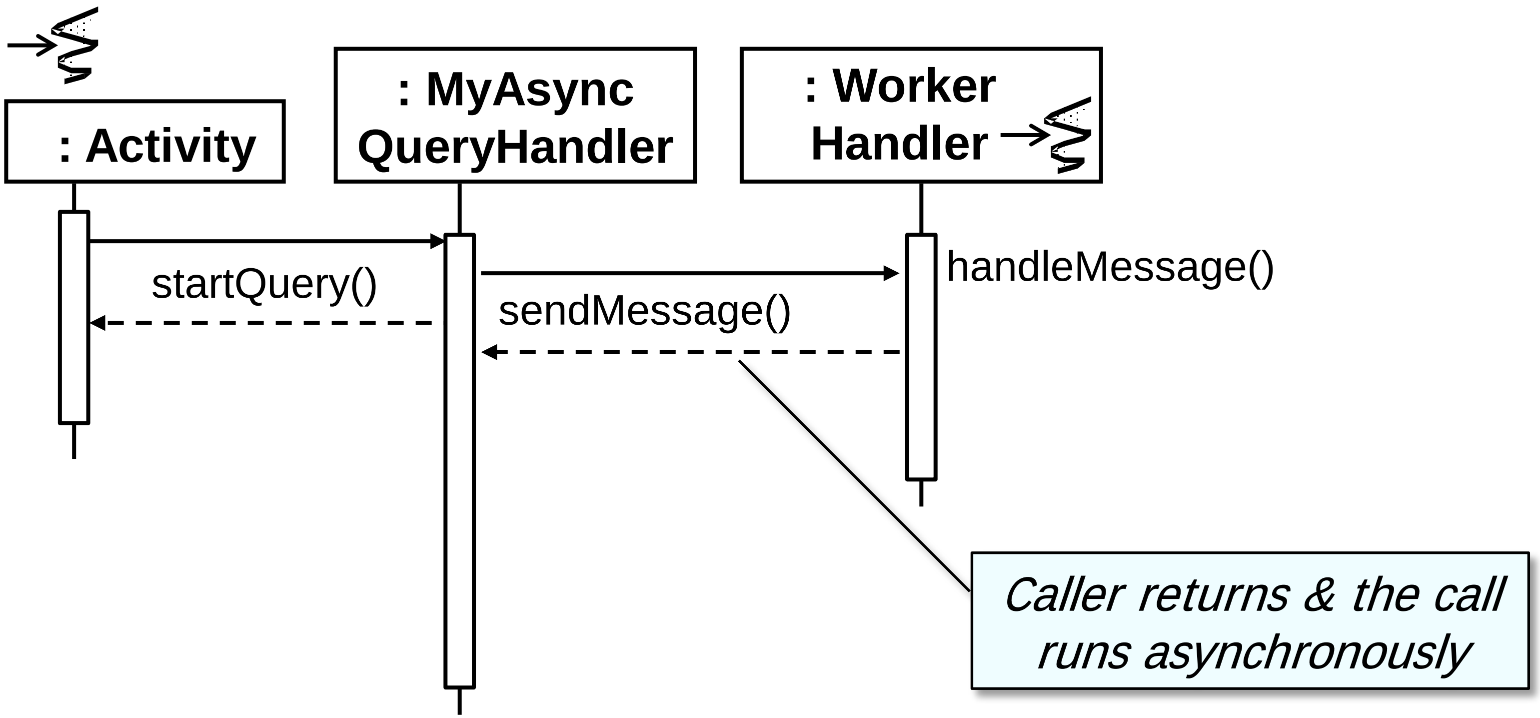
See developer.android.com/reference/android/content/AsyncQueryHandler.html

Active Object POSA2 Concurrency

Applying Active Object in Android

- AsyncQueryHandler is a helper class that invokes ContentResolver calls asynchronously to avoid blocking the UI Thread

Asynchronous Query

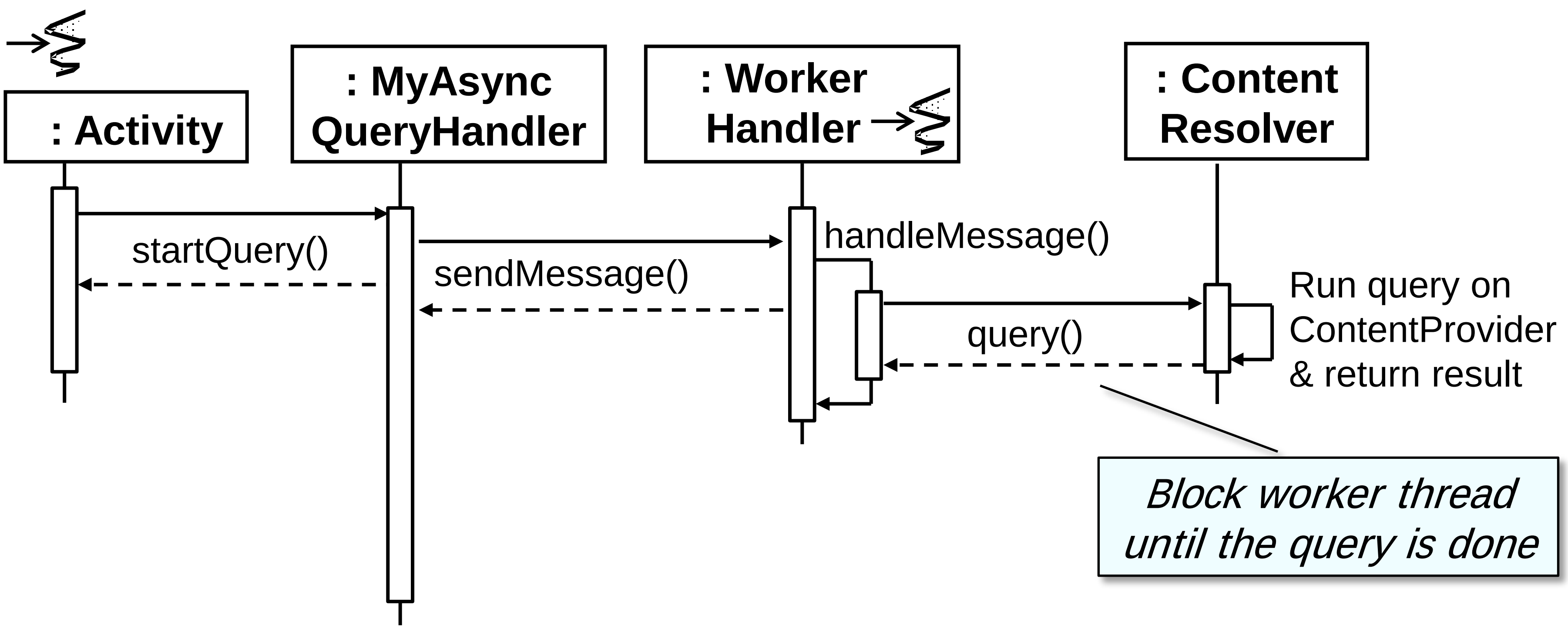


Active Object POSA2 Concurrency

Applying Active Object in Android

- AsyncQueryHandler is a helper class that invokes ContentResolver calls asynchronously to avoid blocking the UI Thread

Asynchronous Query

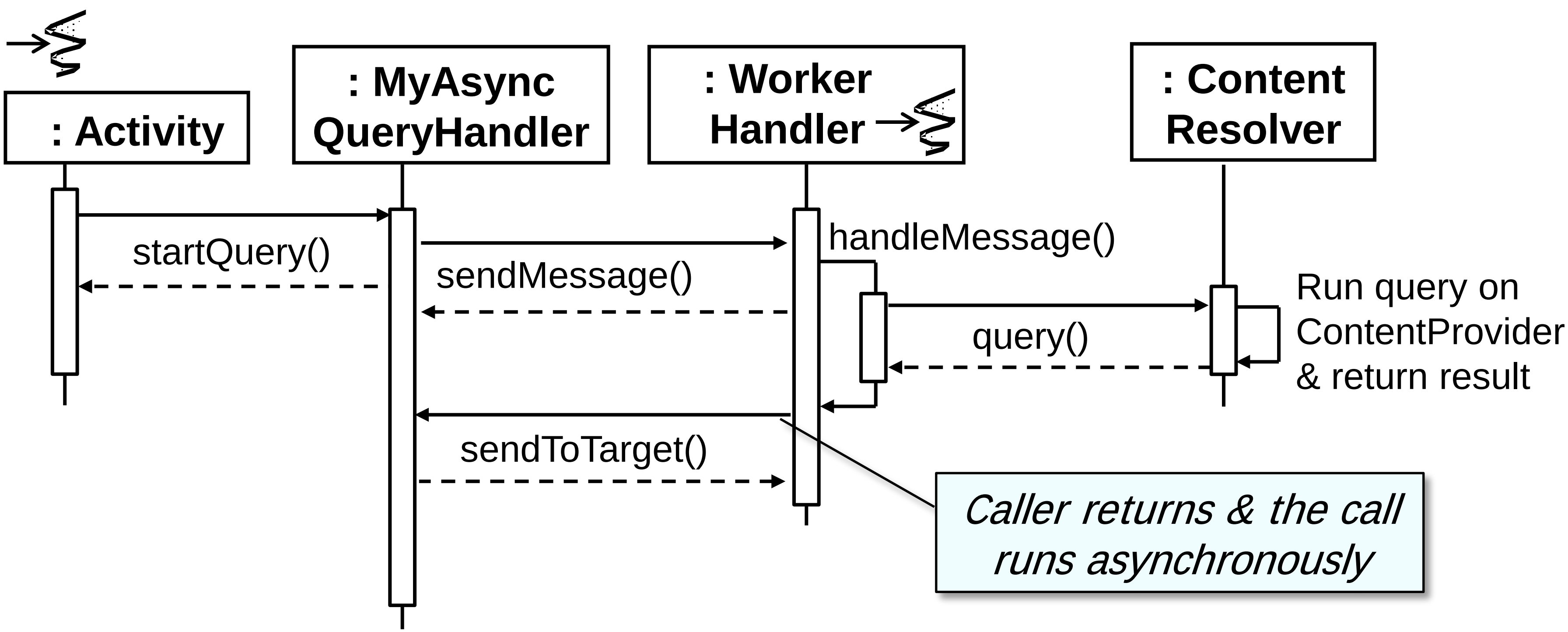


Active Object POSA2 Concurrency

Applying Active Object in Android

- AsyncQueryHandler is a helper class that invokes ContentResolver calls asynchronously to avoid blocking the UI Thread

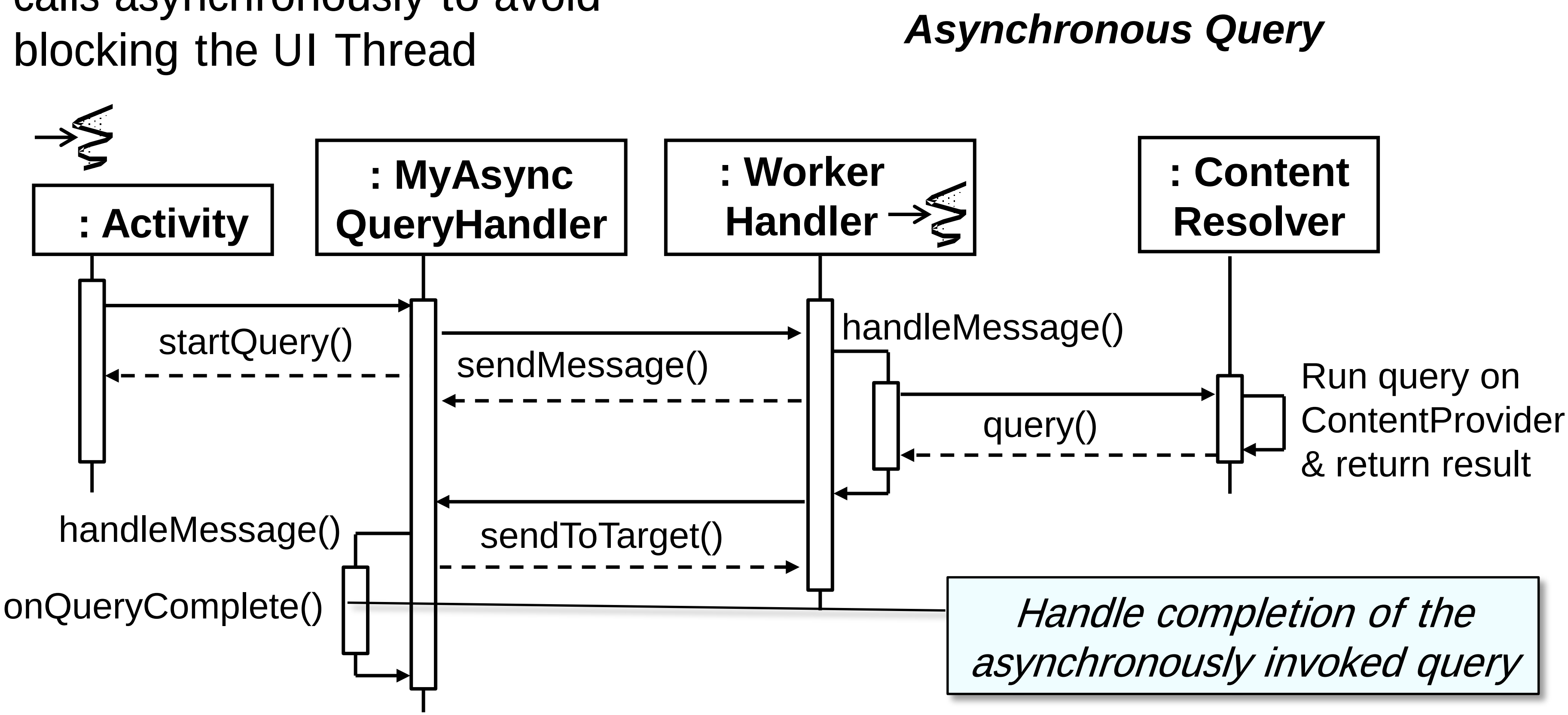
Asynchronous Query



Active Object POSA2 Concurrency

Applying Active Object in Android

- AsyncQueryHandler is a helper class that invokes ContentResolver calls asynchronously to avoid blocking the UI Thread



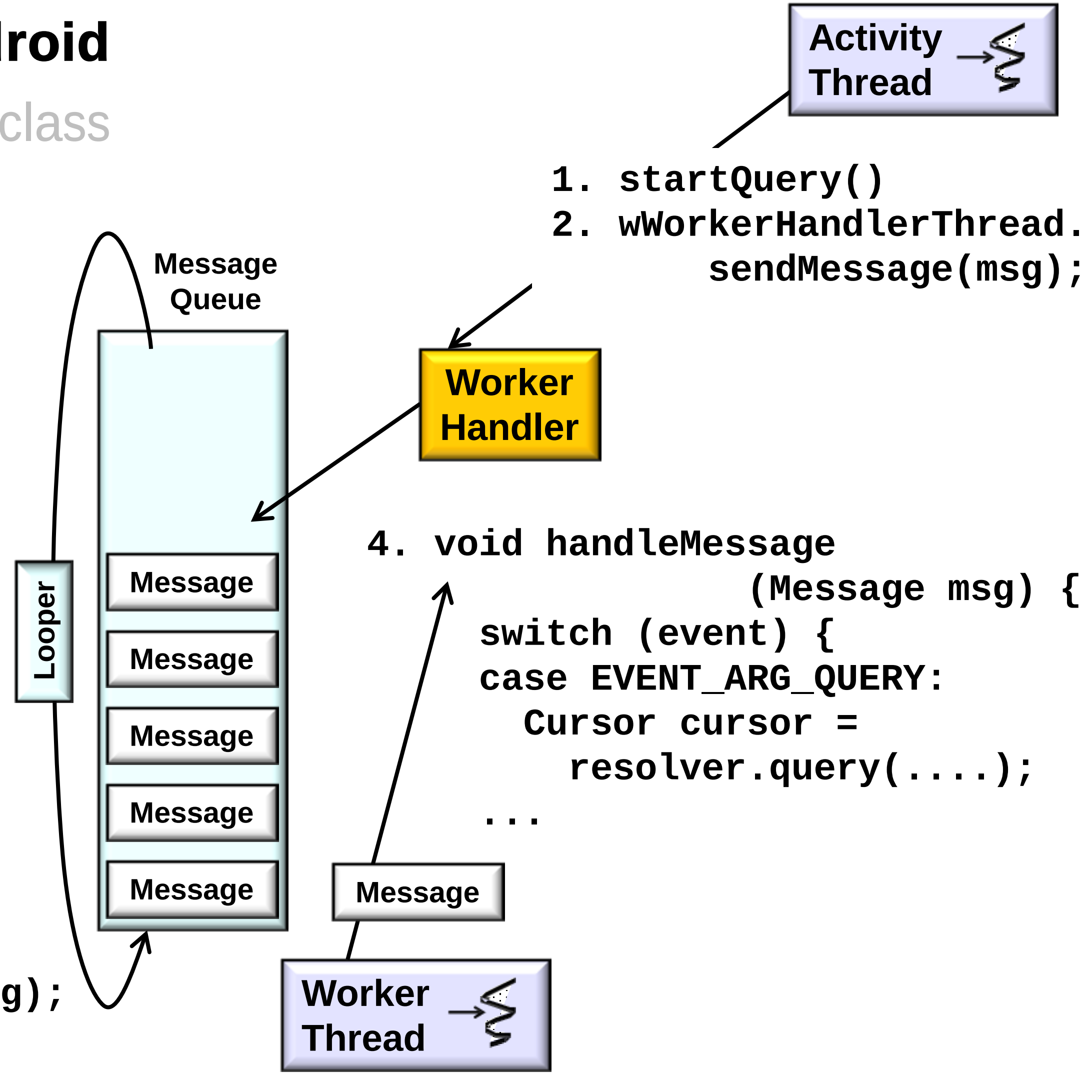
See developer.android.com/reference/android/content/AsyncQueryHandler.html

Active Object POSA2 Concurrency

Applying Active Object in Android

- AsyncQueryHandler is a helper class that invokes ContentResolver calls asynchronously to avoid blocking the UI Thread
- Internally, AsyncQueryHandler uses a (subset of the) *Active Object* pattern

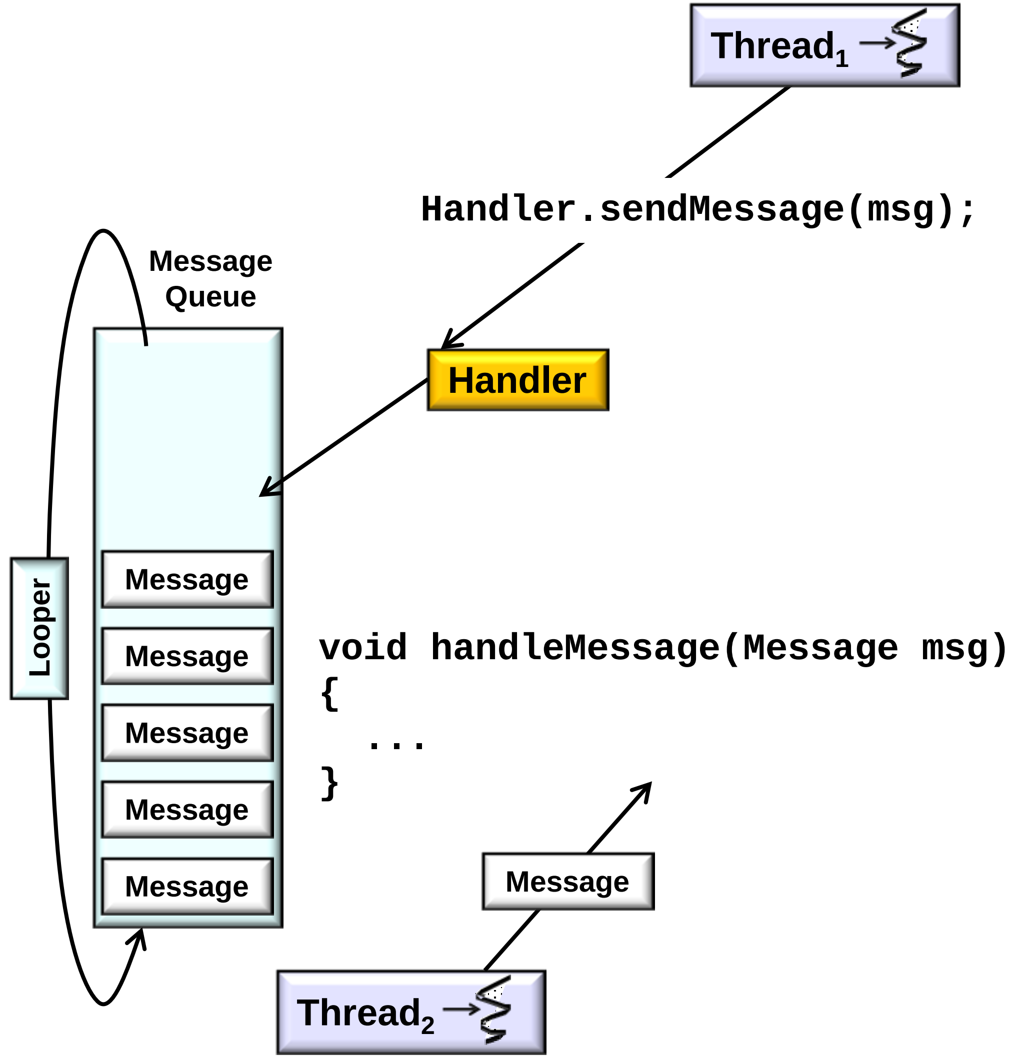
```
3. void loop() {
     ...
     for (;;) {
       Message msg =
         queue.next();
       ...

       msg.target.
         dispatchMessage(msg);
       ...
```



Active Object POSA2 Concurrency

Consequences

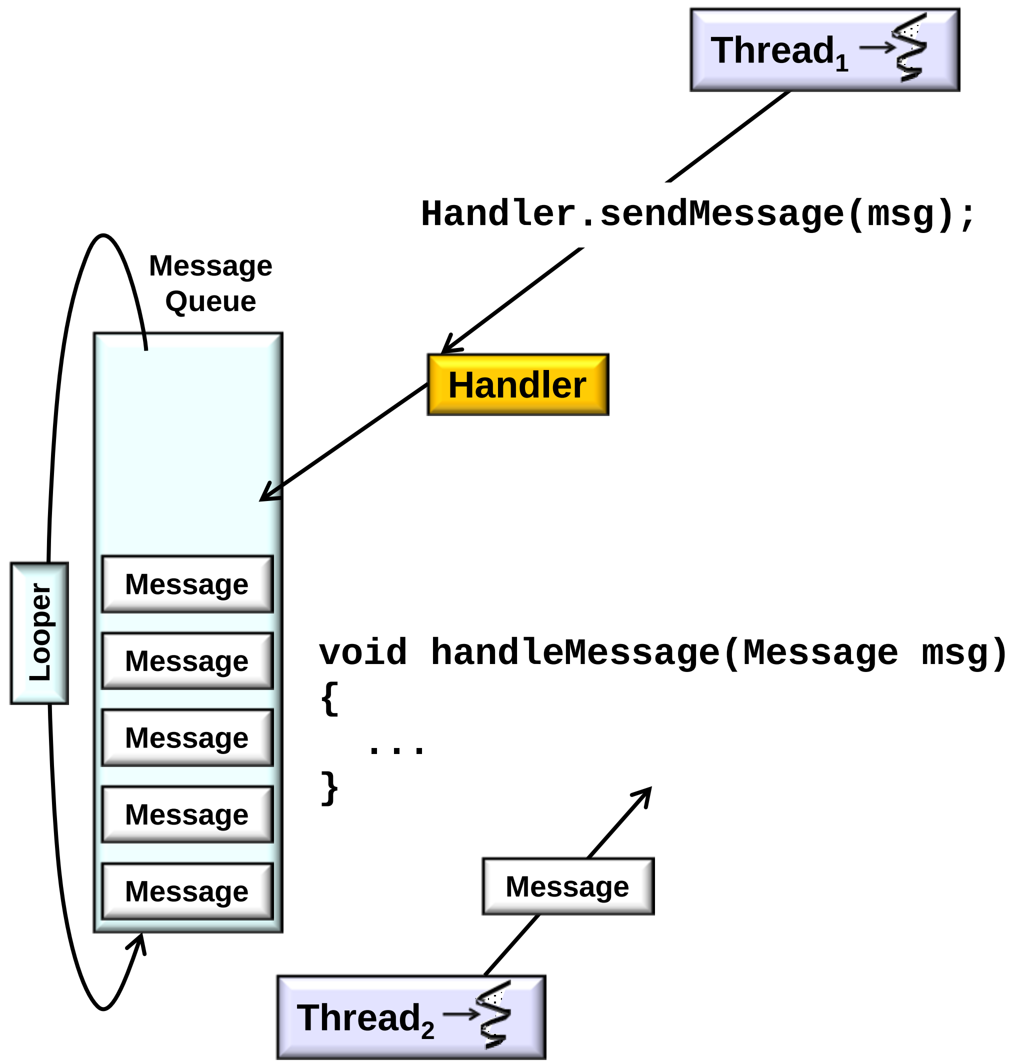
+ Enhances concurrency & simplifies synchronized complexity
  - Client threads & asynchronous method executions can run concurrently



Active Object POSA2 Concurrency

Consequences

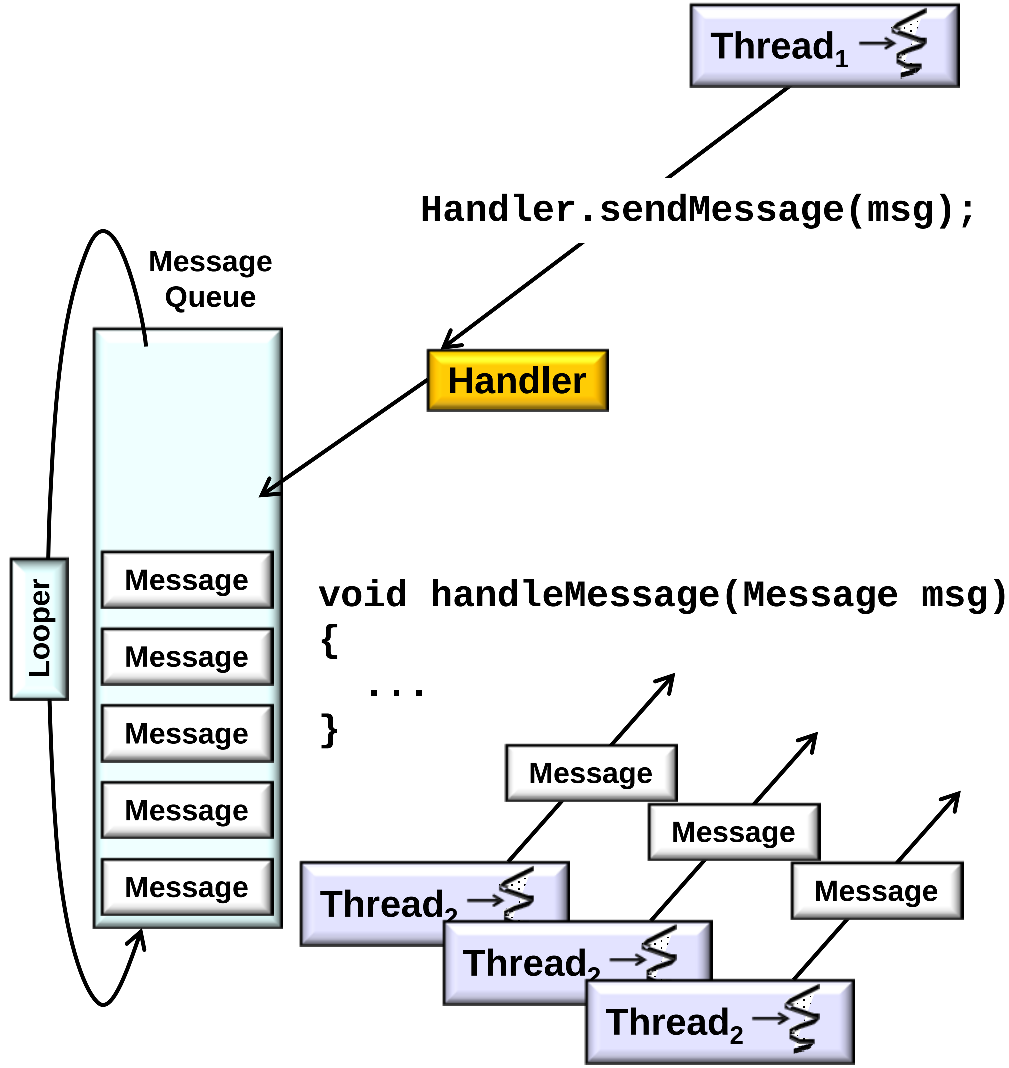
+ Enhances concurrency & simplifies synchronized complexity
  - Client threads & asynchronous method executions can run concurrently
  - A scheduler can evaluate synchronization constraints to serialize access to servants



Active Object POSA2 Concurrency

Consequences

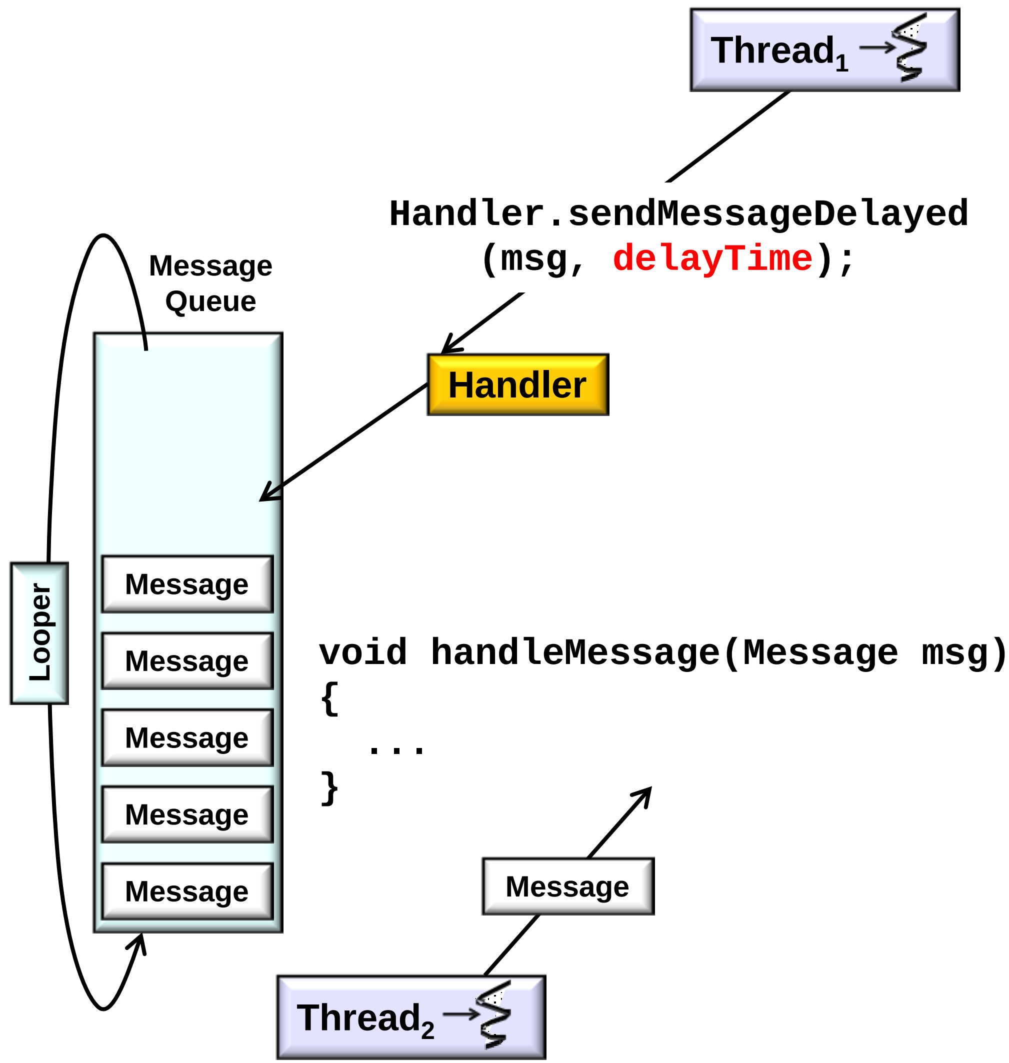
+ Enhances concurrency & simplifies synchronized complexity
+ Transparent leveraging of available parallelism
  - Multiple active object methods can execute in parallel if the scheduler is configured using a thread pool & supported by the OS/hardware



Active Object POSA2 Concurrency

Consequences

+ Enhances concurrency & simplifies synchronized complexity
+ Transparent leveraging of available parallelism
+ Method execution order can differ from method invocation order
  - Methods that are invoked asynchronously can execute according to synchronization constraints defined by guards & scheduling policies

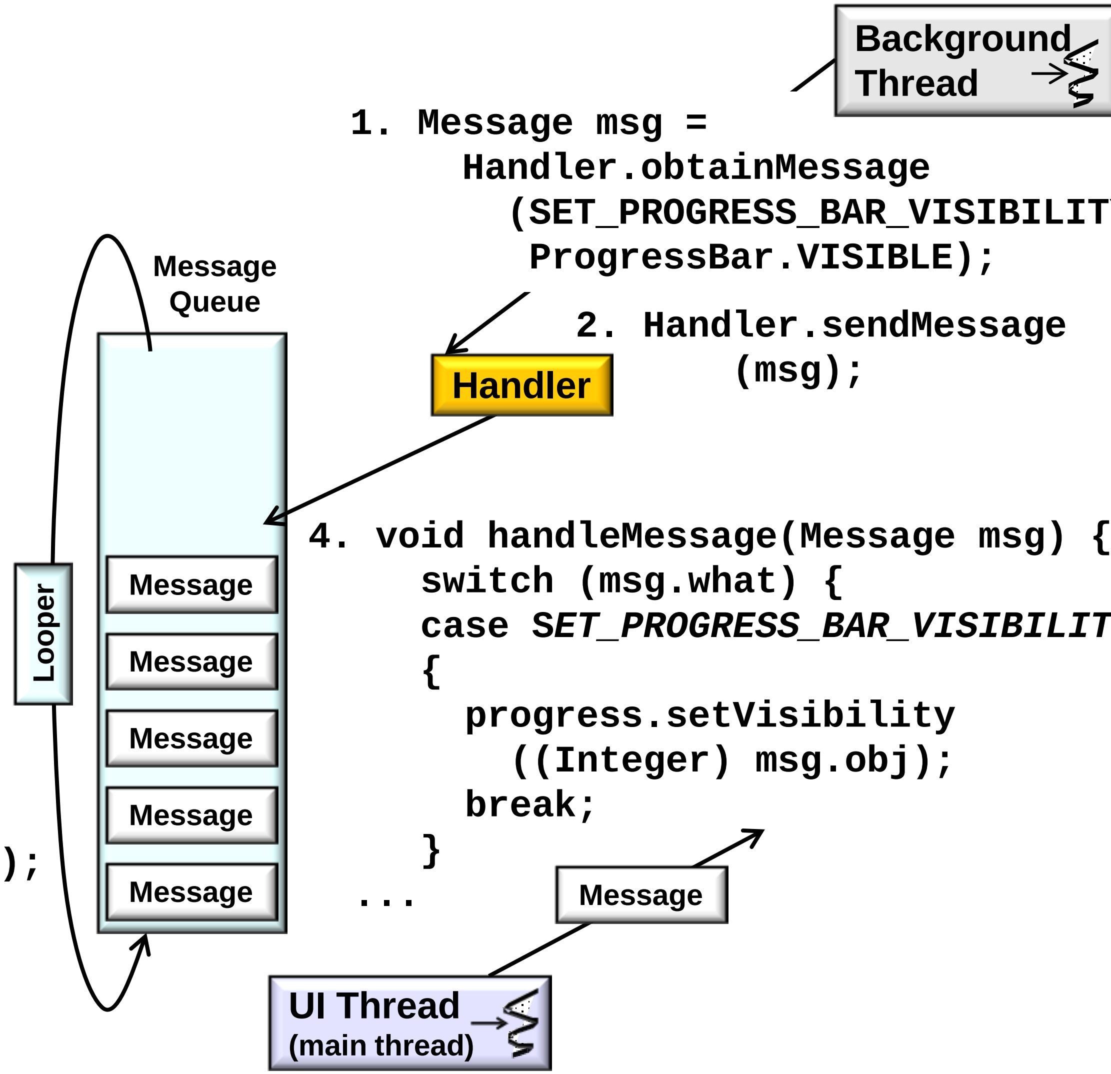


Active Object POSA2 Concurrency

Consequences

– Runtime overhead

- Stemming from




```
1. Message msg =
     Handler.obtainMessage
       (SET_PROGRESS_BAR_VISIBILITY,
        ProgressBar.VISIBLE);
```

```
2. Handler.sendMessage
       (msg);
```

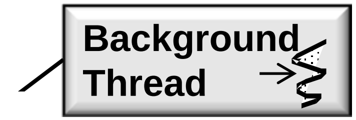
```
3. void loop() {
     ...
     for (;;) {
       Message msg = queue.next();
       ...
       msg.target.
         dispatchMessage(msg);
       ...
```

```
4. void handleMessage(Message msg) {
     switch (msg.what) {
     case SET_PROGRESS_BAR_VISIBILITY:
     {
       progress.setVisibility
         ((Integer) msg.obj);
       break;
     }
   ...
```

Active Object POSA2 Concurrency

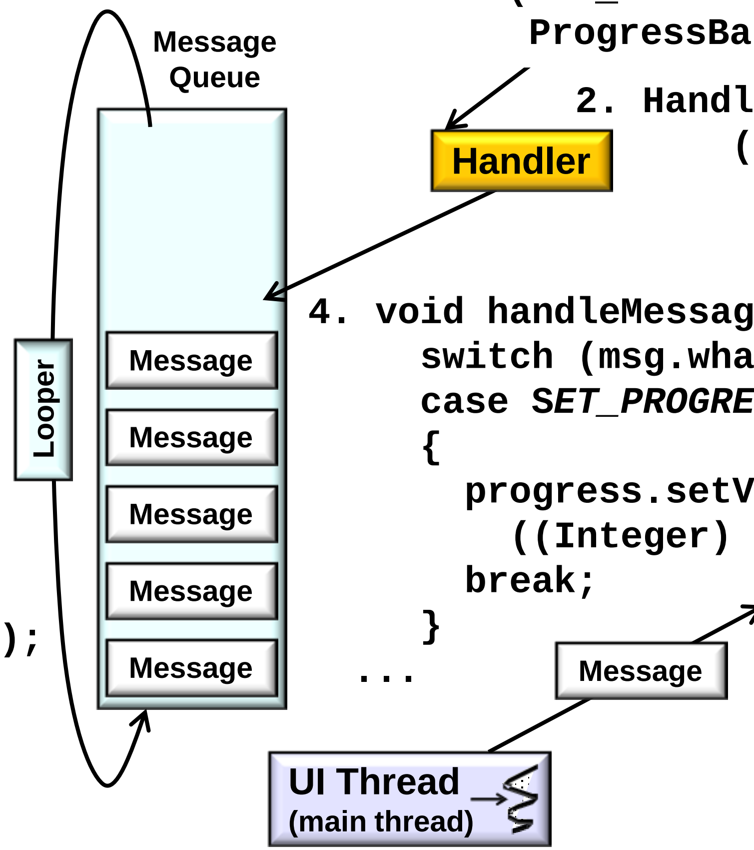
Consequences

– Runtime overhead
  - Stemming from
    - Dynamic memory (de)allocation




```
1. Message msg =
     Handler.obtainMessage
       (SET_PROGRESS_BAR_VISIBILITY,
        ProgressBar.VISIBLE);
```

```
2. Handler.sendMessage
         (msg);
```



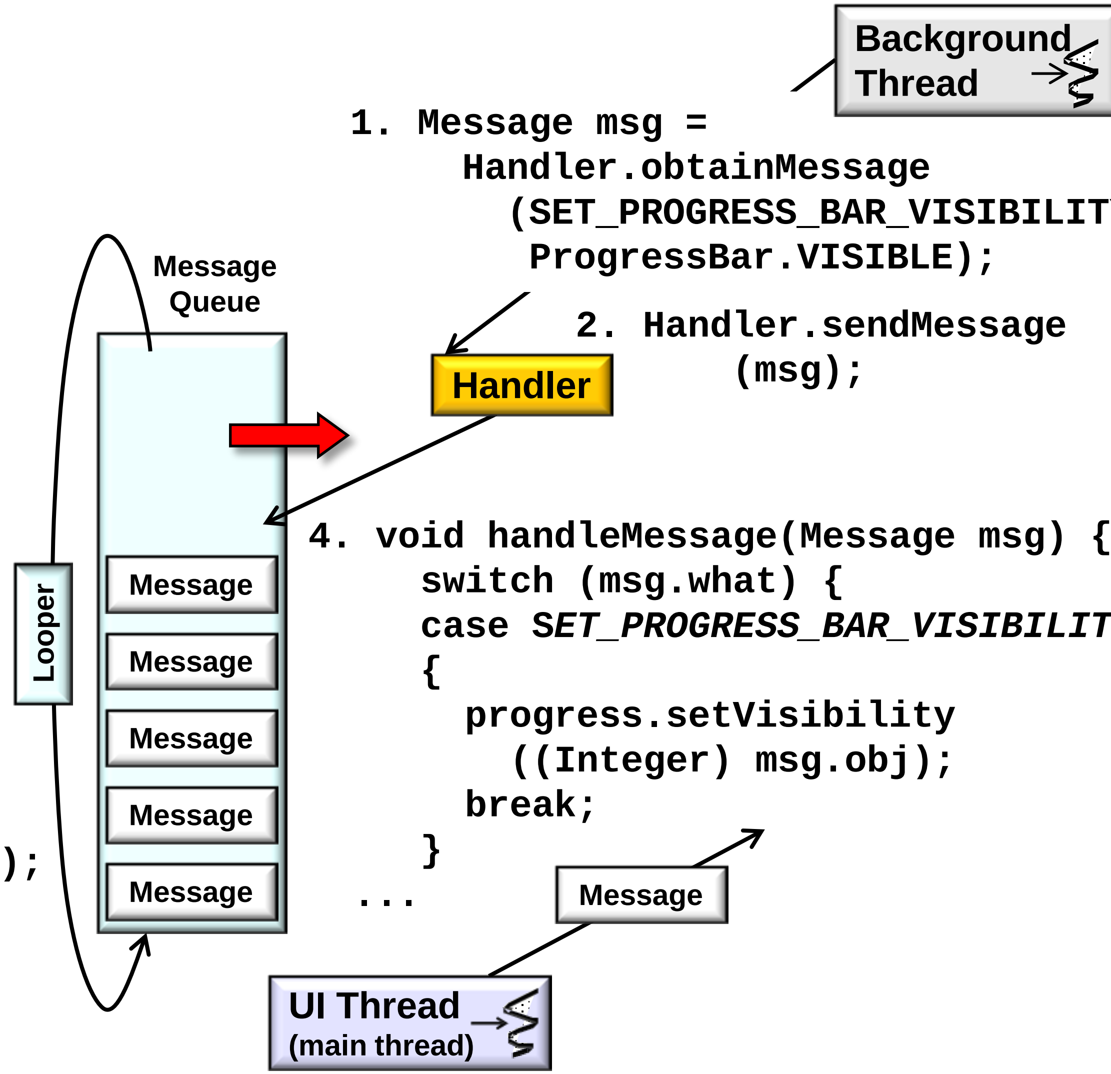

```
3. void loop() {
     ...
     for (;;) {
       Message msg = queue.next();
       ...
       msg.target.
         dispatchMessage(msg);
       ...
```

```
4. void handleMessage(Message msg) {
     switch (msg.what) {
     case SET_PROGRESS_BAR_VISIBILITY:
     {
       progress.setVisibility
         ((Integer) msg.obj);
       break;
     }
   ...
```

Active Object POSA2 Concurrency

Consequences

– Runtime overhead
  - Stemming from
    - Dynamic memory (de)allocation
    - Synchronization operations




```
1. Message msg =
     Handler.obtainMessage
       (SET_PROGRESS_BAR_VISIBILITY,
        ProgressBar.VISIBLE);
```

```
2. Handler.sendMessage
          (msg);
```

```
3. void loop() {
     ...
     for (;;) {
       Message msg = queue.next();
       ...
       msg.target.
         dispatchMessage(msg);
       ...
```

```
4. void handleMessage(Message msg) {
     switch (msg.what) {
     case SET_PROGRESS_BAR_VISIBILITY:
     {
       progress.setVisibility
         ((Integer) msg.obj);
       break;
     }
     ...
```

Active Object POSA2 Concurrency

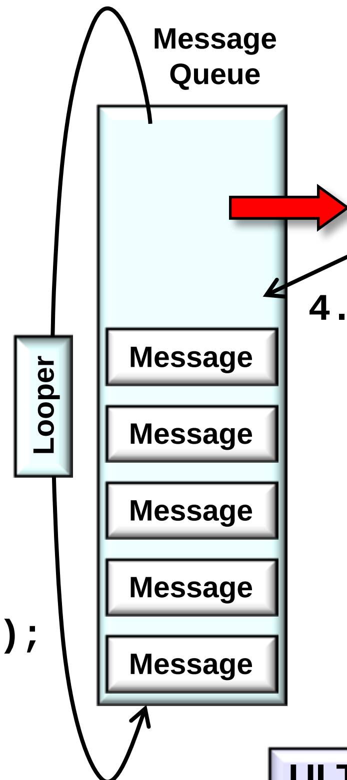
Consequences

– Runtime overhead
  - Stemming from
    - Dynamic memory (de)allocation
    - Synchronization operations
  - Context switches

```
1. Message msg =
     Handler.obtainMessage
       (SET_PROGRESS_BAR_VISIBILITY,
        ProgressBar.VISIBLE);
```

```
2. Handler.sendMessage
         (msg);
```

```
3. void loop() {
     ...
     for (;;) {
       Message msg = queue.next();
       ...
       msg.target.
         dispatchMessage(msg);
       ...
```



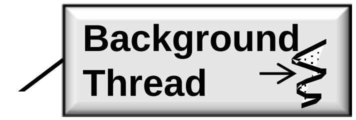

```
4. void handleMessage(Message msg) {
     switch (msg.what) {
     case SET_PROGRESS_BAR_VISIBILITY:
     {
       progress.setVisibility
         ((Integer) msg.obj);
       break;
     }
   ...
```

Active Object POSA2 Concurrency

Consequences

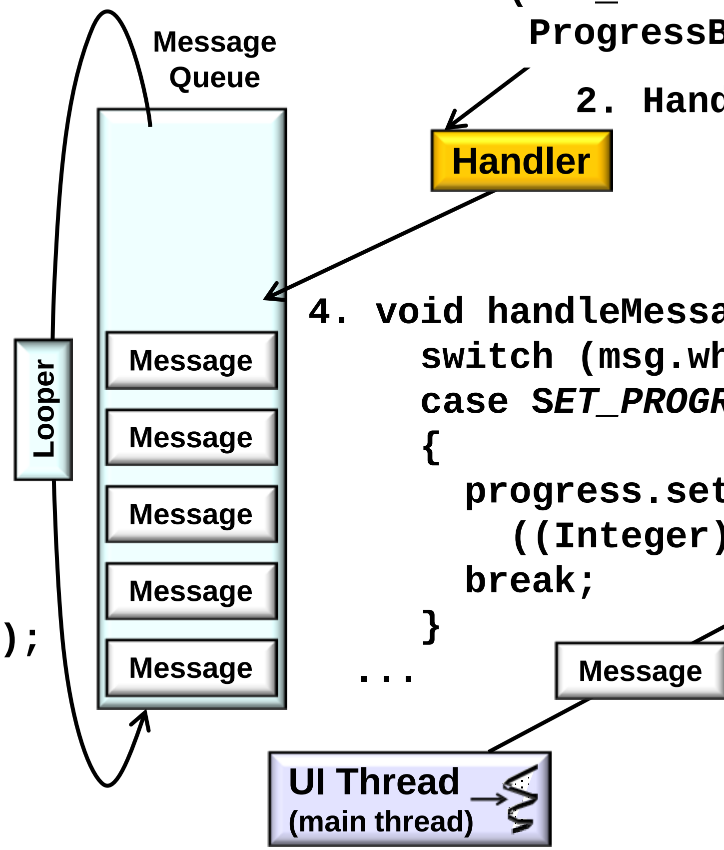
– Runtime overhead

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




```
1. Message msg =
     Handler.obtainMessage
       (SET_PROGRESS_BAR_VISIBILITY,
        ProgressBar.VISIBLE);
```

```
2. Handler.sendMessage
         (msg);
```




```
3. void loop() {
     ...
     for (;;) {
       Message msg = queue.next();
       ...
       msg.target.
         dispatchMessage(msg);
       ...
```

```
4. void handleMessage(Message msg) {
     switch (msg.what) {
     case SET_PROGRESS_BAR_VISIBILITY:
     {
       progress.setVisibility
         ((Integer) msg.obj);
       break;
     }
   ...
```

Active Object POSA2 Concurrency

Consequences

- Higher overhead
- 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

Active Object POSA2 Concurrency

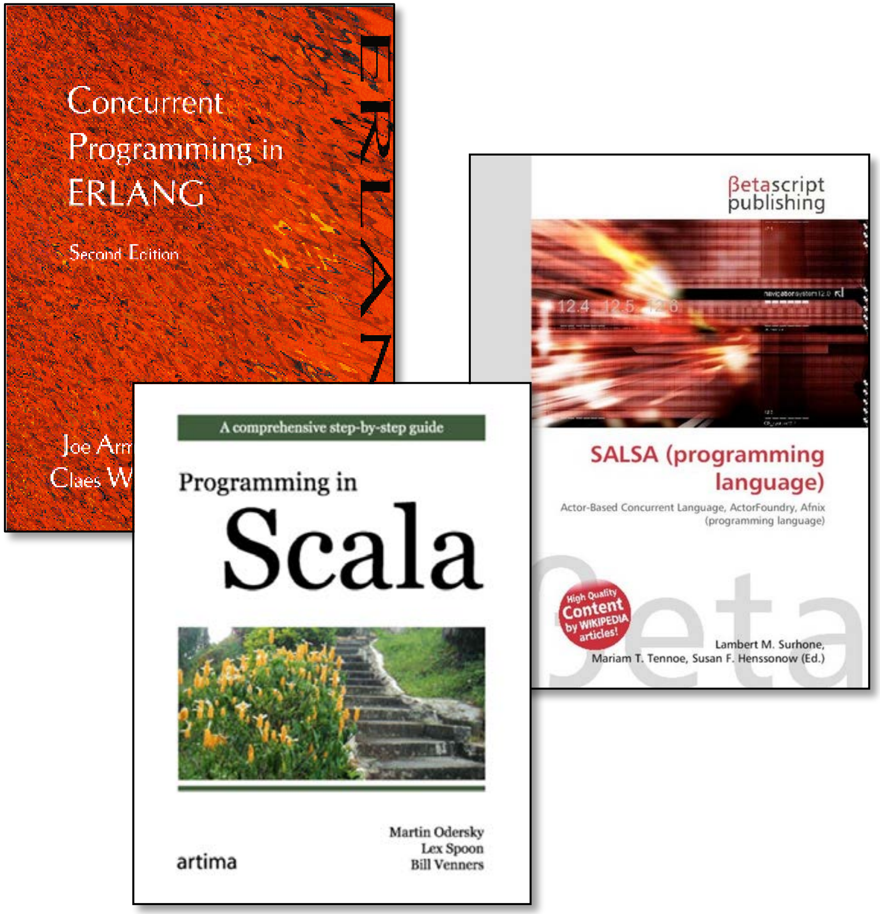
Known Uses

- Programming languages based on the Actor model
  - A mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent digital computation

Active Object POSA2 Concurrency

Known Uses

- Programming languages based on the Actor model
  - A mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent digital computation
  - In response to a message that it receives, an actor can make local decisions, create more actors, send more messages, & determine how to respond to the next message received



en.wikipedia.org/wiki/Actor_model has more info

Active Object POSA2 Concurrency

Known Uses

- Programming languages based on the Actor model
- Active Object in C++11

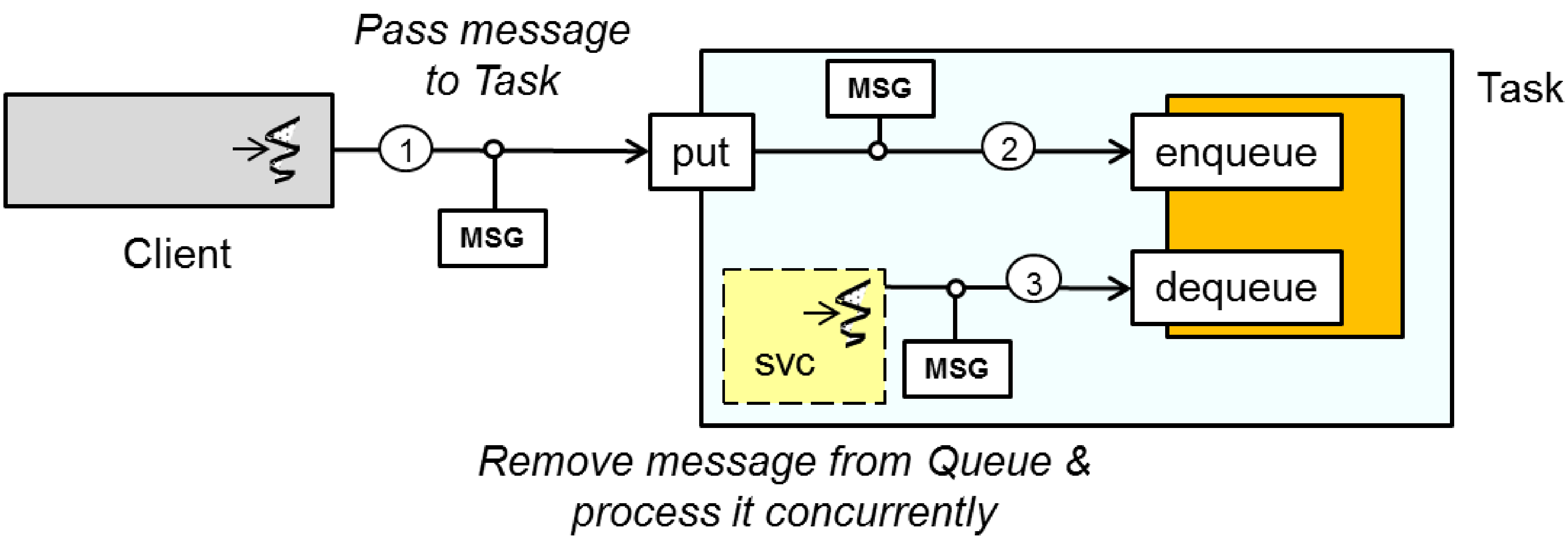
```
class Active {
public:
  typedef function<void()> Message;

  Active(): done(false)
  { th = unique_ptr<thread>(new thread([=]{ this->run(); })); }
  ~Active() { send([&]{done = true;}); th->join(); }

  void send(Message m) { mq.send(m); }

private:
  message_queue<Message> mq; bool done; unique_ptr<thread> th;
  void run(){ while(!done){ Message msg = mq.receive(); msg();}}
};
```

www.drdobbs.com/parallel/prefer-using-active-objects-instead-of-n/225700095

Active Object POSA2 Concurrency

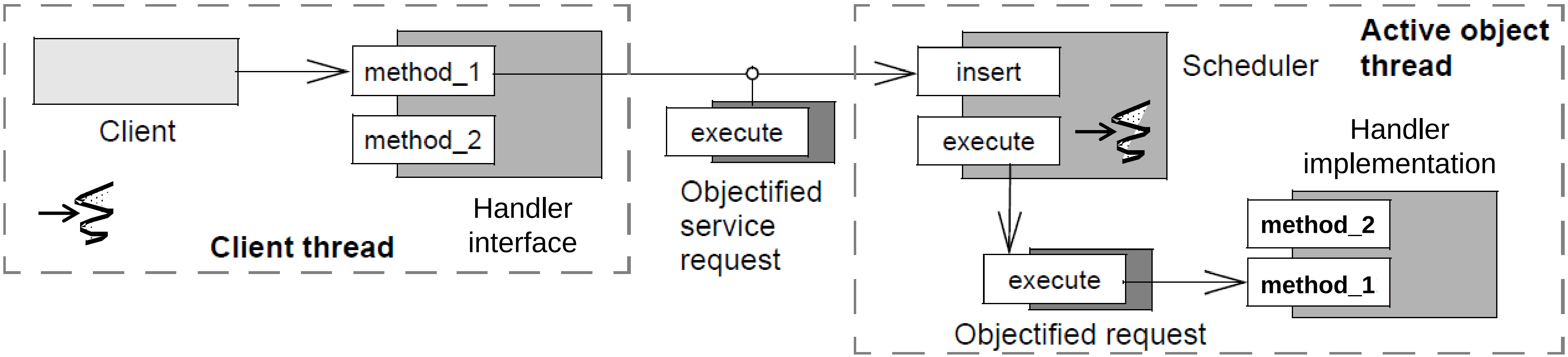
Known Uses

- Programming languages based on the Actor model
- Active Object in C++11
- The ACE Task framework



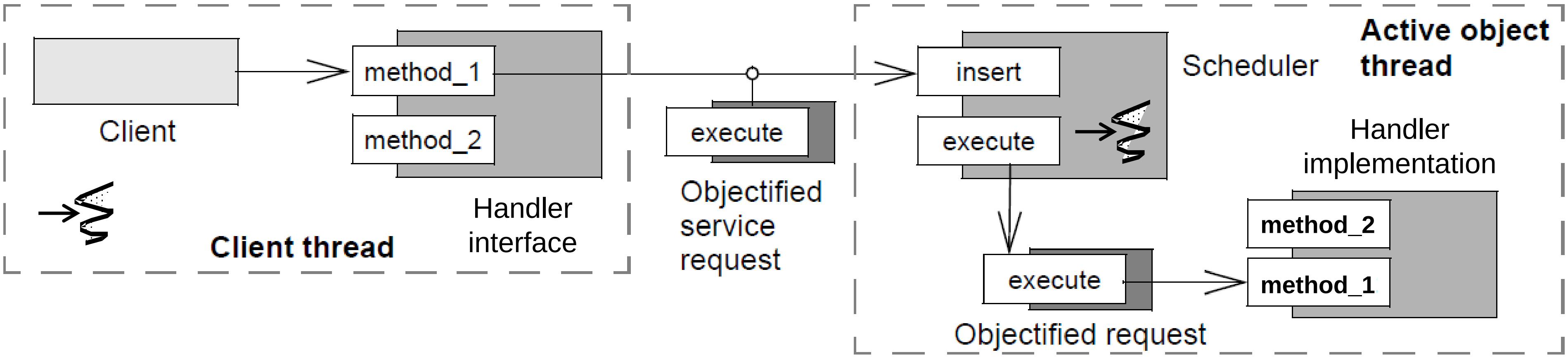
See www.dre.vanderbilt.edu/~schmidt/PDF/ACE-concurrency.pdf for more info

Summary



- Clients may need to issue requests on components without blocking until the requests execute
  - It should also be possible to schedule the execution of client requests according to certain criteria
    - e.g., request priorities or deadlines

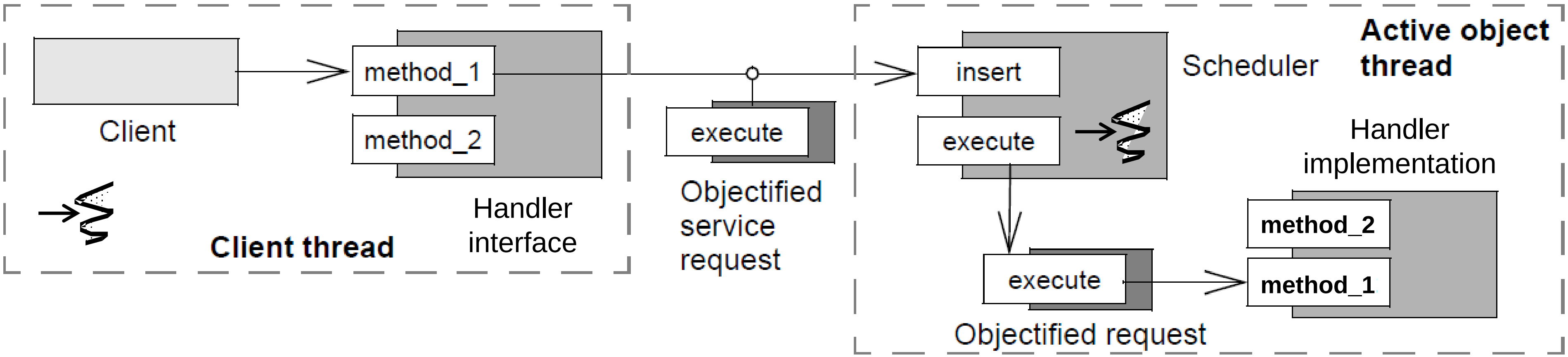
Summary



- Clients may need to issue requests on components without blocking until the requests execute
- The *Active Object* pattern helps keep service requests independent so they can be serialized & scheduled transparently to the component & its clients

See www.dre.vanderbilt.edu/~schmidt/PDF/Act-Obj.pdf for *Active Object*

Summary



- Clients may need to issue requests on components without blocking until the requests execute
- The *Active Object* pattern helps keep service requests independent so they can be serialized & scheduled transparently to the component & its clients
- It's instructive to compare *Active Object* with *Monitor Object*
  - *Active Object* is more powerful, but also more complicated (& potentially more overhead)

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