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

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

```java
Handler.sendMessage(msg);

void handleMessage(Message msg) {
  ...
}
```
Challenge: Invoking Methods in Another Thread

Problems

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

```java
Handler.sendMessage(msg);

void handleMessage(Message msg)
{
    ...
}
```
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

```java
Thread1

Looper

Handler

Message Queue

Message

Message

Message

Message

Message

Handler.sendMessage(msg);

void handleMessage(Message msg) {
    ...
}

Thread2
```
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

```java
void handleMessage(Message msg) {
    
    
}
```
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

```
Client

sendMessage

Handler interface

Objectified service request

insert

execute

Scheduler

Handler implementation

sendMessage

handleMessage
```
Active Object

**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.

---

www.dre.vanderbilt.edu/~schmidt/PDF/Act-Obj.pdf has more info
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

Diagram:
- Client
- method_1
- method_2
- Handler interface
- Objectified service request
- execute
- insert
- Scheduler
- Handler implementation
- Active object thread
- method_2
- method_1
**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

POSAC2 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

POS A2 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

Structure & Participants

Proxy
method_1()
method_N()

Scheduler
dispatch()
insert()

Activation List
insert()
remove()

Servant
method_1()
method_N()

Future
<<invoke>>
<<write to>>
<<obtain result from>>

MethodRequest
can_run()
call()

Client

Concrete MethodRequest 1
Concrete MethodRequest N

Handler.sendMessage()
Active Object  

POSA2 Concurrency

Structure & Participants

- **Proxy**
  - method_1()
  - method_N()

- **Scheduler**
  - dispatch()
  - insert()

- **Activation List**
  - insert()
  - remove()

- **Servant**
  - method_1()
  - method_N()

- **Future**
  - <<obtain result from>>

- **Client**

- **Concrete MethodRequest 1**

- **Concrete MethodRequest N**

- **Handler**

- **MethodRequest**
  - can_run()
  - call()
Active Object  POSA2 Concurrency

Structure & Participants

- **Proxy**
  - `method_1()`
  - `method_N()`

- **Scheduler**
  - `dispatch()`
  - `insert()`

- **Activation List**
  - `insert()`
  - `remove()`

- **Future**

- **MethodRequest**
  - `can_run()`
  - `call()`

- **Client**

- **Concrete MethodRequest 1**

- **Concrete MethodRequest N**

- **Servant**
  - `method_1()`
  - `method_N()`

- **Message**
Active Object

POSAG2 Concurrency

Structure & Participants

Proxy

method_1()
method_N()

Scheduler

dispatch()
insert()

MessageQueue

Activation List

insert()
remove()

Future

<<invoke>>

<<write to>>

<<obtain result from>>

<<invoke>>

Future

<<write to>>

MethodRequest

can_run()
call()

<<maintain>>

Servant

<<execute>>

method_1()
method_N()

<<obtain result from>>

Client

Concrete MethodRequest 1

Concrete MethodRequest N
Active Object

POS A2 Concurrency

Structure & Participants

- Proxy
  - method_1()
  - method_N()
- Scheduler
  - dispatch()
  - insert()
- Activation List
  - insert()
  - remove()
- Looper
- Future
- MethodRequest
  - can_run()
  - call()
- Servant
  - method_1()
  - method_N()
- Client

Concrete MethodRequest 1

Concrete MethodRequest N
Active Object

Structure & Participants

Proxy
- method_1()
- method_N()

Scheduler
- dispatch()
- insert()

Activation List
- insert()
- remove()

Future
- <<invoke>>
- <<instantiate>>
- <<obtain result from>>

MethodRequest
- can_run()
- call()
- * <<maintain>>
- <<execute>>

Servant
- method_1()
- method_N()

Client

Concrete MethodRequest 1

Concrete MethodRequest N

MyHandler
Active Object

Structure & Participants

Android Concurrency

POSAs2 Concurrency

Proxy

method_1()
method_N()

Scheduler

dispatch()
insert()

Activation List

insert()
remove()

Servant

method_1()
method_N()

Future

<<invoke>>

<<write to>>

<<obtain result from>>

MethodRequest

can_run()
call()

<<maintain>>

<<execute>>

Client

MyActivity

MyActivity

Concrete MethodRequest 1

Concrete MethodRequest N
Active Object

Dynamics

Client invokes a method call on a proxy

: Client
: Proxy
: Future
: Scheduler
: Activation List
: Servant

Client method() <<create>>
Future insert() dispatch()
Future can_run() remove()
Future call() method()
Proxy converts method call into method request, passes to scheduler, & returns future to client
Scheduler enqueues method request
Scheduler dequeues method request at some point & runs it on the servant in a separate thread.
Clients can obtain result from futures via polling, or callbacks.
Active Object

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);
        ...
Active Object

Implementation

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

```java
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](https://android-developers.google.cn) has the source code.
Active Object

**Implementation**

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

```java
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

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

```java
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

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

```java
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

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

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()
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
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

Block Activity thread until the query is done

Run query on ContentProvider & return result

See developer.android.com/reference/android/content/ContentResolver.html
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
Applying Active Object in Android

- AsyncQueryHandler is a helper class that invokes ContentResolver calls asynchronously to avoid blocking the UI thread.
Applying Active Object in Android

AsyncQueryHandler is a helper class that invokes ContentResolver calls asynchronously to avoid blocking the UI thread.

Asynchronous Query

- Activity
- MyAsyncQueryHandler
- WorkerHandler
- ContentResolver

Start query
send message
Handle message
Query
Run query on ContentProvider & return result

Block worker thread until the query is done

See developer.android.com/reference/android/content/AsyncQueryHandler.html
Applying Active Object in Android

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

**Asynchronous Query**

- Activity
- MyAsync QueryHandler
- Worker Handler
- Content Resolver

Call startQuery()

Send to target

Handle message

Query

Run query on ContentProvider & return result

Caller returns & the call runs asynchronously

See developer.android.com/reference/android/content/AsyncQueryHandler.html
Applying Active Object in Android

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

Active Object

POS A2 Concurrency

AsyncQueryHandler

Worker Handler

Content Resolver

Activity

MyAsync QueryHandler

Start Query

SendMessage

Handle Message

Handle completion of the asynchronously invoked query

Run query on ContentProvider & return result

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

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);
        ...
    }
```

[Diagram showing the flow of messages and threads]

```
4. void handleMessage(Message msg) {
    switch (event) {
    case EVENT_ARG_QUERY:
        Cursor cursor = resolver.query(....);
        ...
    }
```

Frameworks/base/core/java/android/content/AsyncQueryHandler.java has code
Consequences

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

```java
void handleMessage(Message msg)
{
    ...
}
```

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

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

```java
void handleMessage(Message msg)
{
    ...
}
```

```java
Handler.sendMessage(msg);
```
Active Object

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

```java
Handler.sendMessageDelayed(msg, delayTime);
```

```java
void handleMessage(Message msg) {
    ...
}
```

POSAS2 Concurrency

Thread1

Looper

Message Queue

Handler

void handleMessage(Message msg) {
    ...
}
Active Object

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  

POS A2 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

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;
   }
   ...
}

POS A2 Concurrency
Active Object

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;
     }
   ... 

POSQA2 Concurrency

Android Concurrency
Active Object

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

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

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

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

Known Uses

• Programming languages based on the Actor model

• Active Object in C++11

    ```cpp
    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

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

Pass message to Task

Remove message from Queue & process it concurrently

See [www.dre.vanderbilt.edu/~schmidt/PDF/ACE-concurrency.pdf](http://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](http://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](http://www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf) for comparisons.