Android Services & Local IPC: The Activator Pattern (Part 2)

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

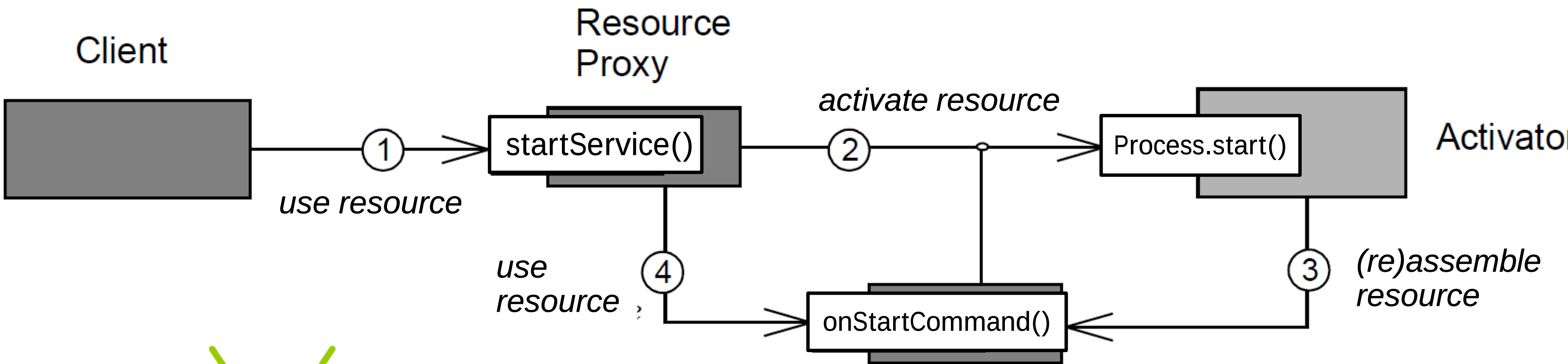
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Module

- Understand how the *Activator* pattern is applied in Android



Activator POSA4 Design Pattern

Implementation

- Define services & service identifiers
  - Encapsulate each distinct unit of app functionality into a self-contained service

```
public abstract class Service
       extends ContextWrapper
       implements
          ComponentCallbacks2
{
  public abstract IBinder
    onBind(Intent intent);

  ...
}
```

frameworks/base/core/java/android/app/Service.java has the source code

Activator POSA4 Design Pattern

Implementation

- Define services & service identifiers
  - Encapsulate each distinct unit of app functionality into a self-contained service
  - Examples of service identifier representations include URLs, IORs, TCP/IP port numbers & host addresses, Android Intents, etc.

Intent Element	Purpose
Name	Optional name for a component
Action	A string naming the action to perform or the action that took place & is being reported
Data	URI of data to be acted on & the MIME type of that data
Category	String giving additional info about the action to execute

frameworks/base/core/java/android/content/Intent.java has the source code

Activator POSA4 Design Pattern

Implementation

- Define services & service identifiers
- Identify services to activate & deactivate on demand
  - Determine overhead of activating & deactivating services on-demand vs. keeping them alive for the duration of the system vs. security implications, etc.

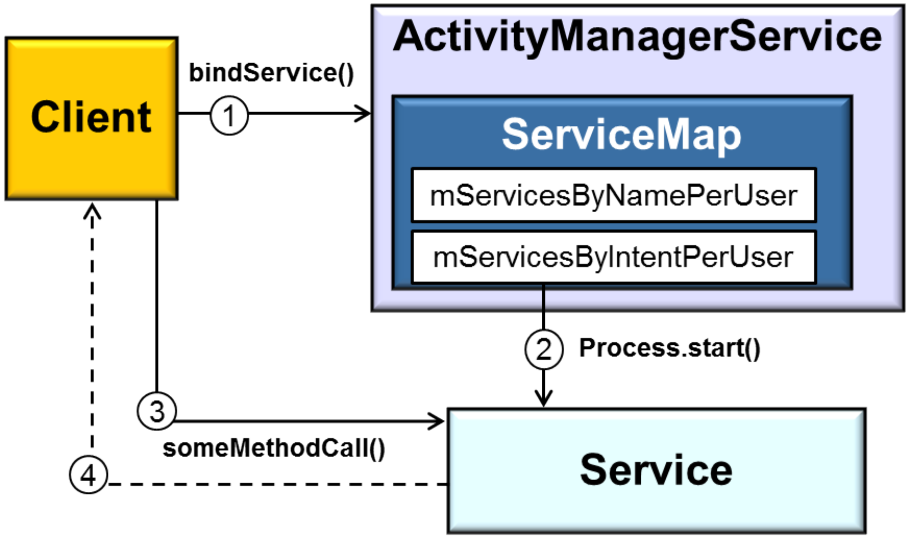
Android Service	Purpose
Media Playback Service	Provides “background” audio playback capabilities
Exchange Email Service	Send/receive email messages to an Exchange server
SMS & MMS Services	Manage messaging operations, such as sending data, text, & PDU messages
Alert Service	Handle calendar event reminders

packages/apps has source code for many Bound & Started Services

Activator POSA4 Design Pattern

Implementation

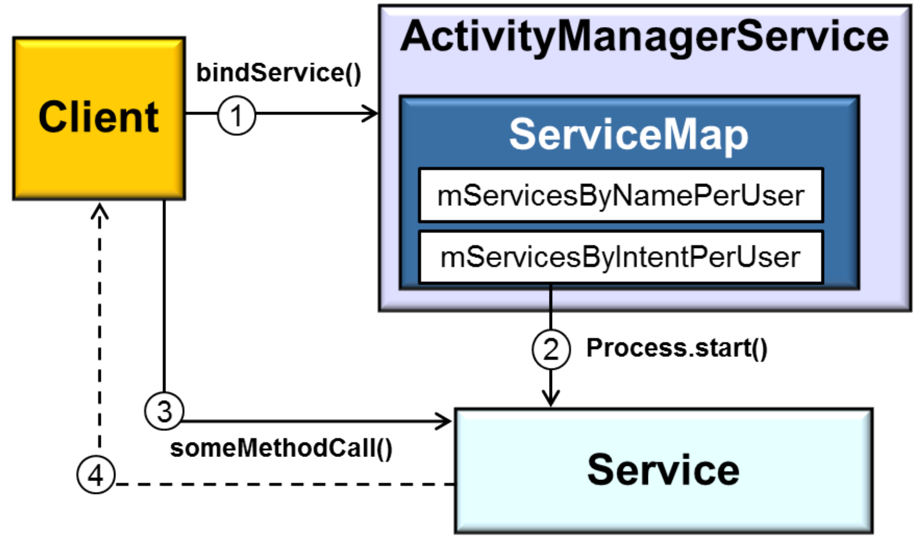
- Define services & service identifiers
- Identify services to activate & deactivate on demand
- Develop service activation & deactivation strategy
    - Define service execution context representation
        - e.g., an OS process/thread or middleware container



Activator POSA4 Design Pattern

Implementation

- Define services & service identifiers
- Identify services to activate & deactivate on demand
- Develop service activation & deactivation strategy
  - Define service execution context representation
  - Define service registration strategy
    - e.g., static text file or dynamic object registration



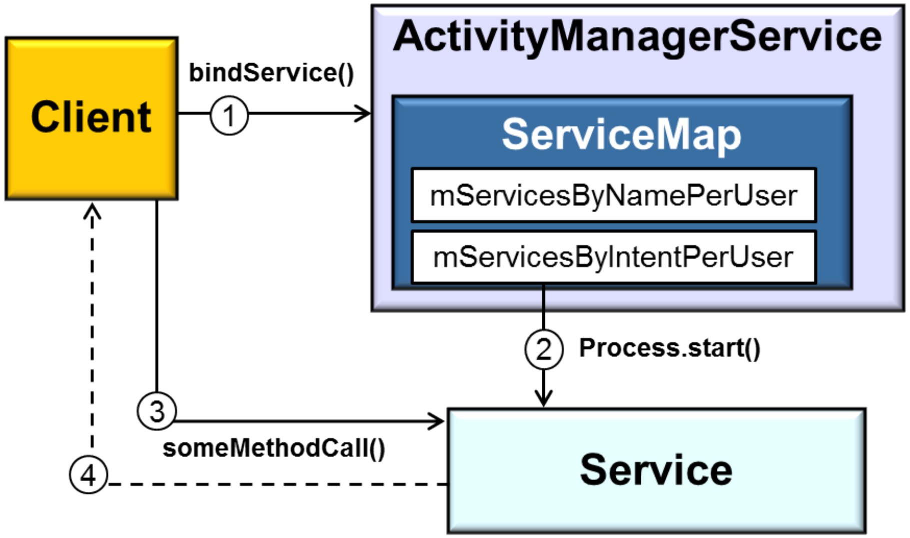

```
<service android:name=
"com.android.music.MediaPlaybackService"
   android:exported="false"/>
```

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

Activator POSA4 Design Pattern

Implementation

- Define services & service identifiers
- Identify services to activate & deactivate on demand
- Develop service activation & deactivation strategy
  - Define service execution context representation
  - Define service registration strategy
  - Define service initialization strategy
    - e.g., stateful vs. stateless services



Android Services are responsible for managing their own persistent state

Activator POSA4 Design Pattern

Implementation

- Define services & service identifiers
- Identify services to activate & deactivate on demand
- Develop service activation & deactivation strategy
  - Define service execution context representation
  - Define service registration strategy
  - Define service initialization strategy
  - Define service deactivation strategy
    - e.g., service-triggered, client-triggered, or activator-triggered deactivation

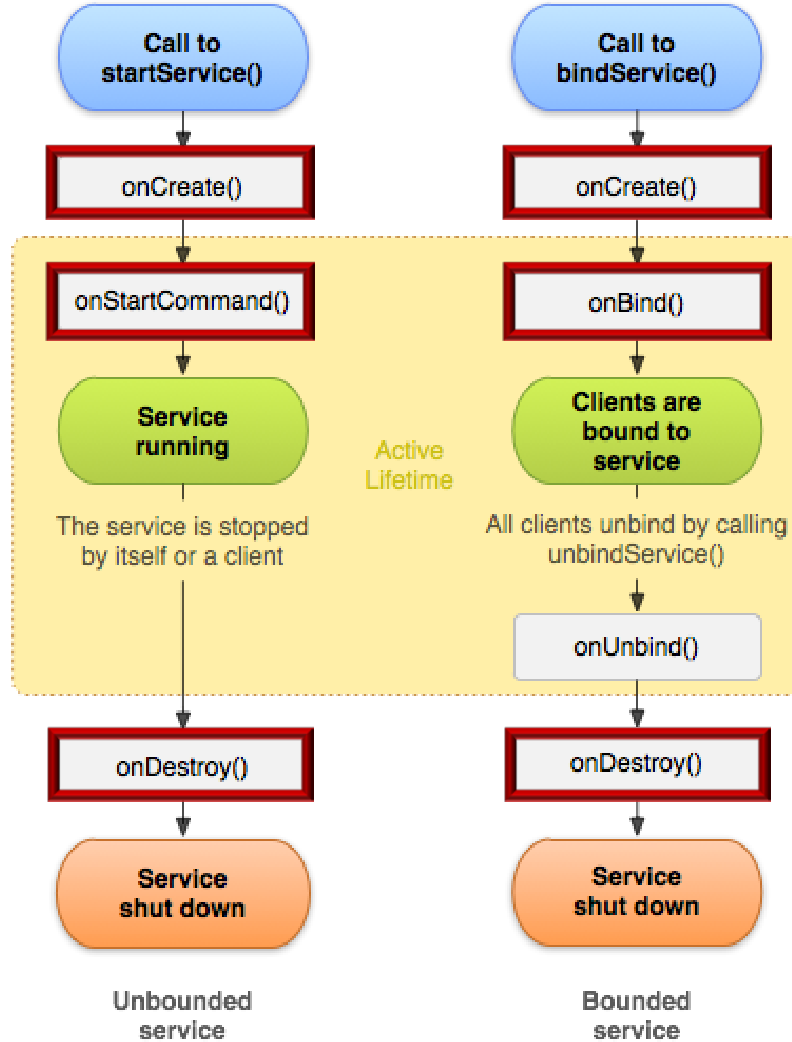
Started Service	Bound Service
• Service runs indefinitely & must stop itself by calling stopSelf() • A component can also stop the service by calling stopService() • When Service is stopped, Android destroys it	• Multiple clients can bind to same Service • When all of them unbind, the system destroys the Service • The Service does not need to stop itself an

developer.android.com/guide/components/services.html#Lifecycle has more

Activator POSA4 Design Pattern

Implementation

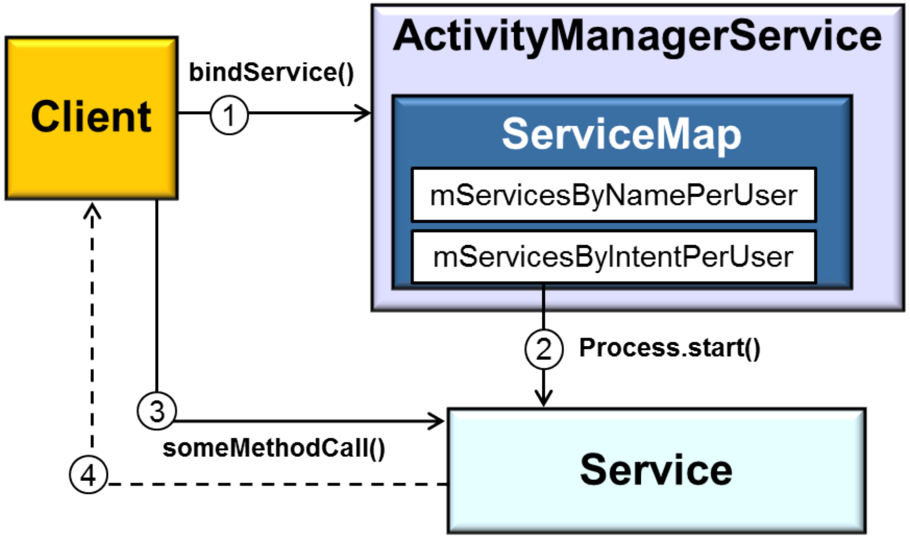
- Define services & service identifiers
- Identify services to activate & deactivate on demand
- Develop service activation & deactivation strategy
- Define interoperation between services & service execution context
  - Typically implemented via some type of lifecycle callback hook methods



Activator POSA4 Design Pattern

Implementation

- Define services & service identifiers
- Identify services to activate & deactivate on demand
- Develop service activation & deactivation strategy
- Define interoperation between services & service execution context
- Implement the activator
  - Determine the association between activators & services
    - e.g., singleton (shared) vs. exclusive vs. distributed activator

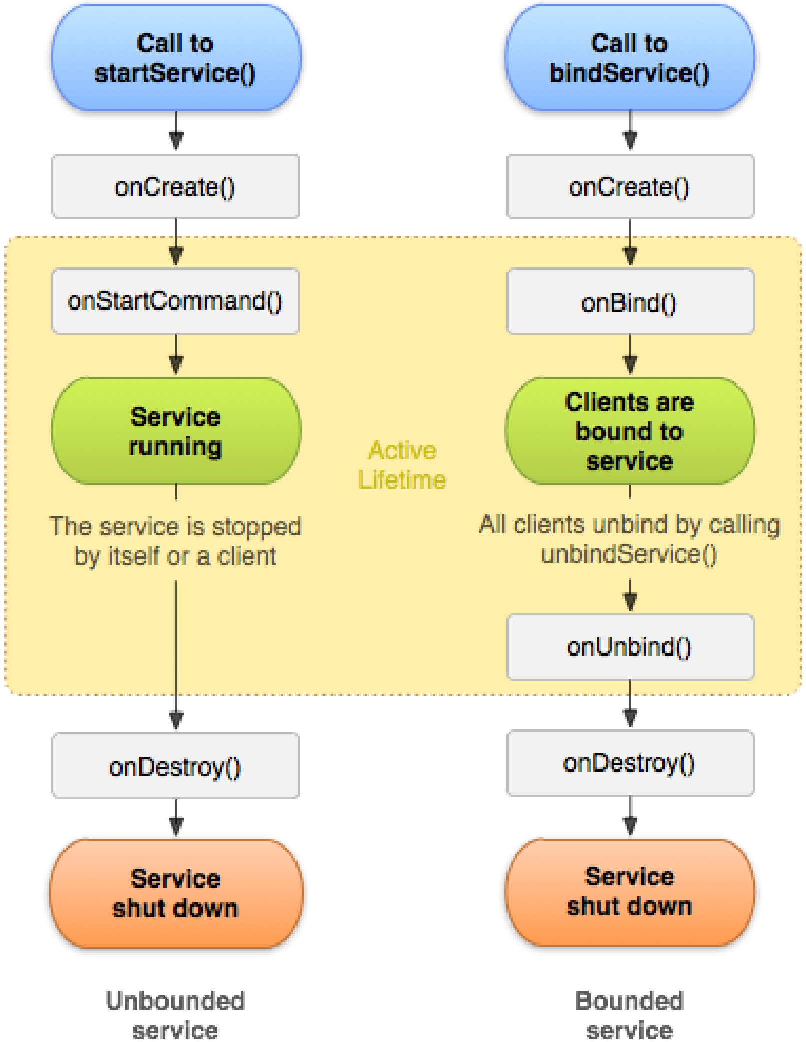


The Android Activity Manager Service is a singleton activator

Activator POSA4 Design Pattern

Implementation

- Define services & service identifiers
- Identify services to activate & deactivate on demand
- Develop service activation & deactivation strategy
- Define interoperation between services & service execution context
- Implement the activator
  - Determine the association between activators & services
  - Determine the degree of transparency
    - e.g., explicit vs. transparent activator

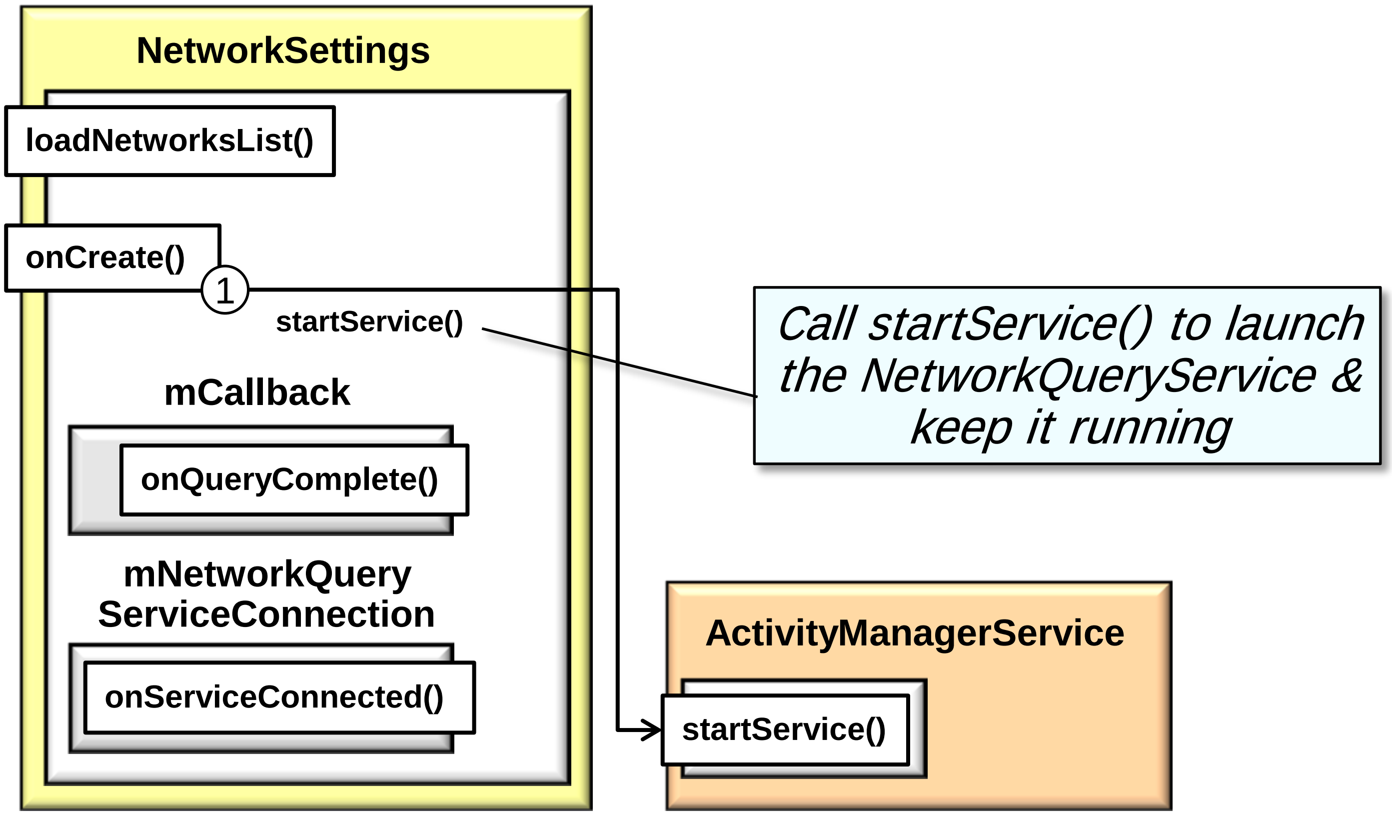


Android Started & Bound Services use an explicit activator model

Activator POSA4 Design Pattern

Applying Activator in Android

- The NetworkSettings Activity uses the *Activator* pattern to launch the NetworkQueryService to assist in querying the network for service availability

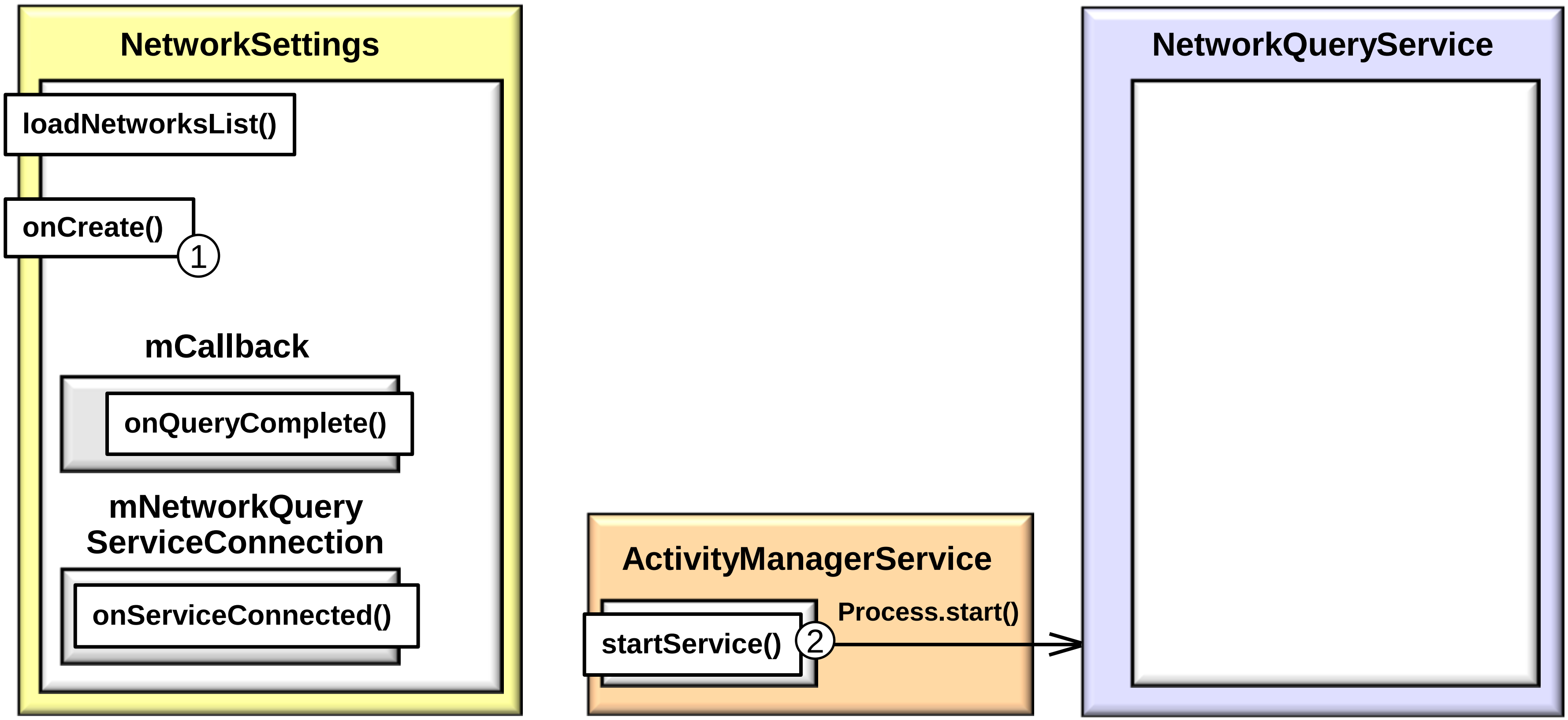


packages/apps/Phone/src/com/android/phone/NetworkSetting.java has source code

Activator POSA4 Design Pattern

Applying Activator in Android

- The NetworkSettings Activity uses the *Activator* pattern to launch the NetworkQueryService to assist in querying the network for service availability

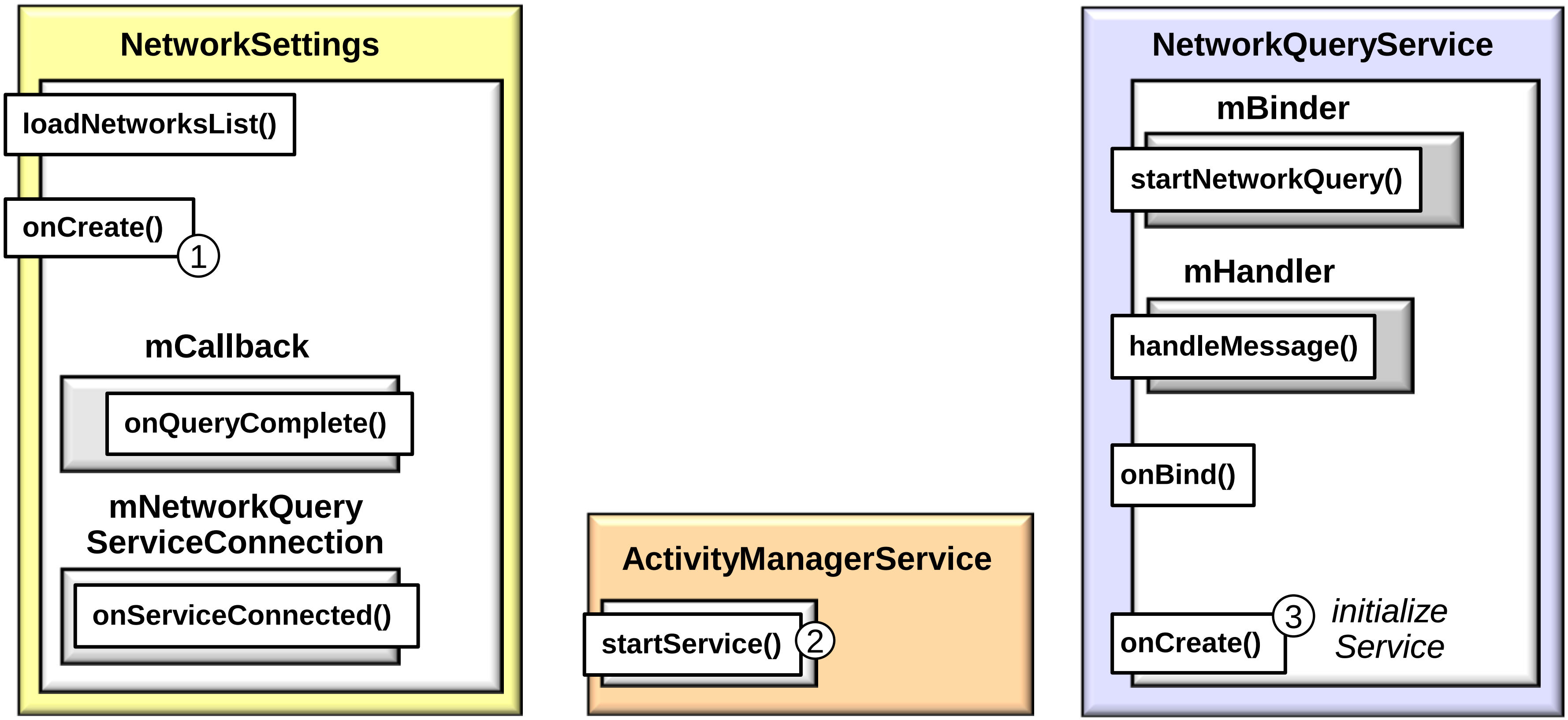


frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

Activator POSA4 Design Pattern

Applying Activator in Android

- The NetworkSettings Activity uses the *Activator* pattern to launch the NetworkQueryService to assist in querying the network for service availability

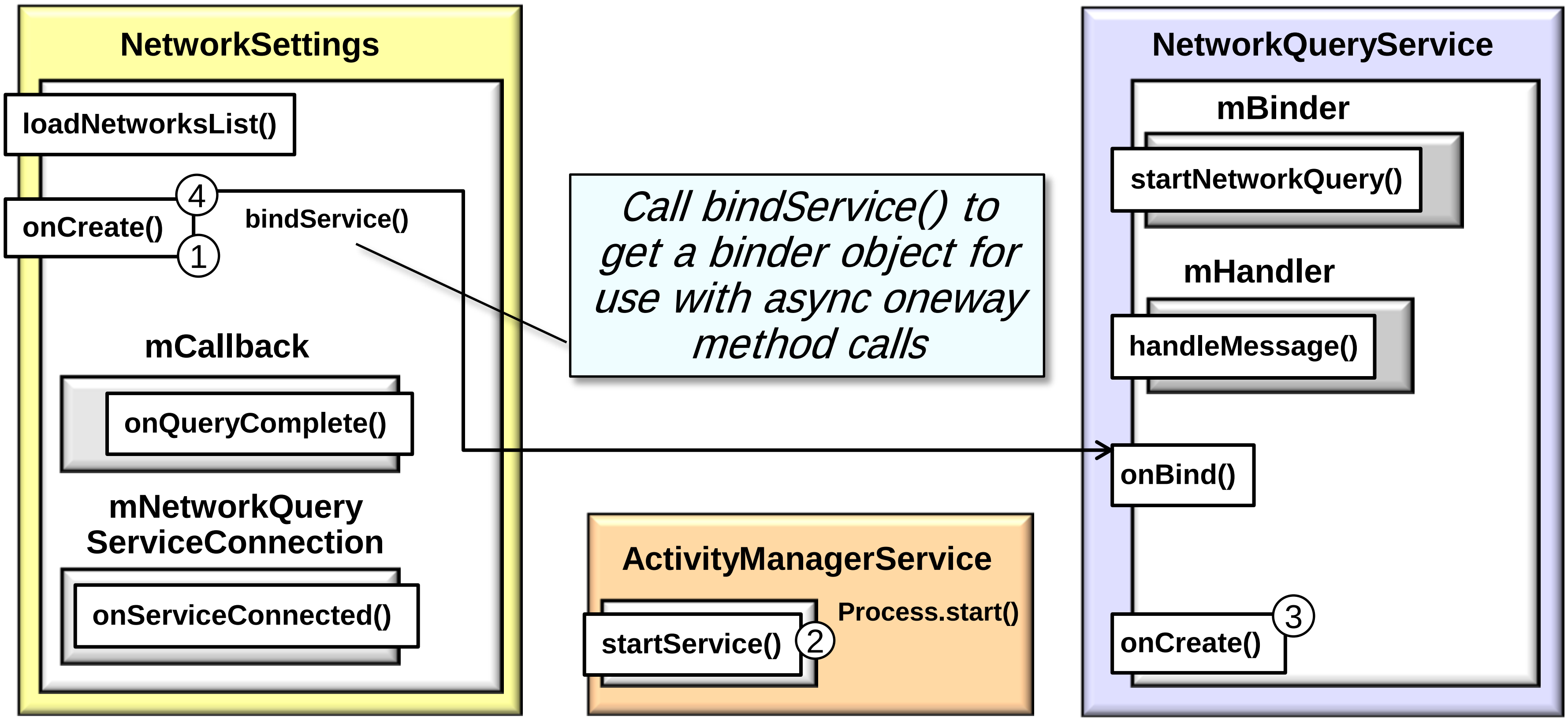


packages/apps/Phone/src/com/android/phone/NetworkQueryService.java has source

Activator POSA4 Design Pattern

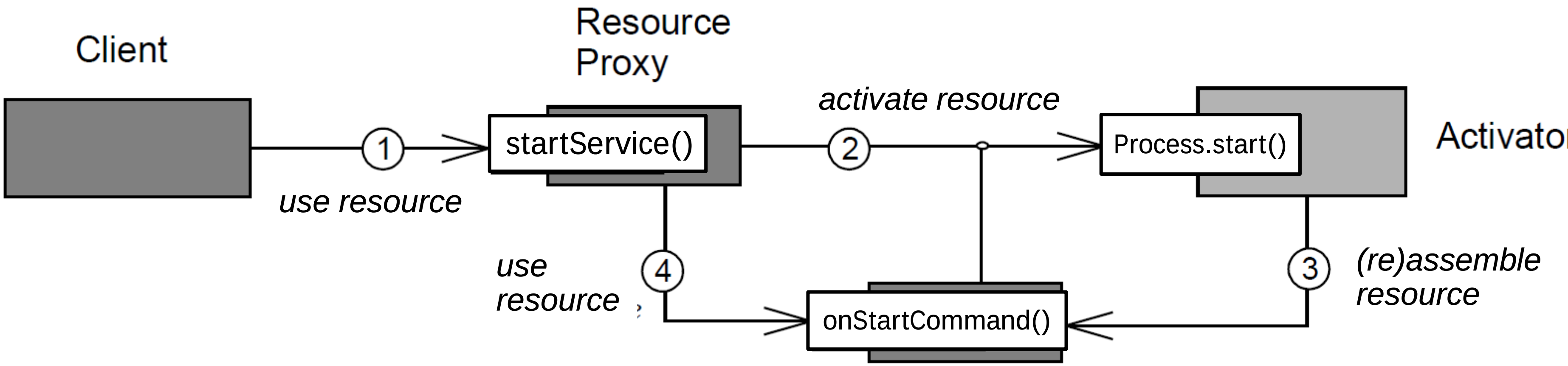
Applying Activator in Android

- The NetworkSettings Activity uses the *Activator* pattern to launch the NetworkQueryService to assist in querying the network for service availability



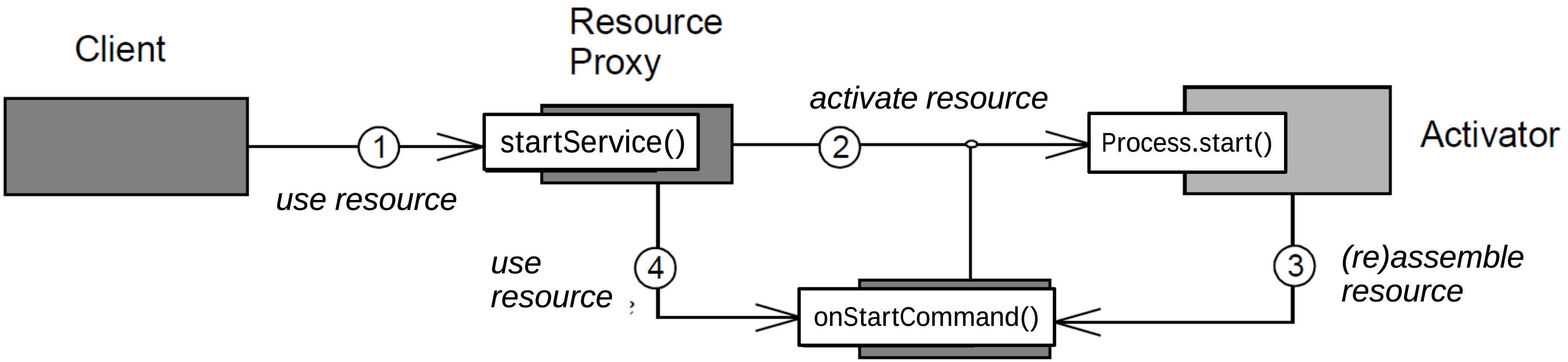
packages/apps/Phone/src/com/android/phone/NetworkSetting.java has source code

Summary



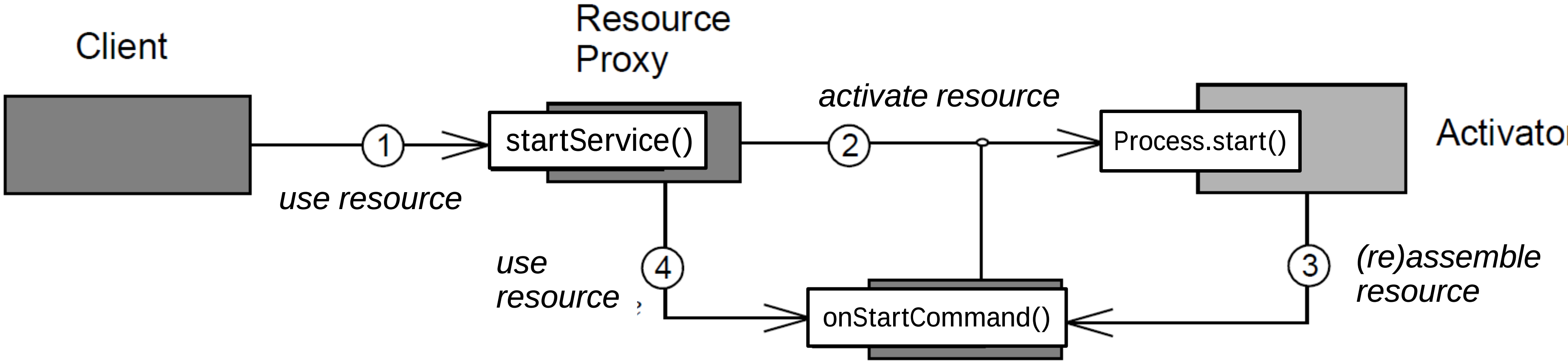
- The Android Started & Bound Services implement the *Activator* pattern

Summary



- The Android Started & Bound Services implement the *Activator* pattern
- These Services can process requests in background processes or threads
  - Processes can be configured depending on directives in the AndroidManifest.xml file

Summary



- The Android Started & Bound Services implement the *Activator* pattern
- These Services can process requests in background processes or threads
  - Processes can be configured depending on directives in the AndroidManifest.xml file
  - Threads can be programmed using IntentService et al.

Android Services & Local IPC: The Proxy Pattern (Part 1)

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

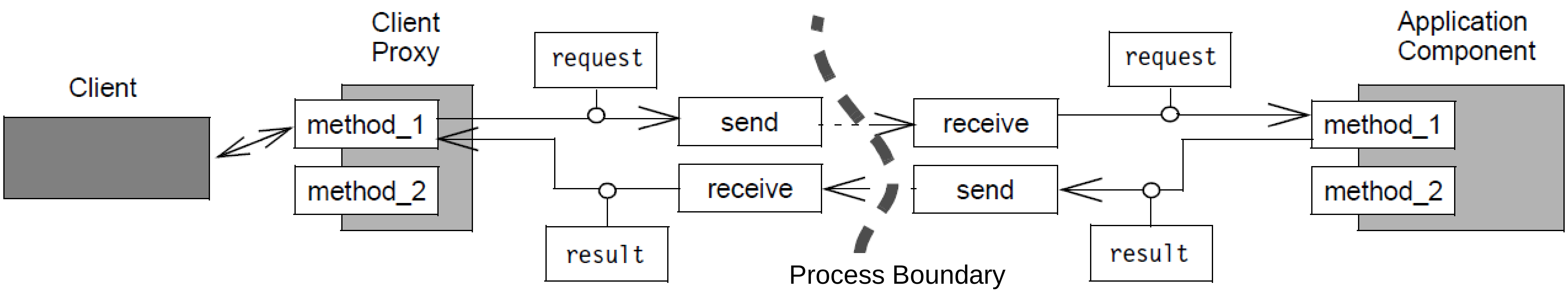
Professor of Computer Science

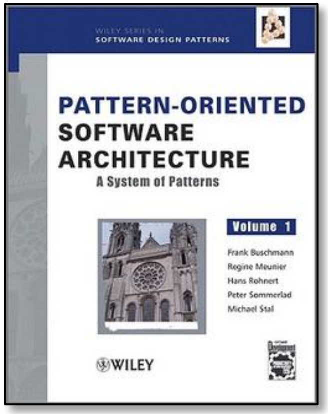
Institute for Software Integrated Systems

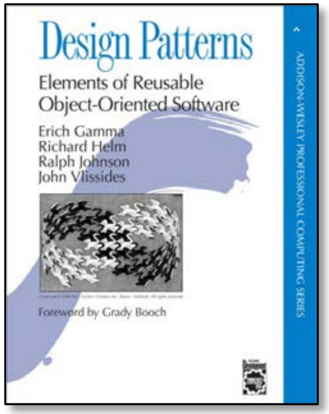
Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Module

- Understand the *Proxy* pattern



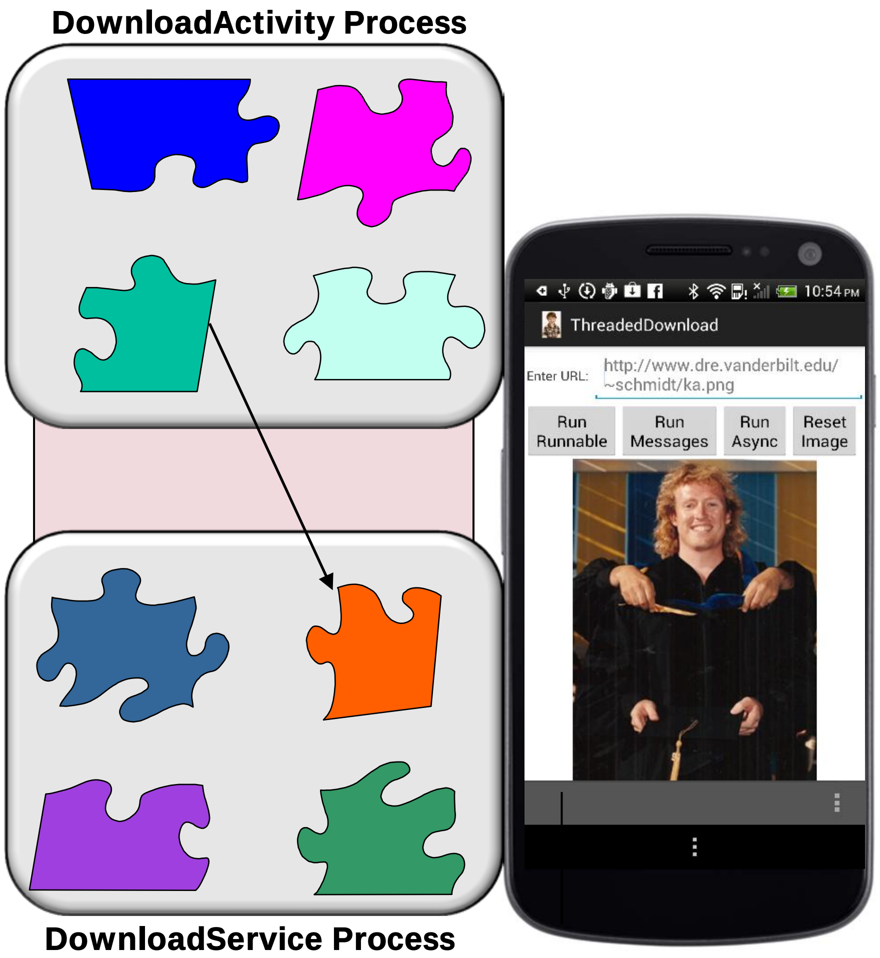




Challenge: Simplifying Access to Remote Objects

Context

- It is often infeasible—or impossible—to access an object directly
  - e.g., may reside in server process

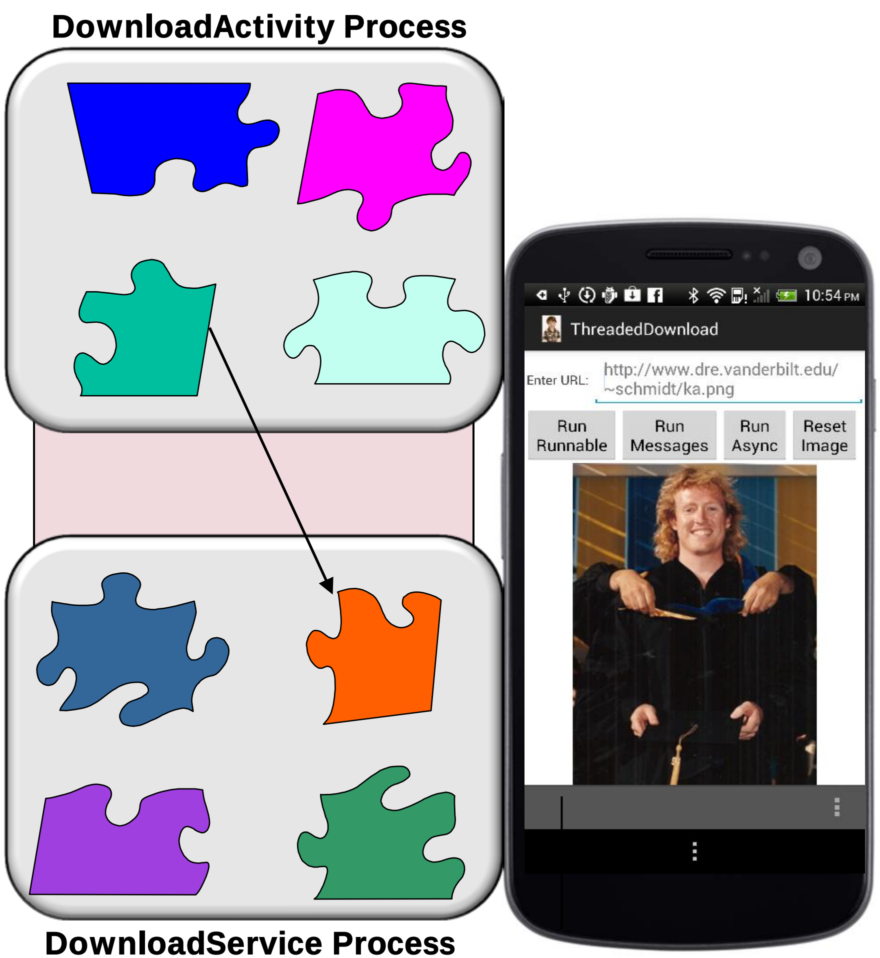


Android's Binder provides a high-performance IPC mechanism

Challenge: Simplifying Access to Remote Objects

Context

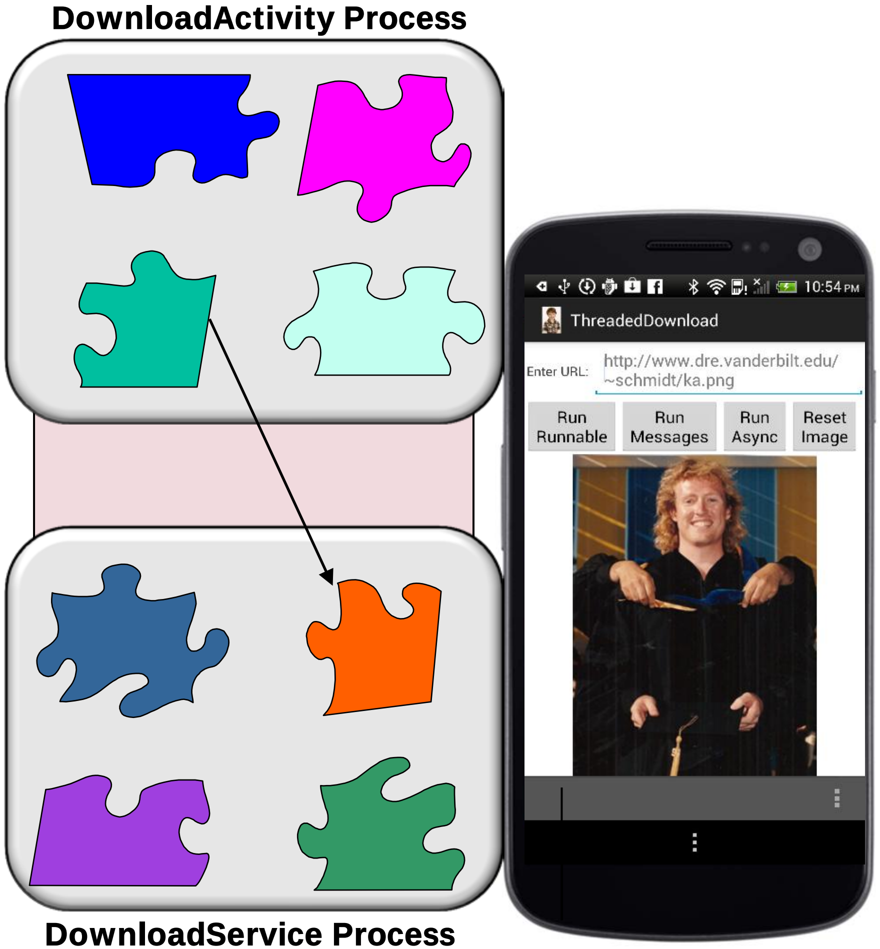
- It is often infeasible—or impossible—to access an object directly
- Partitioning of objects in a system may change as requirements evolve



Challenge: Simplifying Access to Remote Objects

Problems

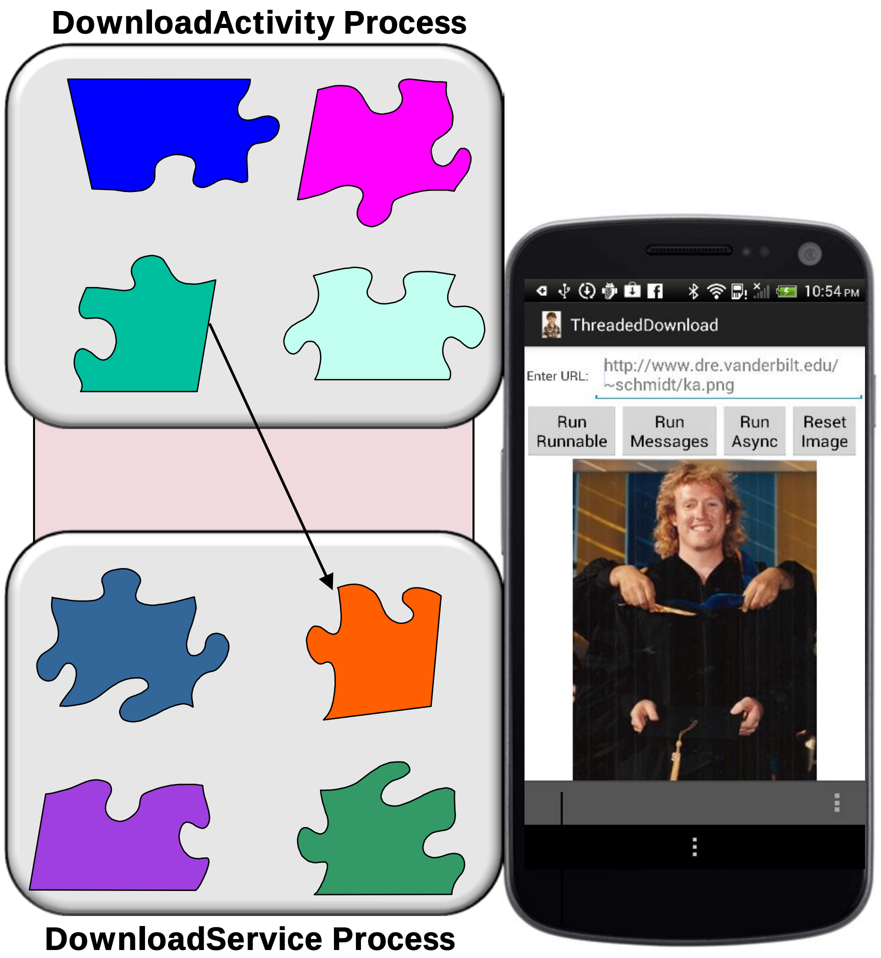
- Manually (de)marshaling messages can be tedious, error-prone, & inefficient



Challenge: Simplifying Access to Remote Objects

Problems

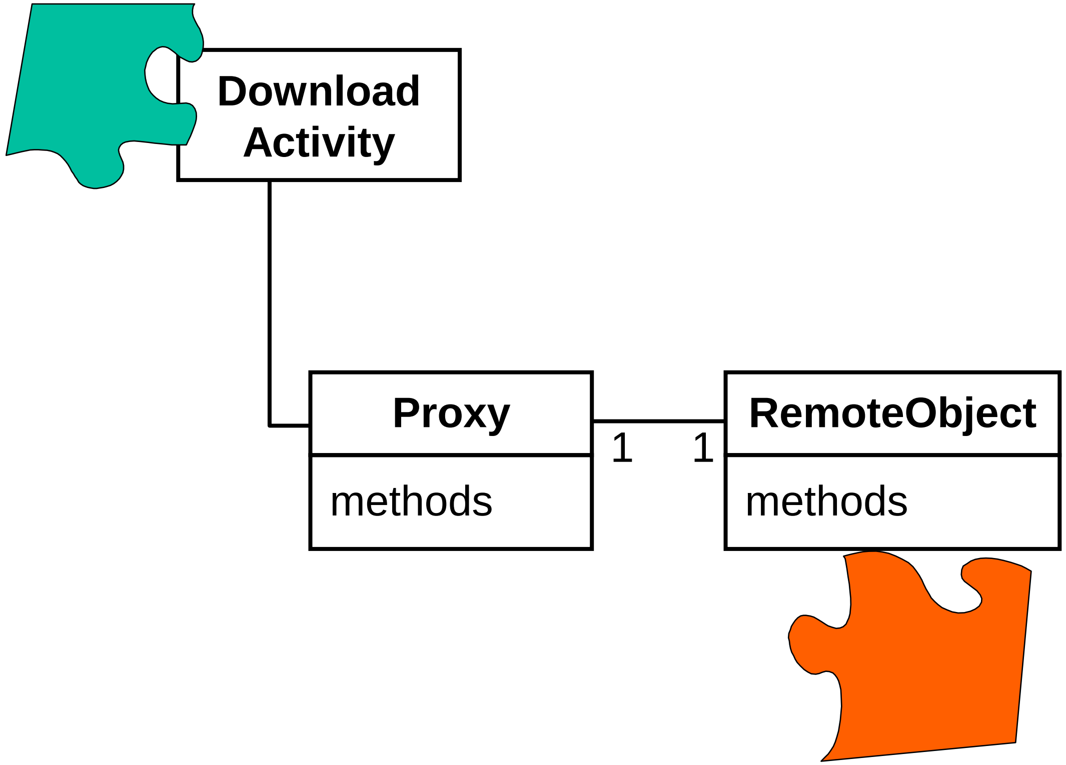
- Manually (de)marshaling messages can be tedious, error-prone, & inefficient
- It is time-consuming to re-write, re-configure, & re-deploy components across address spaces as requirements & environments change



Challenge: Simplifying Access to Remote Objects

Solution

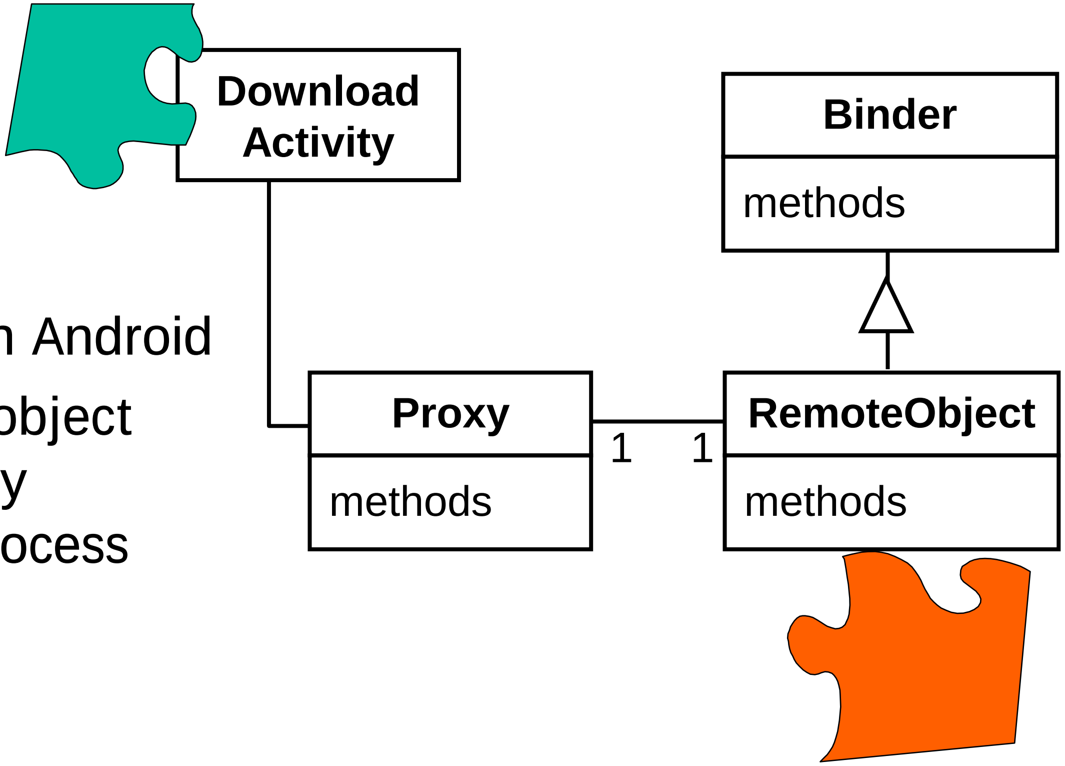
- Define a *proxy* that provides a surrogate thru which clients can access remote objects



Challenge: Simplifying Access to Remote Objects

Solution

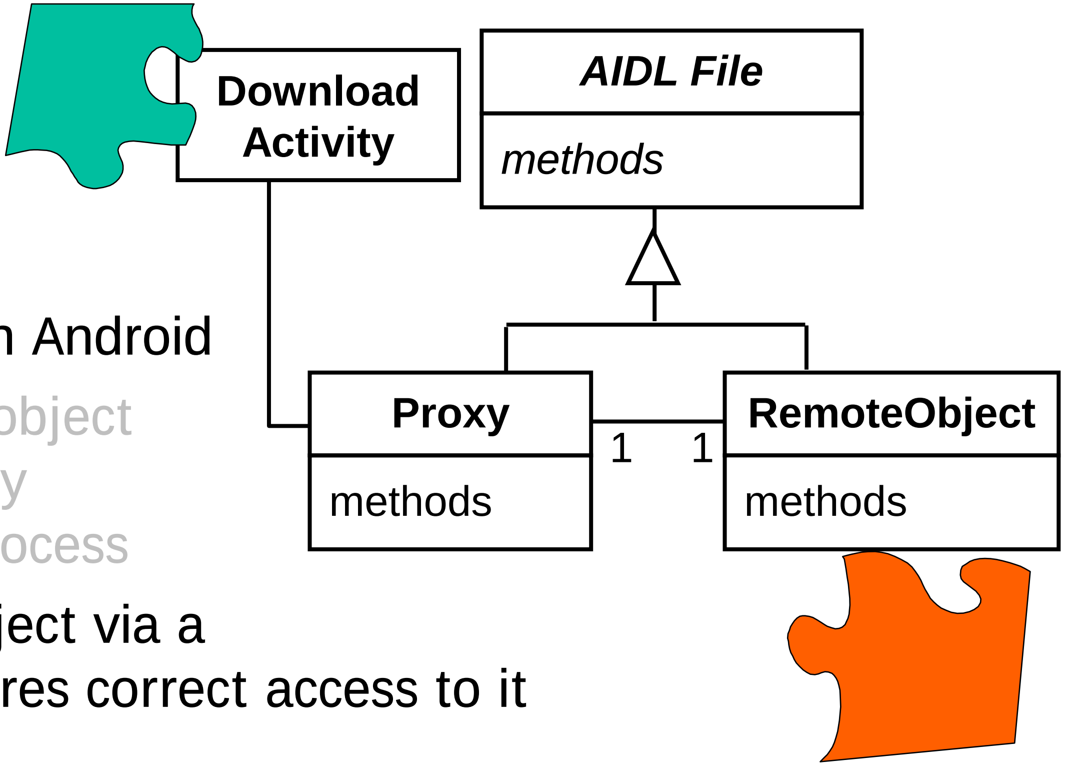
- Define a *proxy* that provides a surrogate thru which clients can access remote objects
- e.g., one way to implement this in Android
  - A service implements a Binder object that a client can't access directly since it may be in a different process



Challenge: Simplifying Access to Remote Objects

Solution

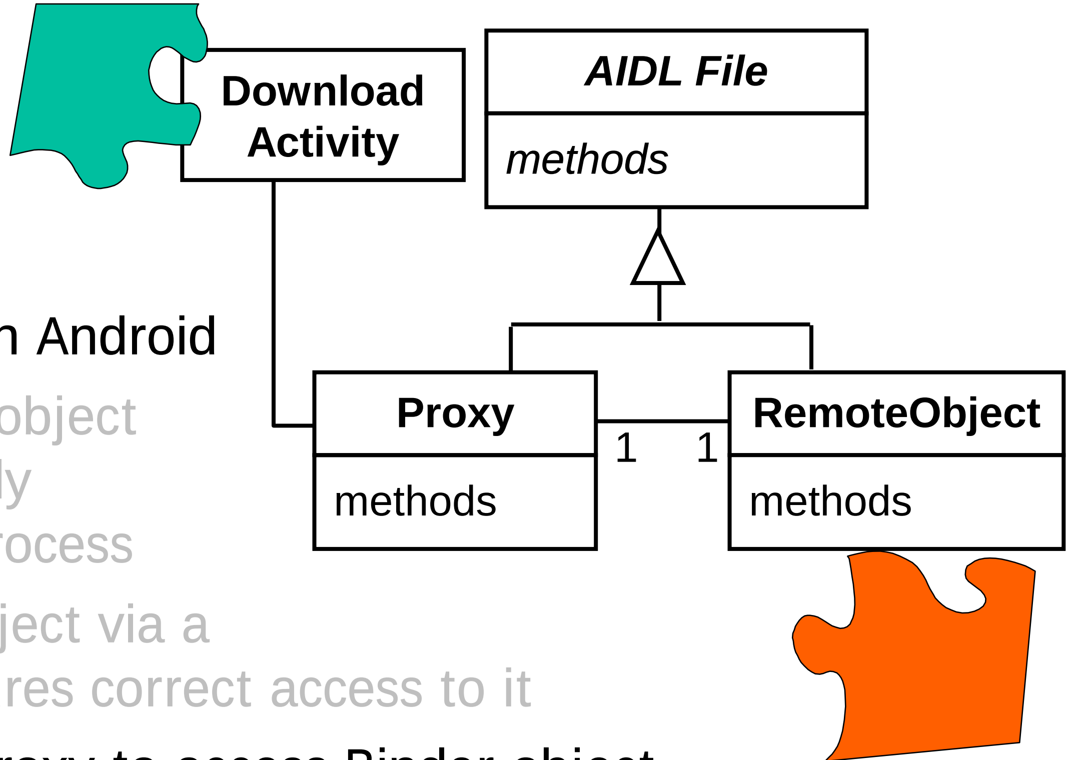
- Define a *proxy* that provides a surrogate thru which clients can access remote objects
- e.g., one way to implement this in Android
  - A service implements a Binder object that a client can't access directly since it may be in a different process
  - Proxy represents the Binder object via a common AIDL interface & ensures correct access to it



Challenge: Simplifying Access to Remote Objects

Solution

- Define a *proxy* that provides a surrogate thru which clients can access remote objects
- e.g., one way to implement this in Android
  - A service implements a Binder object that a client can't access directly since it may be in a different process
  - Proxy represents the Binder object via a common AIDL interface & ensures correct access to it
  - Clients calls a method on the proxy to access Binder object
    - Whether the object is in-process or out-of-process can be controlled via the AndroidManifest.xml config file



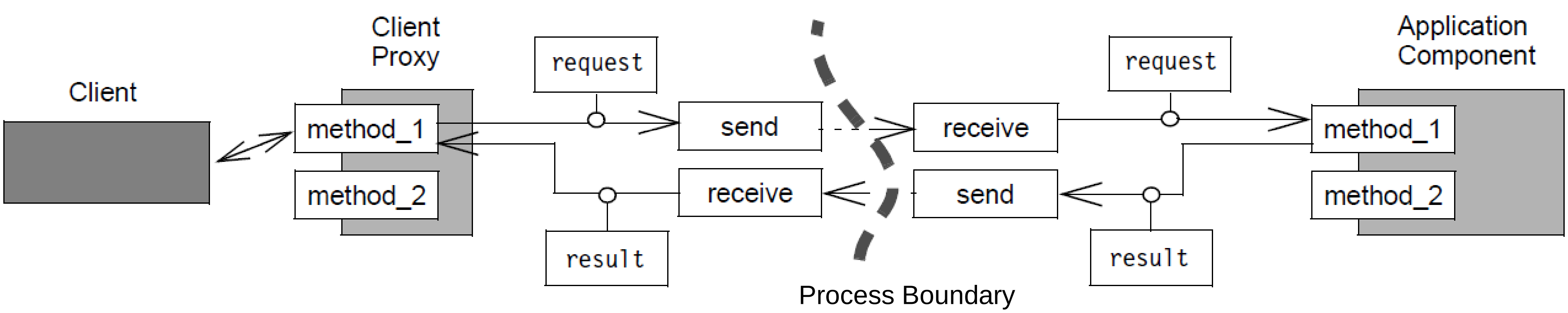
The *Proxy* works together with Binder RPC to implement the *Broker* pattern

Proxy GoF Object Structural

Intent

POSA1 also contains the *Proxy* pattern

- Provide a surrogate or placeholder for another object to control access to it



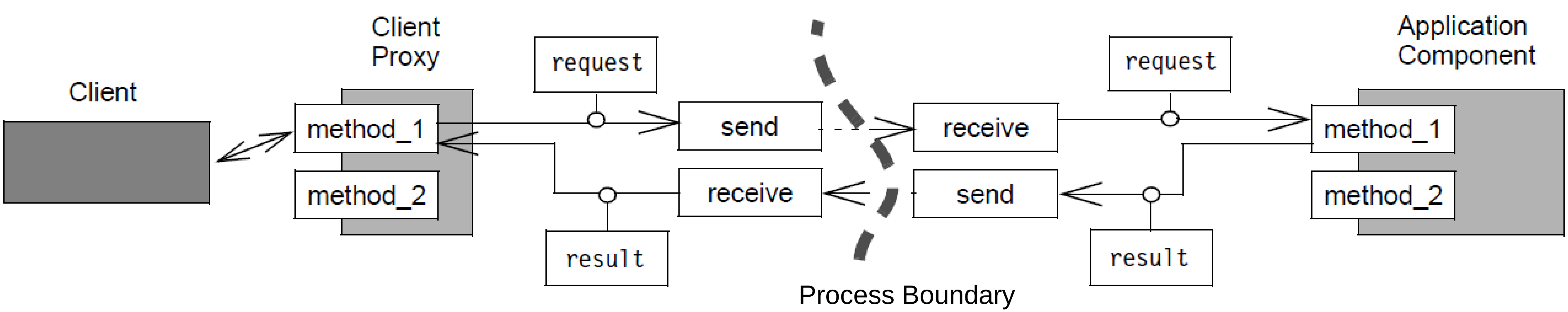
See en.wikipedia.org/wiki/Proxy_pattern for more on *Proxy* pattern

Proxy GoF Object Structural

Applicability

POSA1 also contains the *Proxy* pattern

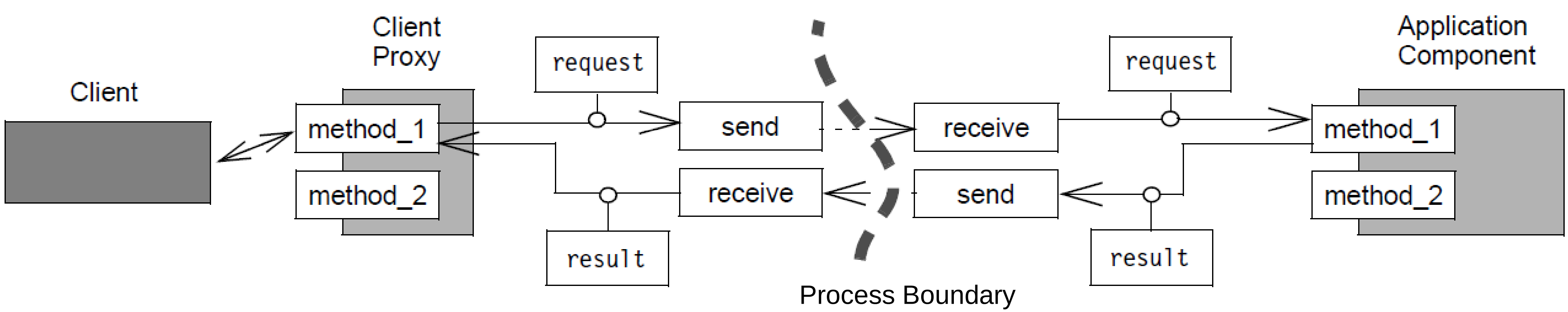
- When there is a need for a more sophisticated reference to a object than a simple pointer or simple reference can provide



Proxy GoF Object Structural

Applicability

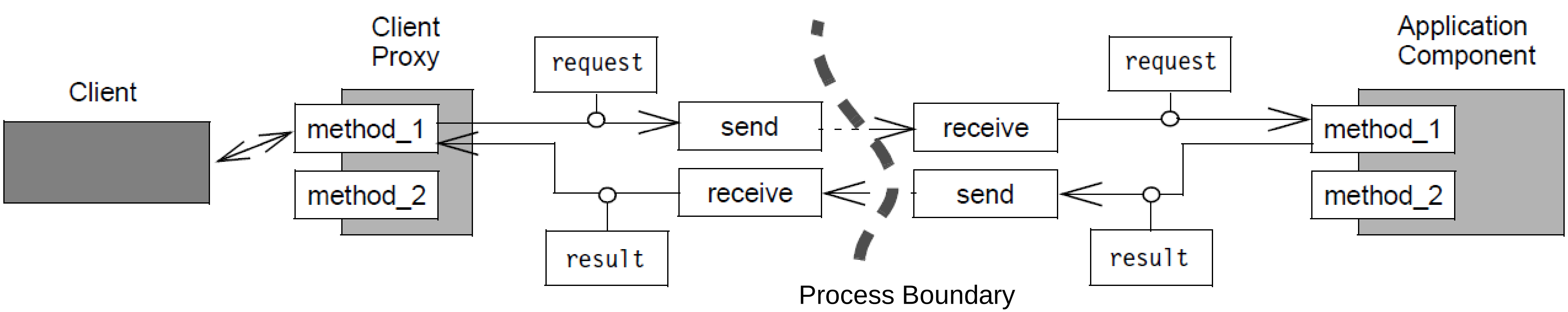
- When there is a need for a more sophisticated reference to a object than a simple pointer or simple reference can provide
- Help ensure remote objects look/act as much like local components as possible from a client app perspective



Proxy GoF Object Structural

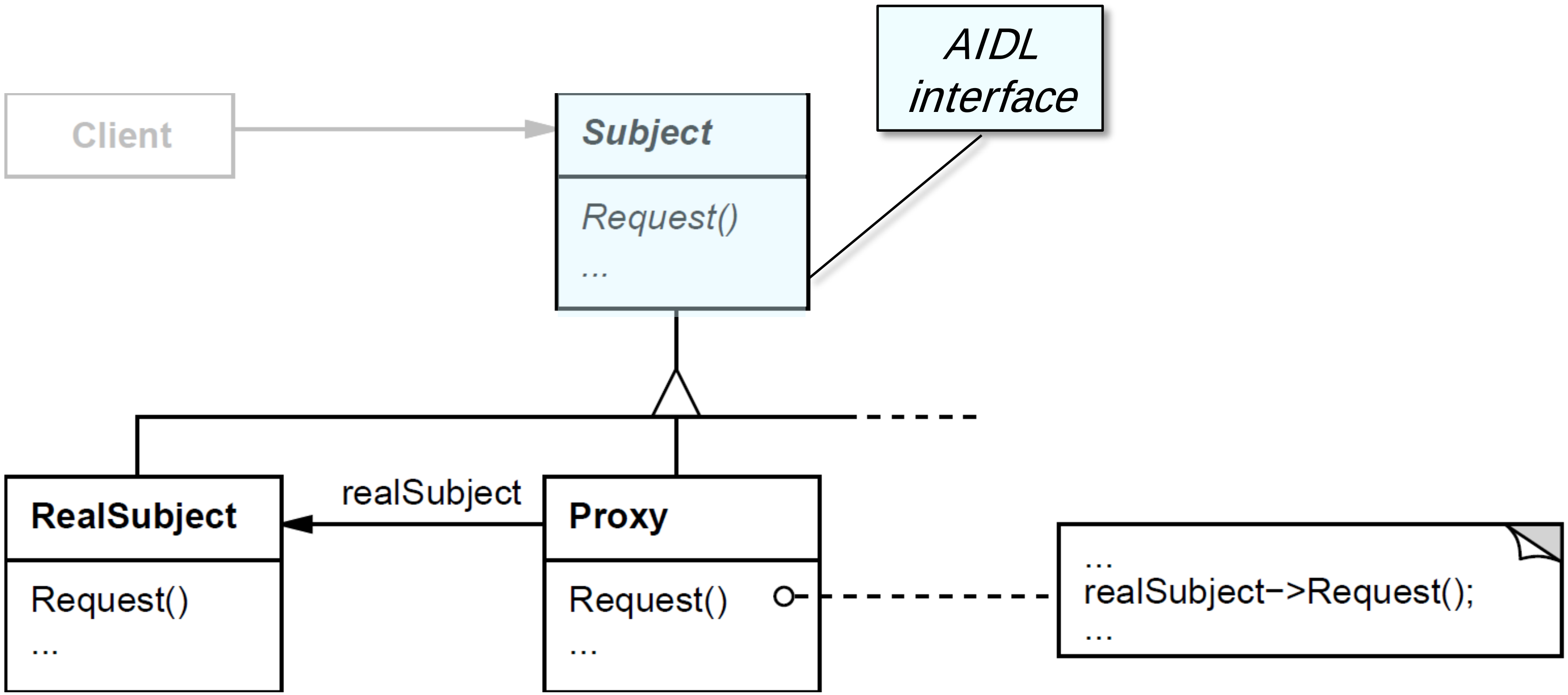
Applicability

- When there is a need for a more sophisticated reference to a object than a simple pointer or simple reference can provide
- Help ensure remote objects look/act as much like local components as possible from a client app perspective
- When there's a need for statically-typed method invocations



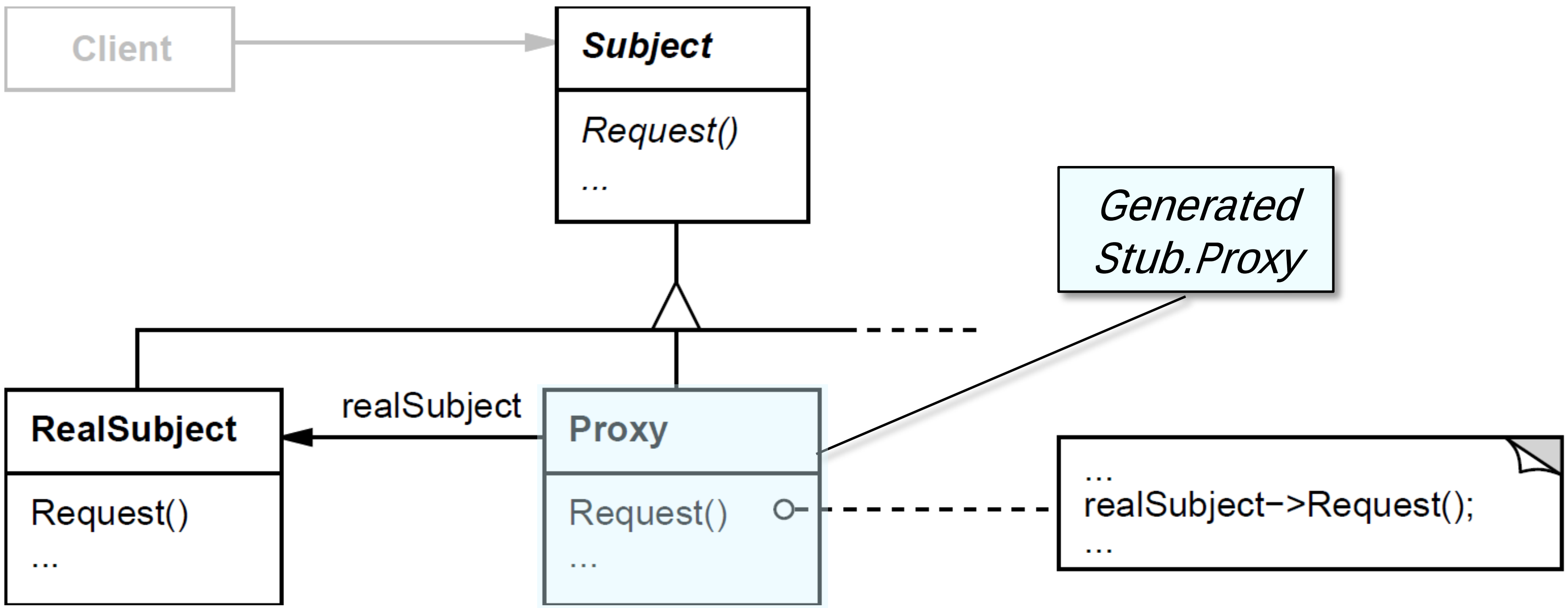
Proxy GoF Object Structural

Structure & Participants



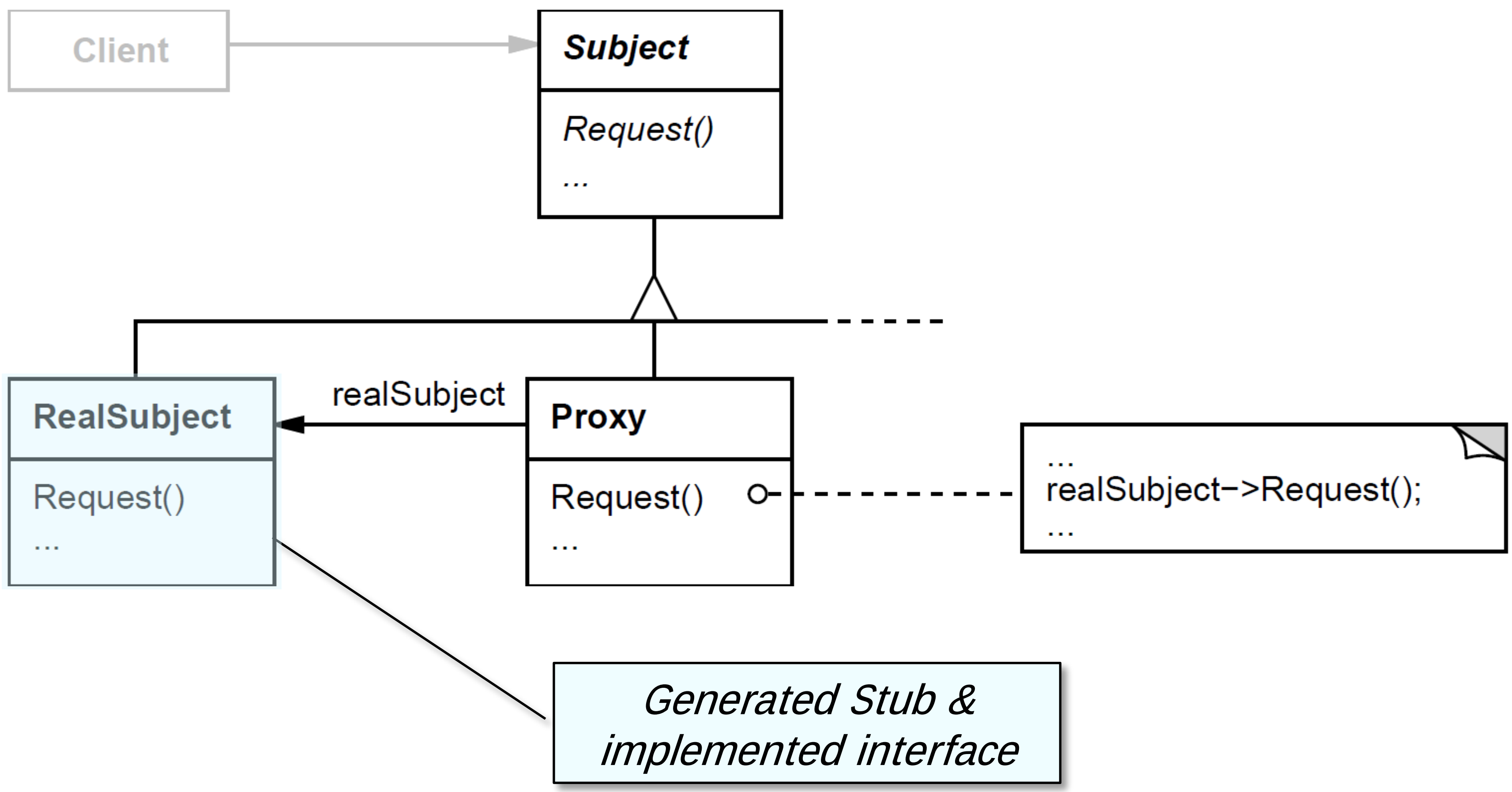
Proxy GoF Object Structural

Structure & Participants



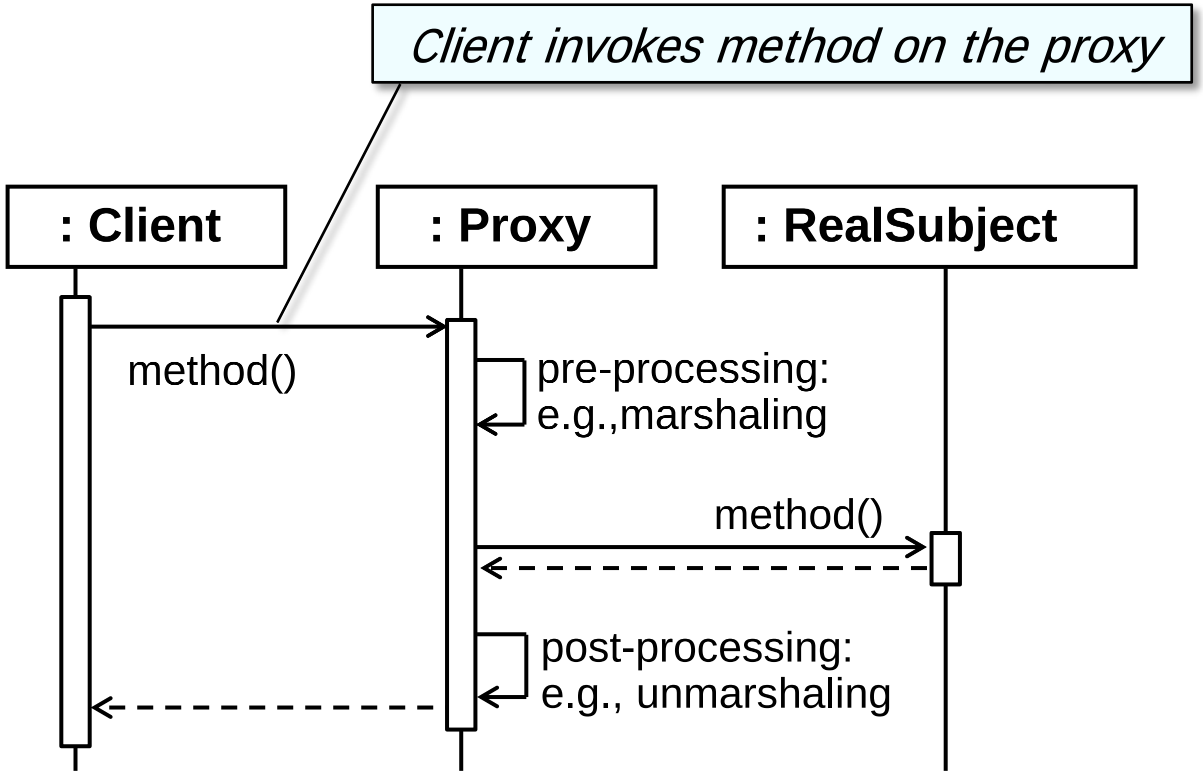
Proxy GoF Object Structural

Structure & Participants



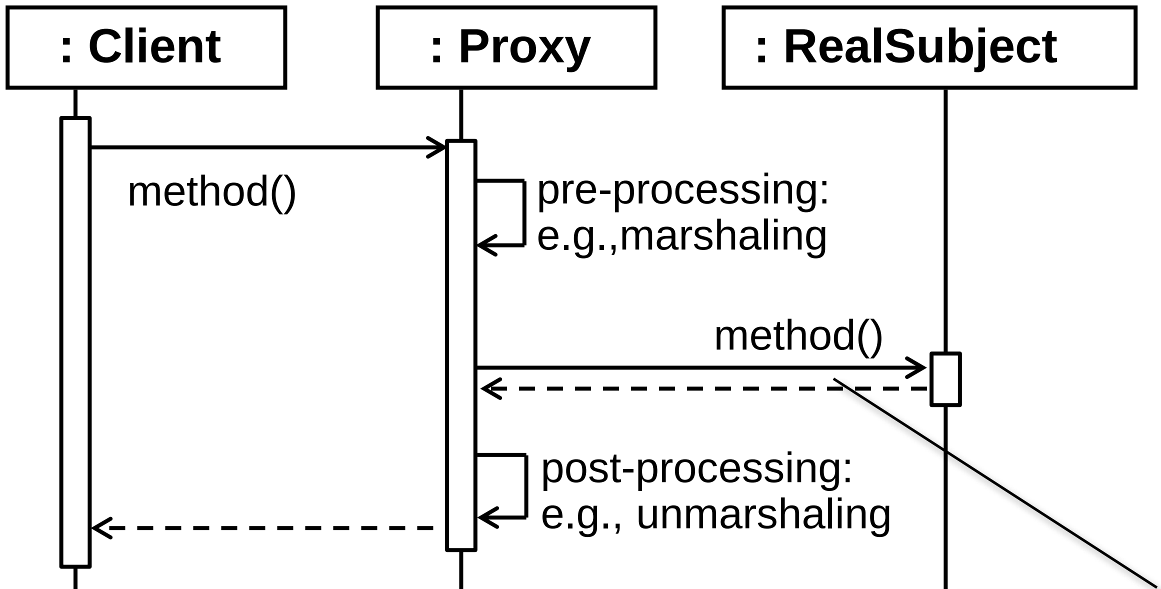
Proxy GoF Object Structural

Dynamics



Proxy GoF Object Structural

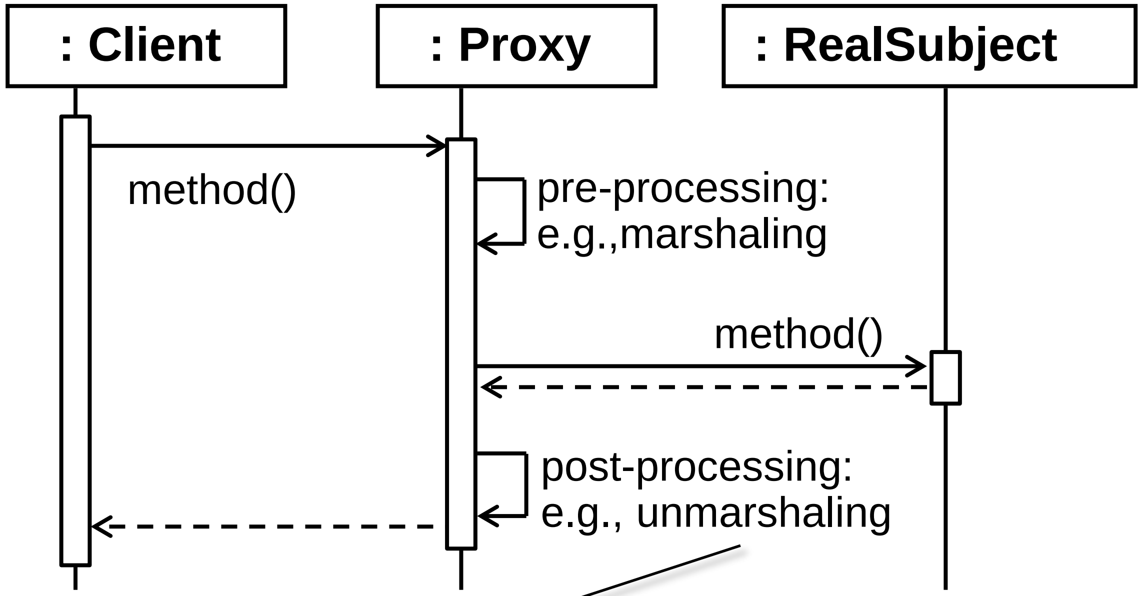
Dynamics



Proxy converts method to message & then invokes method on the real subject

Proxy GoF Object Structural

Dynamics

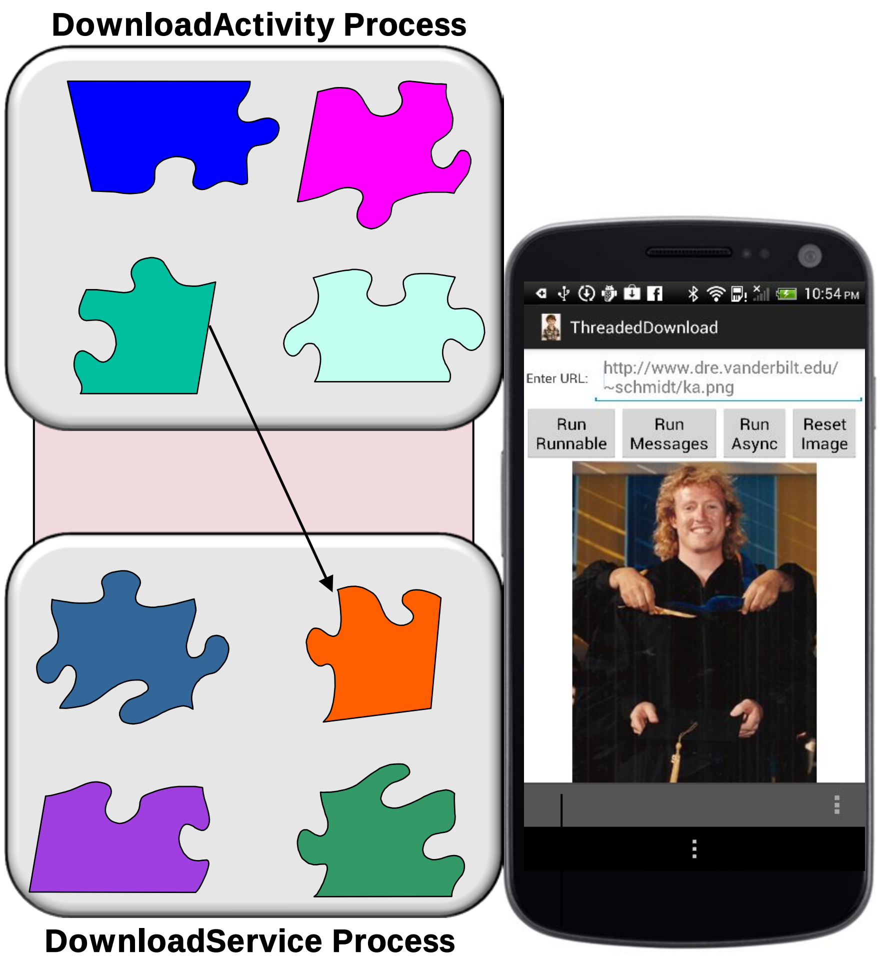


Proxy converts the return from the method call on the RealSubject back into results from the original method

Proxy GoF Object Structural

Consequences

+ Decoupling client from object location

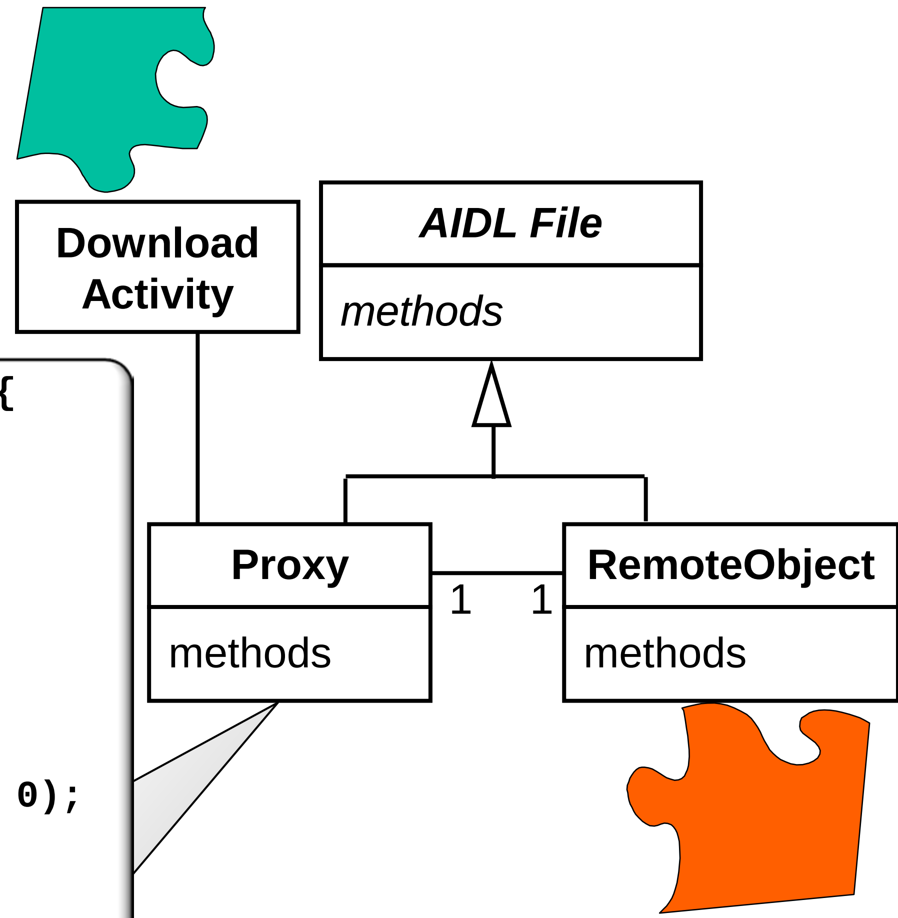


Proxy GoF Object Structural

Consequences

+ Decoupling client from object location

+ Simplify tedious & error-prone details

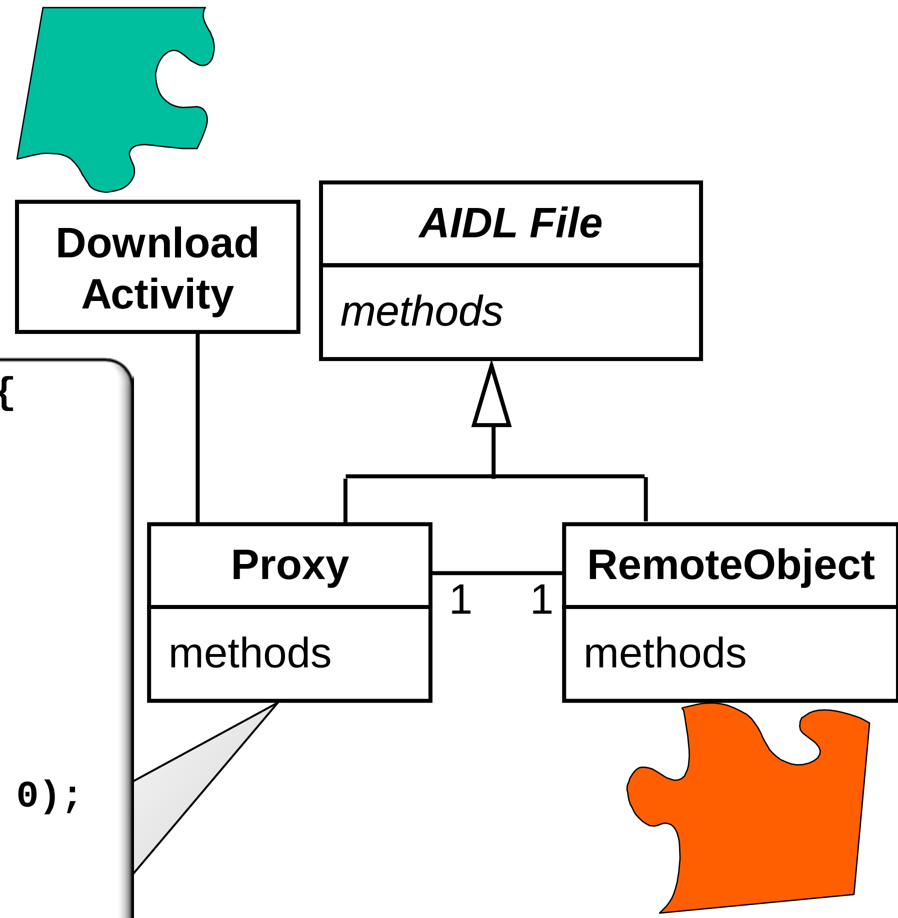



```
public String downloadImage(String uri) ... {
  android.os.Parcel _data =
    android.os.Parcel.obtain();
  android.os.Parcel _reply =
    android.os.Parcel.obtain();
  java.lang.String _result;
  _data.writeInterfaceToken(DESCRIPTOR);
  _data.writeString(url);
  mRemote.transact(Stub.
   TRANSACTION_downloadImage, _data, _reply, 0);
  _reply.readException();
  _result = _reply.readString();
  ...
  return _result;
}
```

Proxy GoF Object Structural

Consequences

– Additional overhead from indirection or inefficient proxy implementations



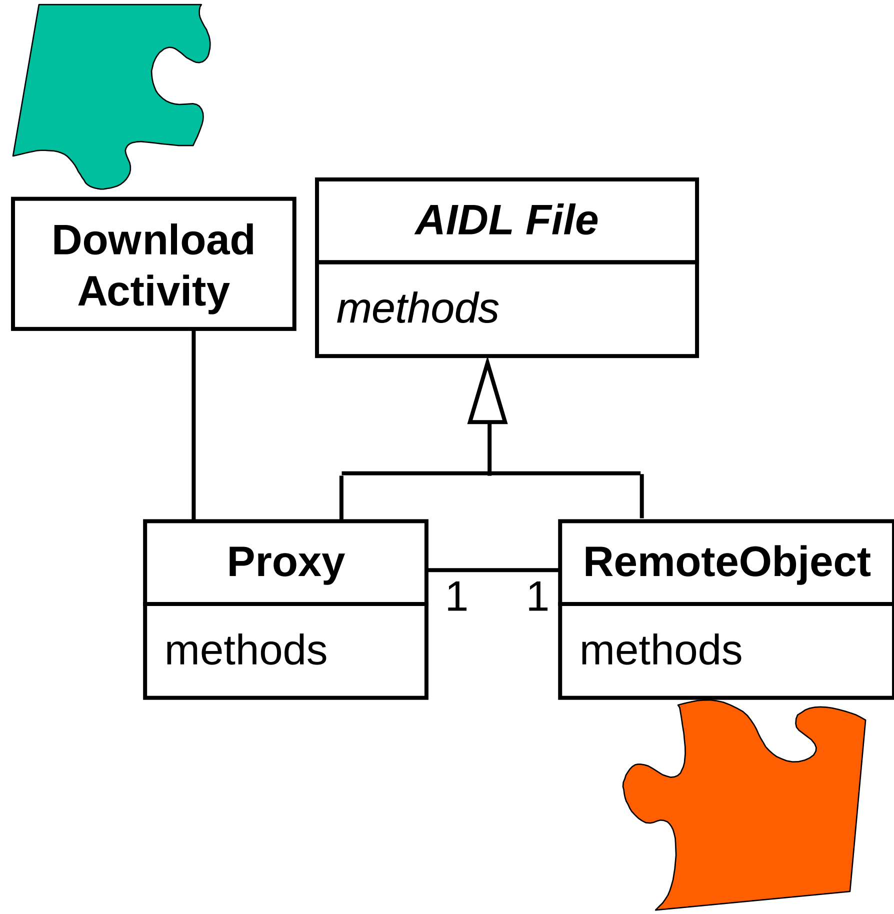

```
public String downloadImage(String uri) ... {
  android.os.Parcel _data =
    android.os.Parcel.obtain();
  android.os.Parcel _reply =
    android.os.Parcel.obtain();
  java.lang.String _result;
  _data.writeInterfaceToken(DESCRIPTOR);
  _data.writeString(url);
  mRemote.transact(Stub.
   TRANSACTION_downloadImage, _data, _reply, 0);
  _reply.readException();
  _result = _reply.readString();
  ...
  return _result;
}
```

Proxy GoF Object Structural

Consequences

– Additional overhead from indirection or inefficient proxy implementations

– May impose overly restrictive type system



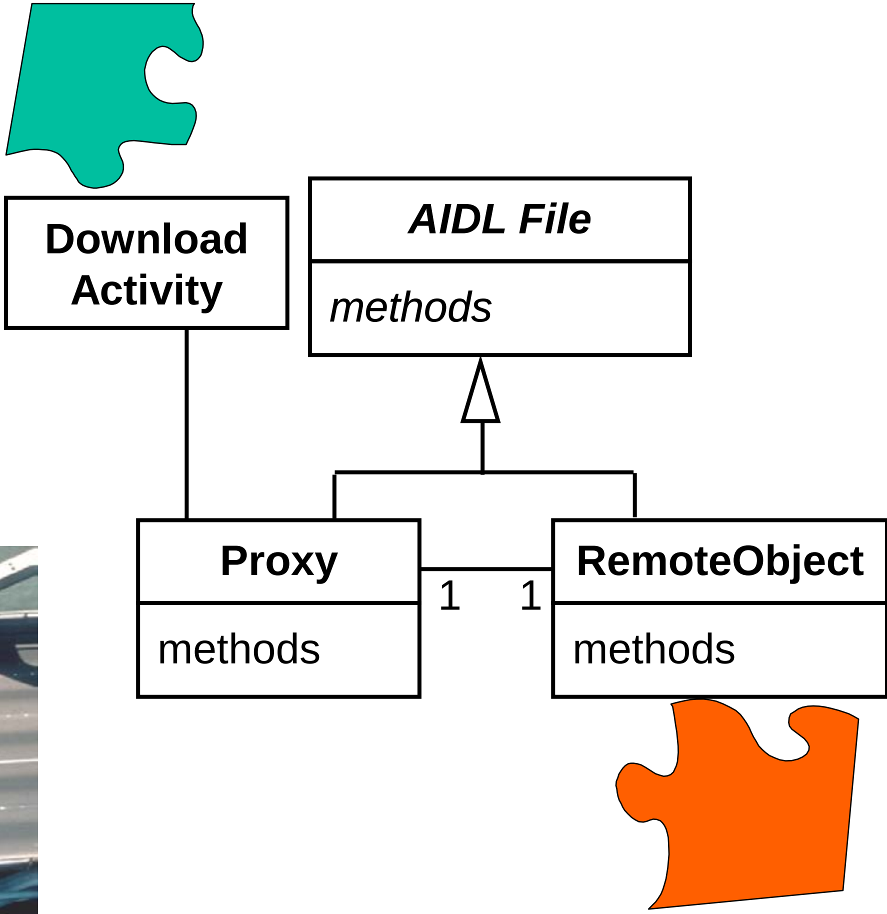
Proxy GoF Object Structural

Consequences

– Additional overhead from indirection or inefficient proxy implementations

– May impose overly restrictive type system

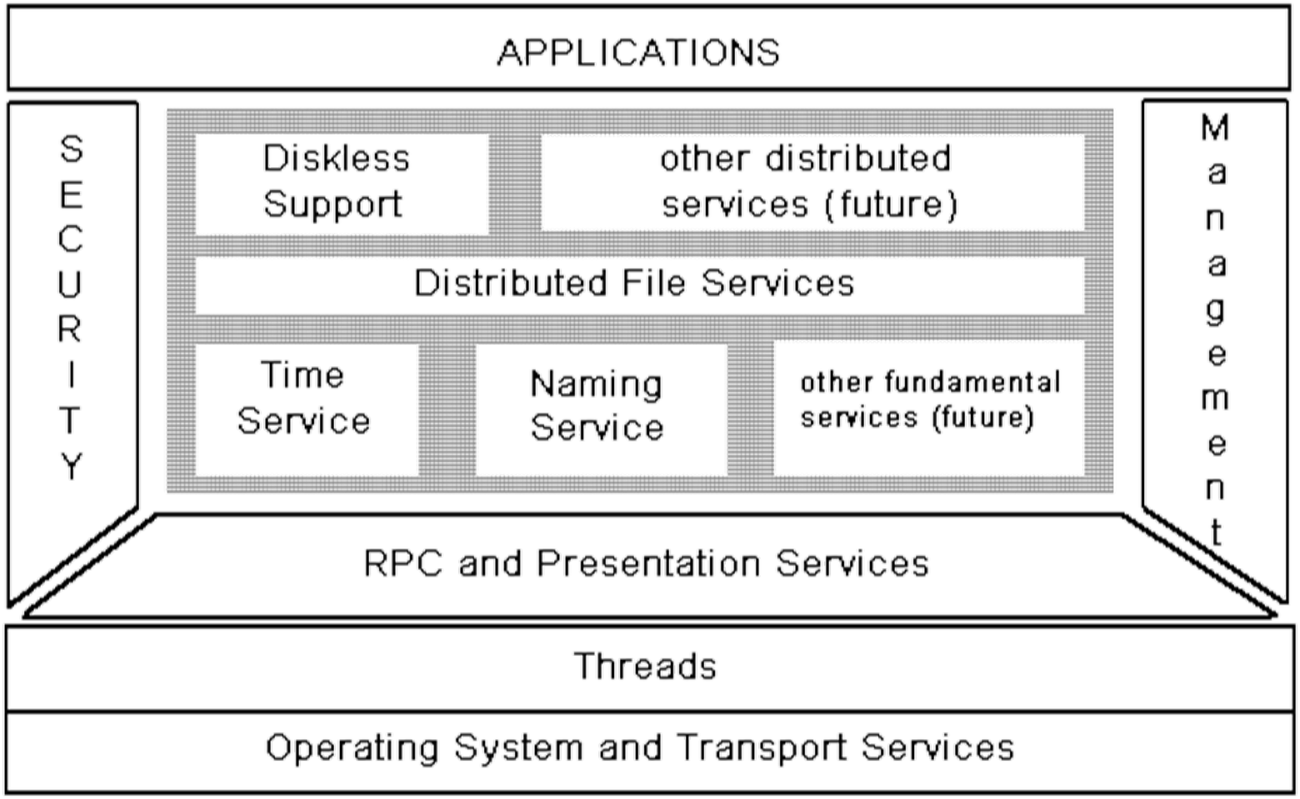
– It's not possible to entirely shield clients from problems with IPC across processes & networks



Proxy GoF Object Structural

Known Uses

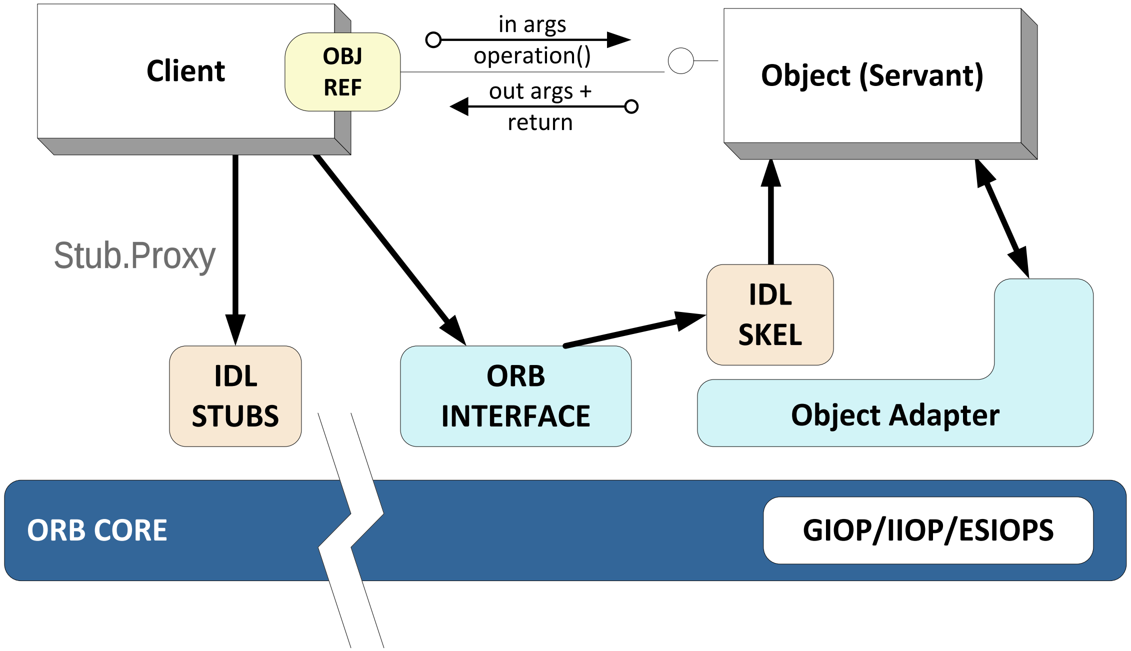
- Remote Procedure Call (RPC) middleware
  - e.g., ONC RPC & OSF Distributed Computing Environment (DCE)



Proxy GoF Object Structural

Known Uses

- Remote Procedure Call (RPC) middleware
- Distributed object computing middleware
  - e.g., Sun Java Remote Method Invocation (RMI) & OMG Common Object Request Broker Architecture (CORBA)

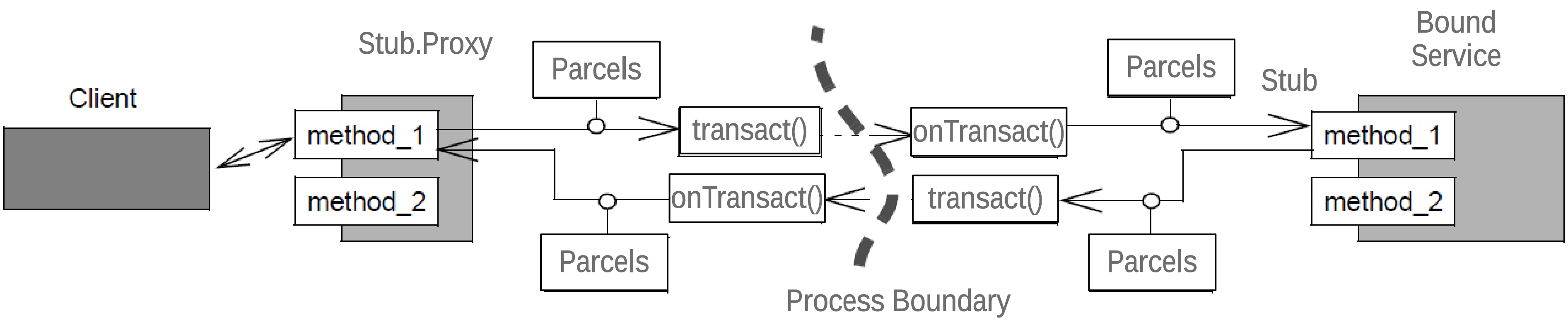


en.wikipedia.org/wiki/Object_request_broker

Proxy GoF Object Structural

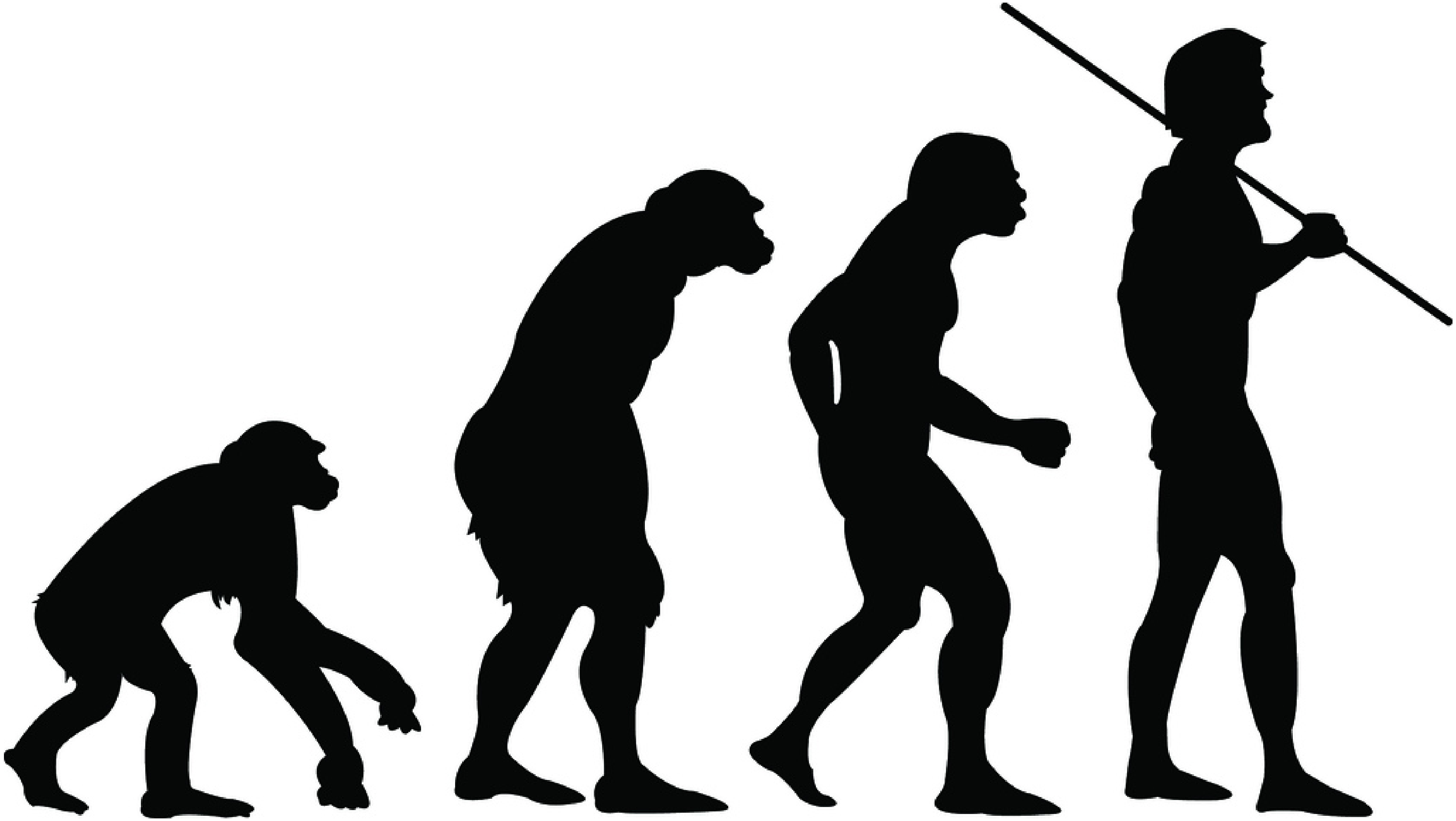
Known Uses

- Remote Procedure Call (RPC) middleware
- Distributed object computing middleware
- Local RPC middleware on smartphones
  - e.g., Android Binder



Evolving Patterns into Programming Languages

- The *Iterator* pattern illustrates a recurring theme throughout the history of computing: *useful patterns evolve into programming language features*



Evolving Patterns into Programming Languages

- Assembly language patterns in the early days of computing led to language features in FORTRAN & C
  - e.g., closed subroutines & control constructs, such as loop, if/else, & switch statements

```
for (int i = 0; i < MAX_SIZE; ++i)
  …

switch (tag_) {
case NUM: …

if (6 == 9)
  printf ("I don’t mind");
```

Evolving Patterns into Programming Languages

- Information hiding patterns done in assembly languages & C led to modularity features in Modula 2, Ada, C++, & Java
  - e.g., modules, packages, & access control sections

```
package com.example.expressiontree;

namespace std { …

class public class LeafNode extends ComponentNode {
   private int item;
   public int item()  { return item; }
   ...
```

Evolving Patterns into Programming Languages

- Preprocessor-style patterns in early C++ toolkits led to C++ templates & template meta-programming patterns are enhancing C++ templates

```
template <typename T>
class argv_iterator :
   public std::iterator <std::forward_iterator_tag,
                                  T> {
public:
   argv_iterator (void) {}
   argv_iterator (int argc,
                 char **argv,
                 int increment);
   ...
```

Evolving Patterns into Programming Languages

- Iterator patterns in C++ Standard Template Library (STL) led to built-in support for range-based for loops in C++11
  - e.g., languages like Java & C# also have built-in iterator support

Java for-each loop

```
for(ExpressionTree it : exprTree)
  doSomethingWithIterator(it);
```

C++11 range-based for loop

```
for (auto &it : expr_tree)
  do_something_with_iterator (it);
```

Not all patterns lend themselves to programming language support!

Android Services & Local IPC: The Proxy Pattern (Part 2)

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

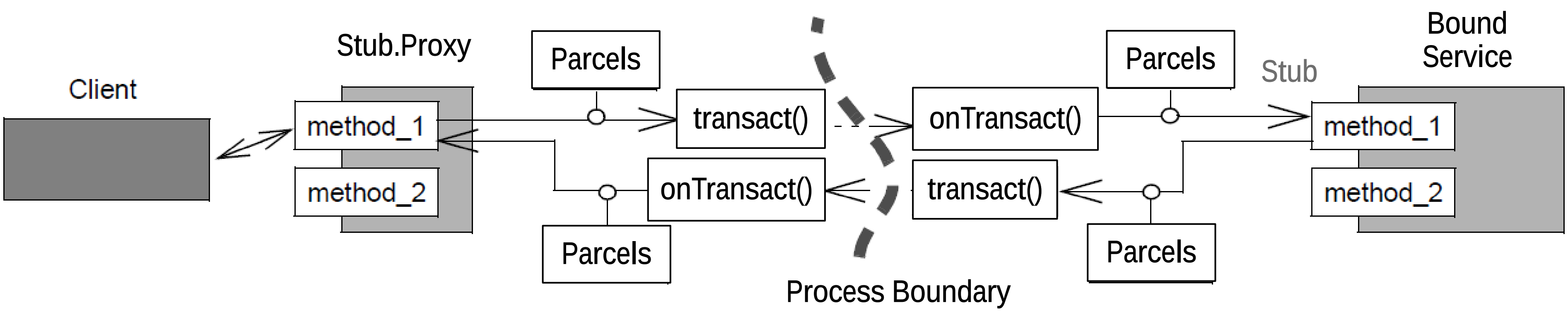
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Module

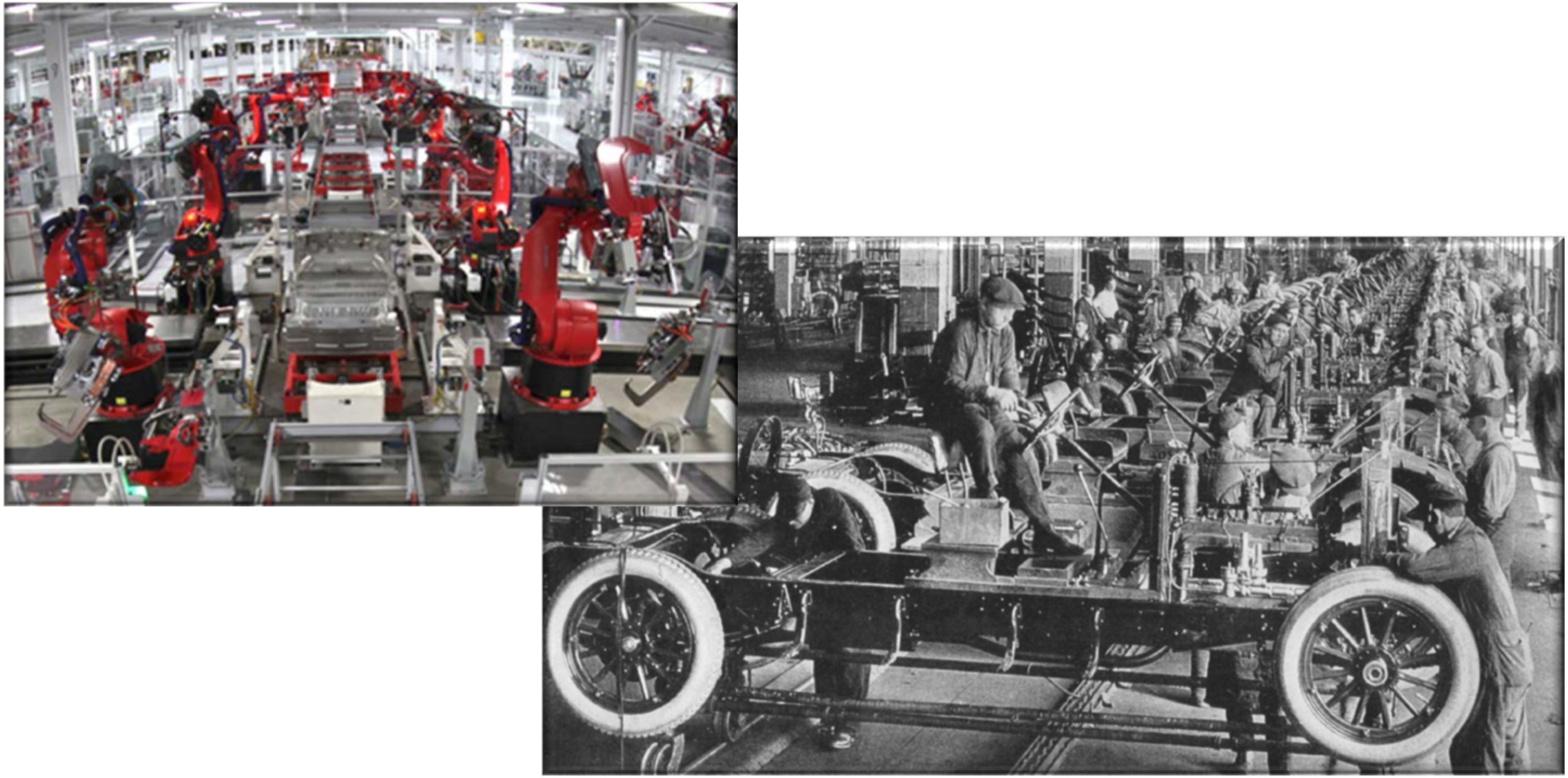
- Understand how the *Proxy* pattern is applied in Android



Proxy GoF Object Structural

Implementation

- Auto-generated vs. hand-crafted



Proxy GoF Object Structural

Implementation

- Auto-generated vs. hand-crafted
- A proxy can cache stable info about the subject to postpone accessing it remotely

Proxy GoF Object Structural

Implementation

- Auto-generated vs. hand-crafted
- A proxy can cache stable info about the subject to postpone accessing it remotely
- Overloading operator−> in C++

```
template <class TYPE> class ACE_TSS {
  TYPE *operator->() const {
    TYPE *tss_data = 0;
    if (!once_) {
      ACE_Guard<ACE_Thread_Mutex>
        g (keylock_);
      if (!once_) {
        ACE_OS::thr_keycreate
          (&key_, &cleanup_hook);
        once_ = true;
      }
    }
    ACE_OS::thr_getspecific
      (key_, (void **) &tss_data);
    if (tss_data == 0) {
      tss_data = new TYPE;
      ACE_OS::thr_setspecific
        (key_, (void *) tss_data);
    }
    return tss_data;
  }
```

www.dre.vanderbilt.edu/~schmidt/PDF/TSS-pattern.pdf

Proxy GoF Object Structural

Android Example in Java

```
public interface IDownload extends android.os.Iinterface {
  public static abstract class Stub extends android.os.Binder
                                    implements IDownload {
```

Local-side IPC implementation class

```
    public Stub() {
      this.attachInterface(this, DESCRIPTOR);
    }

    ...
```

Construct the stub & attach it to the interface

Proxy GoF Object Structural

Android Example in Java

```
public interface IDownload extends android.os.Iinterface {
  public static abstract class Stub extends android.os.Binder
                                    implements IDownload {



  public static IDownload asInterface(android.os.IBinder obj) {
      if ((obj==null)) return null;
      android.os.IInterface iin = (android.os.IInterface)
        obj.queryLocalInterface(DESCRIPTOR);
      if (((iin != null) && (iin instanceof IDownload)))
        return ((IDownload)iin);
      return new IDownload.Stub.Proxy(obj);
    }

    ...
```

Cast an IBinder object into an IDownload interface, generating a proxy if needed

Proxy GoF Object Structural

Android Example in Java

```
public interface IDownload extends android.os.Iinterface {
  public static abstract class Stub ... {
    private static class Proxy implements IDownload {
```

Used by a client to call a remote method

```
      private android.os.IBinder mRemote;

      Proxy(android.os.IBinder remote) {
        mRemote = remote;
      }
      ...
```

Cache Binder for subsequent use by Proxy

This code fragment has been simplified a bit to fit onto the slide

Proxy GoF Object Structural

Android Example in Java

```
public interface IDownload extends android.os.Iinterface {
  public static abstract class Stub ... {
    private static class Proxy implements IDownload {
      ...
```

Marshal the parameter, transmit to the remote object, & demarshal the result

```
      public String downloadImage(String uri) ... {
        android.os.Parcel _data = android.os.Parcel.obtain();
        android.os.Parcel _reply = android.os.Parcel.obtain();
        _data.writeString(url);
        mRemote.transact(Stub.TRANSACTION_downloadImage, _data,
                         _reply, 0);
        _reply.readException();
        java.lang.String _result = _reply.readString();
        ...
        return _result;
   ...
```

This code fragment has been simplified a bit to fit onto the slide

Proxy GoF Object Structural

Android Example in Java

```
public interface IDownload extends android.os.Iinterface {
  public static abstract class Stub extends android.os.Binder
                                    implements IDownload {
```

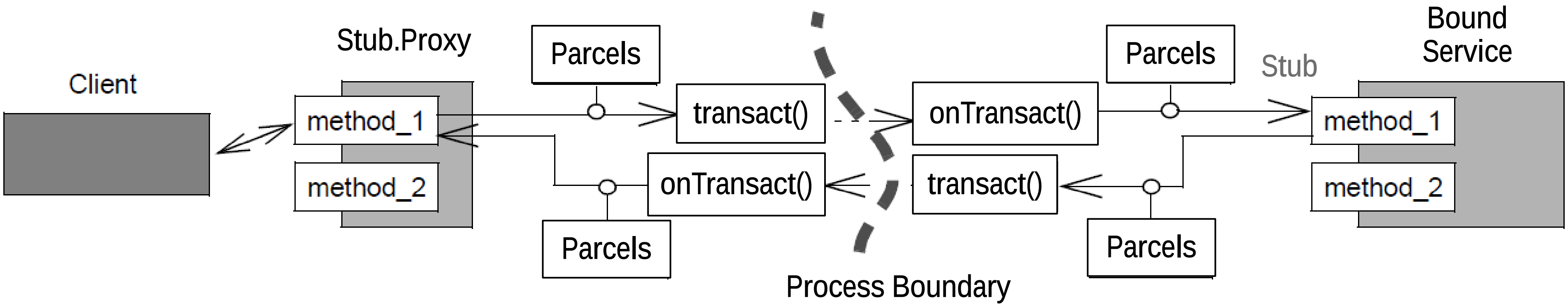
This method is dispatched by Binder RPC to trigger a callback on our downloadImage()

```
  public boolean onTransact(int code, android.os.Parcel data,
                     android.os.Parcel reply, int flags) ... {
    switch (code) {
    case TRANSACTION_downloadImage:
      data.enforceInterface(DESCRIPTOR);
      java.lang.String _arg0 = data.readString();
      java.lang.String _result = this.downloadImage(_arg0);
      reply.writeNoException();
      reply.writeString(_result);
      return true;
      ...
```

Demarshal the parameter, dispatch the upcall, & marshal the result

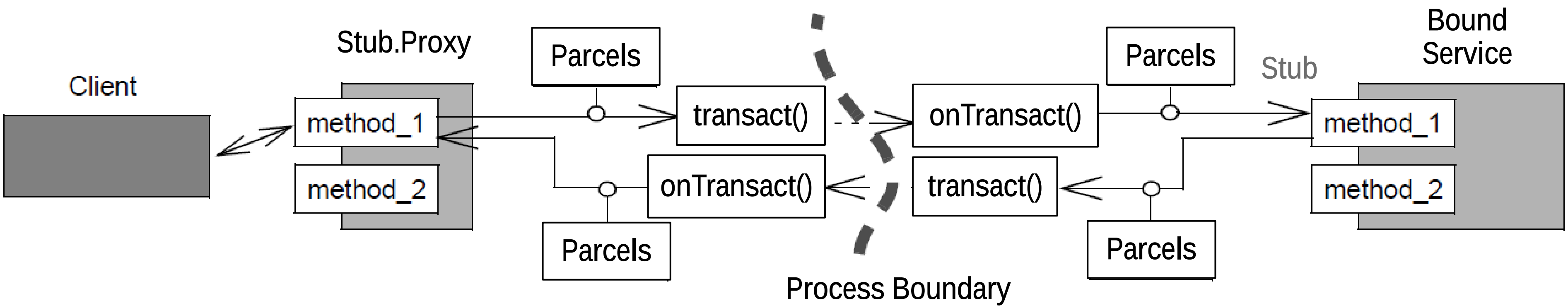
This code fragment has been simplified a bit to fit onto the slide

Summary



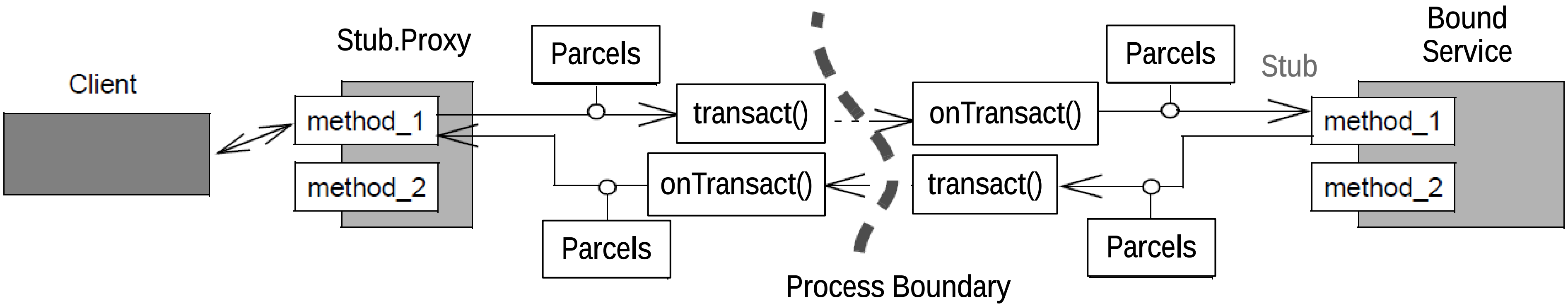
- The Android generated AIDL proxies implement the *Proxy* pattern

Summary



- The Android generated AIDL proxies implement the *Proxy* pattern
- Proxies support a remote method invocation style of IPC
  - As a result, there is no API difference between a call to a local or a remote component, which enhances location-independent communication within an Android App

Summary



- The Android generated AIDL proxies implement the *Proxy* pattern
- Proxies support a remote method invocation style of IPC
- In addition, a proxy can shield its clients from changes in the represented component's 'real' interfaces, which avoids rippling effects in case of component evolution

Android Services & Local IPC: The Broker Pattern (Part 1)

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

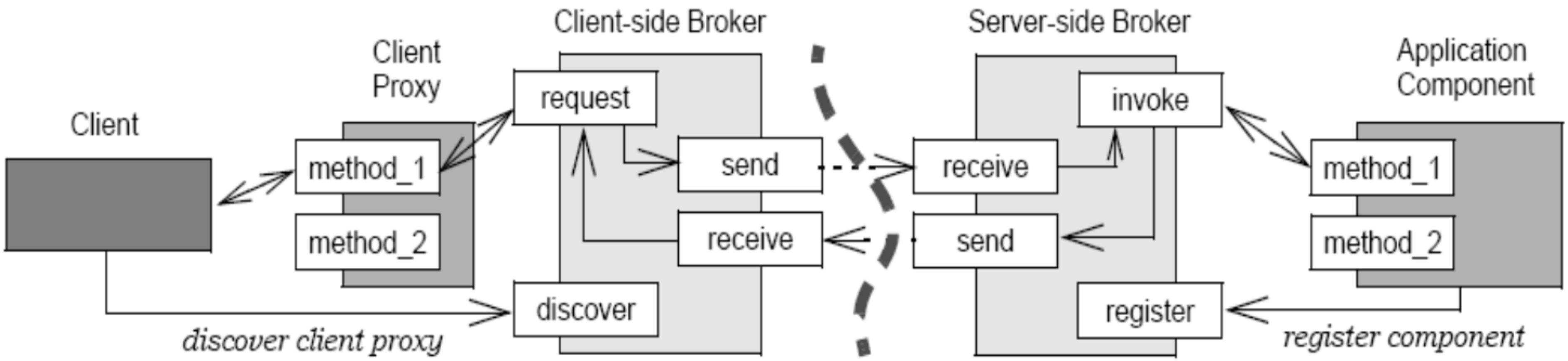
Professor of Computer Science

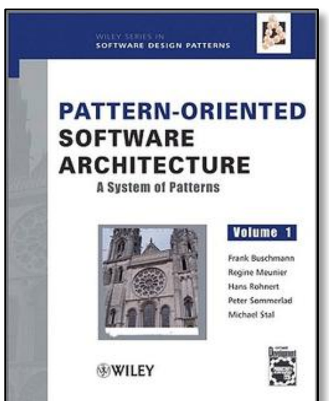
Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Module

- Understand the *Broker* pattern






Broker Revisited

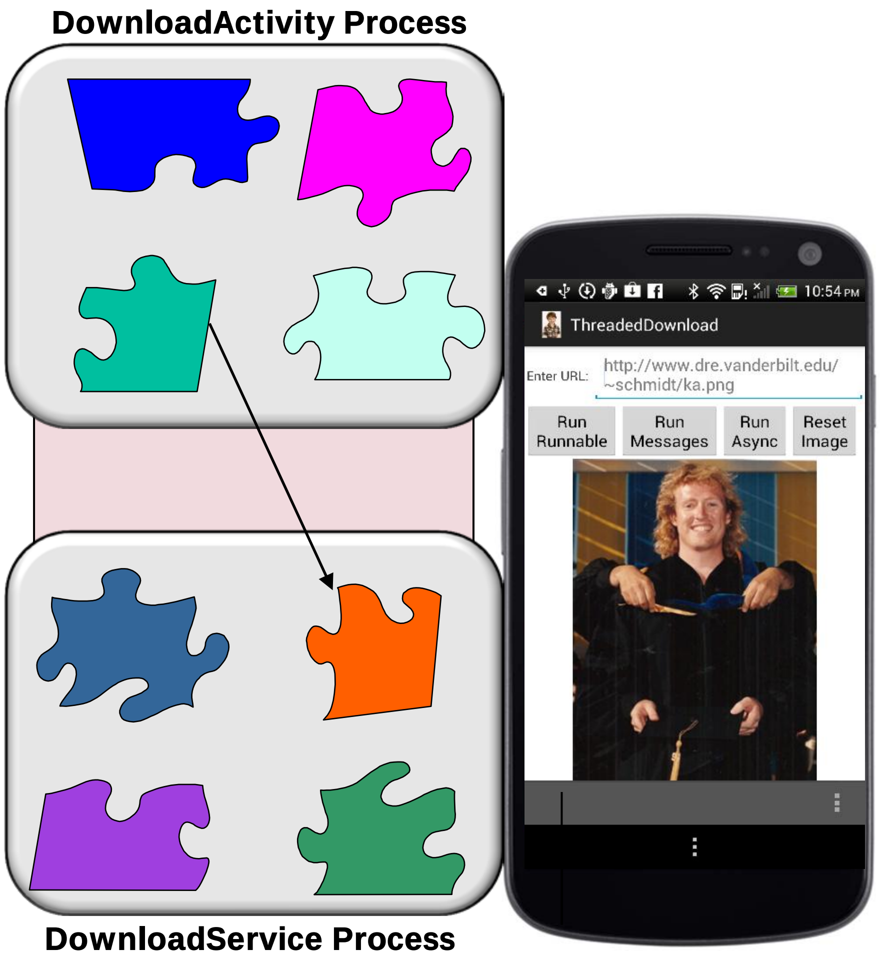
Corporate Technology, Siemens AG, Munich, Germany

See www.kircher-schwanninger.de/michael/publications/BrokerRevisited.pdf

Challenge: Isolating Communication Concerns

Context

- A system with multiple (potentially) remote objects that interact synchronously or asynchronously

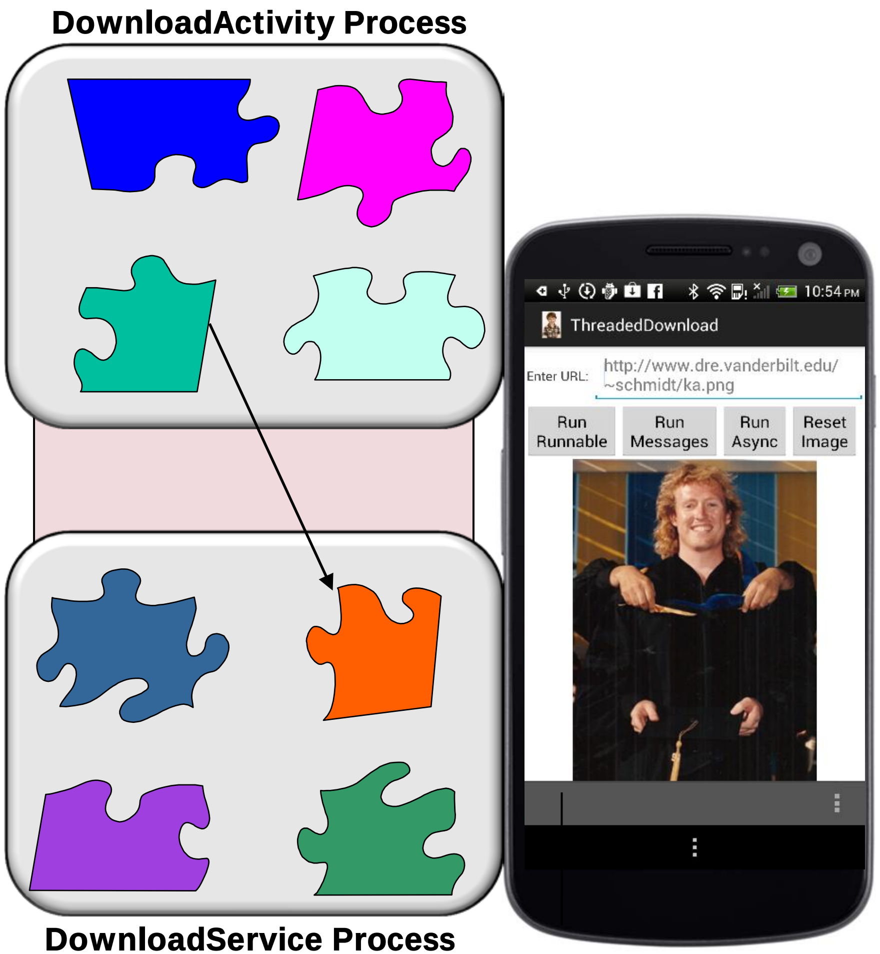


Android's Binder provides a high-performance IPC mechanism

Challenge: Isolating Communication Concerns

Problems

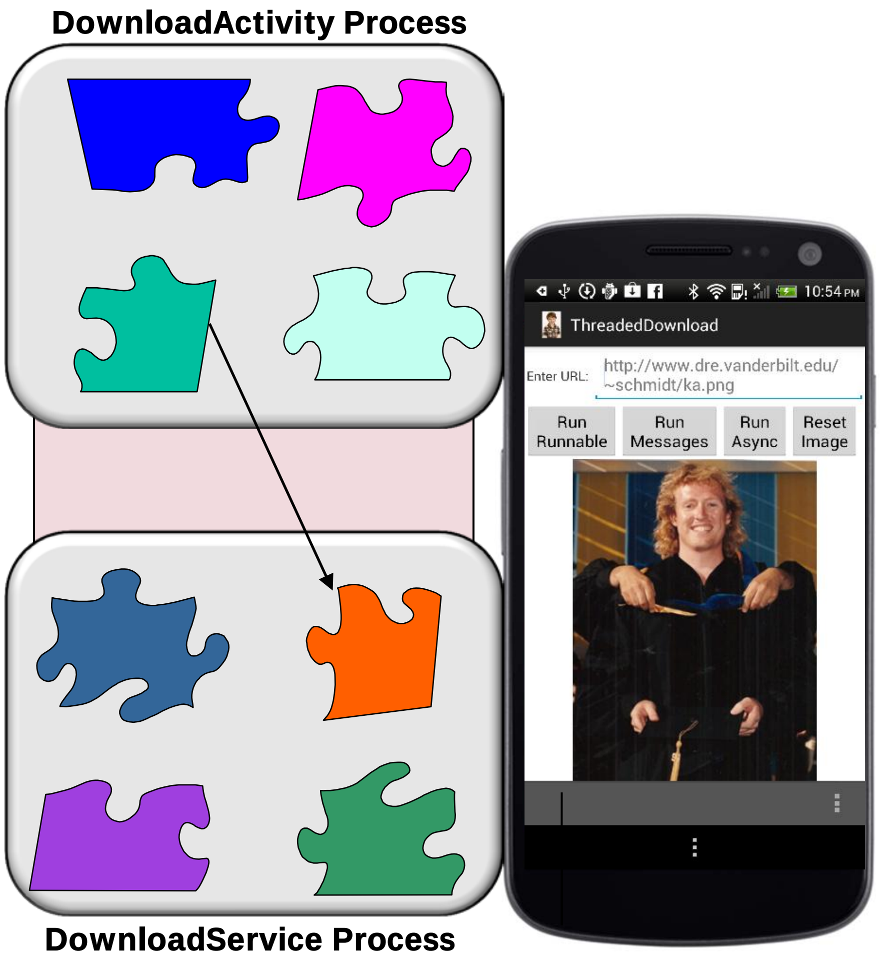
- App developers shouldn't need to handle
  - Low-level message passing, which is fraught with accidental complexity



Challenge: Isolating Communication Concerns

Problems

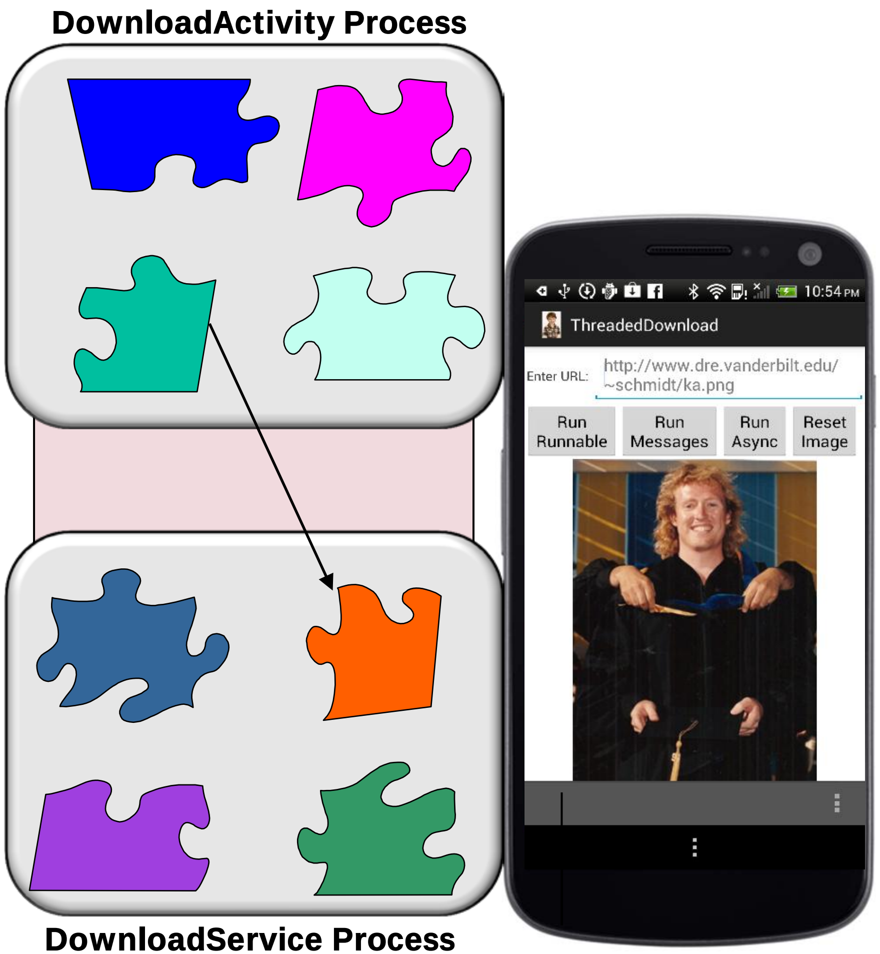
- App developers shouldn't need to handle
  - Low-level message passing, which is fraught with accidental complexity
  - Networked computing diversity
    - e.g., heterogeneous languages, operating systems, protocols, hareware, etc.



Challenge: Isolating Communication Concerns

Problems

- App developers shouldn't need to handle
  - Low-level message passing, which is fraught with accidental complexity
  - Networked computing diversity
  - Inherent complexities of communication
    - e.g., partial failures, security mechanisms, latency, etc.

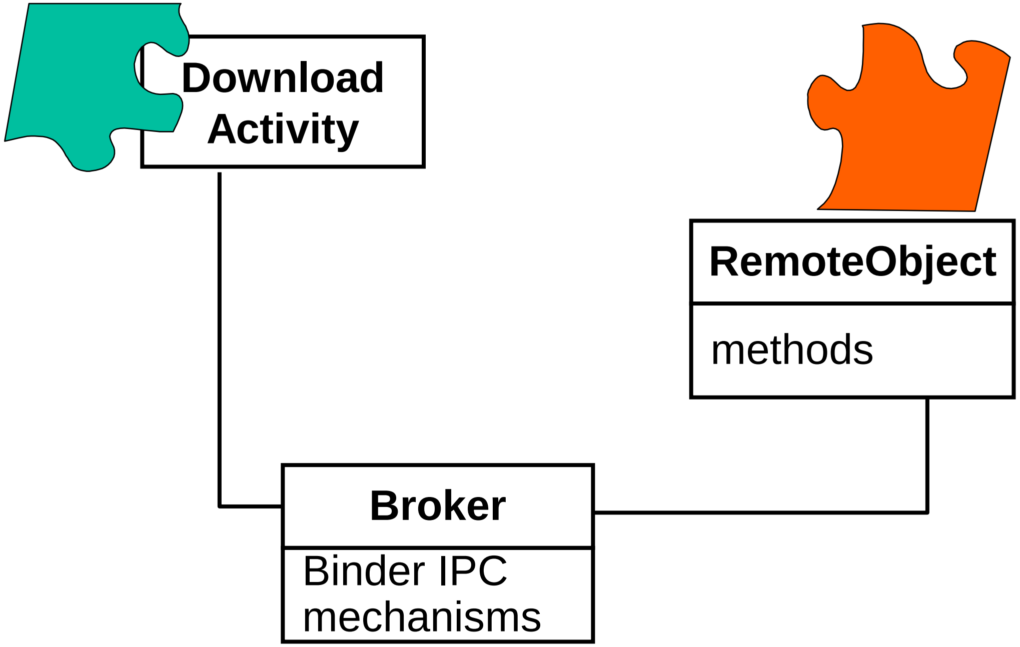


See www.dre.vanderbilt.edu/~schmidt/comm-foreword.html for more info

Use a Broker to Handle Communication Concerns

Solution

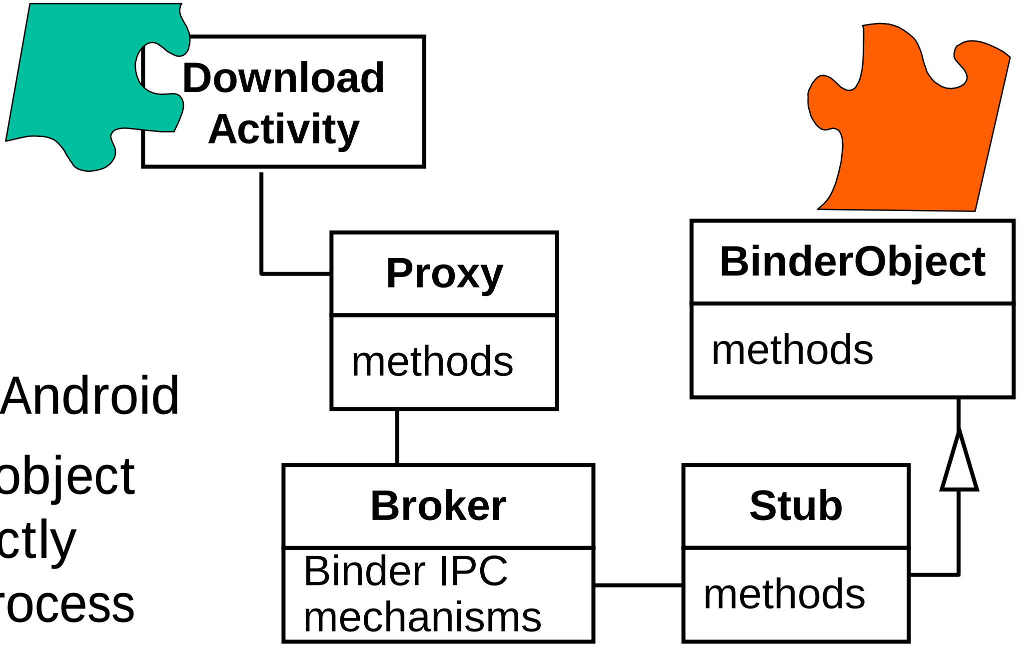
- Separate system communication functionality from app functionality by providing a *broker* that isolates communication-related concerns



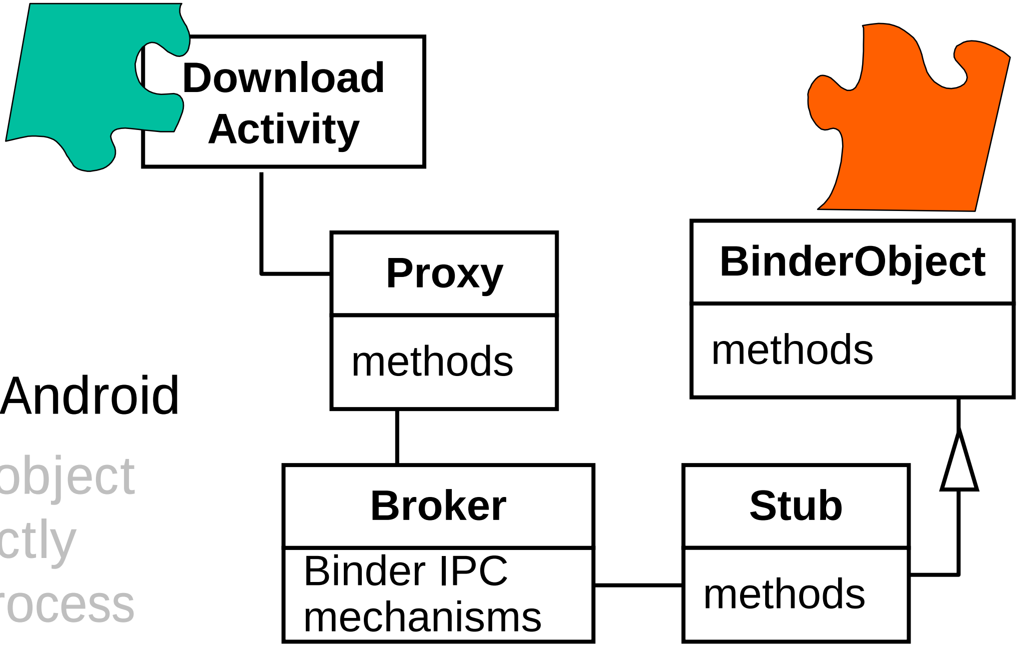
Use a Broker to Handle Communication Concerns

Solution

- Separate system communication functionality from app functionality by providing a *broker* that isolates communication-related concerns
- e.g., one way to implement this in Android
  - A Service implements an Binder object that a client can't accessible directly since it may reside in different process



Use a Broker to Handle Communication Concerns



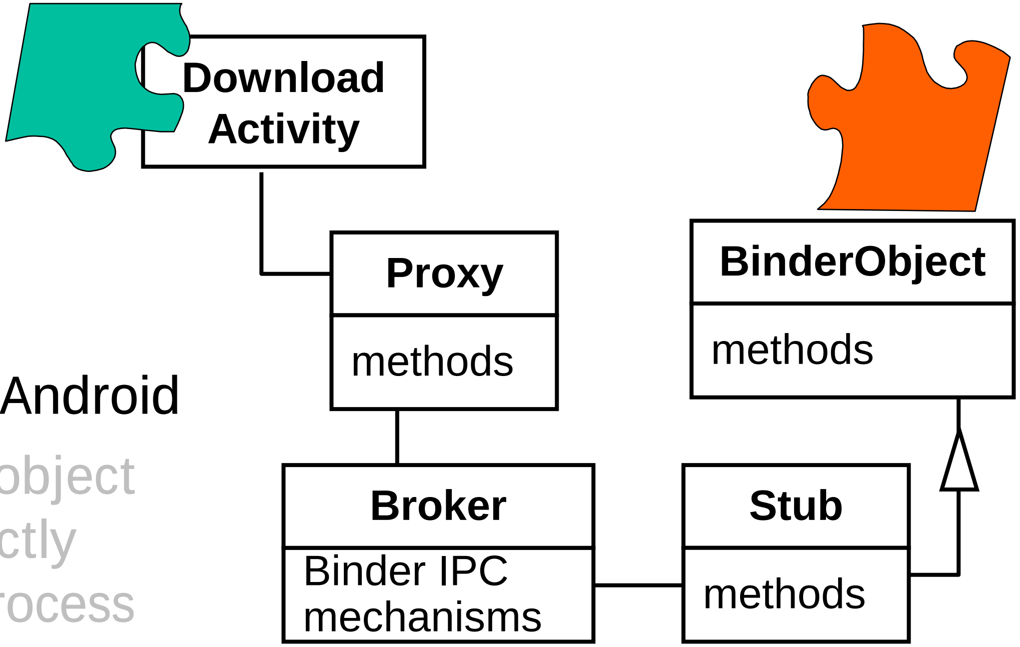
Solution

- Separate system communication functionality from app functionality by providing a *broker* that isolates communication-related concerns
- e.g., one way to implement this in Android
  - A Service implements an Binder object that a client can't accessible directly since it may reside in different process
  - Clients call a method on the proxy, which uses the Android Binder IPC mechanism (broker) to communicate with the object across process boundaries

Use a Broker to Handle Communication Concerns

Solution

- Separate system communication functionality from app functionality by providing a *broker* that isolates communication-related concerns
- e.g., one way to implement this in Android
  - A Service implements an Binder object that a client can't accessible directly since it may reside in different process
  - Clients call a method on the proxy, which uses the Android Binder IPC mechanism (broker) to communicate with the object across process boundaries
  - The Binder IPC mechanisms use a stub to upcall a method to the object

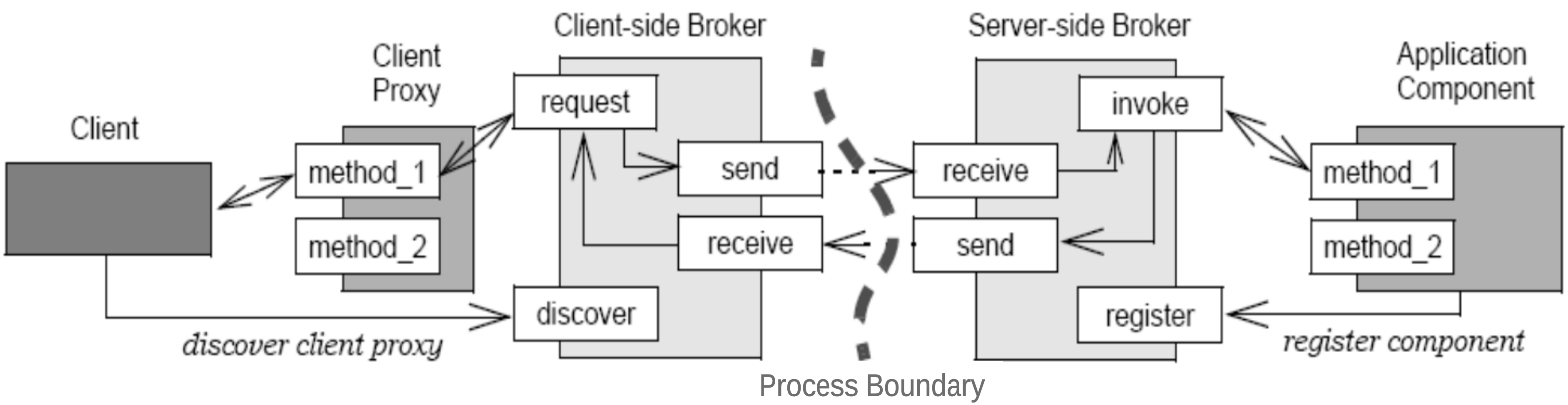


Android Binder uses *Broker* & *Proxy* to support sync & async communication

Broker POSA1 Architectural Pattern

Intent

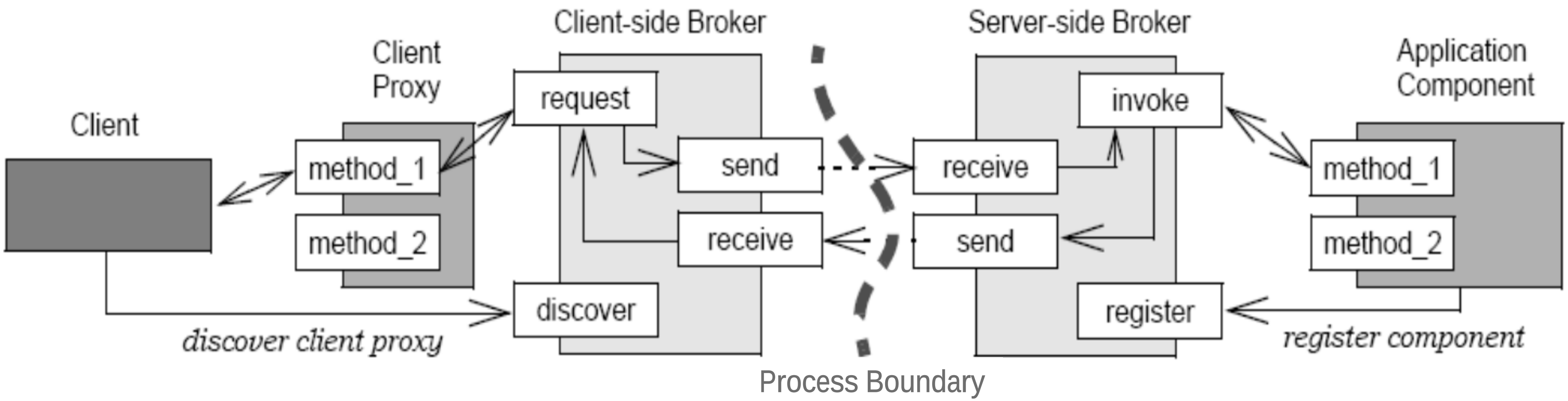
- Connect clients with remote objects by mediating invocations from clients to remote objects, while encapsulating the details of local and/or remote IPC



Broker POSA1 Architectural Pattern

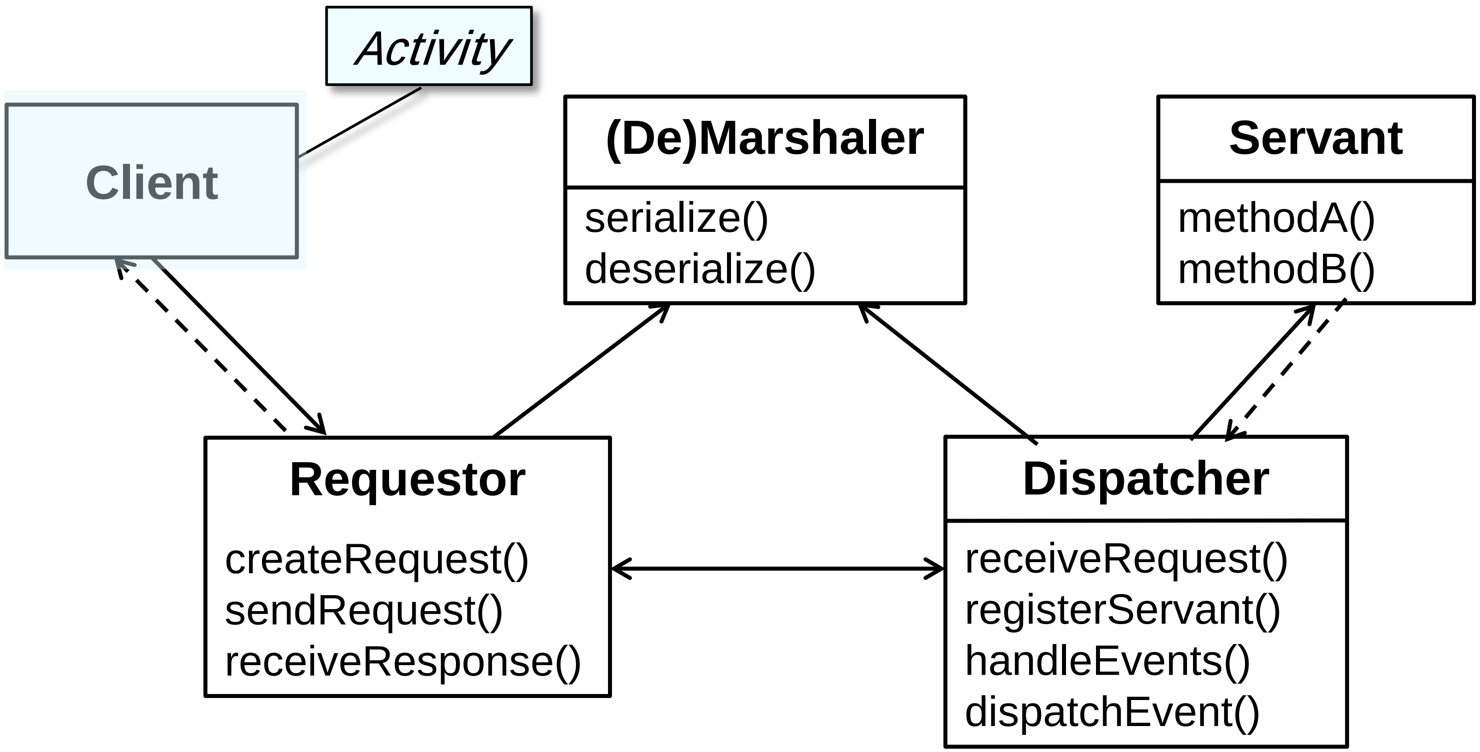
Applicability

- When apps need reusable capabilities that
  - Support (potentially) remote communication in a location transparent manner
  - Detect/handle faults & manage end-to-end QoS
  - Encapsulate low-level systems programming details
    - e.g., memory management, connection management, data transfer, concurrency, synchronization, etc.



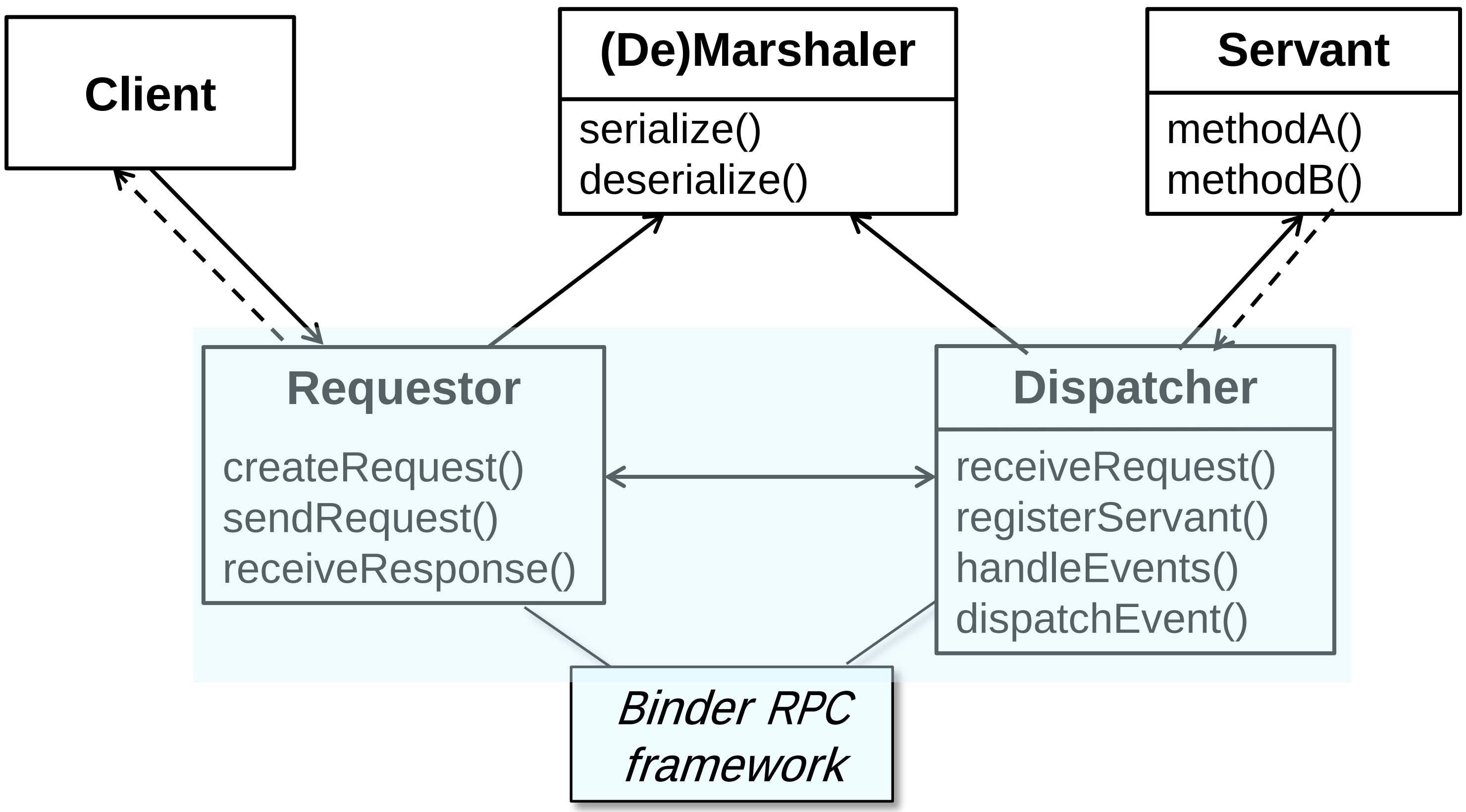
Broker POSA1 Architectural Pattern

Structure & Participants



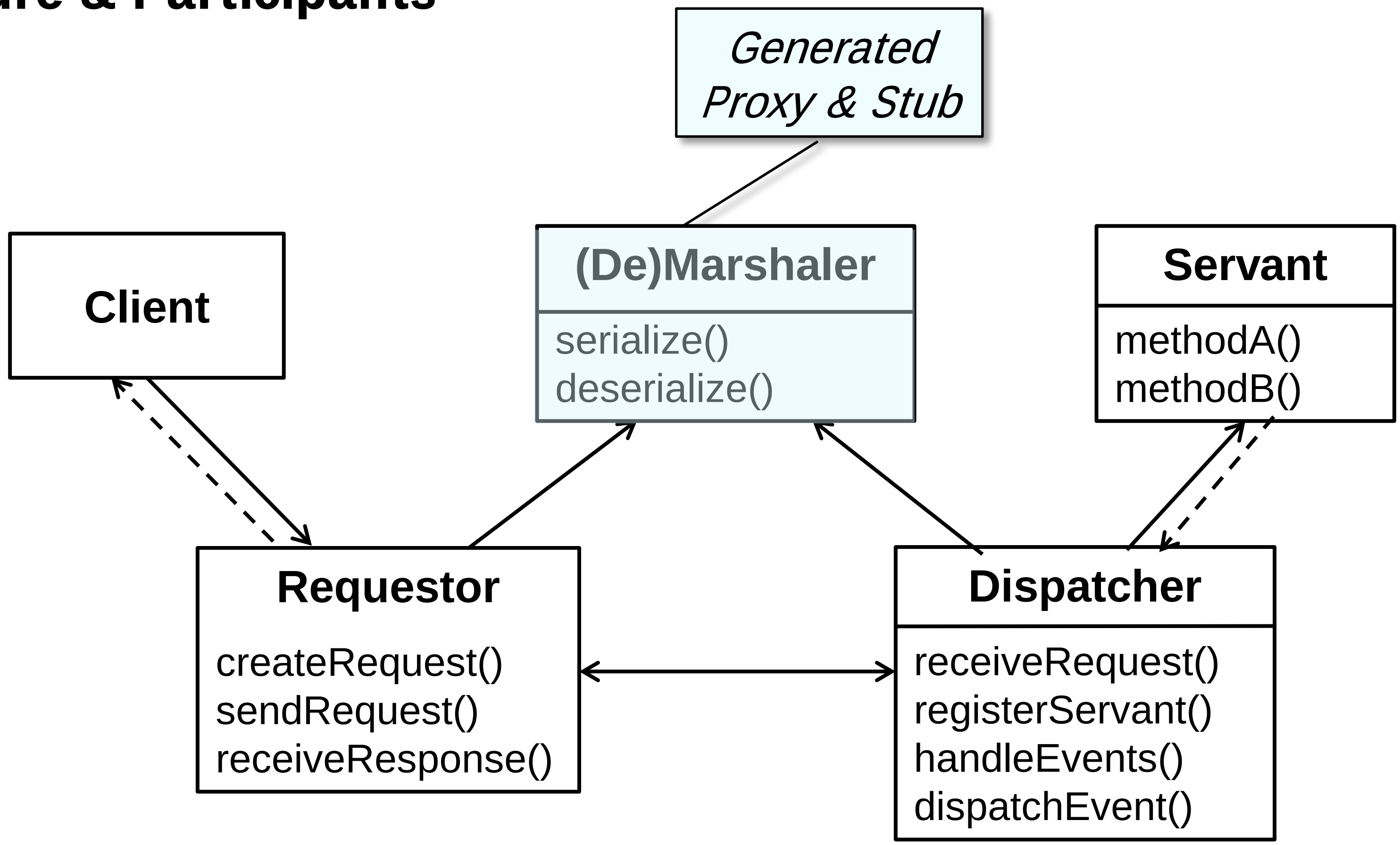
Broker POSA1 Architectural Pattern

Structure & Participants



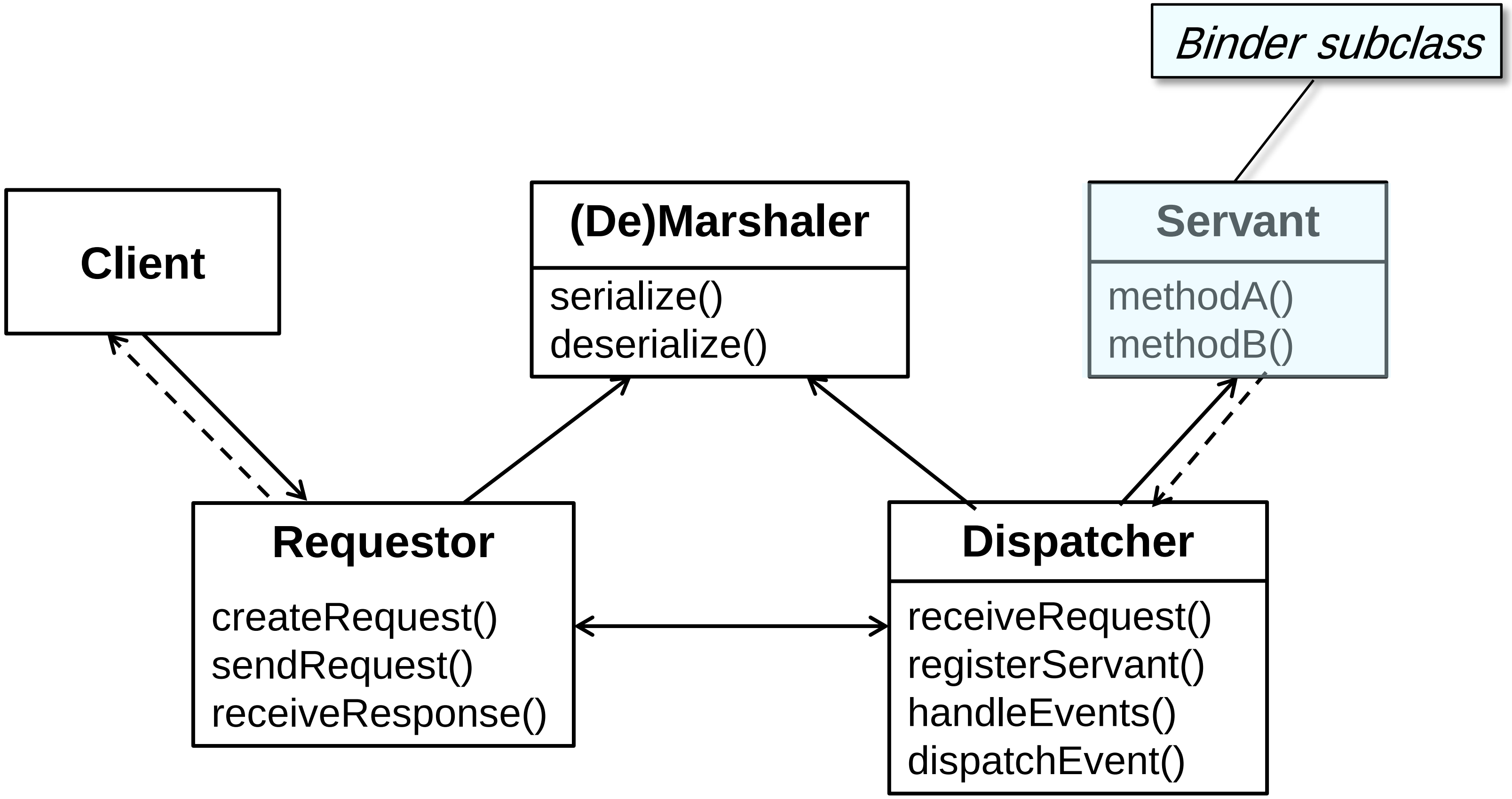
Broker POSA1 Architectural Pattern

Structure & Participants



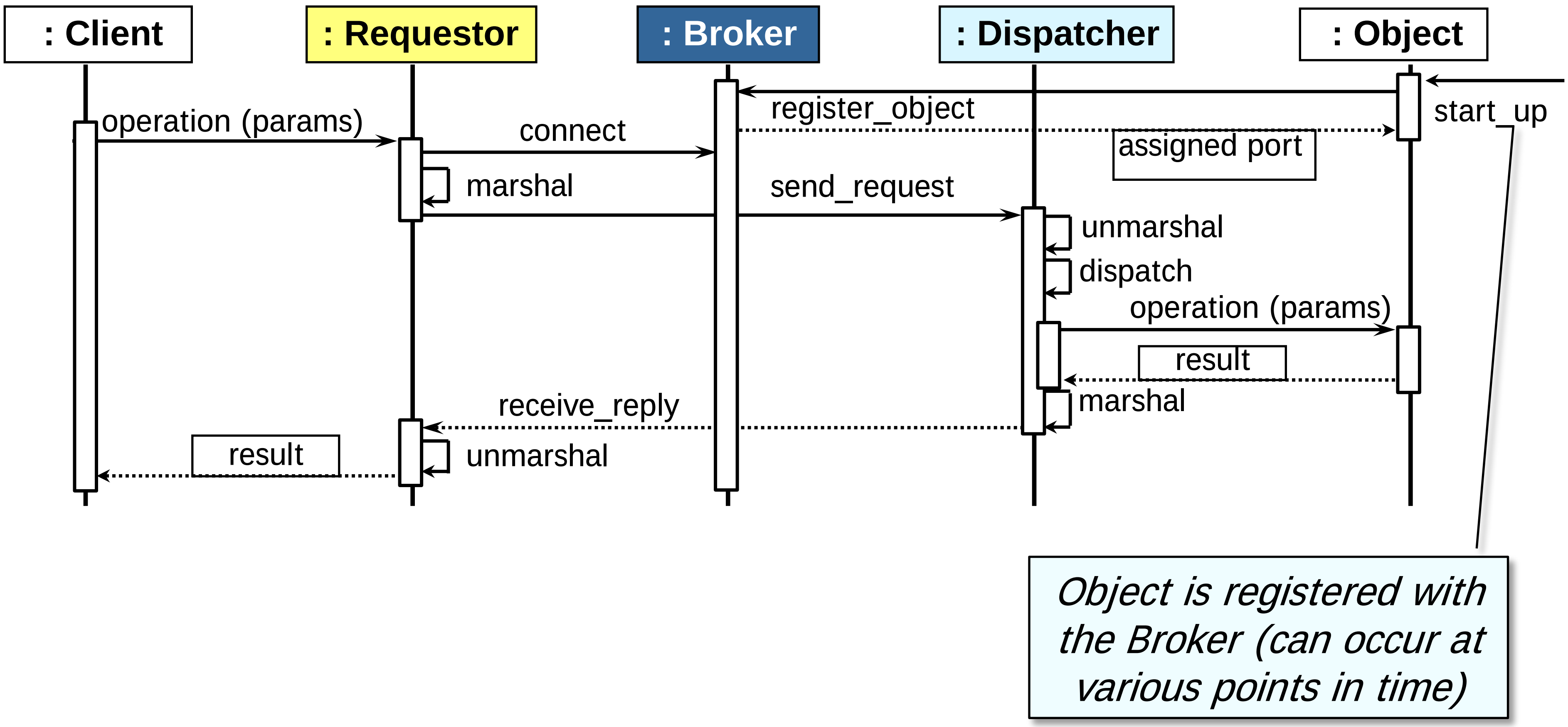
Broker POSA1 Architectural Pattern

Structure & Participants



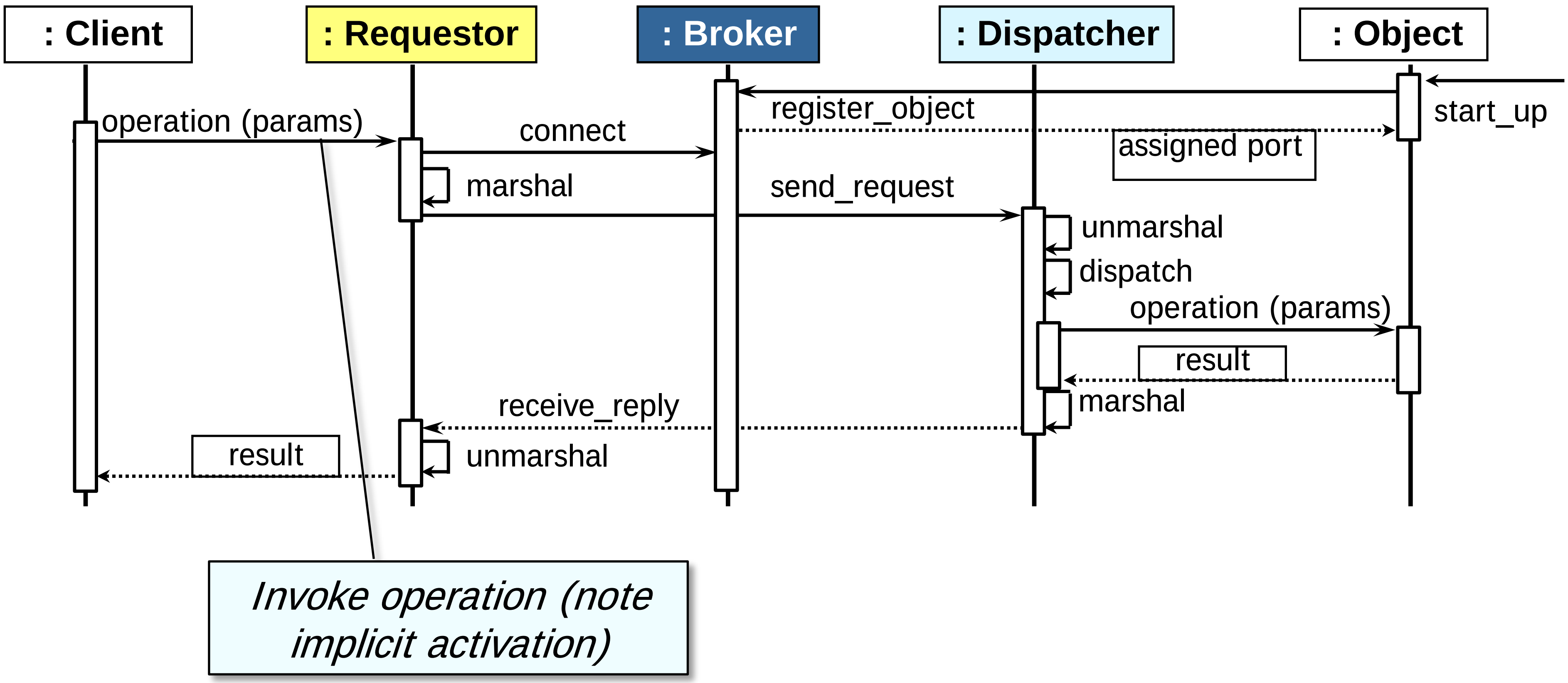
Broker POSA1 Architectural Pattern

Dynamics



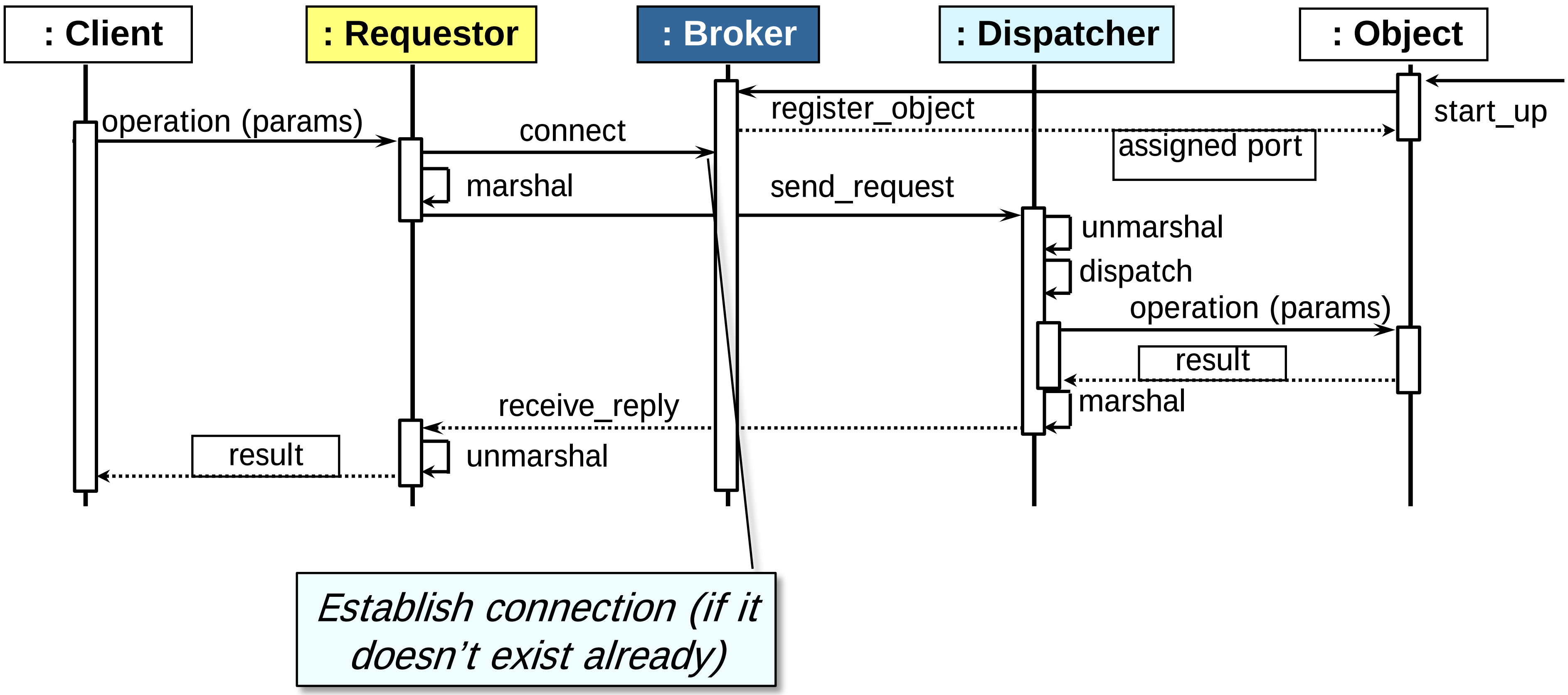
Broker POSA1 Architectural Pattern

Dynamics



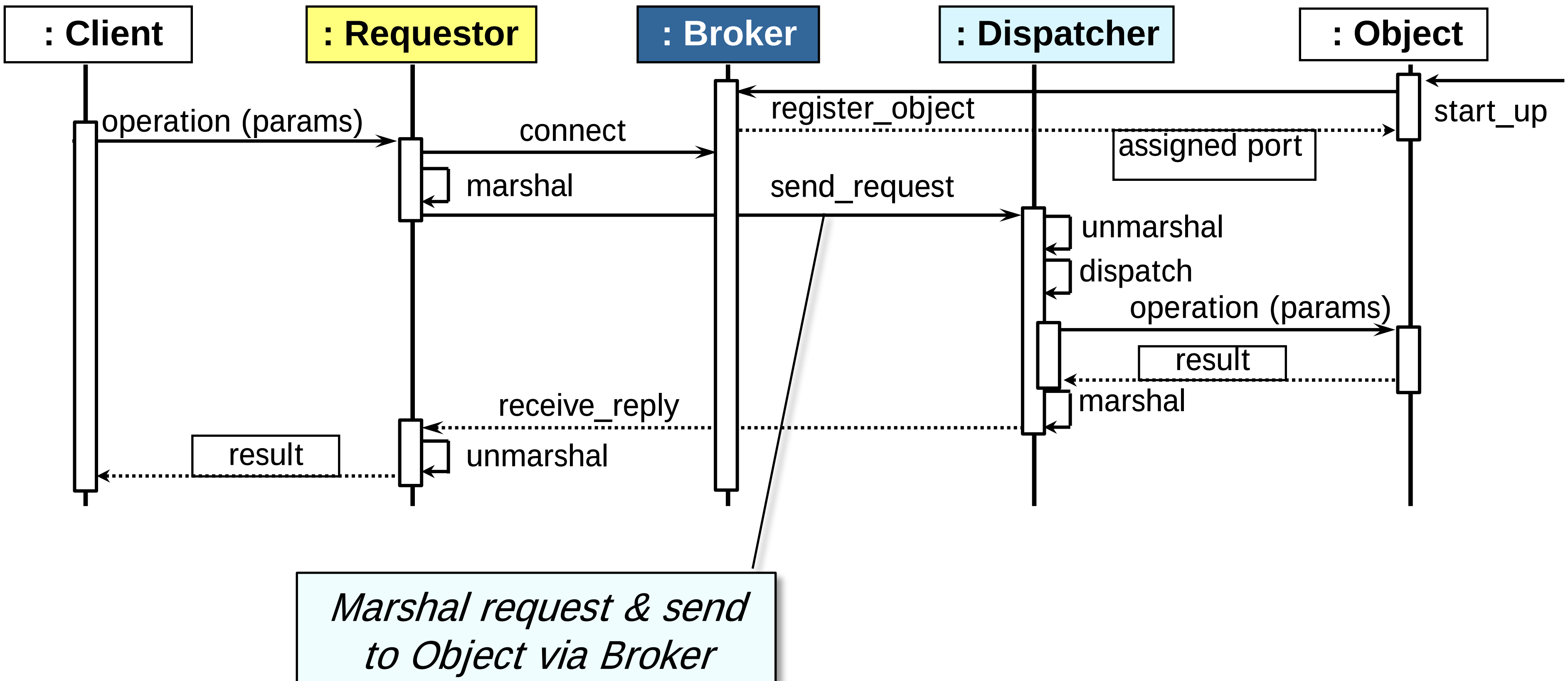
Broker POSA1 Architectural Pattern

Dynamics



Broker POSA1 Architectural Pattern

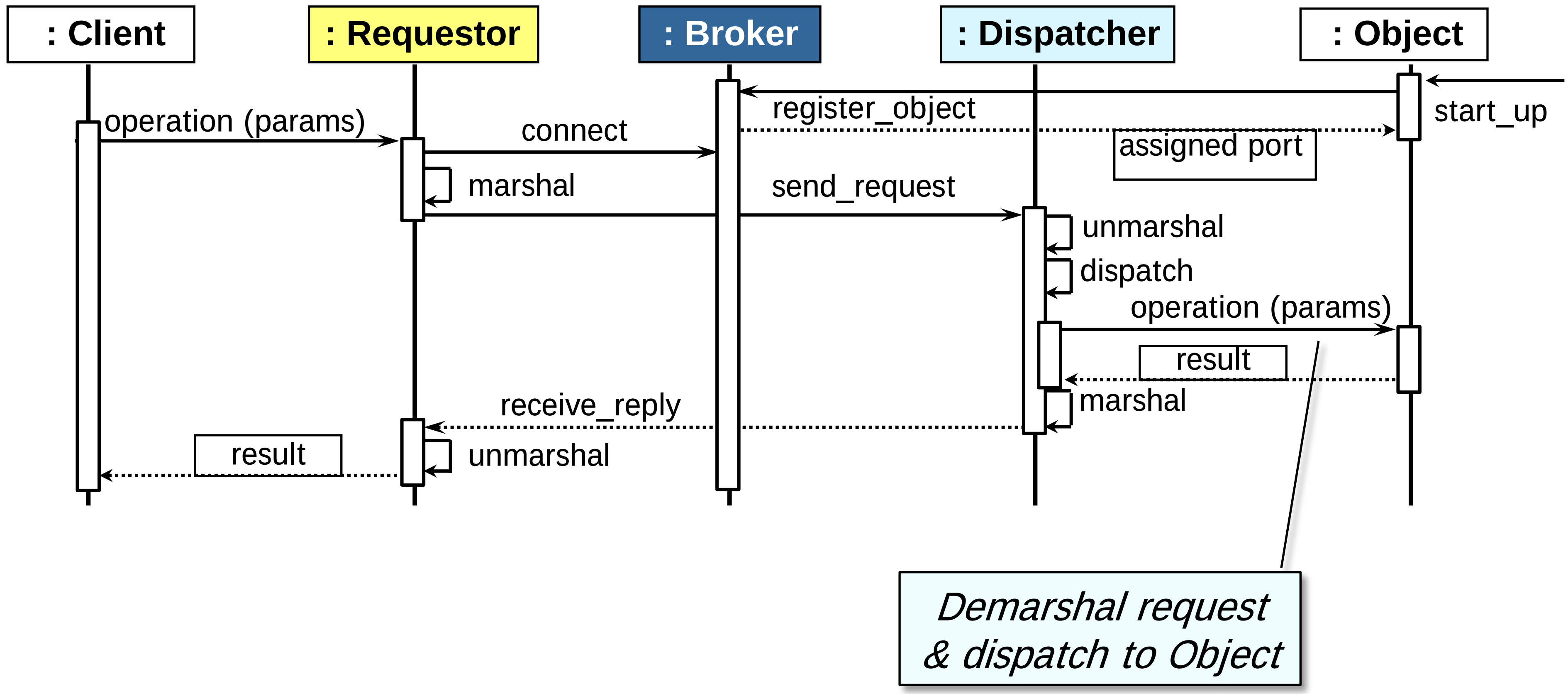
Dynamics



A Broker might or might not run in a separate process or thread

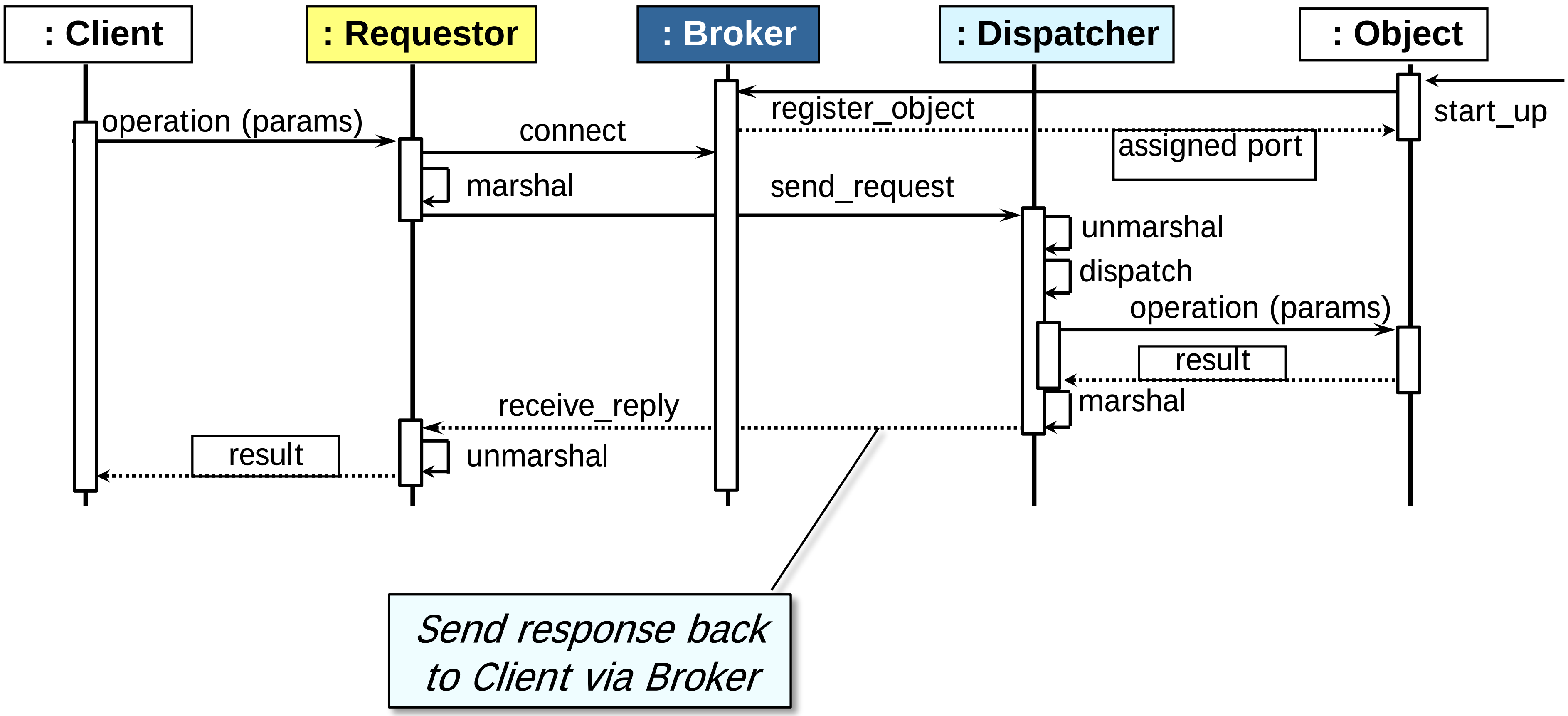
Broker POSA1 Architectural Pattern

Dynamics



Broker POSA1 Architectural Pattern

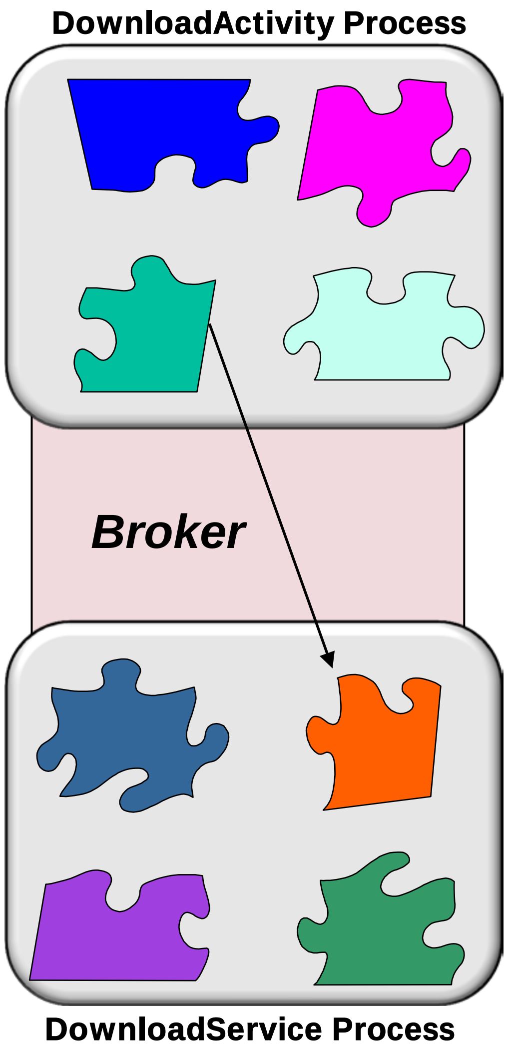
Dynamics



Broker POSA1 Architectural Pattern

Consequences

+ Location independence
  - A broker is responsible for locating servers, so clients need not know where servers are located

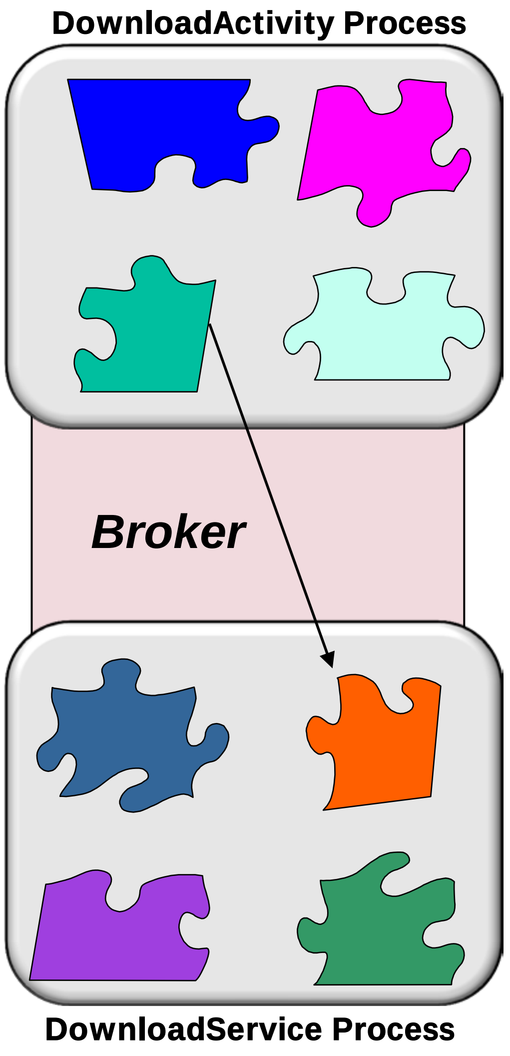


Broker POSA1 Architectural Pattern

Consequences

+ Location independence

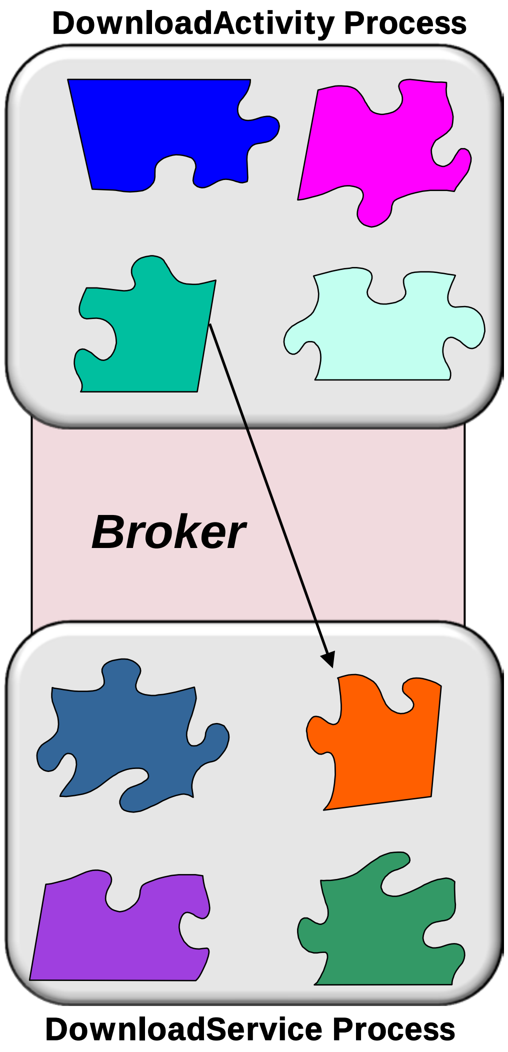
+ Separation of concerns
  - If server implementations change without affecting interfaces clients should not be affected



Broker POSA1 Architectural Pattern

Consequences

+ Location independence
+ Separation of concerns
+ Portability, modularity, reusability, etc.
  - A broker hides OS & network details from clients & servers via indirection & abstraction layers
    - e.g., APIs, proxies, adapters, bridges, wrapper facades, etc.

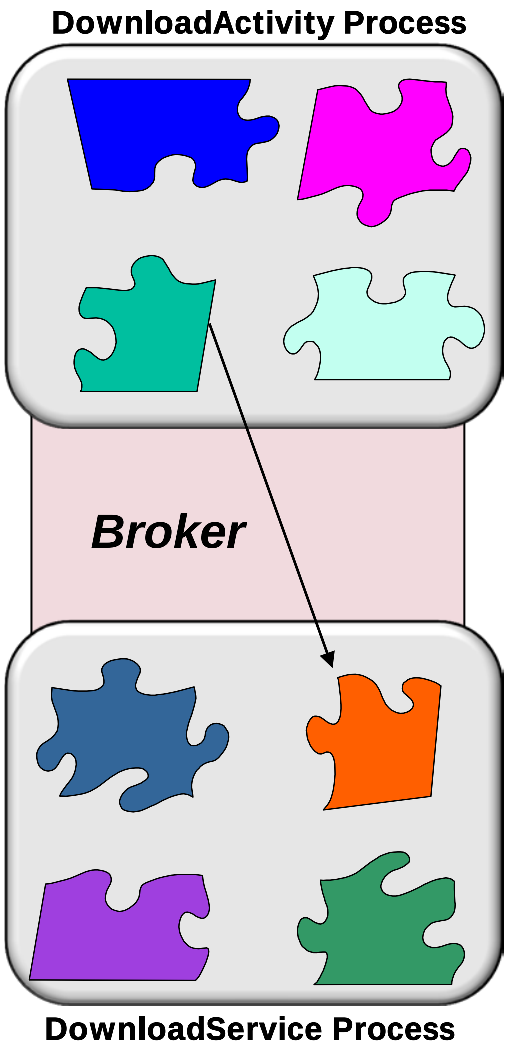


Broker POSA1 Architectural Pattern

Consequences

– Additional time & space overhead

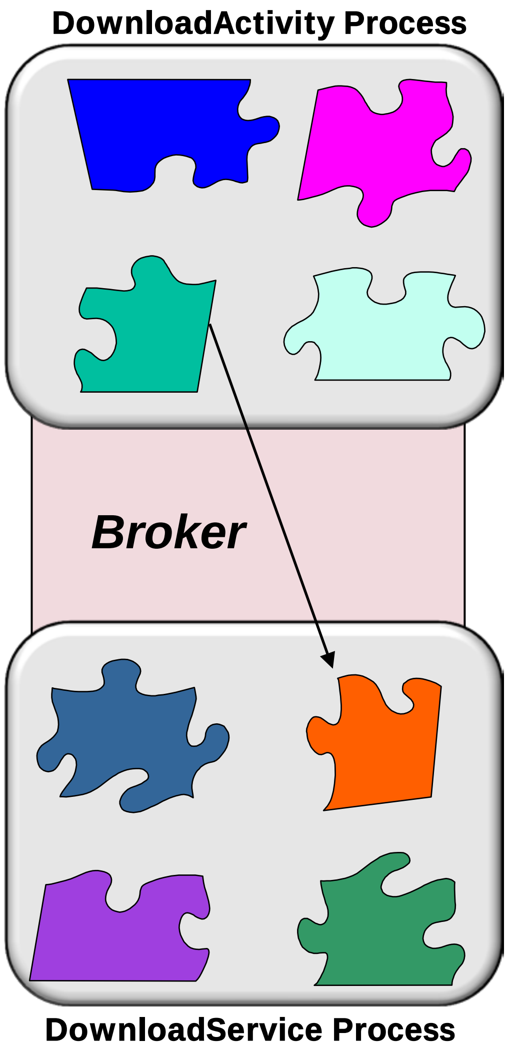
- Applications using brokers may be slower than applications written manually



Broker POSA1 Architectural Pattern

Consequences

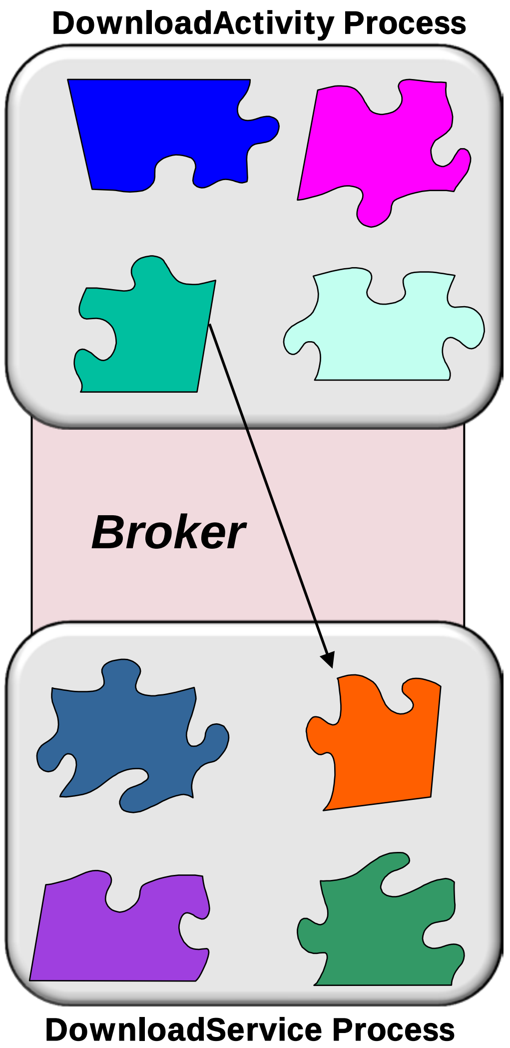
- Additional time & space overhead
- Potentially less reliable
  - Compared with non-distributed software applications, distributed broker systems may incur lower fault tolerance



Broker POSA1 Architectural Pattern

Consequences

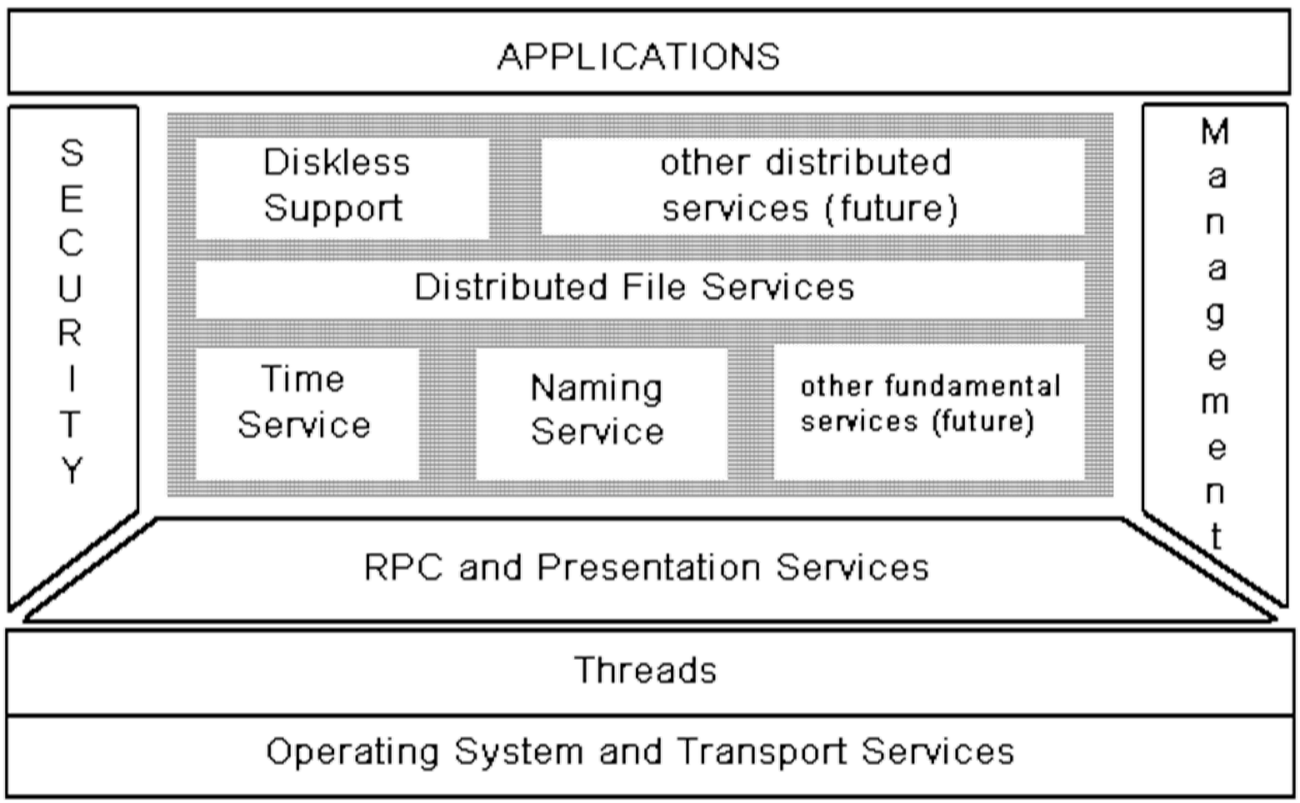
- Additional time & space overhead
- Potentially less reliable
- May complicate debugging & testing
  - Testing & debugging of distributed systems is tedious because of all the components involved



Broker POSA1 Architectural Pattern

Known Uses

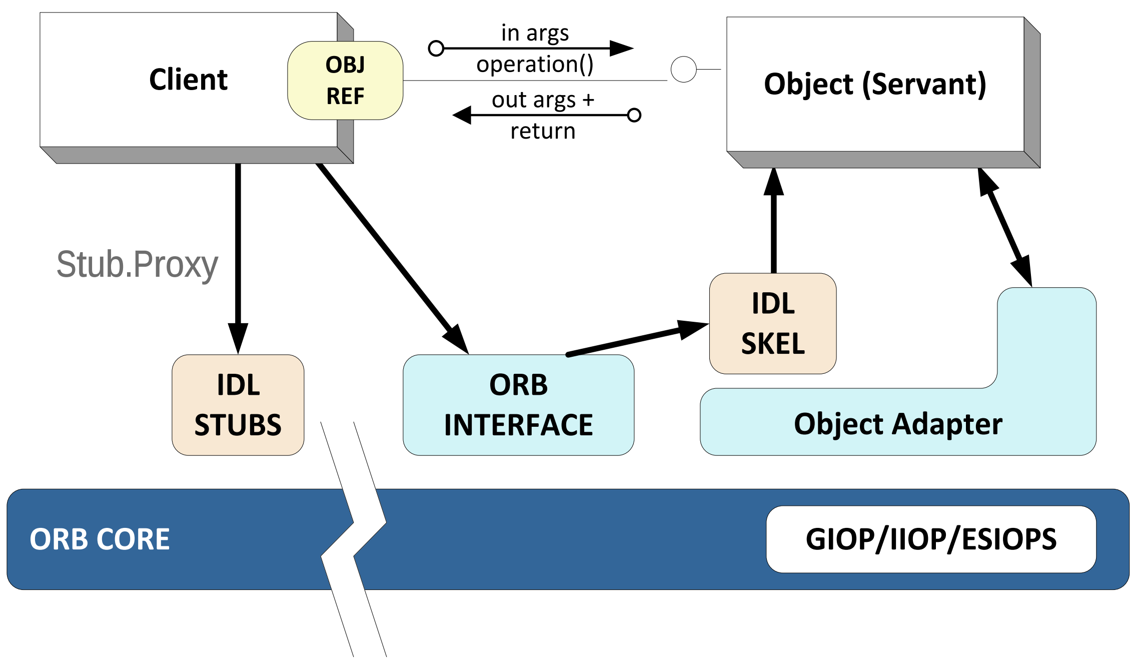
- Remote Procedure Call (RPC) middleware
  - e.g., ONC RPC & OSF Distributed Computing Environment (DCE)



Broker POSA1 Architectural Pattern

Known Uses

- Remote Procedure Call (RPC) middleware
- Distributed object computing middleware
  - e.g., Sun Java Remote Method Invocation (RMI) & OMG Common Object Request Broker Architecture (CORBA)

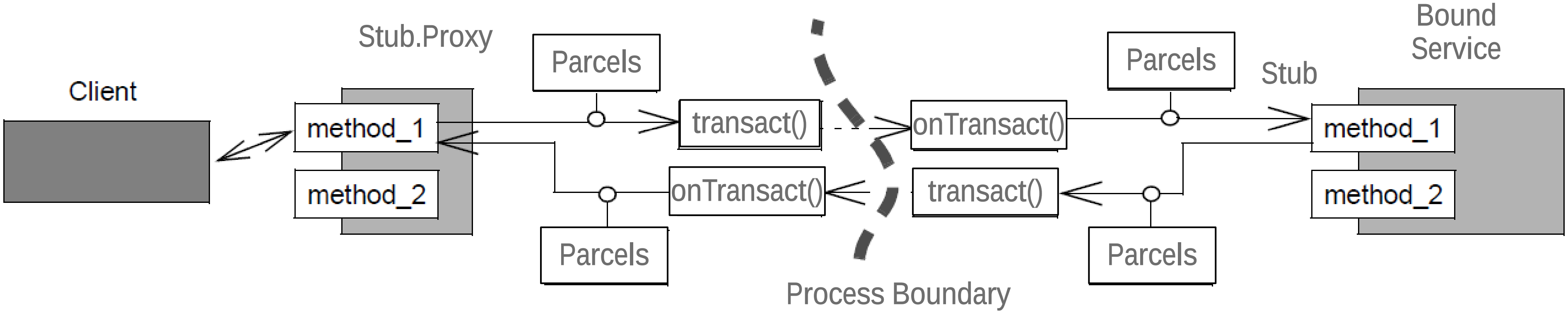


en.wikipedia.org/wiki/Object_request_broker

Broker POSA1 Architectural Pattern

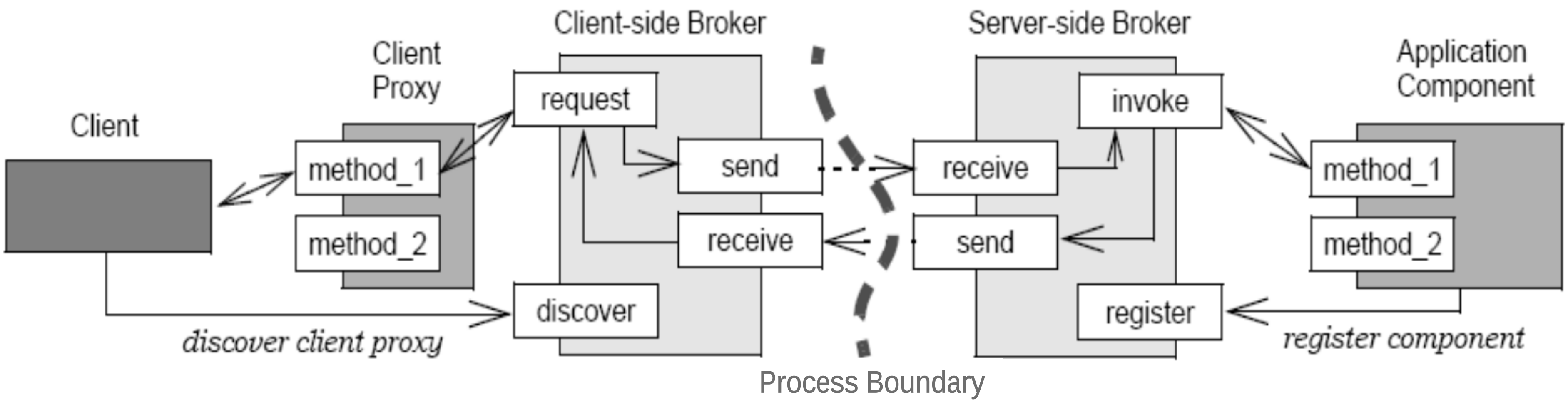
Known Uses

- Remote Procedure Call (RPC) middleware
- Distributed object computing middleware
- Local RPC middleware on smartphones
  - e.g., Android Binder



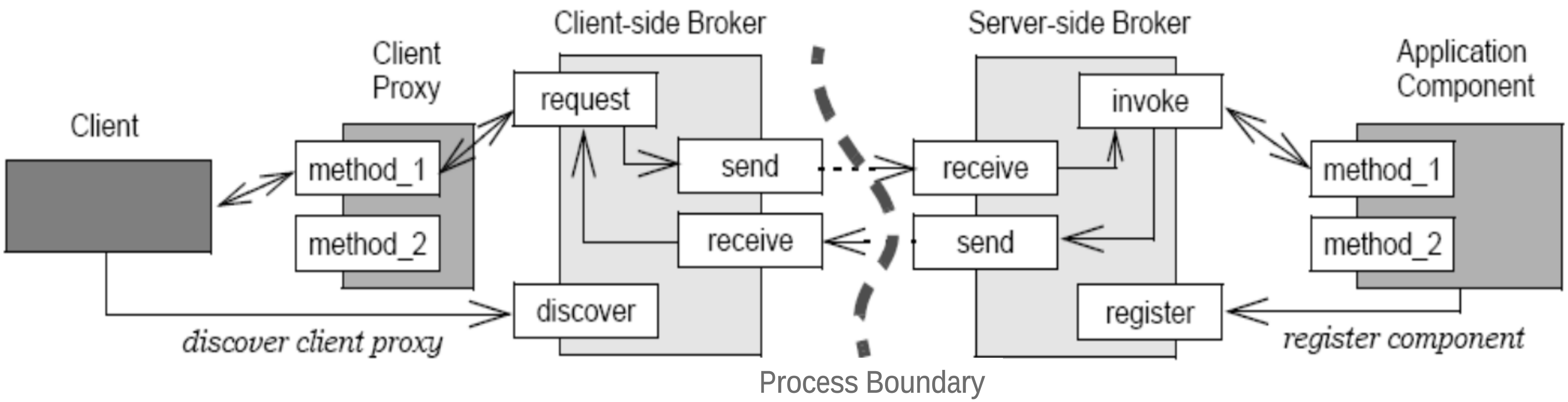
Summary

- *Broker* provides a straightforward means for passing commands between threads and/or processes in concurrent & networked software

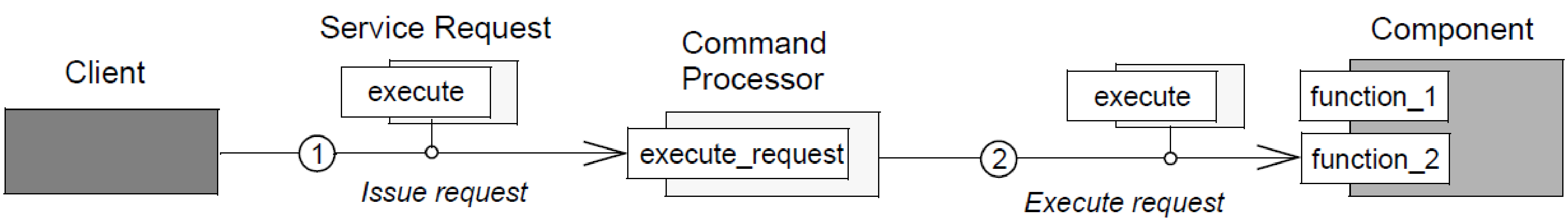


Summary

- *Broker* provides a straightforward means for passing commands between threads and/or processes in concurrent & networked software

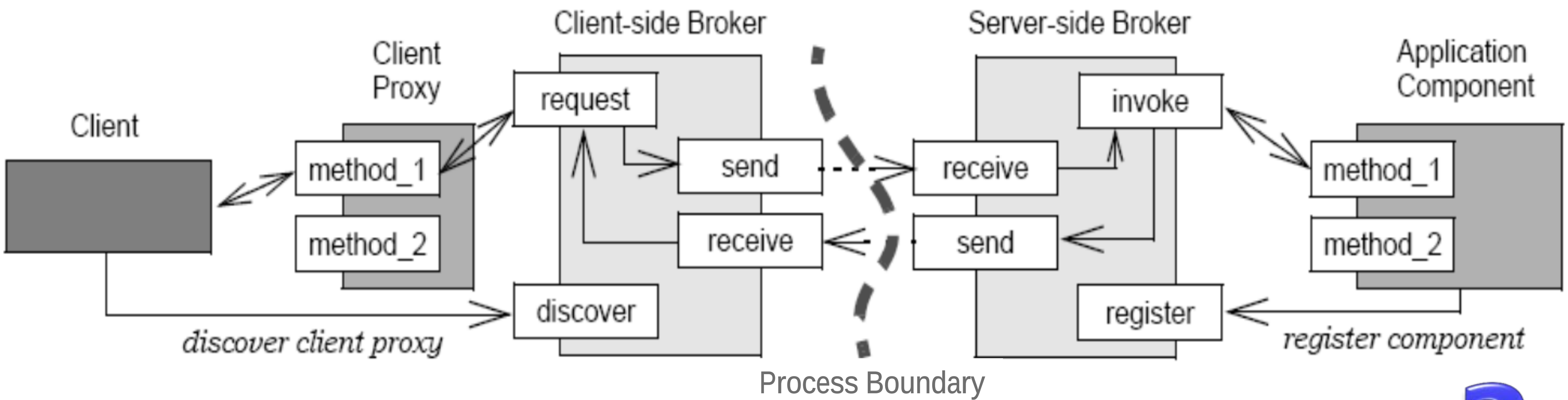


- In contrast, implementations of *Command Processor* use messaging

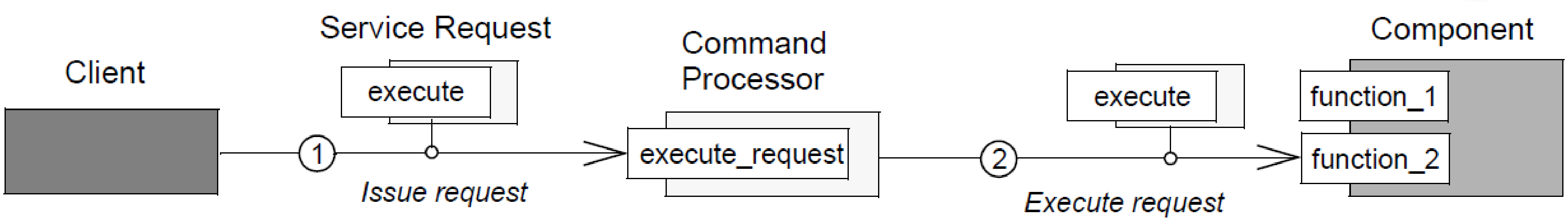


Summary

- *Broker* provides a straightforward means for passing commands between threads and/or processes in concurrent & networked software



- In contrast, implementations of *Command Processor* use messaging



- Software architects must understand the trade-offs between these patterns

Command Processor & *Broker* are "pattern complements"