Android Services & Local IPC: Overview of Programming Bound Services

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 to program Bound Services

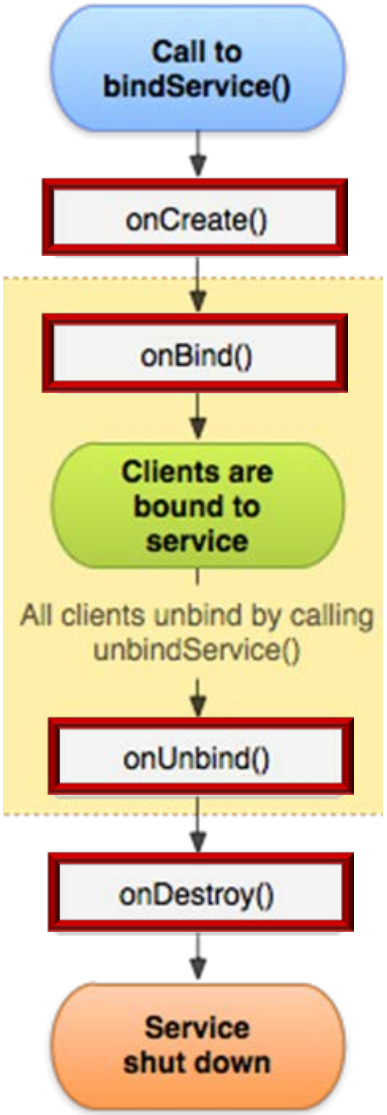
```
public class MyService extends Service {
  ...
  public void onCreate() {...}

  protected void onDestroy() {...}

  public Ibinder onBind(Intent intent) {...}

  public boolean onUnbind(Intent intent) {...}

  public int onStartCommand(Intent intent,
                            int flags,
                            int startId) {...}
  ...
}
```



Programming a Bound Service

- Implementing a Bound Service is similar to a Started Service, e.g.:
  - Inherit from Android Service class

```
public class MyService
             extends Service {
  ...
}
```

Programming a Bound Service

- Implementing a Bound Service is similar to a Started Service, e.g.:
  - Inherit from Android Service class
  - Override onCreate() & onDestroy (optional)
    - These hook methods are called back by Android to initialize & terminate a Service at the appropriate time

```
public class MyService
             extends Service {
  ...
  public void onCreate() {...}

  protected void onDestroy() {...}

  public Ibinder
    onBind(Intent intent) {...}

  public boolean
    onUnbind(Intent intent) {...}

  public int onStartCommand
     (Intent intent,
      int flags,
      int startId) {...}
  ...
}
```

Programming a Bound Service

- Implementing a Bound Service is similar to a Started Service, e.g.:
  - Inherit from Android Service class
  - Override onCreate() & onDestroy (optional)
  - Override the onBind() lifecycle method
    - Returns an Ibinder that defines a communication channel used for two-way interaction

```
public class MyService
                extends Service {
  ...
  public void onCreate() {...}

  protected void onDestroy() {...}

  public Ibinder
    onBind(Intent intent) {...}

  public boolean
    onUnbind(Intent intent) {...}

  public int onStartCommand
     (Intent intent,
      int flags,
      int startId) {...}
  ...
}
```

The object returned here is typically initialized at the class scope or in onCreate()

developer.android.com/reference/android/app/Service.html#onBind(android.content.Intent)

Programming a Bound Service

- Implementing a Bound Service is similar to a Started Service, e.g.:
  - Inherit from Android Service class
  - Override onCreate() & onDestroy (optional)
  - Override the onBind() lifecycle method
  - Can also implement onUnbind()
    - Called when all clients have disconnected from a particular interface published by the Service by calling unBindService()

```
public class MyService
                extends Service {
  ...
  public void onCreate() {...}

  protected void onDestroy() {...}

  public Ibinder
    onBind(Intent intent) {...}

  public boolean
    onUnbind(Intent intent) {...}

  public int onStartCommand
     (Intent intent,
      int flags,
      int startId) {...}
  ...
}
```

developer.android.com/reference/android/app/Service.html#onUnbind(android.content.Intent)

Programming a Bound Service

- Implementing a Bound Service is similar to a Started Service, e.g.:
  - Inherit from Android Service class
  - Override onCreate() & onDestroy (optional)
  - Override the onBind() lifecycle method
  - Can also implement onUnbind()
    - Called when all clients have disconnected from a particular interface published by the service
    - Typically returns false, but can return true to trigger reBind()

```
public class MyService
                extends Service {
  ...
  public void onCreate() {...}

  protected void onDestroy() {...}

  public Ibinder
    onBind(Intent intent) {...}

  public boolean
    onUnbind(Intent intent) {...}

  public int onStartCommand
     (Intent intent,
      int flags,
      int startId) {...}
  ...
}
```

developer.android.com/guide/components/bound-services.html#Lifecycle

Programming a Bound Service

- Implementing a Bound Service is similar to a Started Service, e.g.:
  - Inherit from Android Service class
  - Override onCreate() & onDestroy (optional)
  - Override the onBind() lifecycle method
  - Can also implement onUnbind()
- onStartCommand() is typically not implemented for a Bound Service
  - Only do this if you want to manage the lifecycle of the Bound Service

```
public class MyService
              extends Service {
  ...
  public void onCreate() {...}

  protected void onDestroy() {...}

  public Ibinder
    onBind(Intent intent) {...}

  public boolean
    onUnbind(Intent intent) {...}

  public int onStartCommand
     (Intent intent,
      int flags,
      int startId) {...}
  ...
}
```

developer.android.com/guide/components/bound-services.html#Lifecycle

Programming a Bound Service

- Implementing a Bound Service is similar to a Started Service, e.g.:
  - Inherit from Android Service class
  - Override onCreate() & onDestroy (optional)
  - Override the onBind() lifecycle method
  - Can also implement onUnbind()
  - onStartCommand() is typically not implemented for a Bound Service
  - Include the Service in the AndroidManifest.xml config file

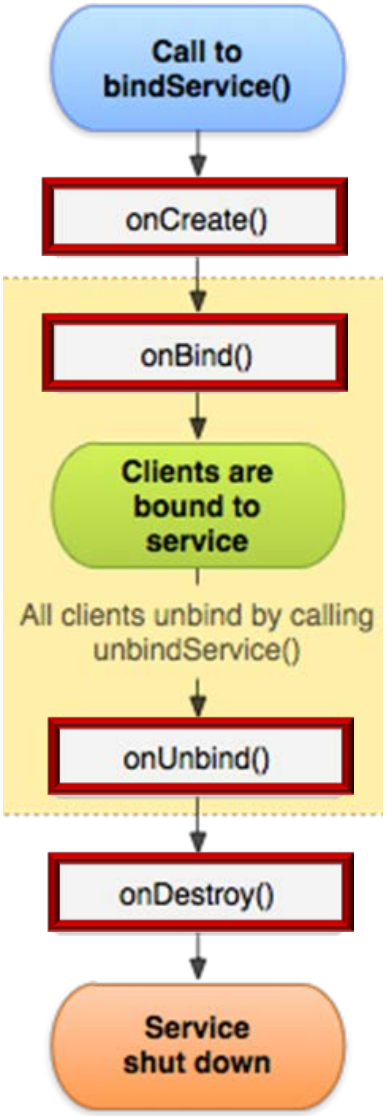
```
<application ... >
    <activity android:name=
        ".MyActivity"
      ...
    </activity>

    <service
       android:exported= "true"
       android:name=
          ".MyService“
        ...
    </service>

</application
```

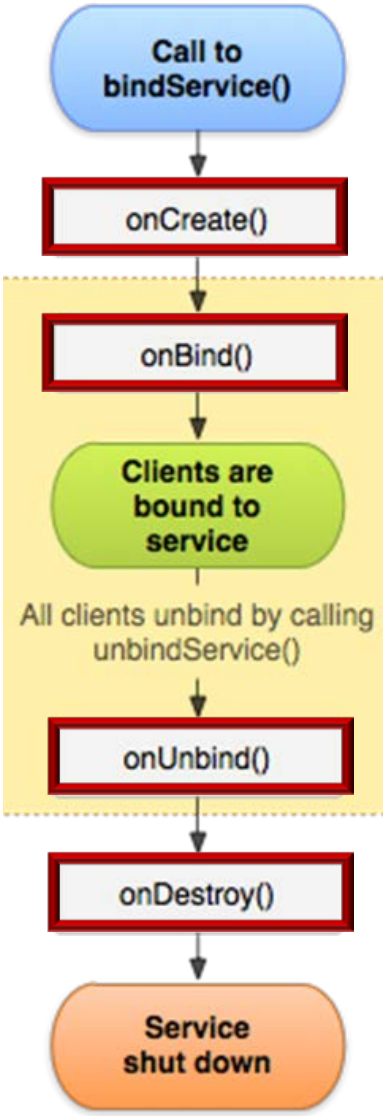
Summary

- Programming two-way communication with Bound Services is straightforward
  - The bulk of the implementations are handled by Android & a client-side callback protocol



Summary

- Programming two-way communication with Bound Services is straightforward
- One of the most important parts of implementing a Bound Service is defining the interface that the onBind() callback method returns
  - Three common ways to implement the Service's IBinder interface are discussed next
    - Extent the Binder class
    - Use a Messenger
    - Use the Android Interface Definition Language (AIDL)



Android Services & Local IPC: Local Bound Service Communication by Extending the Binder Class

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

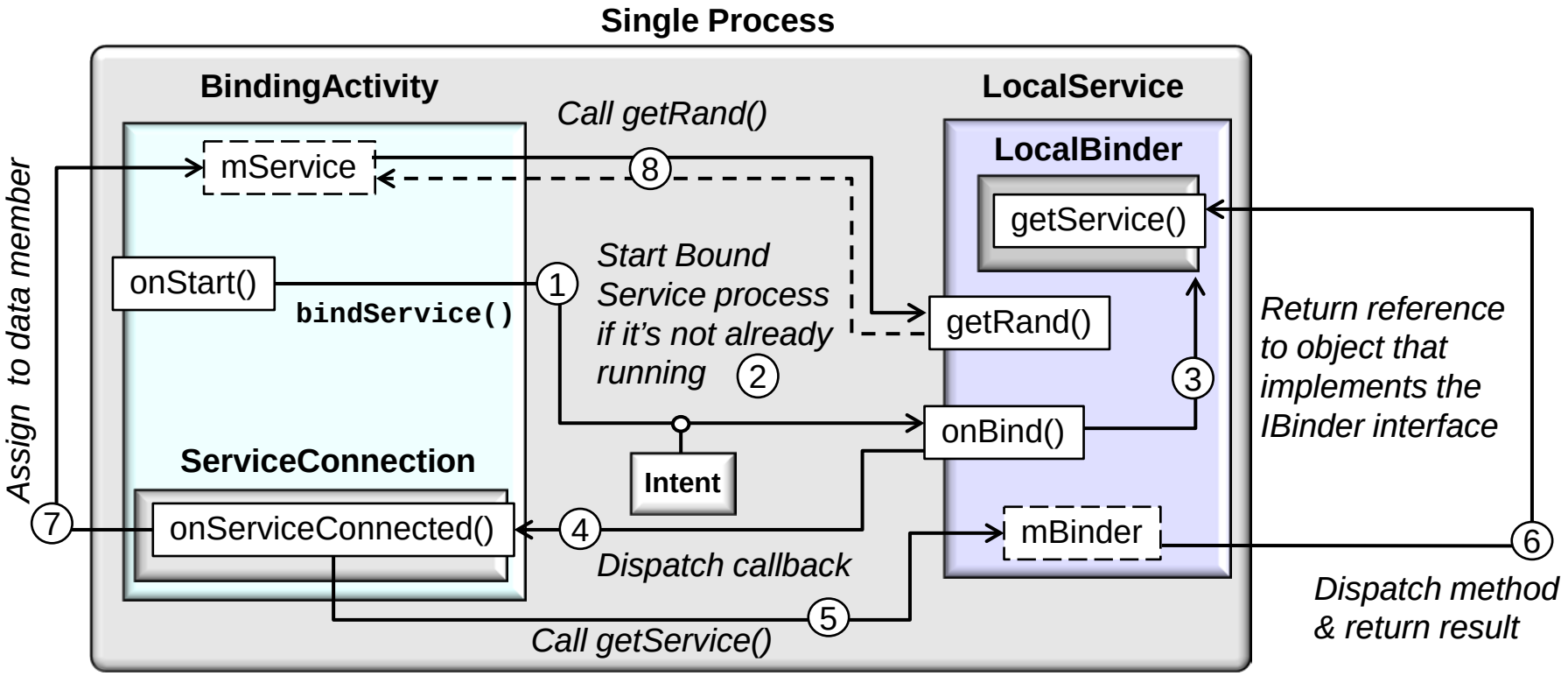
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Module

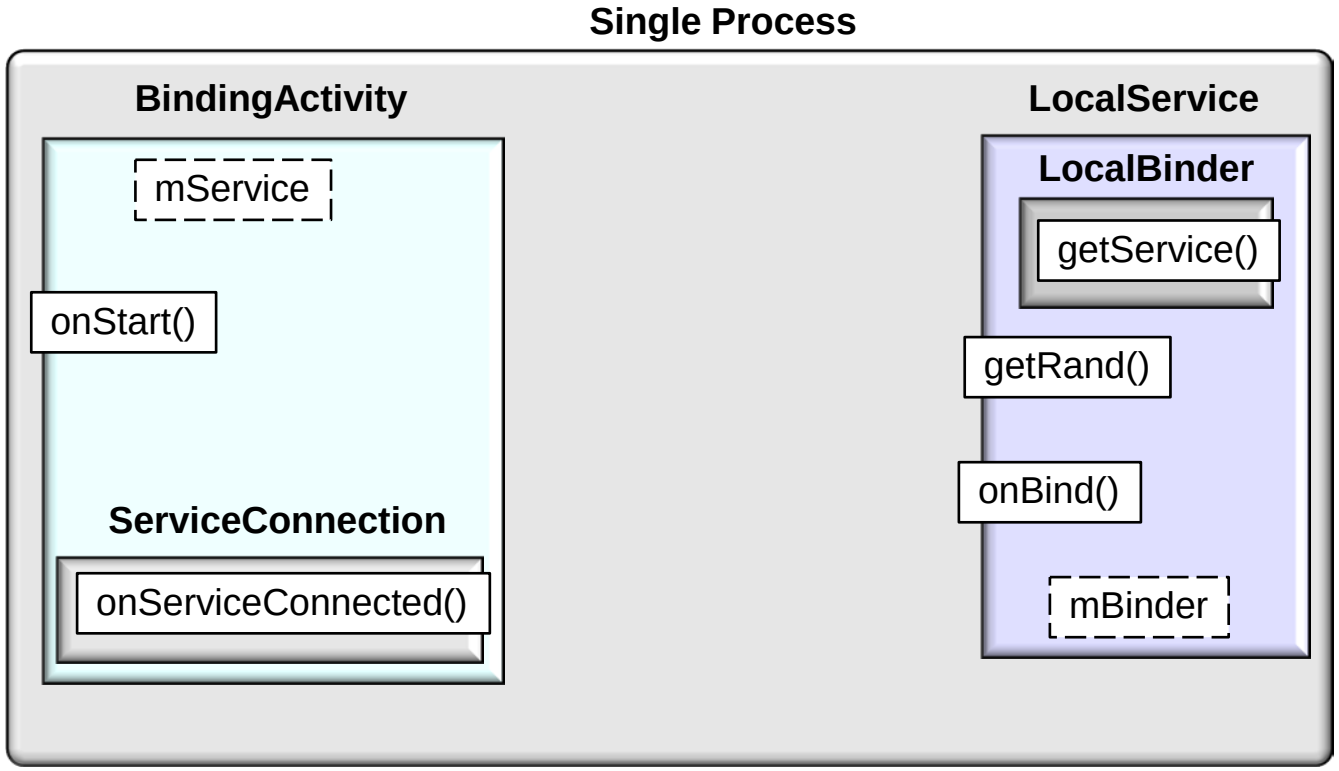
- Understand how to communicate with Local Bound Services by extending the Binder class



See developer.android.com/guide/components/bound-services.html#Binder

Communication via a Local Binder

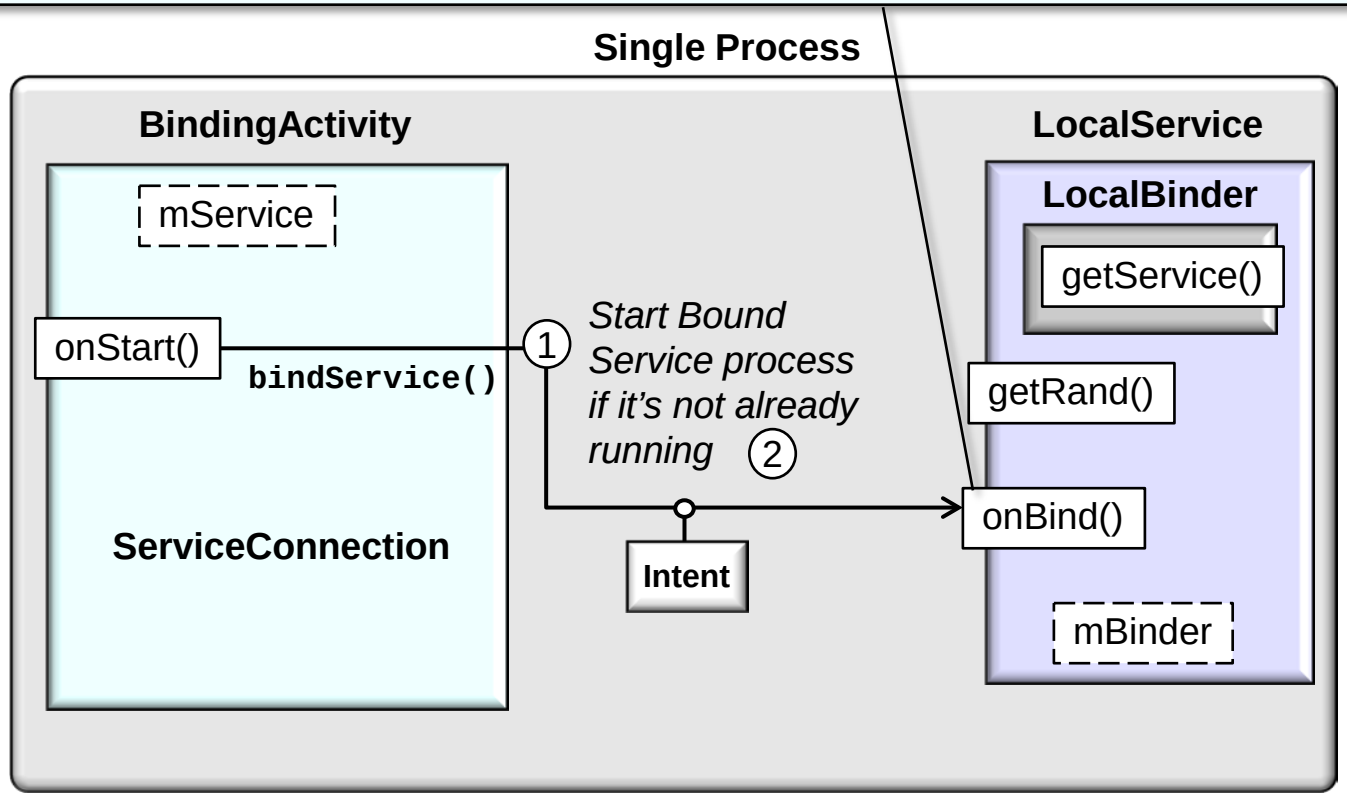
- Sometimes a Bound Service is used only by a local client Activity & need not work across processes
  - In this "collocated" case, simply implement an instance of a Binder subclass that provides the client direct access to public methods in a Service



Communication via a Local Binder

The onBind() method can create a Binder object that either:

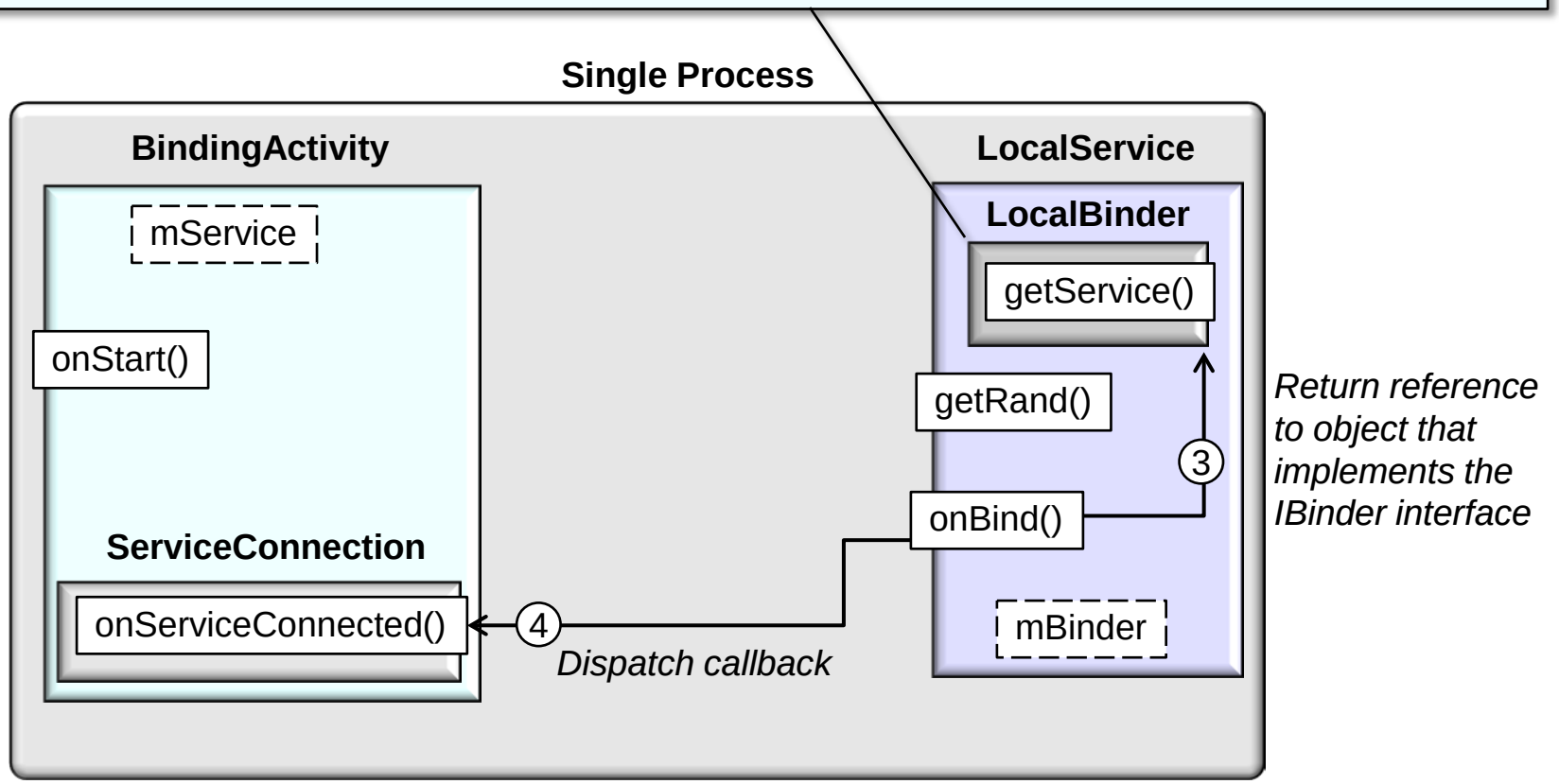
- *Contains public methods the client can call*
- *Returns current Service instance, which has public methods the client can call, or*
- *Returns an instance of another class hosted by Service that the client can call*



Communication via a Local Binder

The LocalBinder "is a" Binder

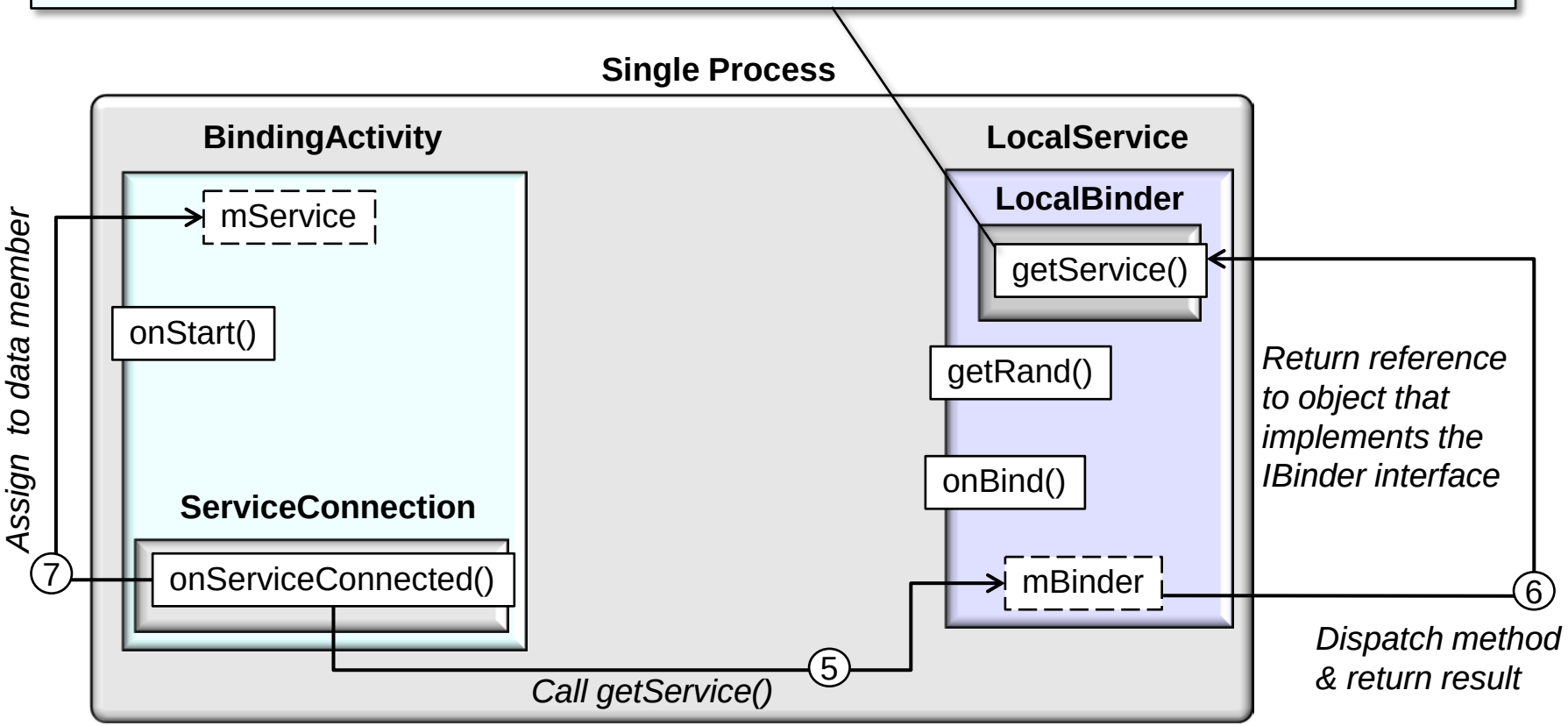
```
public class LocalBinder extends Binder {
   ...
}
```



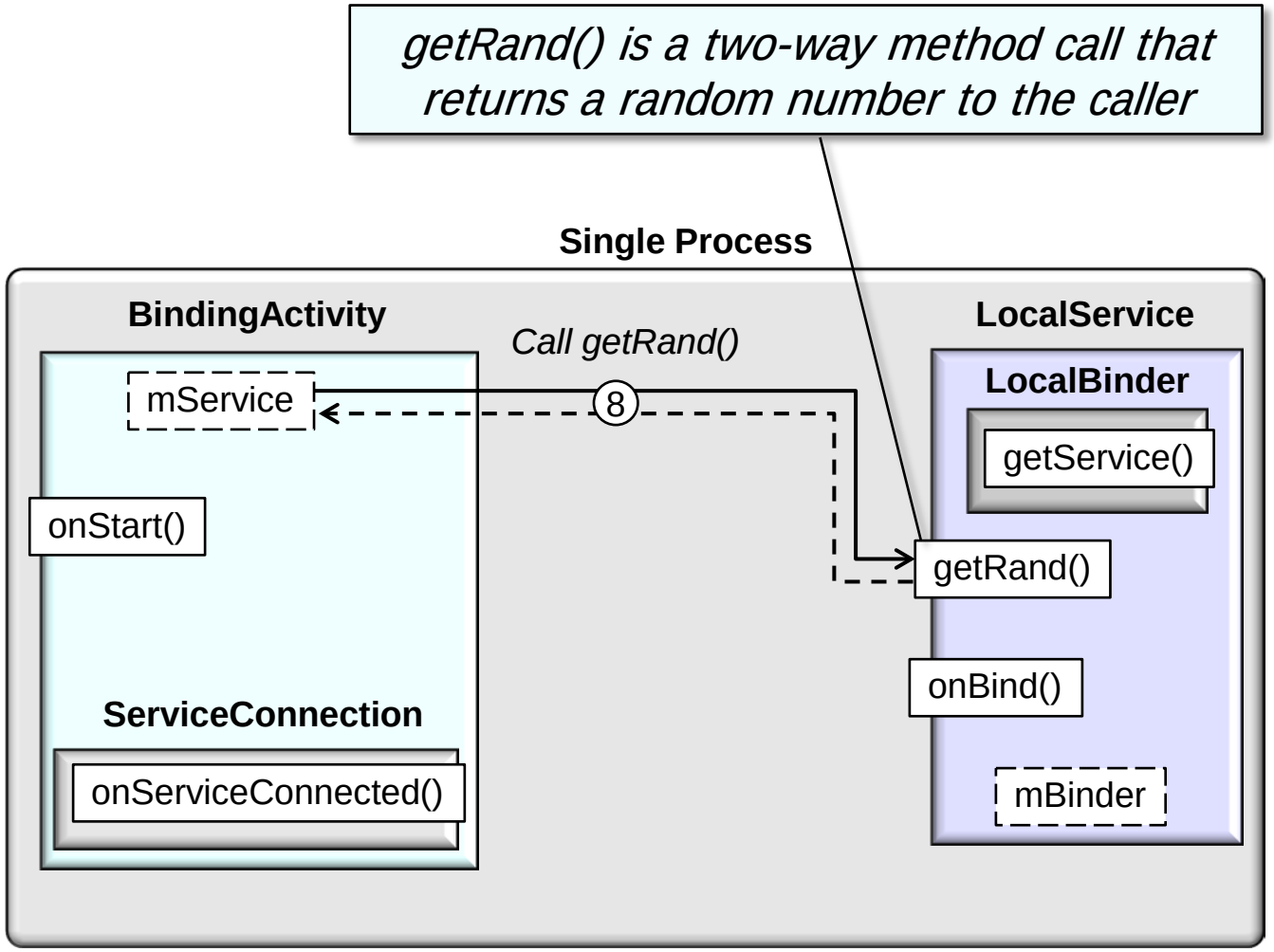
Communication via a Local Binder

The getService() factory method allows clients to call LocalService methods

```
public class LocalBinder extends Binder {
  LocalService getService() { return LocalService.this; }
}
```



Communication via a Local Binder



Example of Service that Extends the Binder

- Create a Binder object that returns the current Service instance, which has public methods the client can call

```
public class LocalService extends Service {
  public class LocalBinder extends Binder
  { LocalService getService() { return LocalService.this; } }

  private final IBinder mBinder = new LocalBinder ();

  public IBinder onBind(Intent intent) { return mBinder; }

  private final Random mGenerator = new Random();

  public int getRand() { return mGenerator.nextInt(100); }
}
```

Return Service instance to client

Factory Method for clients

Called by Android when client invokes bindService() to return Binder instance

Called by clients to generate a random number

Example of Client that Uses the Extended Binder

- The client receive the Binder from the onServiceConnected() callback method & makes calls to the Bound Service using the provided methods

```
public class BindingActivity extends Activity {
  LocalService mService; boolean mBound = false;

  protected void onStart() {
    super.onStart();
    Intent intent = new Intent(this, LocalService.class);
    bindService(intent, mConn, Context.BIND_AUTO_CREATE);
  }

  protected void onStop() {
    super.onStop();
    if (mBound) { unbindService(mConn); mBound = false; }
  }

  public void onButtonClick(View v) {
    if (mBound) Toast.makeText(this, mService.getRand(),
                                Toast.LENGTH_SHORT).show();
  }
```

Object state

Bind to LocalService

Unbind to LocalService

Calls Service's method

Example of Client that Uses the Extended Binder

- The client receive the Binder from the onServiceConnected() callback method & makes calls to the Bound Service using the provided methods

```
public class BindingActivity extends Activity {
  ...
```

Defines Service binding callbacks, passed to bindService()

```
  private ServiceConnection mConn = new ServiceConnection() {

    public void onServiceConnected(ComponentName className,
                                   IBinder service) {
      LocalService.LocalBinder binder =
                              (LocalService.LocalBinder)service;
      mService = binder.getService(); mBound = true;
    }
```

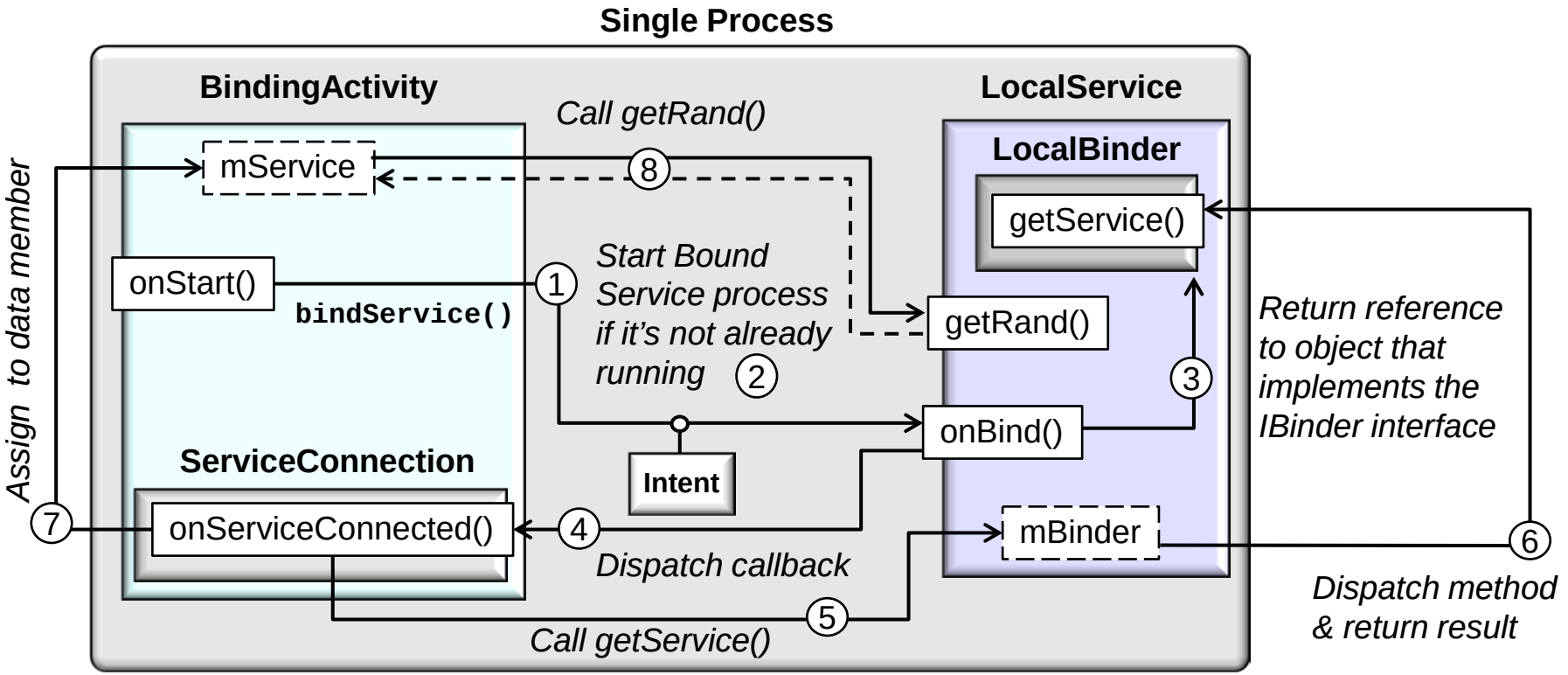
Cast the IBinder & get LocalService instance

```
    public void onServiceDisconnected(ComponentName a)
    { mBound = false; }
  };
}
```

Called when Service is unexpectedly disconnected

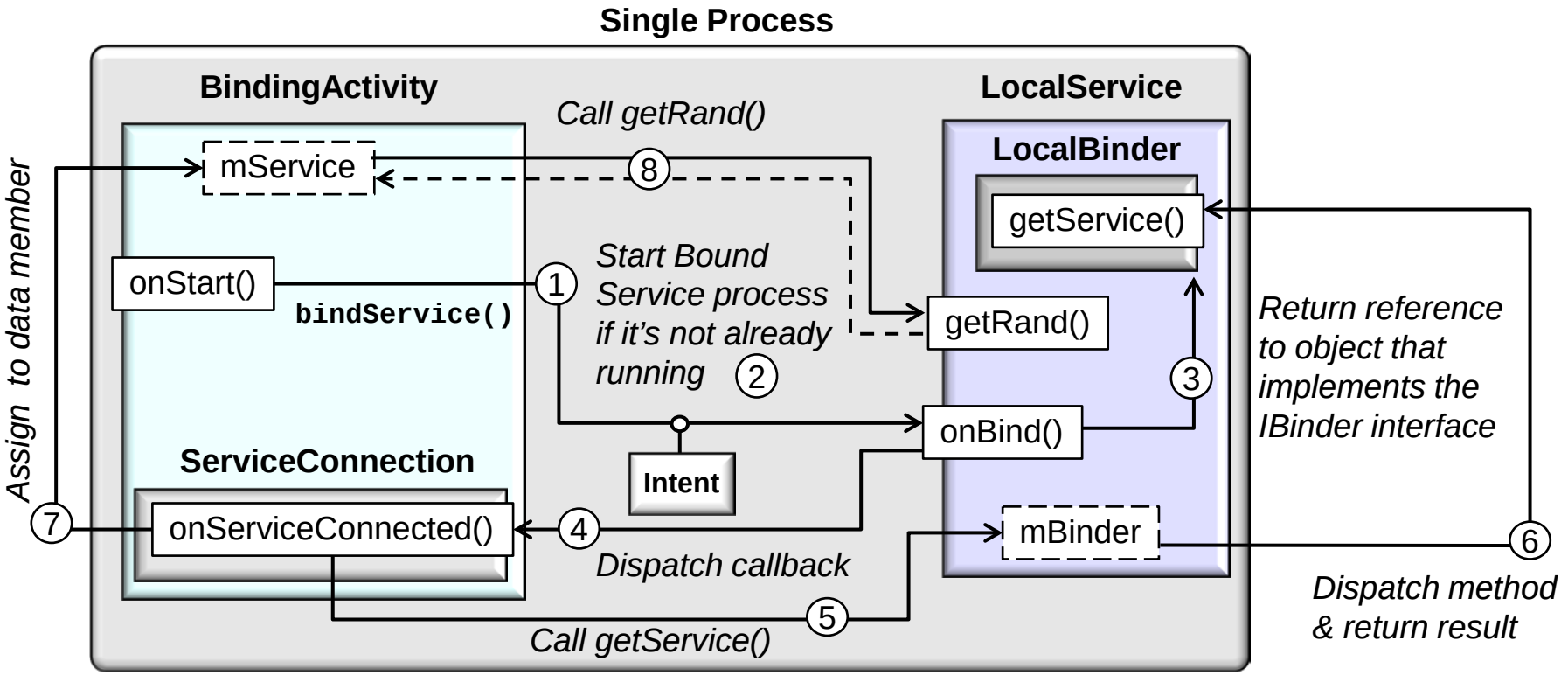
Summary

- Using Local Binders is the preferred technique when a Service is merely a background worker for an Activity
  - The Service & the client must be in the same process because this technique does not perform any (de)marshaling across processes



Summary

- Using Local Binders is the preferred technique when a Service is merely a background worker for an Activity
- The only reason not to create a Bound Service this way is because the Service is used by other Apps or across separate processes
  - Note how the method is dispatched in the same thread as the caller



Android Services & Local IPC: Bound Service Communication Via Messengers

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

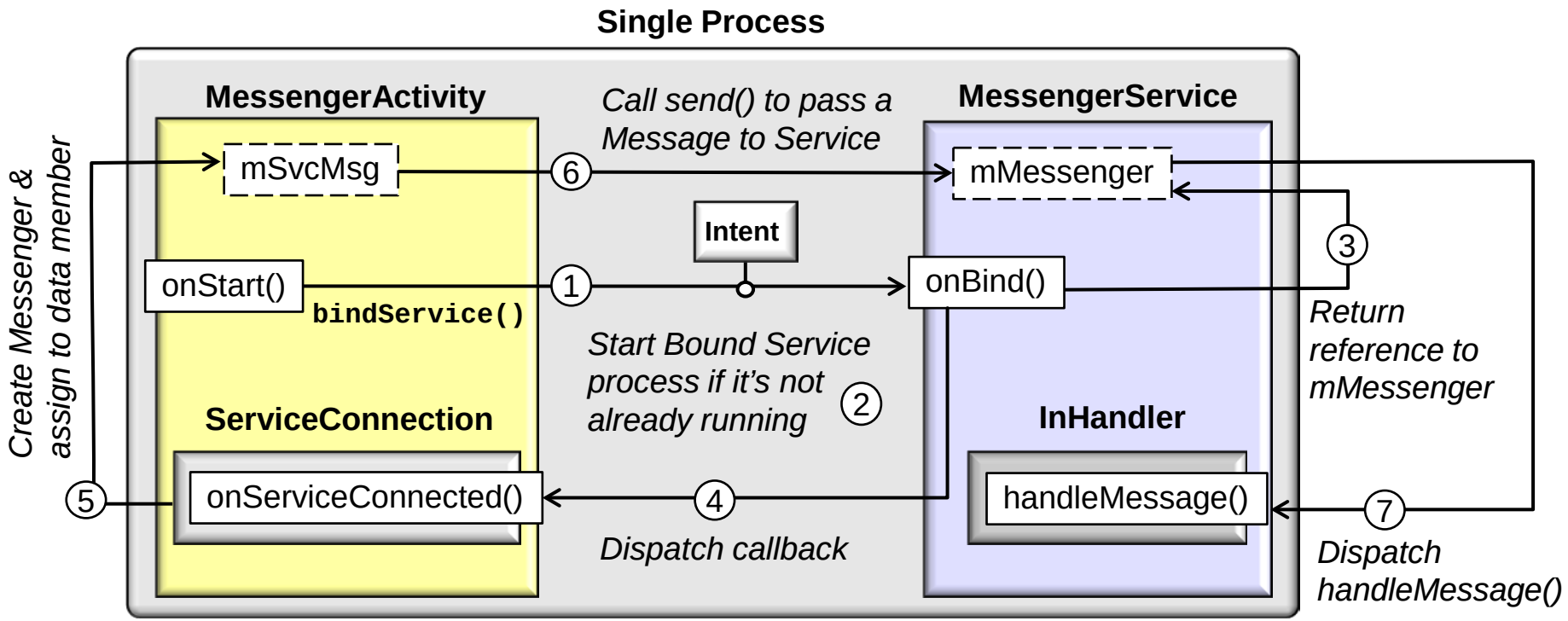
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

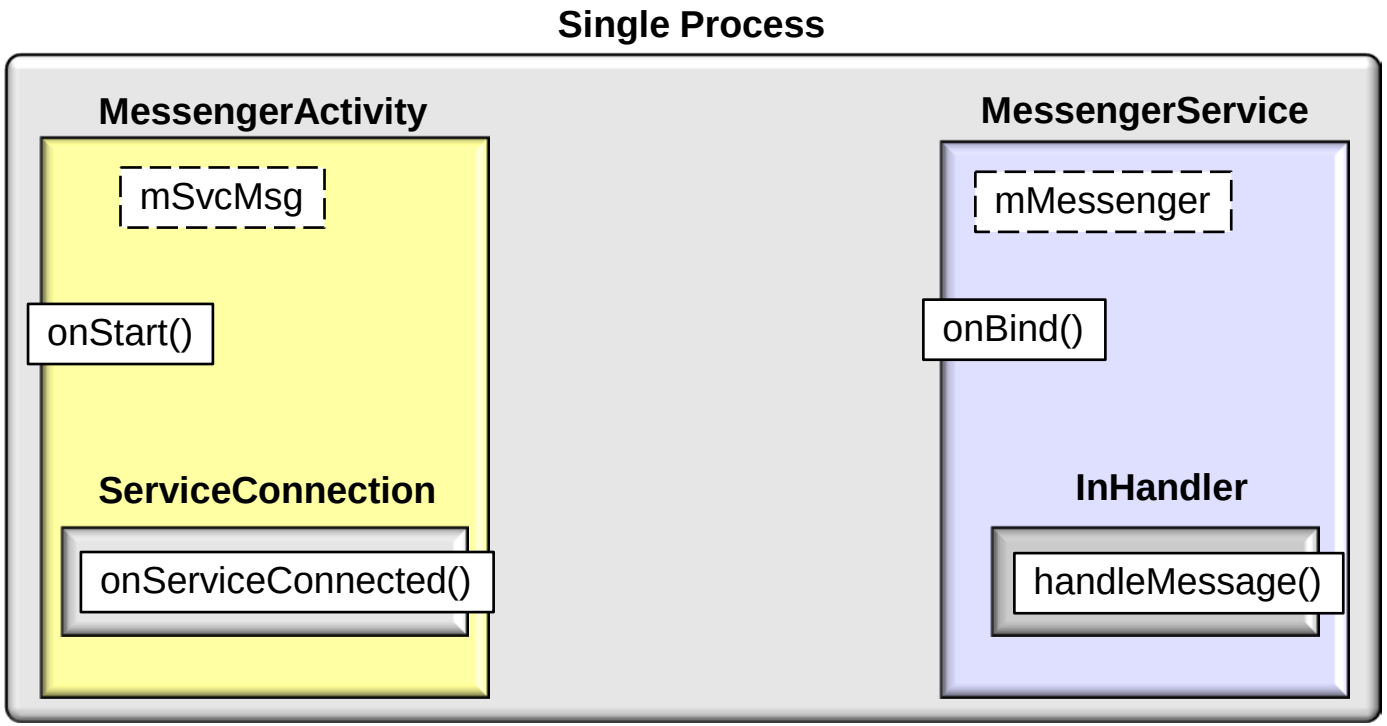
Learning Objectives in this Part of the Module

- Understand how to communicate with Bound Services via Messengers



Using a Messenger in a Bound Service

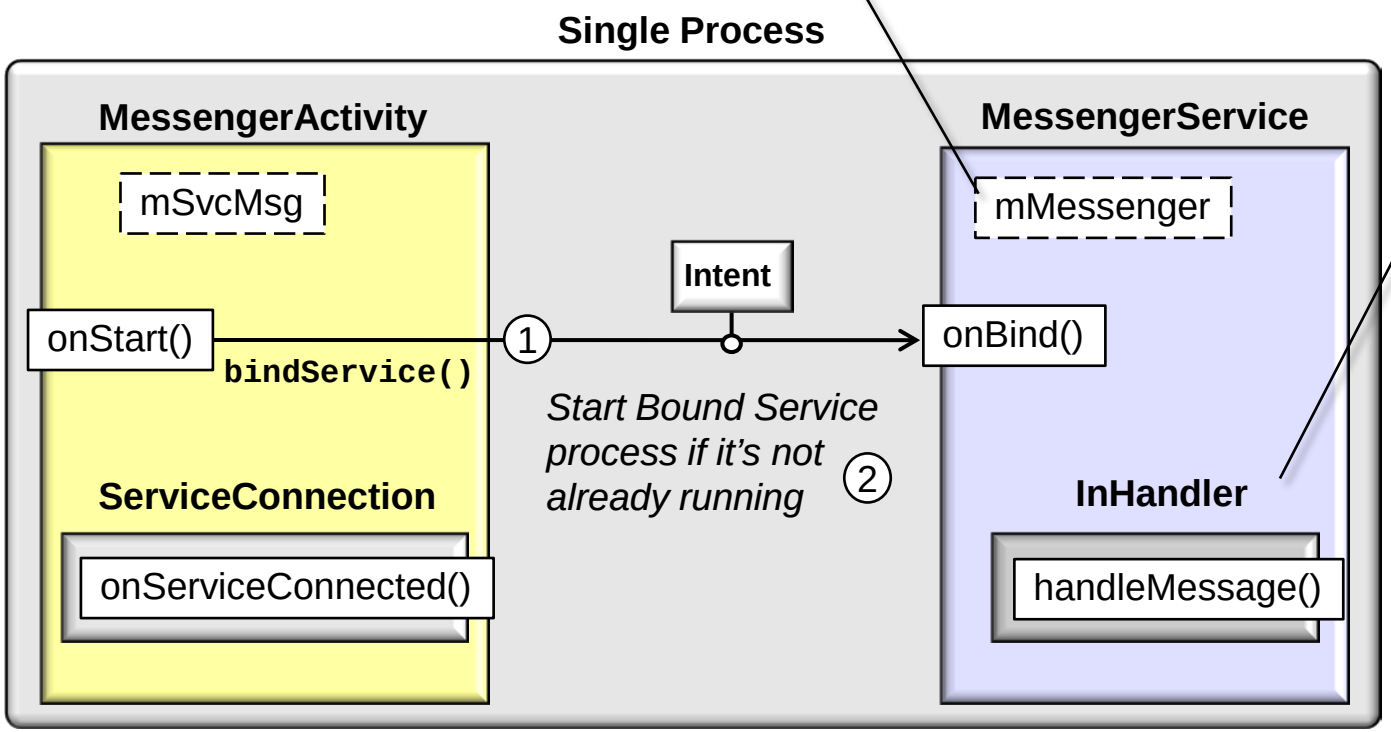
- A Messenger can be used to communicate with a Bound Service
  - Enables interaction between an Activity & a Bound Service without using AIDL (which is more powerful & complicated)



Generalizing to communicate between processes is relatively straightforward

Using a Messenger in a Bound Service

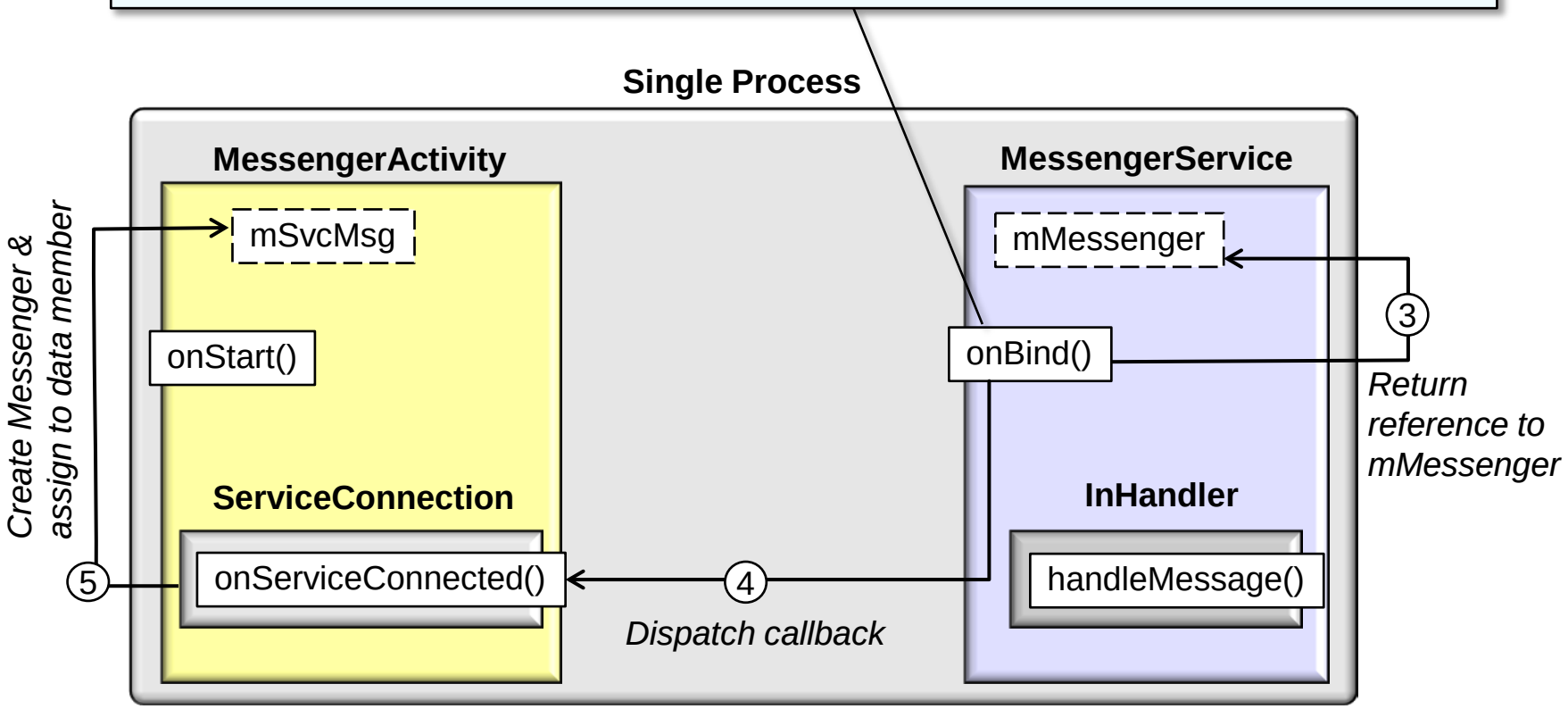
Implement a Handler that receives a callback for each call from a client & reference the Handler in a Messenger object



Using a Messenger in a Bound Service

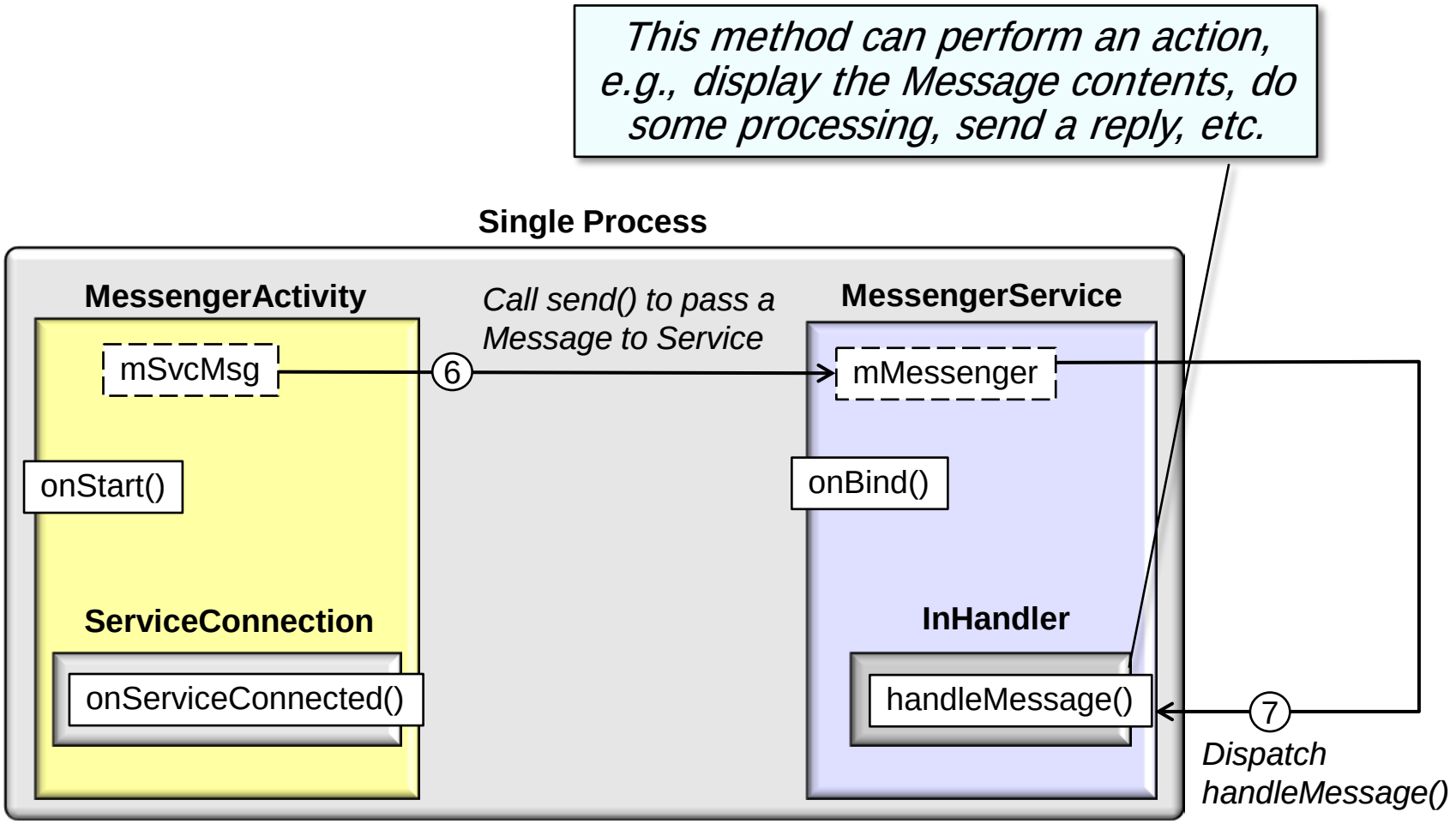
Messenger creates IBinder that Service returns to clients from onBind()

```
public IBinder onBind(Intent intent)
{ return mMessenger.getBinder(); }
```



developer.android.com/reference/android/os/Messenger.html#getBinder()

Using a Messenger in a Bound Service



Example Using a Messenger in a Bound Service

```
public class MessengerService extends Service {
  static final int MSG_PERFORM_ACTION = 1;

  class InHandler extends Handler {
    public void handleMessage(Message msg) {
      switch (msg.what) {
      case MSG_PERFORM_ACTION:
        processMessage(msg); break;
      default: super.handleMessage(msg);
      }
    }
  }

  final Messenger mMessenger = new Messenger(new InHandler());

  public IBinder onBind(Intent intent)
  { return mMessenger.getBinder(); }
}
```

Instruct Service to do some action

Handler for incoming client Messages

Target for clients to send Messages to InHandler

Return Ibinder so clients can send Messages to Service

Example Using a Messenger in an Activity

```
public class MessengerActivity extends Activity {
  ...
  Messenger mSvcMsg = null;

  boolean mBound;

  private ServiceConnection mConnection =
                                  new ServiceConnection() {

    public void onServiceConnected(ComponentName className,
                                   IBinder service) {
      mSvcMsg = new Messenger(service); mBound = true;
    }

    public void onServiceDisconnected(ComponentName className) {
      mSvcMsg = null;  mBound = false;
    }
  };
```

Means to communicate w/Service

Flag indicating if Service is bound

Called when connection with Service has been established, giving the object to interact with the Service

Called when Service is unexpectedly disconnected

Example Using a Messenger in an Activity

```
public class MessengerActivity extends Activity {
  ...
  protected void onStart() {
    super.onStart();
    bindService(new Intent(this, MessengerService.class),
                mConnection, Context.BIND_AUTO_CREATE);
  }

  protected void onStop() {
    super.onStop();
    if (mBound) { unbindService(mConnection); mBound = false; }
  }

  public void onButtonClick(View v) {
    if (!mBound) return;
    Message msg = Message.obtain
      (null, MessengerService.MSG_PERFORM_ACTION, 0, 0);
      ...
      mSvcMsg.send(msg);
  }
  ...
```

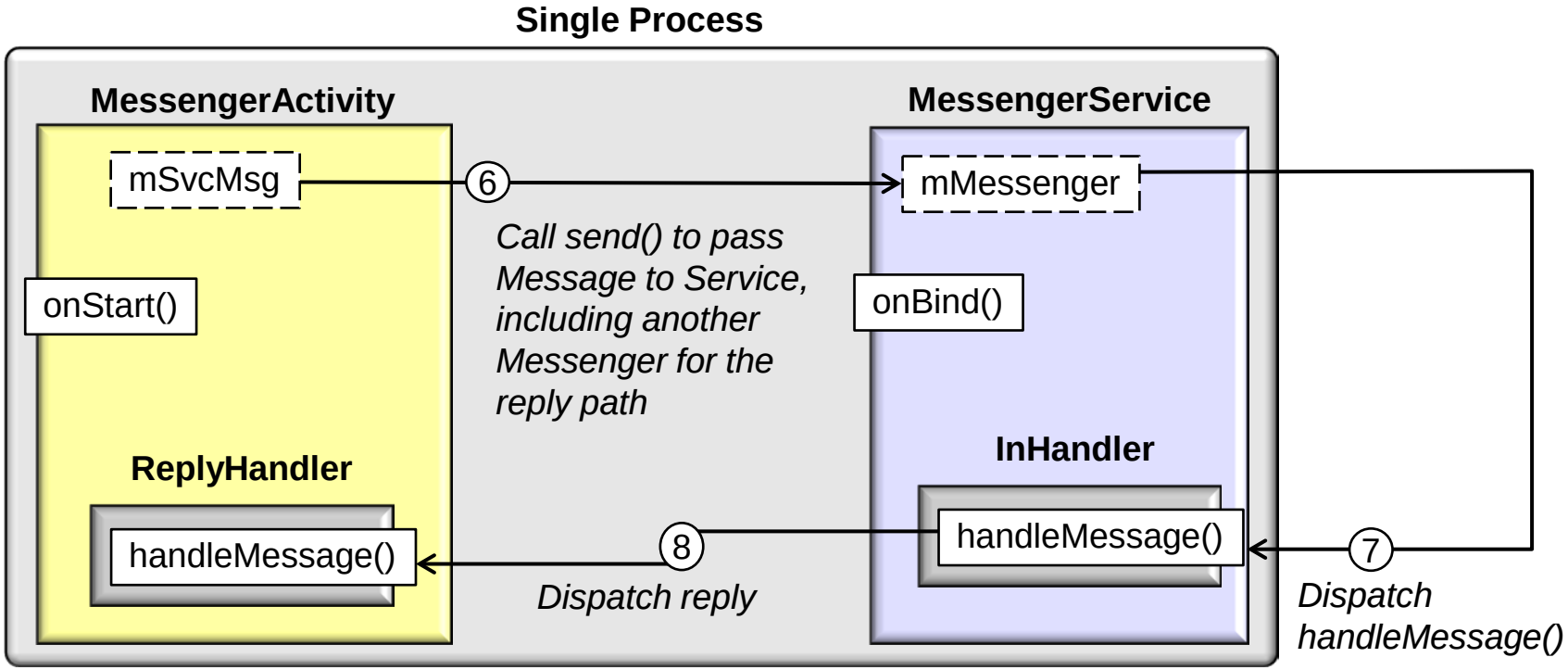
Bind to the service

Unbind from the service

Create & send a Message to Messenger in Service, using a 'what' value

Using Messengers for Two-way Communication

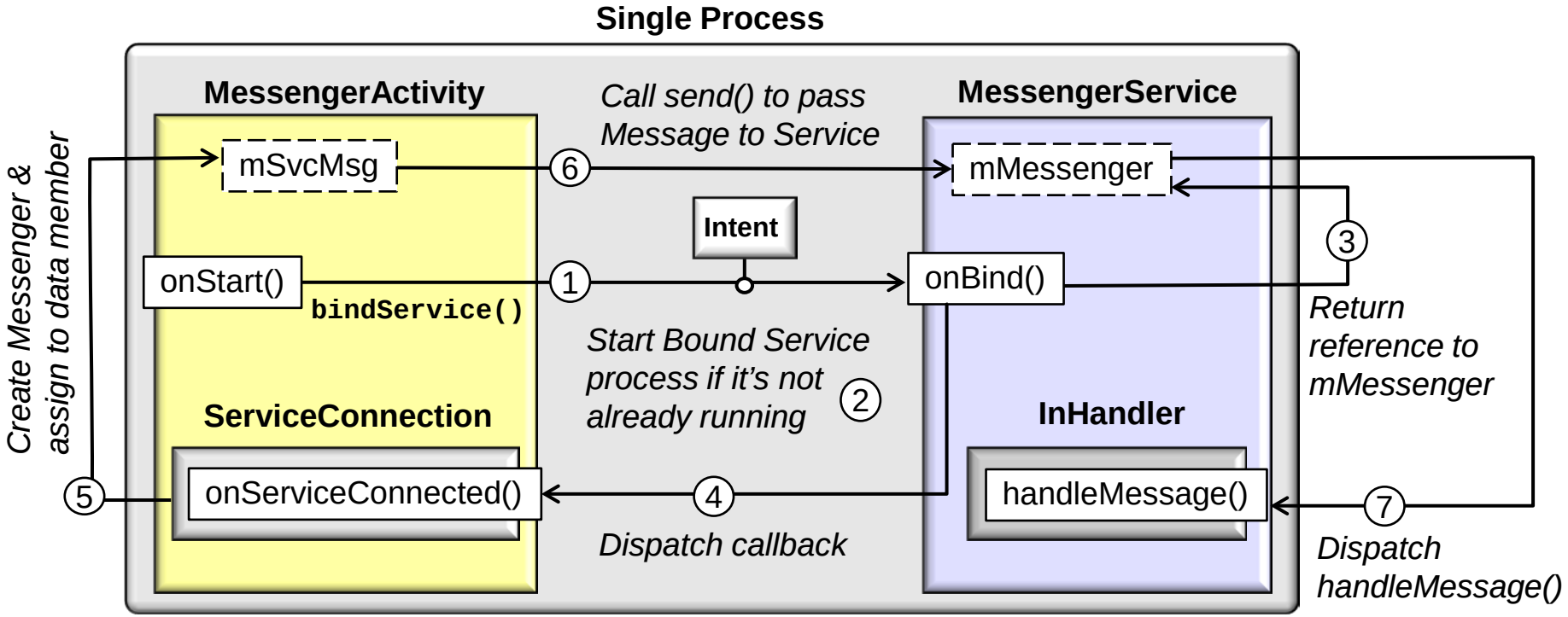
- Two-way communication via Messengers in a Bound Service is a slight variation on the approach described earlier
  - It involves sending a replyMessenger with the original Message, which is then used to call send() back on the client



We didn't show the code for two-way communication in our example

Summary

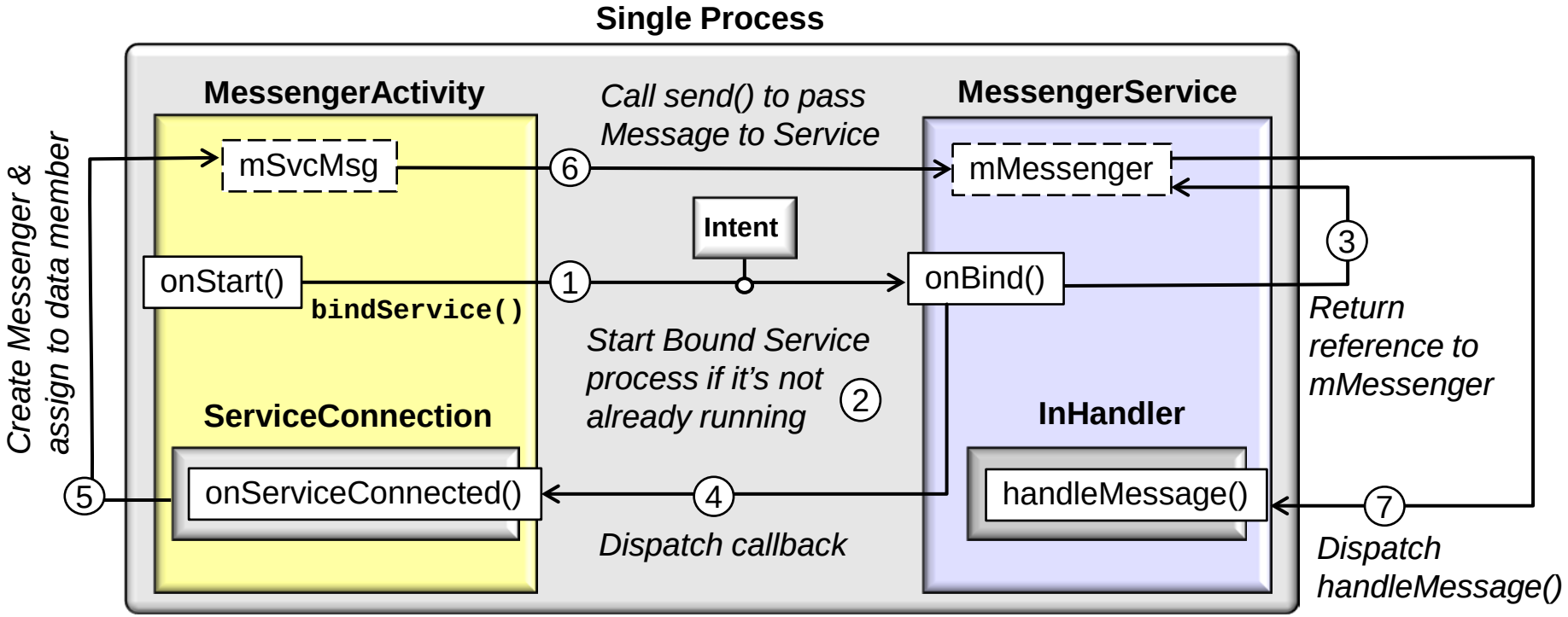
- If an Activity needs to communicate with a Bound Service a Messenger can provide a message-passing interface for this Service
  - This technique makes it easy to perform inter-process communication (IPC) without the need to use AIDL



Some additional programming is required to use Messengers for IPC

Summary

- If an Activity needs to communicate with a Bound Service a Messenger can provide a message-passing interface for this Service
- A Messenger queues the incoming send() calls, which allows the Service to handle one call at a time without requiring thread-safe programming



If your Service must be multi-threaded then you'll need AIDL (covered next)

Android Services & Local IPC: Advanced Bound Service Communication – Overview of the AIDL & Binder RPC

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

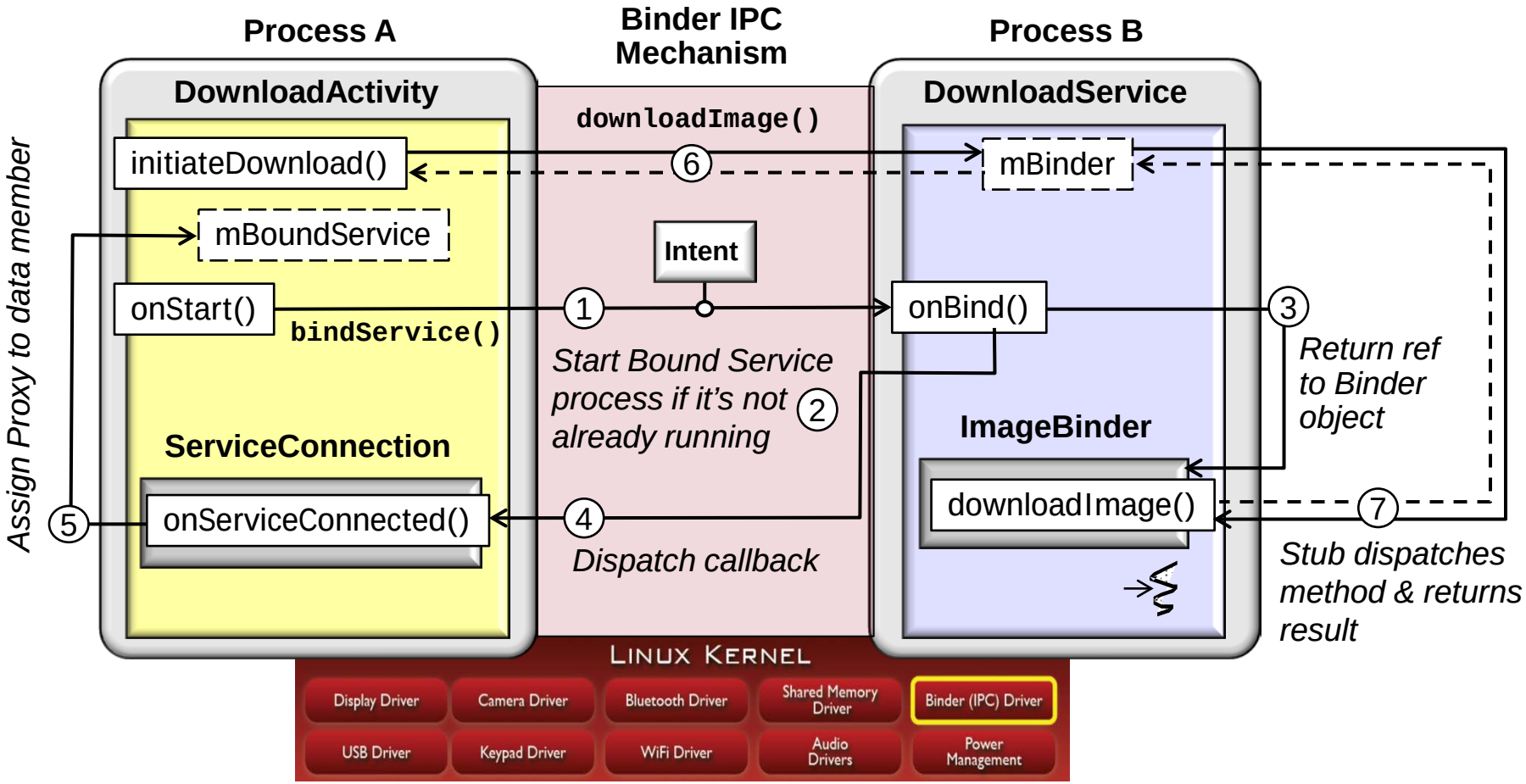
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Module

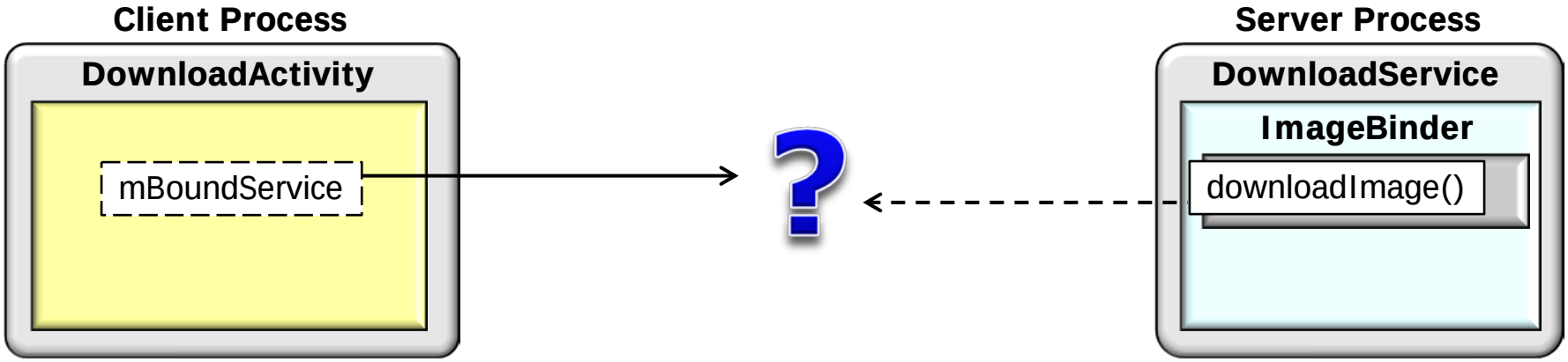
- Understand AIDL & Binder RPC mechanisms for communicating with Bound Services



AIDL & Binder RPC are the most powerful Android local IPC mechanism

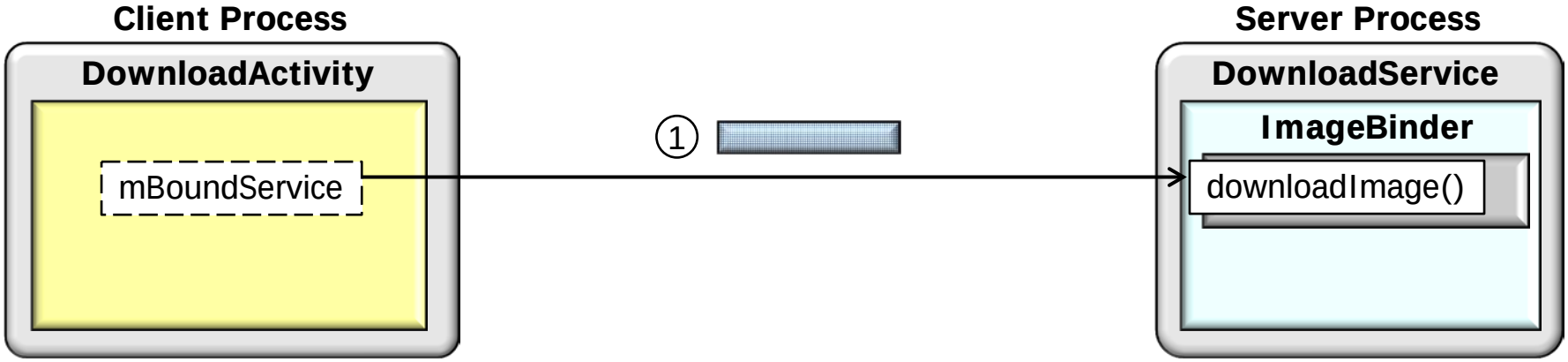
Motivation for AIDL & Binder RPC

- One process on Android cannot normally access the address space of another process
  - Our two previous examples of communicating with Bound Services side-stepped this issue by collocating the Activity & the Service in the same process address space



Motivation for AIDL & Binder RPC

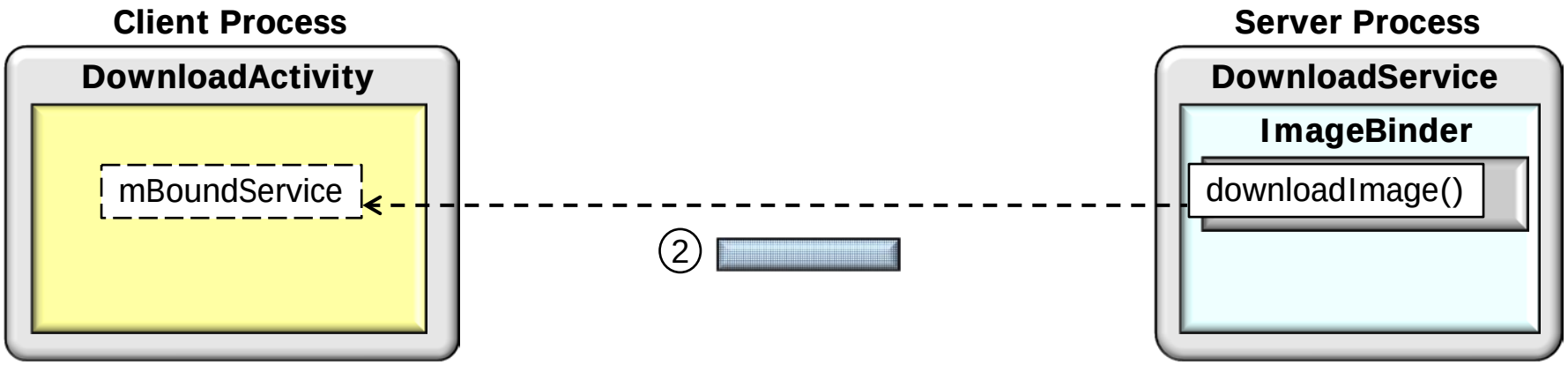
- One process on Android cannot normally access the address space of another process
- To communicate therefore they need to decompose their objects into primitives that the operating system can understand & (de)marshal the objects across the process boundary
  - Marshaling converts data from native format into a linearized format



en.wikipedia.org/wiki/Marshalling_(computer_science) has more info

Motivation for AIDL & Binder RPC

- One process on Android cannot normally access the address space of another process
- To communicate therefore they need to decompose their objects into primitives that the operating system can understand & (de)marshal the objects across the process boundary
  - Marshaling converts data from native format into a linearized format
  - Demarshaling converts data from the linearized format into native format

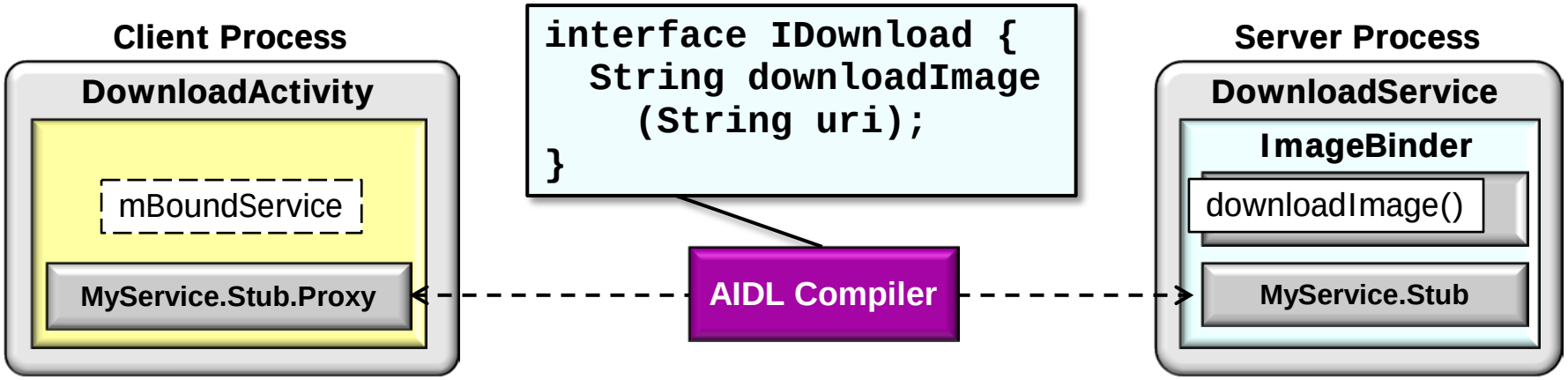


Motivation for AIDL & Binder RPC

- One process on Android cannot normally access the address space of another process
- To communicate therefore they need to decompose their objects into primitives that the operating system can understand & (de)marshal the objects across the process boundary
- The code to (de)marshal is tedious to write, so Android automates it with the Android Interface Definition Language (AIDL) & an associated compiler
  - AIDL is similar to Java interfaces

Motivation for AIDL & Binder RPC

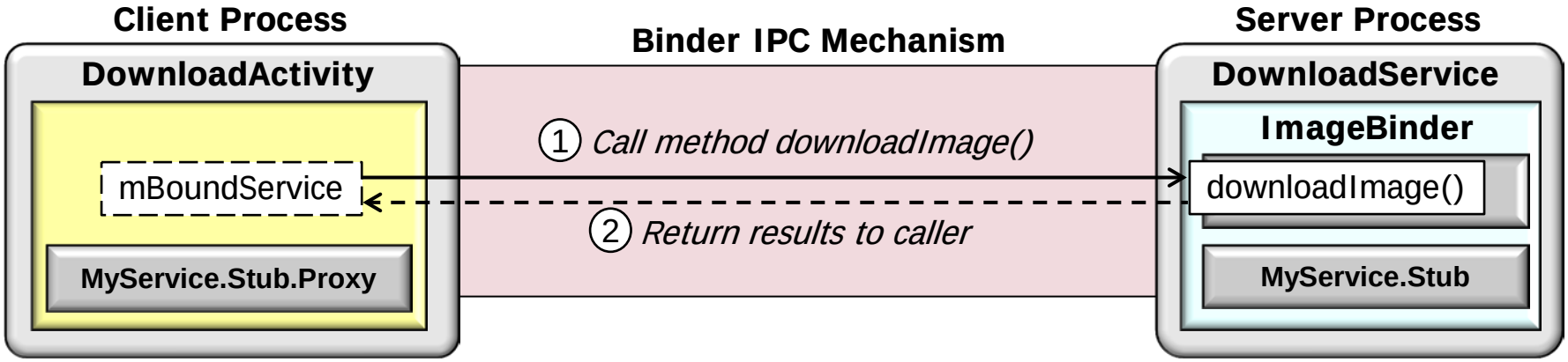
- One process on Android cannot normally access the address space of another process
- To communicate therefore they need to decompose their objects into primitives that the operating system can understand & (de)marshal the objects across the process boundary
- The code to (de)marshal is tedious to write, so Android automates it with the Android Interface Definition Language (AIDL) & an associated compiler
  - AIDL is similar to Java interfaces
  - Compilation is handled automatically by Eclipse



developer.android.com/guide/components/aidl.html has AIDL overview

Motivation for AIDL & Binder RPC

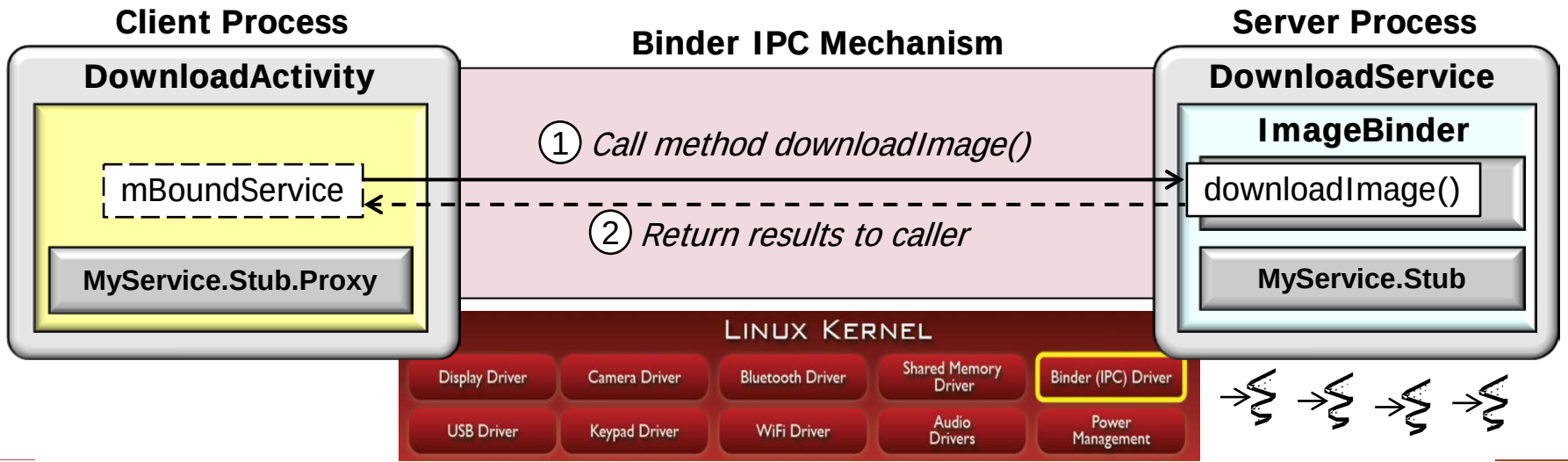
- One process on Android cannot normally access the address space of another process
- To communicate therefore they need to decompose their objects into primitives that the operating system can understand & (de)marshal the objects across the process boundary
- The code to (de)marshal is tedious to write, so Android automates it with the Android Interface Definition Language (AIDL) & an associated compiler
- The Android Binder provides a local RPC mechanism for cross-process calls
  - Apps rarely access the Binder directly, but instead use AIDL Stubs & Proxies



elinux.org/Android_Binder has more info on Android Binder RPC

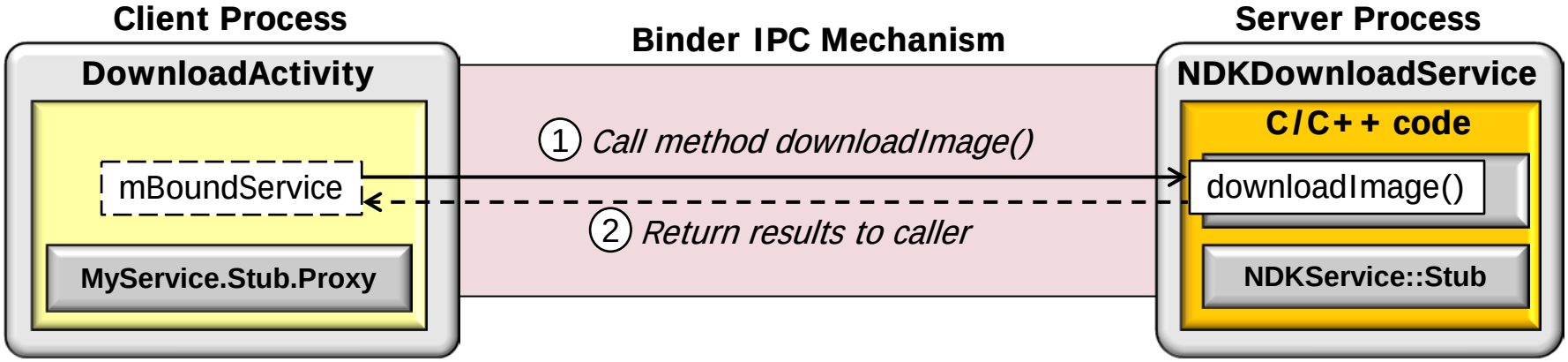
Details of Android Binder & AIDL IPC

- The Binder Driver is installed in the Linux kernel to accelerate IPC
  - It uses shared memory & per-process thread pool for high performance



Details of Android Binder & AIDL IPC

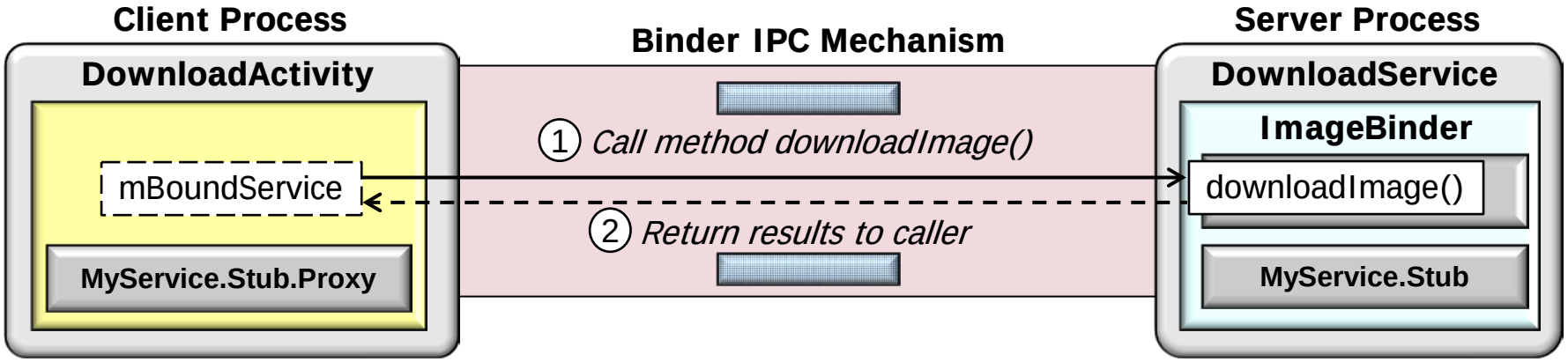
- The Binder Driver is installed in the Linux kernel to accelerate IPC
- Android (system) Services can be written in C/C++, as well as Java



sites.google.com/site/io/anatomy--physiology-of-an-android has more info

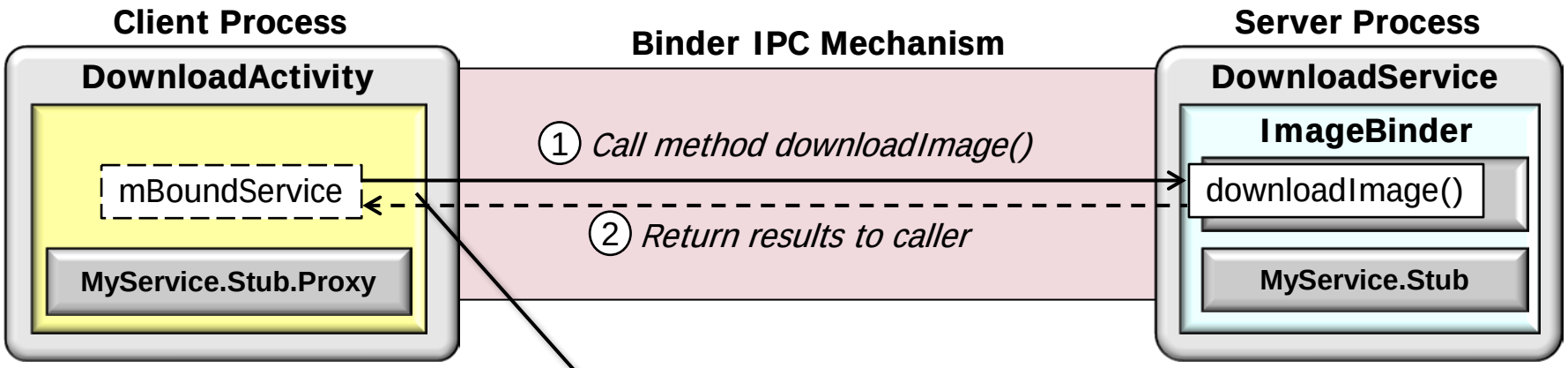
Details of Android Binder & AIDL IPC

- The Binder Driver is installed in the Linux kernel to accelerate IPC
- Android (system) Services can be written in C/C++, as well as Java
- Caller's data is marshaled into parcels, copied to callee's process, & demarshaled into what callee expects



Details of Android Binder & AIDL IPC

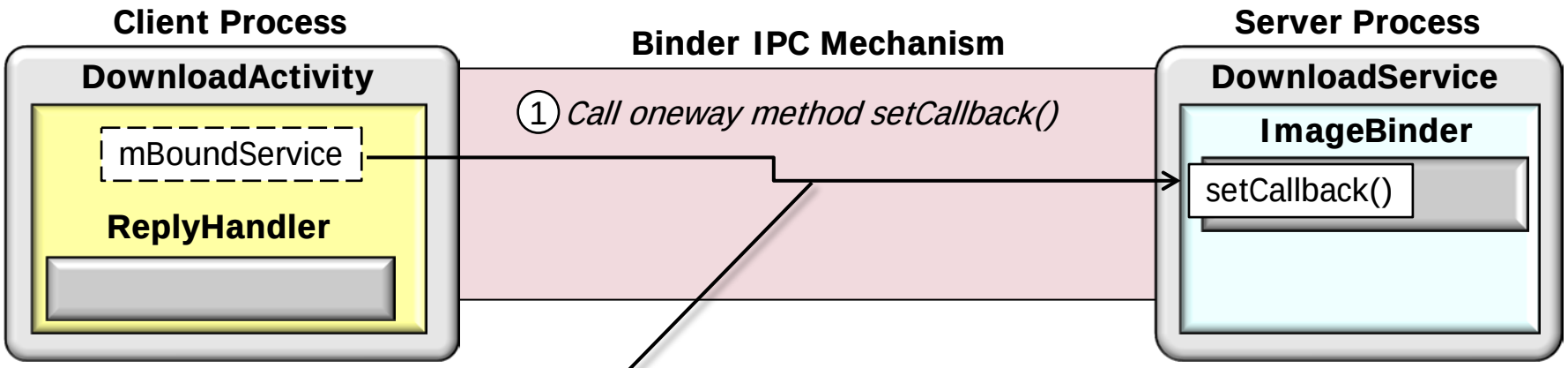
- The Binder Driver is installed in the Linux kernel to accelerate IPC
- Android (system) Services can be written in C/C++, as well as Java
- Caller's data is marshaled into parcels, copied to callee's process, & demarshaled into what callee expects
- Two-way method invocations are synchronous (block the caller)
  - One-way method invocations do not block the caller



Caller thread blocks waiting for results from the Service

Details of Android Binder & AIDL IPC

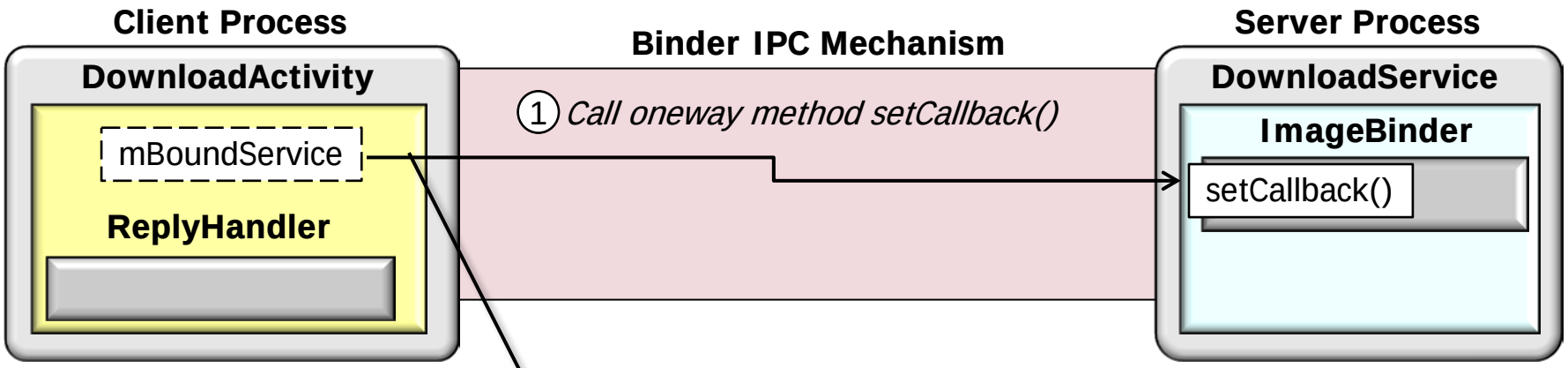
- The Binder Driver is installed in the Linux kernel to accelerate IPC
- Android (system) Services can be written in C/C++, as well as Java
- Caller's data is marshaled into parcels, copied to callee's process, & demarshaled into what callee expects
- Two-way method invocations are synchronous (block the caller)
- Android also supports asynchronous calls between processes
  - Implemented using one-way methods & callback objects



Client passes callback object via oneway method to Service

Details of Android Binder & AIDL IPC

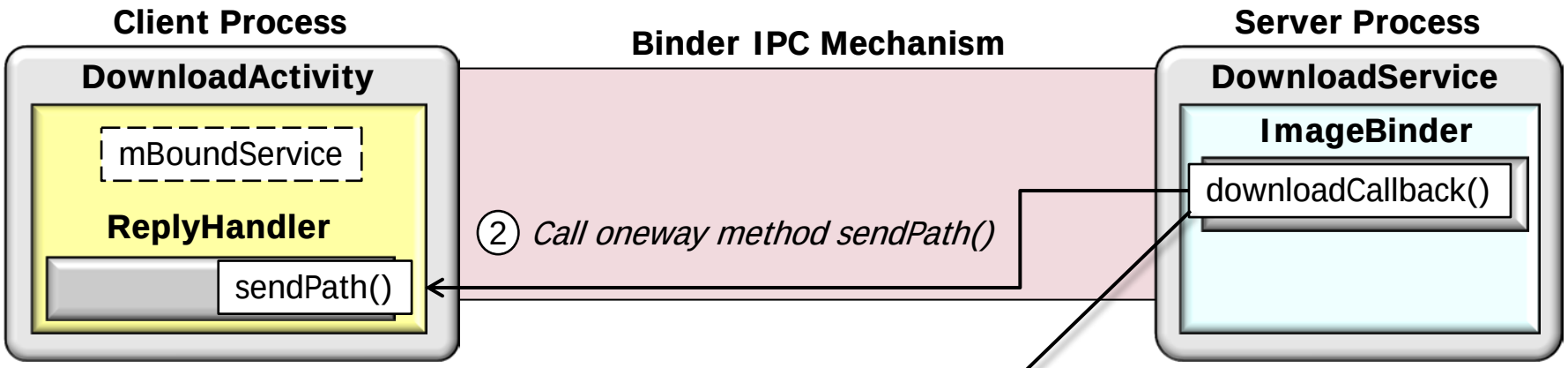
- The Binder Driver is installed in the Linux kernel to accelerate IPC
- Android (system) Services can be written in C/C++, as well as Java
- Caller's data is marshaled into parcels, copied to callee's process, & demarshaled into what callee expects
- Two-way method invocations are synchronous (block the caller)
- Android also supports asynchronous calls between processes
  - Implemented using one-way methods & callback objects



Caller thread doesn't block waiting for results

Details of Android Binder & AIDL IPC

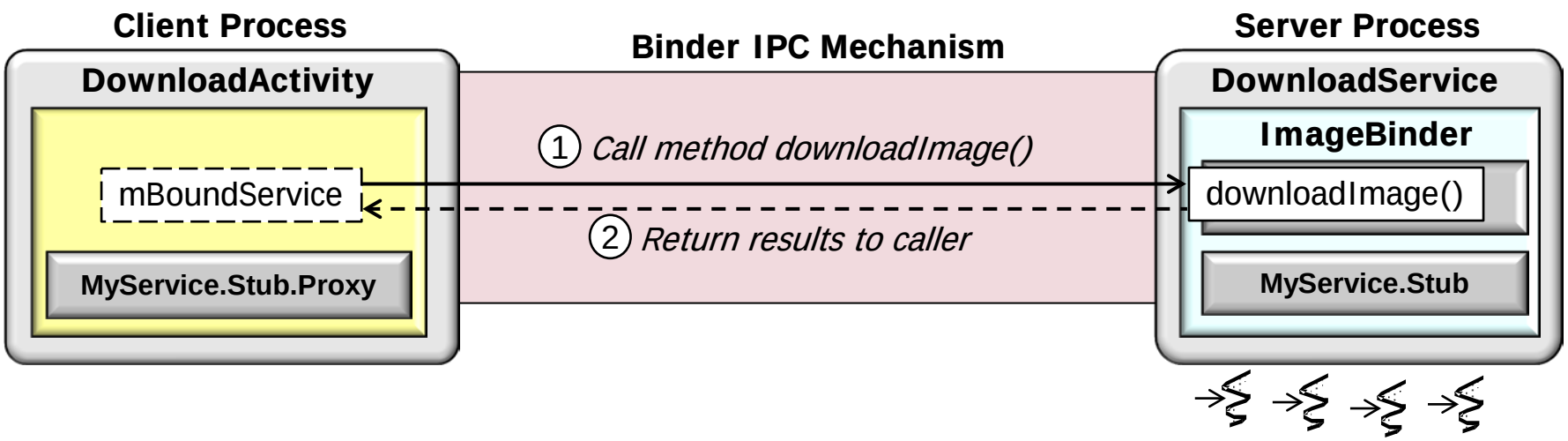
- The Binder Driver is installed in the Linux kernel to accelerate IPC
- Android (system) Services can be written in C/C++, as well as Java
- Caller's data is marshaled into parcels, copied to callee's process, & demarshaled into what callee expects
- Two-way method invocations are synchronous (block the caller)
- Android also supports asynchronous calls between processes
  - Implemented using one-way methods & callback objects



Service invokes a one-way method to return results

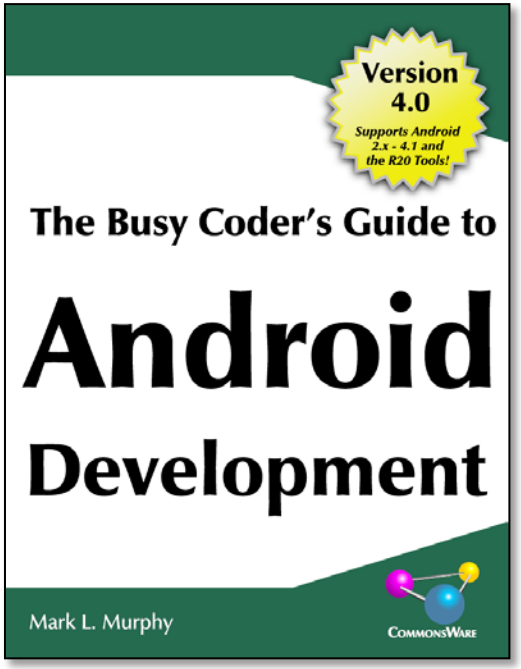
Details of Android Binder & AIDL IPC

- The Binder Driver is installed in the Linux kernel to accelerate IPC
- Android (system) Services can be written in C/C++, as well as Java
- Caller's data is marshaled into parcels, copied to callee's process, & demarshaled into what callee expects
- Two-way method invocations are synchronous (block the caller)
- Android also supports asynchronous calls between processes via callbacks
- Server typically handles one- & two-way method invocations in a thread pool
  - Service objects & methods must therefore be thread-safe



Summary

- Android provides a wide range of local IPC mechanisms for communicating with Bound Services

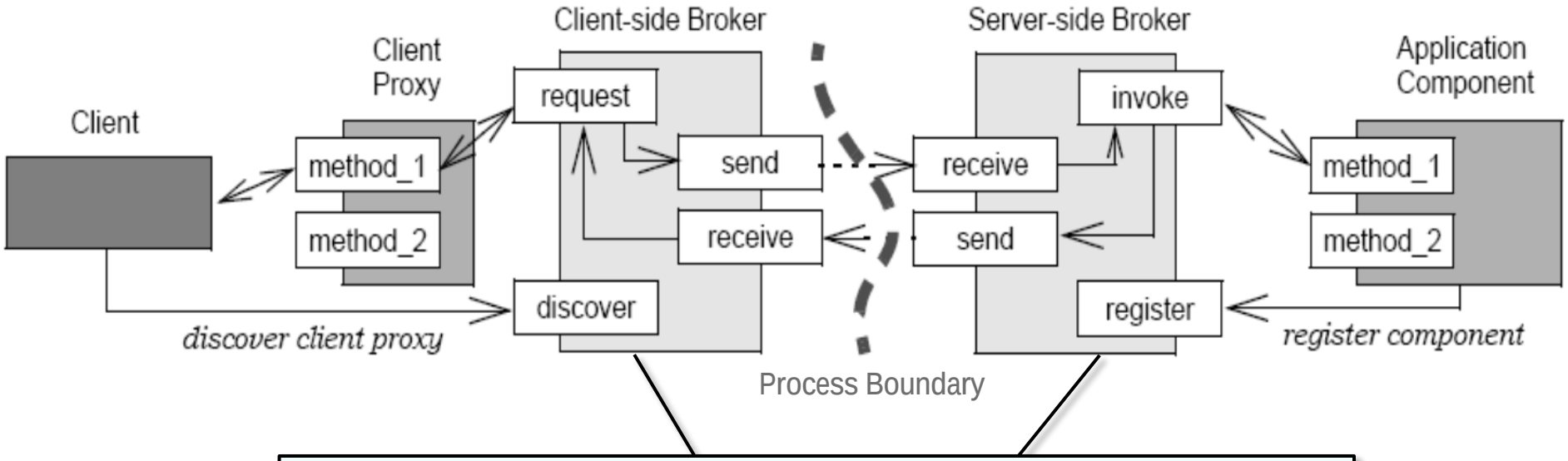


www.vogella.com/tutorials.html

There are many Android tutorials & resources available online

Summary

- Android provides a wide range of local IPC mechanisms for communicating with Bound Services
- AIDL is a language for defining Binder-based interfaces to Bound Services
  - It's used with the Binder RPC mechanism to implement the *Broker* pattern

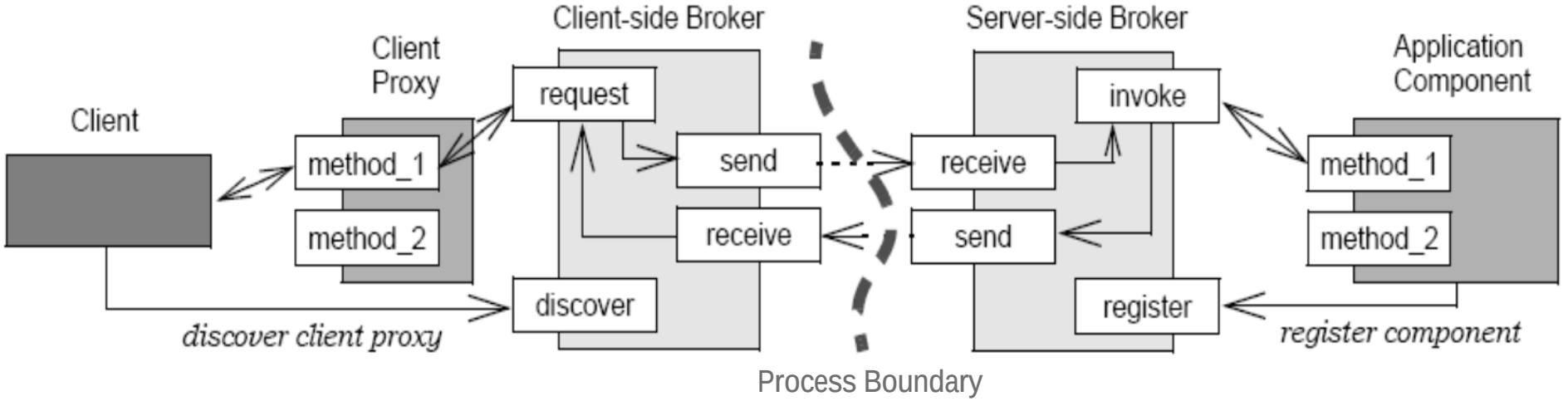


Broker connects clients with remote objects by mediating invocations from clients to remote objects, while encapsulating the details of IPC or network communication

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

Summary

- Android provides a wide range of local IPC mechanisms for communicating with Bound Services
- AIDL is a language for defining Binder-based interfaces to Bound Services
  - It's used with the Binder RPC mechanism to implement the *Broker* pattern



- Many other patterns are used to implement AIDL & Binder RPC
  - e.g., *Proxy*, *Adapter*, *Activator*, etc.