Architecture of the CORBA Component Model

C++ Language Mapping: Server Side

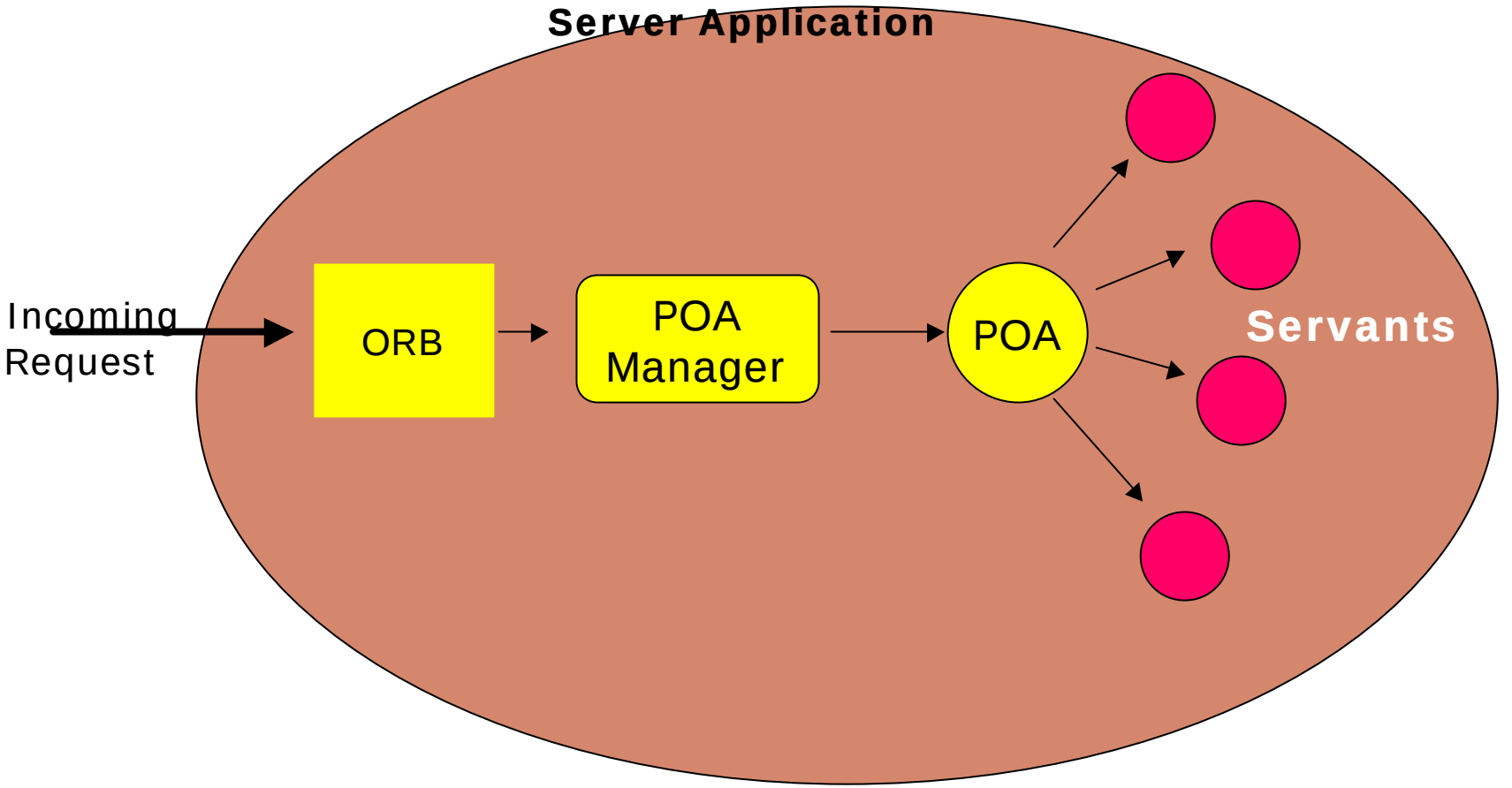
Introduction

The server-side mapping is a generalization of the client-side mapping. The developer needs to

- connect servants to skeletons
- receive parameters and return results
- create object references for objects
- initialize the server-side run-time
- start an event loop for processing requests

Interaction of servants with the ORB is controlled through the object adapter (OA). The Portable Object Adapter (POA) is ORB-independent.

Flow of Incoming Requests



Object Implementations

- IDL compiler generates skeleton class for each interface
- Skeleton class has prefix POA
  - Interface ::X maps to class ::POA_X
  - Interface M::X maps to class POA_M::X

```
// IDL
interface X{
   long get_value();
};
// C++
class POA_X: public virtual PortableServer::ServantBase
   { public:
       virtual CORBA::Long get_value()
   throw(CORBA::SystemException)=0;
};
```

Object Implementations (2)

- Servant classes inherit from skeleton classes
  - Implementation with delegation is also possible
  - Inheritance must be virtual

```
class MyXImpl: public virtual POA_X {
public:
   MyXImpl(CORBA::Long initial): m_value(initial){}
   virtual CORBA::Long get_value()
   throw(CORBA::SystemException);
private:
   CORBA::Long m_value;
   // prevent copying and assignment
   MyObject_impl(const MyObject_impl&);
   void operator=(const MyObject_impl&);
};
```

POA: Portable Object Adapter

Replaces Basic Object Adapter (BOA) of CORBA 2.1. It specifies details of object activation, and allows in a flexible way

- to assign object references to servants
- to transparently „activate“ of servants
- to assign „policies“ to servants

The definition of the POA interfaces itself is in IDL:

- POA interfaces are „local“ interfaces
- C++ implementation objects (servants) have the IDL type **native**

POA Architecture

- Servant: Implementation object, determines run-time semantics of one or more CORBA objects
- ObjectID: unique identification of object within a POA (type: sequence<octet>)
- Active Object Map: table associating ObjectID values and servants
- Incarnate: The action of creating or specifying a servant for a given ObjectID
- Etherealize: The action of detaching a servant from an ObjectID
- Default Servant: Servant that is associated with all ObjectID values not mentioned in the active object map

POA Functions

- Each POA defines a namespace for servants.
- All servants within a POA have the same implementation characteristics (policies). The Root POA has a standardized set of policies.
- Each (active) servant is associated to a POA.
- POAs determine the relevant servant upon incoming requests, and invoke the requested operation at the servant.

Functions of the POA Manager

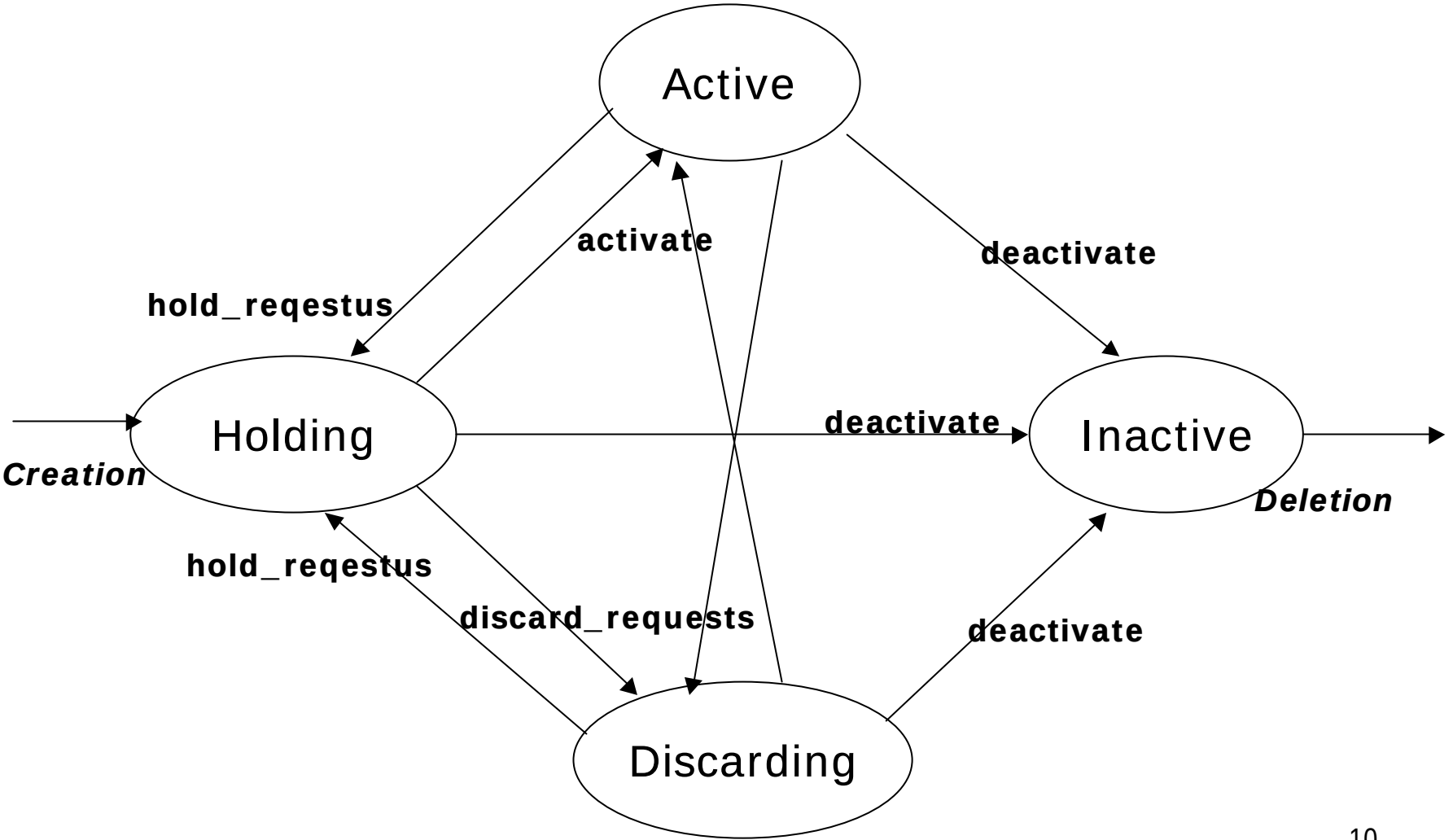
Each POA is assigned a POA manager, which is set when the POA is created.

The POA manager controls the flow of requests for one or multiple POAs

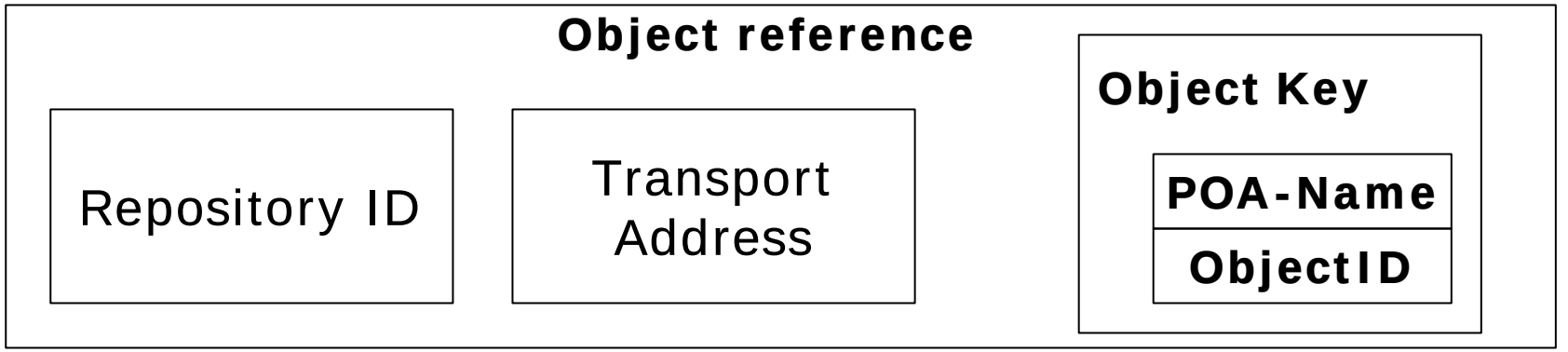
The POA manager always has one of the following states:

- Active: Requests are processed
- Holding: Reqests are saved
- Discarding: Requests are refused with the TRANSIENT exception
- Inactive: Requests are rejected, connections are shut down

POA Manager State Transitions



Structure of an Object Reference



Conceptually, an object reference contains:

- repository ID (optional, contains type information)
- transport address (denotes transport endpoint)
- object key (identifies POA and object within POA)

The object key is in a proprietary format, specific to each ORB

Object Incarnation

- Association of an object reference with a servant
- Two-way association:
  - Given the servant, create an object reference: Activation
  - Givent the object reference, find a servant: Incarnation
- POA interface provides operations for activation; policies decide what modes of activation and incarnation are supported
- C++ mapping adds _this

```
MyXImpl servant(17);
MX_var object = servant._this();
```

- In the default case (POA of servant is RootPOA), _this does:
  - Create an object in the RootPOA (generating a transient ObjectID)
  - Associate the ObjectID with the servant in the active object map of the root POA
  - Create an object reference for this object
  - Return the object reference

Initializing the Server-Side ORB Runtime

- Initialize the ORB: **`CORBA::ORB_init(/* ... */);`**
- Get a reference to the RootPOA: **`orb->resolve_initial_references("RootPOA");`**
- Narrow the RootPOA to PortableServer::POA
- Obtain the servant manager of the root POA: **`poa->the_POAManager();`**
- Activate the servant manager: **`poa->activate();`**
- Create a servant instance
- Activate the servant: **`servant->_this();`**
- Start the ORB mainloop: **`orb->run();`**

Parameter Passing to Servants

- Simple Types: pass by value or reference (depending on direction)
- Fixed-Length complex types: pass by (possibly const) reference (use _out type for output parameters)
- Strings: pass as char* value or reference (use String_out for output parameters)
- Complex types & **any**:
  - in: Pass by const reference
  - out: Pass by reference
  - out: Use _out type
  - result: Return pointer
- Object references: Pass as _ptr value or reference

Returning Sequences Pitfall

Given the IDL

```
typedef sequence<long> LongSeq;
```

Programmers are tempted to return the value as

```
LongSeq * result = new LongSeq;
result->length(2);
result[0] = 1234;    // wrong
result[1] = 5678;    // wrong
return result;
```

Correction: Use LongSeq_var; return result._retn()

Raising Exceptions

- throw exceptions by value, catch them by reference
  - Exceptions expect member fields in constructor
  - Exception will assume ownership of parameters
- Throwing system exceptions is allowed
  - As a usage guideline, it should be avoided if possible
- ORB will deallocate all in and inout parameters

Further POA Features

- Policies (will discuss in detail later)
- Ties (not presented here)