CptS 464/564
Lecture 11

- Today we're on the "practice" track
  - Using the CORBA Naming Service
  - Using CORBA Exceptions
  - Project 2 Description
- Resume the "theory" track next time

Why use the Naming Service?

- Obtaining IOR's is hard unless you share file space
- IOR's aren't very human-friendly
- IOR's may be temporary
- …

```
// Find the Naming Service
   CORBA::Object_var naming_obj =
     orb->resolve_initial_references( "NameService" );
   CosNaming::NamingContext_var root =
     CosNaming::NamingContext::_narrow( naming_obj.in() );
   if( CORBA::is_nil( root.in() ) ) {
     cerr << "Nil Naming Context reference" << endl;
     throw 0;
```

How does resolve_initial_references resolve “NameService”?

Command-line switch –ORBInitRef NameService=corbaloc::nif-s1:26000/NameService

```
// Resolve the Messenger object
    CosNaming::Name name;
    name.length( 2 );
    name[0].id = CORBA::string_dup( "example" );
    name[1].id = CORBA::string_dup( "Messenger" );
    CORBA::Object_var obj = root->resolve( name );

    // Narrow
    Messenger_var messenger = Messenger::_narrow( obj.in() );
    if( CORBA::is_nil( messenger.in() ) ) {
      cerr << "Not a Messenger reference" << endl;
      throw 0;
    }
```

Notice that a name has two parts – like a filename basepart and extension. You can use any strings you want for these. Also, notice that a hierarchical namespace can be created in the nameserver – we aren't using that feature here.

```
// Bind the Messenger object
  name.length( 2 );
  name[0].id = CORBA::string_dup( "example" );
  name[1].id = CORBA::string_dup( "Messenger" );

  PortableServer::ObjectId_var oid =
    poa->activate_object( messenger_servant );
  CORBA::Object_var messenger_obj = poa->id_to_reference( oid.in() );
  try {
    root->rebind( name, messenger_obj.in() );
  }
  catch ( const CosNaming::NamingContext::NotFound & ) {
    cout << "Can't bind example/Messenger" << endl;
  }
```

Note in the MessengerName example code also creates a dummy new naming context (“example”) which is not used. The pair (“example”, “Messenger”) here is bound in the root context of the name service.

Other techniques using orb->string_to_object()

orb->string_to_object(“IOR:xxxxxxx…”)

orb->string_to_object(“file://<absolute path name>”)

orb->string_to_object(“corbaloc:[iiop]:host:port/object-key

orb->string_to_object(“corbaname:[iiop]:host:port/NameService#[context]/key

Any or all of these could be applicable in doing the project!

CORBA Exceptions

- Why use exceptions?
  - Avoid cluttering programs with code to handle unusual cases
  - Because they are dynamically scoped, allow handling problems at upper layers of program for all code “below”
  - Maybe controversial, especially in CORBA where out parameters are supported

When to use exceptions

- Unusual cases
- Cases where common error handling can be beneficial
- Warning: in the example that follows, the case for using an exception is weak!

```
// messenger.idl

interface Messenger
   {
      exception ImNotHere {
         string reason;
      };

      boolean send_message ( in    string user_name,
                             in    string subject,
                             inout string message ) raises (ImNotHere);
   };
```

Notice the declaration of the exception: exceptions can carry values explaining why they were raised.

Notice the addition of the exception to the method.

```
CORBA::String_var reason = CORBA::string_dup("Go talk to someone else");
throw Messenger::ImNotHere(reason);
```

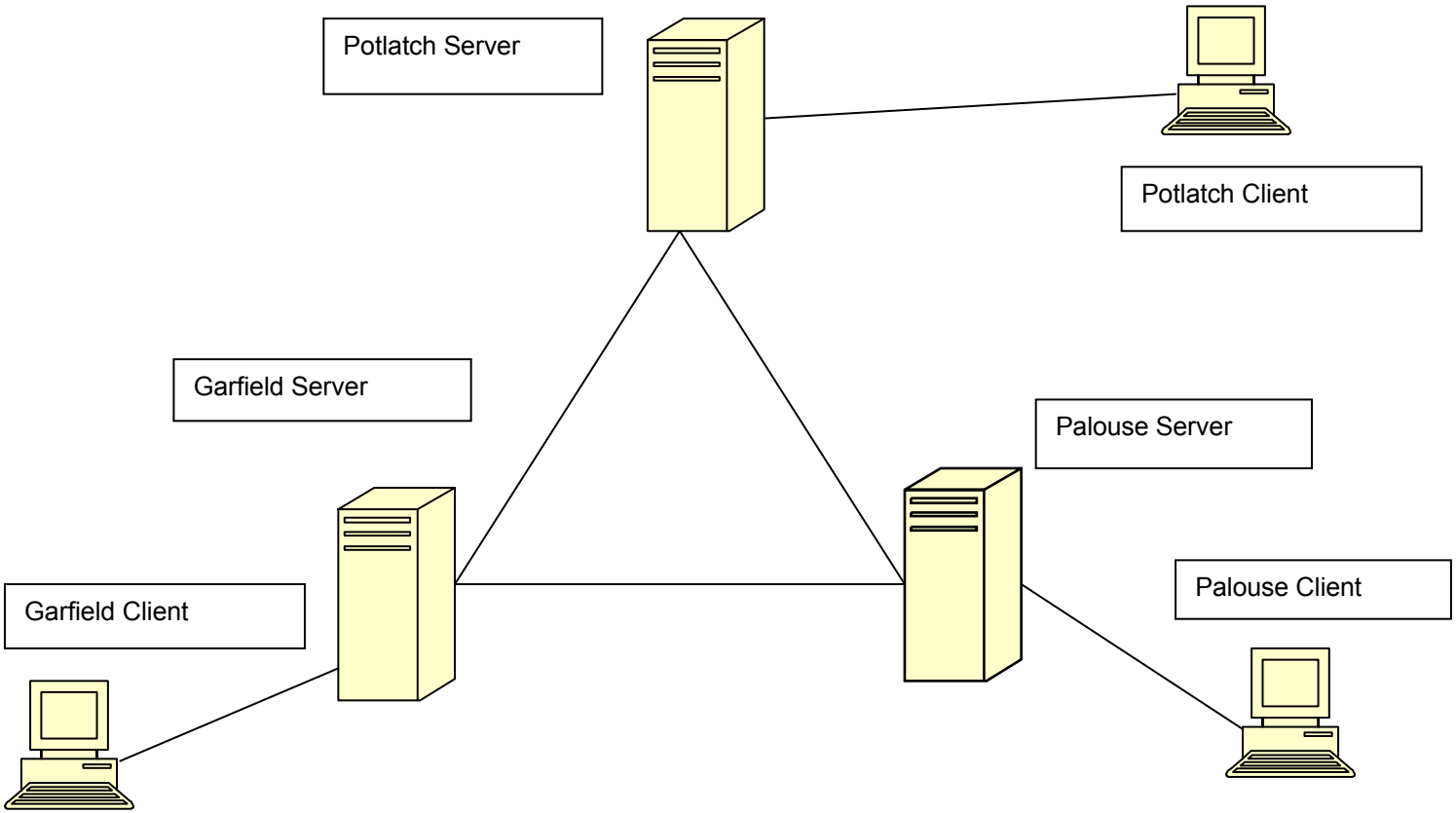
Raising an exception: note the use of a String_var variable and string_dup – be careful of your storage management issues.

```
{
try { … }

 catch( const Messenger::ImNotHere &inhEx ) {
    cerr << "Caught an ImNotHere exception: " << inhEx << endl;
    cerr << "reason: " << inhEx.reason << endl;
    return 1;
    }

  catch( const CORBA::COMM_FAILURE &commEx ) {
    cerr << "Caught a COMM_FAILURE: " << commEx << endl;
    return 1;
    }

  catch( const CORBA::Exception &ex ) {
    cerr << "Uncaught CORBA exception: " << ex << endl;
    return 1;
    }
}
```


Potlatch Server
Potlatch Client
Garfield Server
Palouse Server
Garfield Client
Palouse Client

```
// Library.idl
module Project2  {
  interface Library {
    typedef long BookIDType;
    typedef short Date;
      // Dates are used for due dates, and are treated abstractly.
      // A date is just the number of days from the start of
      // running of the system.
    struct BookInfo {
      BookIDType bookID;
      string bookTitle;
    };
    exception CheckedOut {
        Date dueDate;
    };
    exception NotInCollection {
      // no further data needed
      // raised when an operation is attempted on a book that
      // is not in the library's collection
    };
    exception NotMine {
      // no further data needed
      // raised when an attempt is made to check in a book to
      // the wrong library
    };
```

```
// Methods used by clients on their local library.
// The local library may make further calls on
// other libraries to resolve a client's request
boolean queryBook(in BookIDType bookID, out string libName)
        raises (NotInCollection);
  // returns true if available (i.e. checked in),
  // false if it is part of the collection of some library but not available
  // sets libName to the name of a library that has the book in its
collection
  // raises NotInCollection if the book ID is not known at any library
string checkoutBook(in BookIDType bookID)
       raises (NotInCollection, CheckedOut);
  // if successful, returns the library name from which the book was checked
out
  // raises NotInCollection if the book is not part of any library's
collection
  // raises CheckedOut if the book is in some library's collection
  // yet it is checked out everywhere
  // The due date in the exception must be the earliest due date at any
library.
void returnBook(in string libraryName, in BookIDType bookID)
     raises (NotInCollection);
  // raises NotMine if the named library doesn't own the book,
  // or its copy is not checked out
BookInfo getBookInfo(in BookIDType bookID) raises
(NotInCollection);
  // get the info for a book; raise the exception if it is not in any
collection
};
```

```
// The library serving these calls never makes further calls
// in order to resolve them. The listed exceptions are raised
// if the call cannot be satisfied locally.

boolean queryBookLocal(in BookIDType bookID);
  // Tests whether a book is available (i.e., checked in)
  // in the library serving the call. If so, returns true;
  // returns false
  // if it is part of the local collection but not checked in.
  // raises NotInCollection if it is not part of the local
collection

void checkoutBookLocal(in BookIDType bookID)
     raises (NotInCollection, CheckedOut);
  // Checks out a book from the library serving the call.

void returnBookLocal(in BookIDType bookID) raises(NotMine);
  // Returns a book to the library serving the call.

BookInfo getBookInfoLocal(in BookIDType bookID)
         raises (NotInCollection);
  // Gets info for a book from the library serving the call.
```