Overview

• Lots of details here, very few testable
• Goal is to let you understand better middleware plumbing
• I will carefully articulate in the last lecture what you are responsible for on the final exam
CORBA Features

• Transparencies
  – Programming language
  – CORBA vendor
  – Operating Systems
  – Location
  – Network HW/SW
  – Access

• Dynamic Binding

• Dynamic Typing

• Object Orientation
  – Encapsulation
  – Polymorphism
  – Inheritance

• Instantiation

• Extended Services
  – Naming/trader
  – Events/notification
  – Transactions
  – Security, domains
  – ……

In an open specification with multivendor support

CORBA: A “Software Bus”

- Hardware bus: lets chips “plug and play” in a standard way
- Software bus: same idea but for software objects
Object Management Architecture (OMA)

- **Object Services**: useable by all objects
  - Events, Trader, Security, Naming, Transactions, …

- **Common Facilities**: useable by all applications
  - Scripting, compound documents, …. 

- **Domain interfaces**: industry-specific APIs
  - Finance, telecom, …. 

- **Application Interface**:
  - What you provide…. 

**ORB (Object Request Broker)**
ORB, Proxies, and POA

- Object Proxy (aka stub)
- Skeleton
- Portable Object Adaptor (POA)
- Client

Same Interface!

• Note: POA has a tree of objects, starting with root “/” (more later…)
Notes on this Example

• Showing generic CORBA application
  – Should work with any CORBA vendor’s implementation
  – Exact steps, command names, etc. for TAO are mentioned in documentation (Chapter.3)

• Lots of low-level details, esp. on server side
  – You will just mimic code fragments for the different steps (startup, etc)
  – Basically the same, just the interface/class name is different, for almost all of what you will do here
CORBA C++ App. Development Steps

1: Object Spec. in IDL

2: Run IDL Compiler

3: Add client program code

4: Add object implementation & Server code

5: Run C++ Compiler (MPC)

5: Run C++ Compiler (MPC)

6: Start server program

7: Start client program

Object Request Broker (ORB)
interface Messenger {
    boolean send_message ( 
        in string user_name, 
        in string subject, 
        inout string message );
};
Another IDL Example (no more details...)

module BankExample {

    interface Account {
        exception BadCheck {float fee;};

        float deposit(in float amount);
        float writeCheck(in float amount, in long checknum)
            raises (BadCheck);
    }

    interface AccountManager {
        Account openAccount(in string name);
    }

};

Note 1) Module-namespace control 2) exceptions 3) Obj ref
2: Run IDL Compiler

- Compiles Messenger.idl
- Generates client-side code
- starter implementation class in MessengerI.*
- Client-side stubs in MessengerC.*
- Server-side skeletons in MessengerS.*
- Reminder: “server” is a process/program, “servant” is a running piece of code that provides functionality for an object reference
IDL to C++ mapping notes

- IDL isolates implementation from interface
  - Allows clients and servants to be in different languages
  - Allows for an interface to not be bound to a single implementation
    - Useful for many server-side optimizations
    - E.g., multiple servants can service requests to a single object reference
  - Allows for code to be inserted above ORB while still meeting stub API
    - E.g., QuO delegates
    - Caching
- Mappings of Basic IDL types (see CORBA manual for details)
# IDL to C++ mapping notes

<table>
<thead>
<tr>
<th>IDL type</th>
<th>Java</th>
<th>C++</th>
<th>CORBA C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>short</td>
<td>short</td>
<td>CORBA::Short</td>
</tr>
<tr>
<td>long</td>
<td>int</td>
<td>---¹</td>
<td>CORBA::Long</td>
</tr>
<tr>
<td>unsigned long</td>
<td>int</td>
<td>unsigned long</td>
<td>CORBA::ULong</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>float</td>
<td>CORBA::Float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>double</td>
<td>CORBA::Double</td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
<td>unsigned char</td>
<td>CORBA::Boolean</td>
</tr>
<tr>
<td>long long</td>
<td>long</td>
<td>---¹</td>
<td>CORBA::LongLong</td>
</tr>
</tbody>
</table>

1: Platform dependent: use the standardized CORBA types for portability
// C++
#include "MessengerC.h"
#include <iostream>
int main(int argc, char* argv[]) 
{
  try {
    // Initialize the ORB.
    CORBA::ORB_var orb =
      CORBA::ORB_init(argc, argv);
    // Read Messenger object's IOR.
    CORBA::Object_var obj = orb-
      >string_to_object("file://Messenger.io
      r");
    if( CORBA::is_nil(obj.in())) {
      std::cerr << "Could not get Messenger IOR." << std::endl;
      return 1;
    }
    Messenger_var messenger =
      Messenger::_narrow(obj.in());
    if( CORBA::is_nil(messenger.in())) {
      std::cerr << "IOR was not a Messenger object reference." << std::endl;
      return 1;
    }
    // Send a message
    CORBA::String_var msg =
      CORBA::string_dup("Hello!");
    messenger->send_message("TAO User",
      "Test", msg.inout());
    // Print the Messenger's reply.
    ..
  } catch (CORBA::Exception & ex) {...} 
}
4A: Add Object Implementations
(Messenger_impl.h)

2: Run IDL Compiler

Client-Side Generated Code

Includes Messenger class in MessengerC.h

Server-Side Generated Code

Includes POA_Messenger class in MessengerS.h

MessengerI class written by programmer (4A) and used by server
4A: Add Object Implementations (cont.)

#include "Messenger_i.h" // renamed from Messengerl.h
#include <iostream>
CORBA::Boolean Messenger_i::send_message (const char* user_name, const char* subject, char*& message)
ACE_THROW_SPEC ((CORBA::SystemException))
{
    std::cout << "Message from: " << user_name << std::endl;
    std::cout << "Subject: " << subject << std::endl;
    std::cout << "Message: " << message << std::endl;
    CORBA::string_free(message);
    message = CORBA::string_dup("Thanks for the message.");
    return 1;
}
4B: Implement the Server(Server.cpp)

1. Initializes the ORB
2. Creates a Portable Object Adaptor (POA).
3. Creates a (C++) MessengerI object. Note that this is not yet CORBA object.
4. Instantiates a CORBA object from the local one
5. Writes the IOR for the impl objects to a file.
6. Activates the POA manager (and the POA)
7. Waits for incoming requests

Note: Lotsa details, you will be just copying and changing..
#include "Messenger_i.h"

int main(int argc, char* argv[]) {
    try {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // Get POA
        PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

        // Activate the POAManager.
        PortableServer::POAManager_var mgr = poa->the_POAManager();
        mgr->activate();

        // Create a servant.
        Messenger_i servant; a reference to the RootPOA.

        // Register the servant & write it to a file.
        PortableServer::ObjectId_var oid = poa->activate_object(&servant);
        obj = poa->id_to_reference(oid.in());
        CORBA::String_var str = orb->object_to_string(obj.in());
        ofstream iorFile("Messenger.ior");
        iorFile << str.in() << std::endl;
        iorFile.close();
        std::cout << "IOR written to file Messenger.ior" << std::endl;

        // Accept requests from clients.
        orb->run();
        orb->destroy();
        return 0;}
    catch (CORBA::Exception& ex) { }
    return 1;
}
TAO Notes

• Cross platform make solution
  – Supports multiple build environment (VC++, GNU Make, etc)
• Based on using Make Project Creator (MPC)
  – Just create a .mpc file for each project
• Example:
  
  ```
  project(*Server): taoexe, portableserver {
    Source_Files {
      Messenger_i.cpp
      MessengerServer.cpp
    }
  }
  
  project(*Client): taoexe {
    Source_Files {
      MessengerC.cpp // prevents implicit MessengerS.cpp
      MessengerClient.cpp
    }
  }
  ```

More CORBA Topics

- Example with naming service
- User-defined exceptions
- Overview of CORBA hooks and architectural modules
- CORBA hooks
- ORB core
- CORBA: Object
- Dynamic Invocation Interface (DII)
- Object References
- Interface Repository
- Implementation Repository
- GIOP and IIOP
Messenger Server Finding the Naming Service

// Note – you will ALWAYS just copy and tweak this code.
// For more info see TAO manual section 24.3.3

// 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;
}
// Bind the example Naming Context, if necessary
CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup( "example" ); // use userid
try {
    CORBA::Object_var dummy = root->resolve( name );
}
catch ( const CosNaming::NamingContext::NotFound & ) {
    CosNaming::NamingContext_var dummy =
        root->bind_new_context( name );
}

// … continued on next slide…
Messenger Server Binding a Name

// Bind the Messenger object
name.length( 2 );
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;
}
cout << "Messenger obj bound in Naming Service" << endl;
Messenger Client Using the Naming Service

// … find naming service just like server did, same code

// 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;
}

}
User-Defined Exceptions

- CORBA has >20 pre-defined system exceptions that you should catch with every CORBA call of any kind
  - UNKNOWN, BAD_PARAM, NO_MEMORY, IMPL_LIMIT, COMM_FAILURE, INV_OBJREF, NO_PERMISSION, …
  - Thrown by ORB implementation
- Programmers can also declare exceptions in IDL
  - Thrown by server side
Messenger IDL with Exceptions

// messenger.idl

interface Messenger { 
  exception ImNotHere { 
    string reason; 
  }; 

  boolean send_message ( 
    in    string user_name, 
    in    string subject, 
    inout string message ) raises (ImNotHere); 
};
// MessengerClient.cpp

try {
    // …. Init ORB, get IOR from file or name service, make invocations …
}

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::SystemException &sysEx) {……}
Messenger Throwing User-Level Exceptions

CORBA::Boolean
Messenger_impl::<= send_message(const char* user_name, const char* subject, char*& message)
    throw(Messenger::ImNotHere, CORBA::SystemException)
{
    if (strcmp(user_name, "The_Professor") == 0) {
        // ...
    } else if (strcmp(user_name, "Joe_Freshman") == 0) {
        // don't want to talk to a freshman--send him the ImNotHere exception
        // and tell him to talk to someone else
        cerr << "Throw ImNotHere" << endl;
        cerr << endl;
        CORBA::String_var reason =
            CORBA::string_dup("Go talk to someone else");
        throw Messenger::ImNotHere(reason);
    } else {
        // ...
    }
Major CORBA Design Principles

• Separation of interface and implementation
  – Clients depend on interfaces, not implementations
• Location transparency
  – Service use is orthogonal to service location
• Access transparency
  – Invoke CORBA objects just like local ones
• Typed interfaces
  – Object references are typed by interfaces
• Support of multiple inheritance of interfaces
  – Inheritance extends, evolves, and specialized behavior
  – Note: not implementation of multiple implementations!
• Support of multiple interaction styles
  – Client/server
    • Some support for mobile code, too, with Objects by Value (OBV)
  – Peer processes
  – Publish/Subscribe (aka “push”)
CORBA Modules & System Builders' Hooks

Interface Repository

IDL Compiler

Implementation Repository

CORBA Modules & System Builders' Hooks

ORB Core

Client

Stub/proxy (SII)

DII

ORB Interface

Smart Stub

Interceptor

Interceptor

Interceptor

Interceptor

Servant

Skeleton

DSI

Object Adaptor

ORB Core

Interceptor

Interceptor

Interceptor

Interceptor

Interceptor

Interceptor

Interceptor

Interceptor

Interceptor

Interceptor

Interceptor

Interceptor

This slide adapted from FTCS-29 Tutorial by Shalini Yajnik of Lucent Technologies

Standard Interfaces  IDL-generated  ORB-Specific
ORB Core Overview

Features (server-side)
- Connection management
- Memory management
- Request transfer
- Endpoint demuxing
- Concurrency control

Other utility methods
- `object_to_string()` and `string_to_object()`
- Etc.
CORBA: Object class

• Base class for all proxies
• Useful utility methods:
  - _is_a()
  - _is_equivalent()
  - _duplicate()
  - _release()
  - _is_local()
  - _is_remote()
• Request methods for DII (more soon…)
SII

• Static Invocation Interface (SII)
  – Most common way to use IDL
  – All operations specified in advance and known to client (by proxies/stubs) and server (by skeletons)
  – Simple
  – Typesafe
  – Efficient
  – Its what you used so far in this class
• Dynamic Invocation Interface (DII)
  – Less common way to use IDL
  – Lets clients invoke operations on objects whose IDL is not known to them at compile time (main advantage of DII)
    • Browsers of all sorts (interface browser, etc)
    • Debuggers
  – Also can use `send_deferred()` and `poll_response()`
    • asynchronous (non-blocking) APIs for sending request and getting reply
  – Clients construct a `CORBA::Request` (local) object, “pushing” arguments and operation name etc. on it like a stack
    • Exactly what a proxy does: same API to ORB Core
DII Example

- Notes (don’t need to memorized gory details….)
  - CORBA::Request object represents one invocation of one method of one CORBA object
  - CORBA::Any encapsulates any CORBA type

```c++
// Create request that will be sent to the manager object
CORBA::Request_var request = manager->_request("open");

// Create argument to request
CORBA::Any customer;
customer <<= (const char *) name;
CORBA::NVList_ptr arguments = request->arguments();
arguments->add_value( "name" , customer, CORBA::ARG_IN );

// Set result type
request->set_return_type(CORBA::_tc_Object);

// Invoke operation. NOTE: Could have used send_deferred()
request->invoke();

// Get the return value
CORBA::Object_var account;
CORBA::Any& open_result = request->return_value();
open_result >>= CORBA::Any::to_object(account.out());
```
Object References

• Object reference
  – Opaque handle for client to use
  – Identifies exactly one CORBA object
  – IOR == “Interoperable Object Reference”

• References may be passed among processes on different hosts
  – As parameters or “stringified”
  – ORB will convert into form suitable for transmission over network
  – ORB on receiver side will create a proxy and return a pointer to it
  – Basically functions as a remote “pointer” that works across heterogeneity in language, OS, net, vendor, …
Object References (cont.)

- Object Key
  - Opaque to client
  - ORB-specific
- Object ID
  - Can be created by user or POA (more in POA slides…)

IOR:

| Repository ID | Profile for Protocol1 | Profile for Protocol2 |

IIOP Profile:

| TAG_INTERNET_IOP | IIOP version | Host addr. | Port | Object Key | Components… |

Object Key (one possible implementation):

| POA ID | Object ID |
Interface Repository

• Stores information on interfaces which can be looked up later by others at runtime. Tells about
  – Interface names
  – Method signatures
  – ...
  – Exactly the information in an IDL file.

• Allows for runtime discovery of interfaces.
  – Can be used by other useful hooks, such as the DII, DSI, and Interceptors.
Implementation Repository (IR)

- Stores information on the implementations available for a given interface
  - Mainly bindings between interface names and executable files that implement them
- This allows the ORB to
  - activate servants to process object invocations
  - Construction of remotely accessible factory objects
General Interoperability Protocol (GIOP)

Abstract protocol to allow for interoperability between different vendors’ ORBs. To do this, it defines:

- Interoperable Object Reference (IOR) format
- Inter-ORB message formats (and their protocol state machine for interacting)

1. Request: from client, sending an invocation. Contains
   - GIOP.MessageHeader
   - GIOP.RequestHeader
   - GIOP.RequestBody

2. Reply: from server, responding to Request.

3. CancelRequest: from client, telling it to ignore a Request already sent.

4. LocateRequest: from client, to find out if a server can service a particular request, or if it has a forwarding IOR to the actual server implementation.

5. LocateReply: from server, in response to LocateRequest.

6. CloseConnection: from server, indicating it is closing the connection.

7. MessageError: by both...

- Wire protocol (data transfer syntax): Common Data Representation (CDR). A coding for all IDL types, structured types, exceptions, object references. Covers coding into an octet stream, alignment boundaries, how to indicate byte ordering used.
Example of GIOP Format

module GIOP {
    enum MsgType {Request, Reply, CancelRequest, LocateRequest, LocateReply, CloseConnection, MessageError};
    struct MessageHeader {
        char magic[4];
        Version GIOP_version;
        octet byte_order;
        octet message_type;
        unsigned long message_size;
    };
    struct RequestHeader {
        IIOP::ServiceContextList service_context;
        unsigned long request_id;
        boolean response_requested;
        sequence<octet> object_key;
        string operation;
        Principal requesting_principal;
    }
    ...
}
GIOP Transport Layer Assumptions

• Connection-oriented
  – I.e., transport management deals with opening and closing and using connections

• Connections are like
  – Two roles
    • Clients: open connections to servers
    • Servers: listen for connections
  – Clients and servers may only send a subset of the message types
  – Can be closed in an orderly fashion, or abortive close (both clearly defined)
IIOP

• IIOP is simply GIOP (an abstract protocol, remember) implemented over TCP/IP
• Must be implemented by every ORB
  – Gives a universal way for ORBs to communicate
  – A given ORB may implement different transports underneath GIOP, also
• CommunicationID = {IP address, TCP port}