Lecture 6

Architectural Patterns:
Broker

Broker Pattern

The Broker pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

The Problem

- The problem is how to get more work done efficiently
- A common solution to this problem is to divide and conquer
- In the object arena, one way to do this is to disperse the functionality across multiple cooperating objects, some of which are twins (identical implementations of the same object), which coordinate in solving the problem more efficiently

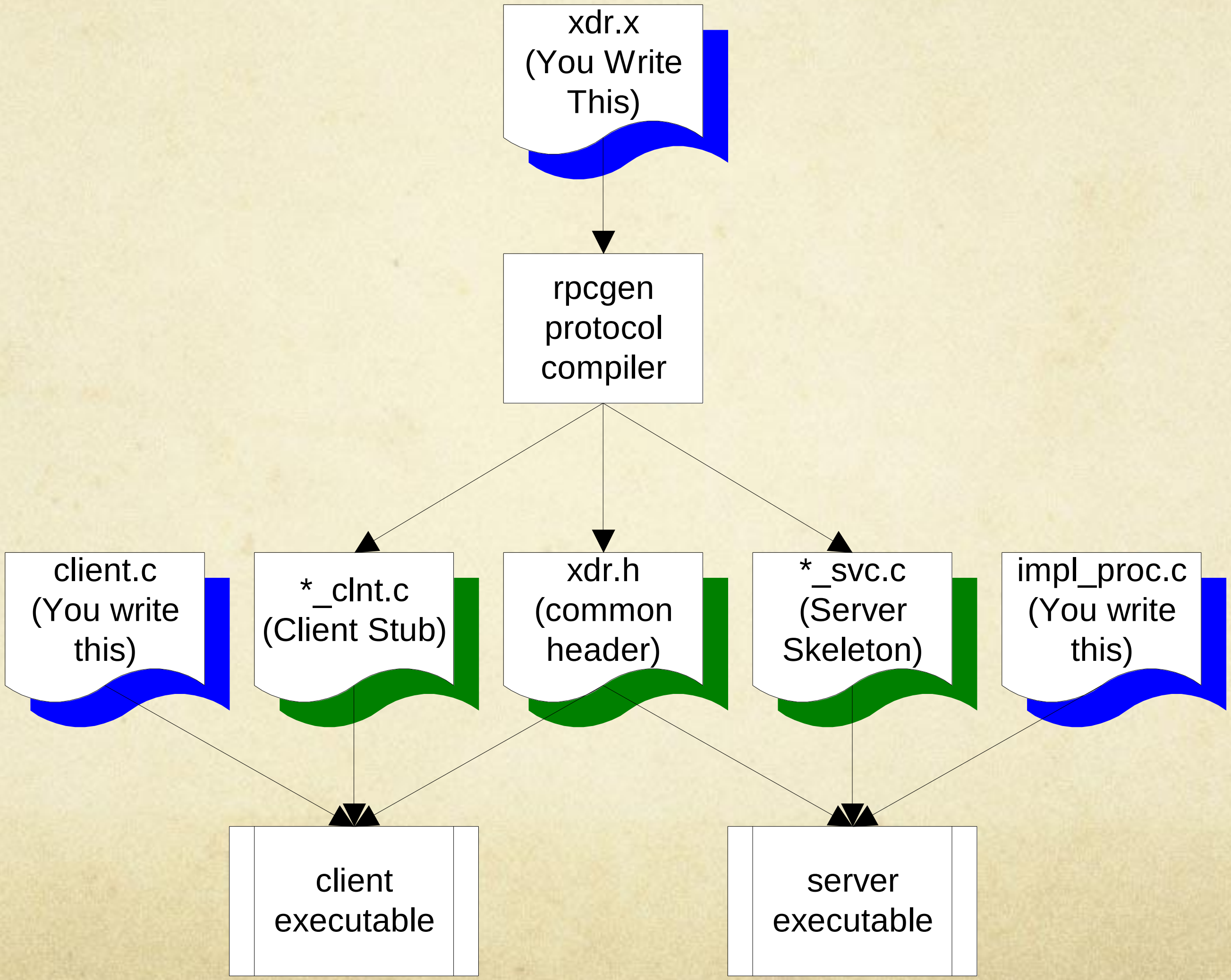
Remote Procedure Calls (RPC)

Digital ONC RPC

The Point

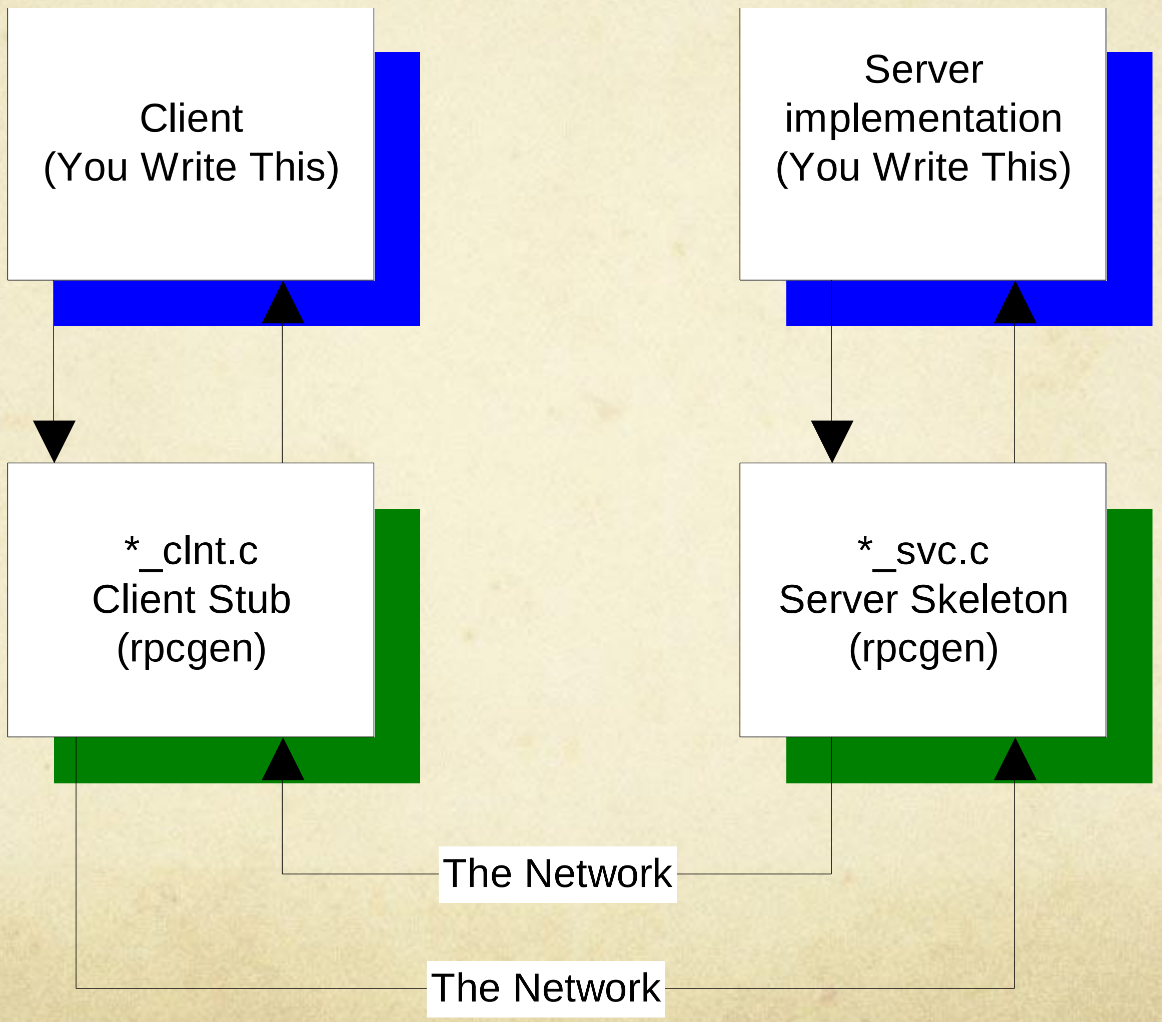
- “What’s the difference between local and remote procedure calling?”
  - “Very little—that’s the point”
- Remote Procedures generally accept and return *pointers* to data

The Process



xdr.x (You Write This)
rpcgen protocol compiler
client.c (You write this)
*_clnt.c (Client Stub)
xdr.h (common header)
*_svc.c (Server Skeleton)
impl_proc.c (You write this)
client executable
server executable

Call Sequence



Remote Services

- SUN Remote Procedure Call
  - If the time to transfer the data is more than the time to execute a remote command, the latter is generally preferable.
  - UDP protocol is used to initiate a remote procedure, and the results of the computation are returned.

SUN RPC

- Communication is message-based
- When a server starts, it binds an arbitrary port and publishes that port and the PROGRAM and VERSION with the portmapper daemon (port 111)
- When a client starts, it contacts the portmapper and asks where it can find the remote procedure, using PROGRAM and VERSION ids. The portmapper daemon returns the address and client and server communicate directly.

Sample Protocol Definition File (.x file)

this XDR file (somefile.x):

```
program NUMPROG
{
   version NUMVERS
   {
      int READNUM(int) = 1; /* procedure number 1 */
   } = 1; /* interface VERSION number */
} = 0x2000002; /* PROGRAM number */
```

is turned into this header file by rpcgen (somefile.h):

```
#define NUMPROG 0x2000002
#define NUMVERS 1

#if defined(__STDC__) || defined(__cplusplus)
#define READNUM 1
extern  int * readnum_1(int *, CLIENT *);
extern  int * readnum_1_svc(int *, struct svc_req *);
```

RPC Paradigms for Client Server

- Fat Client-DBMS (2 Tier)
  - VB <=> Sybase (ODBC)
  - Motif C++ <=> DBMS (ctlib)
- Fat Client-Application Server-DBMS
  - C Front End <=> C Business Logic <=> DBMS

RPC Under the Hood

- RPC is important because it handles network details for you:
  - Network Details
    - Byte Ordering (Big Endian, Little Endian)
  - Alignment Details
    - 2/4 Byte alignment
  - String Termination (NULL ?)
  - Pointers (how to handle migration of pointers?)

RPC eXternal Data Representation

- XDR provides:
  - Network Transparency
    - Single Canonical Form using Big-Endian
    - 4-Byte alignment
    - XDR passes all data across the wire in a byte stream
  - Filters

XDR Filters (Types)

- Integer: int (4 bytes)
- Unsigned Integer: unsigned int (4 bytes)
- char: int (4 byte signed integer)
- Double: double (8 bytes IEEE754 FP)
- Float: float (4 bytes IEEE754 FP)
- int week[7]
- int orders <50> (variable length array)
- opaque data<1000> any data

Building an RPC Application

- Create XDR file (~mark/pub/518/rpc/[linux|sun]/numdisp.x)
- run rpcgen to create
  - client stub: numdisp_clnt.c
  - server skeleton: numdisp_svc.c
  - common header: numdisp.h
- write client.c and numdisp_proc.c
- compile client and server (in subdirs)
- run (client on sheik, server on cheiron)
- *Example:*
  - *client on sheik linux: ~mark/pub/51081/rpc/linux,*
  - *server on cheiron OS X: ~/UofC/51081/pub/51081/rpc/mac*

Once upon a time . . .

- Remote Procedure Calls
  - functionality could be offloaded into a remote procedure, which could be called allowing the programmer to use the familiar procedure call rather than having to worry about low level networking programming with sockets and TLI (Transport Level Interface), and
  - Networking details (byte ordering, alignment differentiation, string termination, etc.) were still hidden from the programmer, easing her development effort, and making *cross platform development easier*, and
  - underlying transport protocols and mechanisms are *hidden from* the developer

Problems with RPC

- But, RPC mechanisms played the square peg to the round object model:
  - incompatible versions (Sun, OSF DCE, ParcPlace XNS Courier (Novell Netware)
  - programmer must know *where* (i.e., what server) the remote procedure resides (its socket)
  - Objects could be broken down manually by the programmer for transport over RPC, but an object-RPC mapping at runtime was needed to be written by the programmer
  - Only data could be passed across the wire, essentially a command and its parameters

Problems with RPC

- Additionally:
  - encapsulation was entirely broken, state exposure
  - no object identity, or runtime interface discovery (DII)
  - object model was broken, you could not call a remote method on an object
  - polymorphism (not caring the exact subclass you're talking to) was impossible
  - External Data Representation assumed a pure C Language implementation, making it *monolingual*
- The basic problem was:
  - How to provide the *effect* of RPC within the *model* of OO?

Broker Pattern in Detail

The Broker pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

Broker Pattern Details

- Use a broker when your environment is heterogeneous and needs a distributed system operating with independent loosely-coupled but cooperating components (aka objects)
- Building a complex distributed system as a set of decoupled and interoperating components, rather than a monolithic application, results in:
  - Greater flexibility
  - Greater maintainability
  - Greater scalability
- However, in order for loosely-coupled components to communicate, some kind of inter-process communication is required

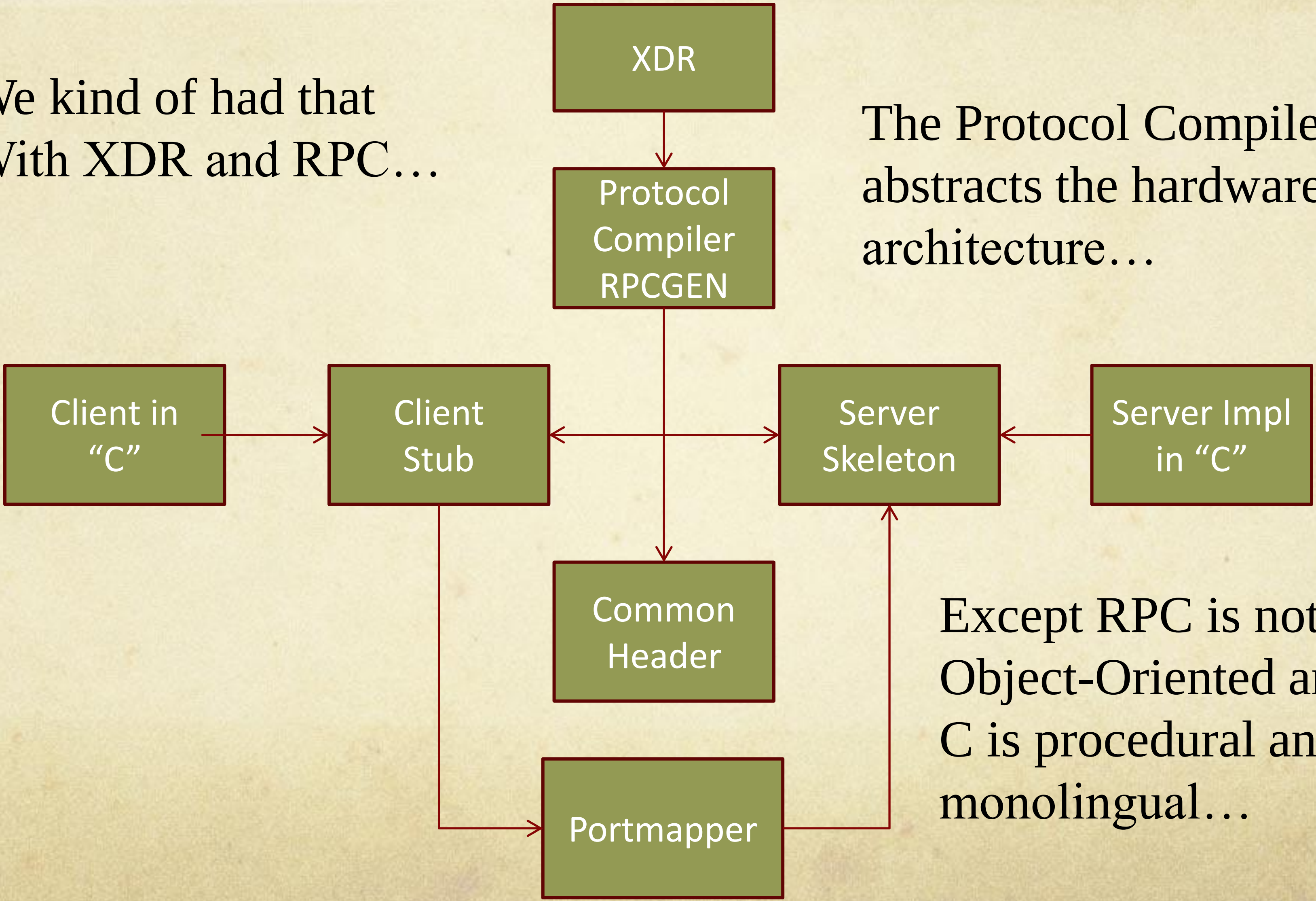
Broker Pattern Details

- Services for adding, removing, substituting, activating, and locating distributed components are also required
- Specifically:
  - Components should be able to access the services of other components through remote, location-transparent methods
  - Components should be substitutable at runtime according to interface
  - The overall architecture should hide system- and implementation-specific details from the developers
  - Provide an interface definition language that allows interfaces to be generically defined and a protocol compiler that translates those generic interfaces into language-specific classes

RPC Model Revisited

We kind of had that
With XDR and RPC…

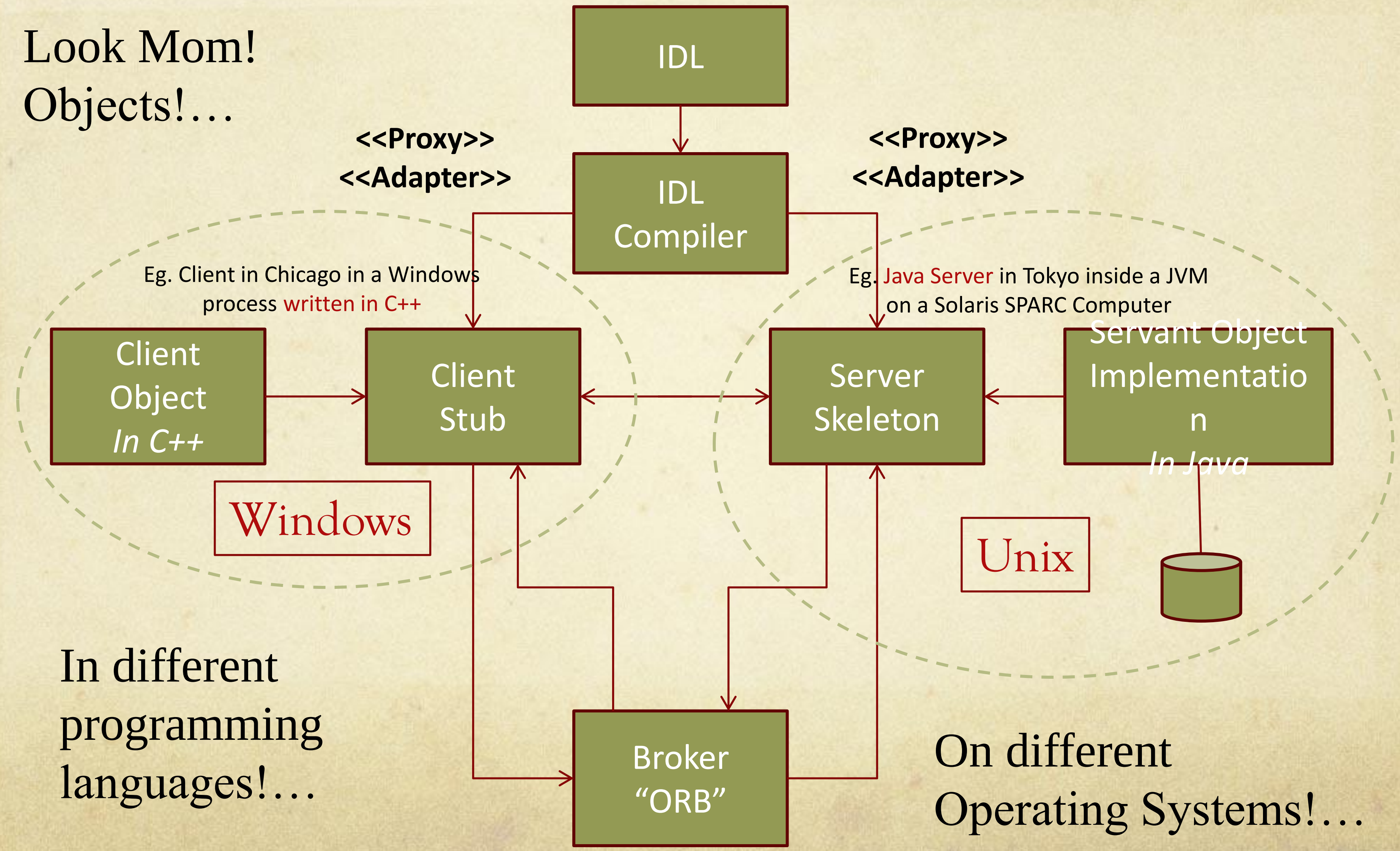
The Protocol Compiler abstracts the hardware architecture…



Except RPC is not Object-Oriented and C is procedural and monolingual…

Broker Model

Look Mom!
Objects!…


IDL
IDL Compiler
<<Proxy>>
<<Adapter>>
<<Proxy>>
<<Adapter>>
Eg. Client in Chicago in a Windows process written in C++
Eg. Java Server in Tokyo inside a JVM on a Solaris SPARC Computer
Client Object In C++
Client Stub
Server Skeleton
Servant Object Implementation In Java
Windows
Unix
Broker "ORB"

In different programming languages!…

On different Operating Systems!…

CORBA: The Common Object Request Broker Architecture

A (Serious) Broker Implementation

The Broker pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

The Object Management Group

- Formed in 1989 to provide common guidelines on how to *integrate* and *interoperate* new technology with legacy systems.
- The OMG was formed to create a component-based software *marketplace* by hastening the introduction of standardized object software
- Made up of close to 800 different vendors, some of the original members:
  - 3Com
  - Data General
  - American Airlines
  - Sun Microsystems
  - Hewlet Packard
  - Unisys Corp.

The Object Management Group

- Voting rights by committee
- Request for Information, Request For Proposals and Specifications (i.e., CORBA 2.3, 3.0)
  - Darwinian process of "survival of the fittest"
  - RFP responses must be about *existing* solutions, not hypothetical
- Not only specify CORBA:
  - Unified Modeling Language since 1997
  - Medical, Electronics, Transportation, Financial, Retail
- Microsoft has been a member since 1992

The OMG Specifications

- OMA: Object Management Architecture
  - Two Models:
    - Object Model
      - Object: Encapsulated Entity with immutable distinct identity which offers services through well-defined interfaces. The model also supports interface inheritance.
    - Reference Model
      - provides interface categories for groups of objects
      - Interface categories are subsumed by an Object Request Broker (ORB)
        - CORBAServices (Naming, Events, Trading, Security, Transaction, etc.)
        - CORBAFacilities: Vertical Domain Interfaces (service unique to finance, healthcare, manufacturing, etc)
        - Application Interfaces (non-standardized)

CORBA “Selling Points”

- Location Transparency (a client doesn’t have to care *where* an object is)
- Relocation Transparency (an object implementation may be moved without affecting the client)
- Encapsulation Transparency (a client does not have to care *which object implementation* or *how* an object implements its semantics, thus allowing an object’s internal representation to change)
- Communication Transparency (network transports and data link layers are irrelevant to the client)
    - communication can be conducted over Ethernet, ATM, token ring, serial ports, etc.

CORBA “Selling Points”

- Invocation Transparency (a client does not have to care whether a call is statically or dynamically conducted)
- Heterogeneous Hardware (a client does not have to worry about the machine hardware)
- Operating System Transparency (a client does not have to worry about what OS)
- Programming Language Transparency (an object’s implementation may be in any number of different languages (C, C++, Smalltalk, Java, COBOL, Ada, Lisp) and the client should not have to care what language that is)

CORBA “Selling Points”

- Inter-ORB communication uses Interoperable Object References, IORs (multiple vendor’s ORBs, COM/DCOM, RMI/IIOP)
  - One vendor’s Java client can use an IOR to contact another vendor’s C++ server
  - example: ~mark/src/550/cpp-java
- Scalability, Load Balancing, failover
- Object Oriented Semantic Integrity (Inheritance, Polymorphism (late binding), Exceptions)
- Supports multi-tier distribution models, including internet connectivity (gatekeeper/http tunneling)

The OMG Specifications

- A CORBA object is a virtual object, an interface with a delegated implementation
- A target object is the *interface* that is the target of a client request
- A client invokes a request on a CORBA object (which the ORB *delivers* to a servant)
- A request is an invocation of a method defined by a CORBA object

The OMG Specifications

- An object reference is a handle used to identify, locate and address a CORBA object
- A servant is a coded implementation of a CORBA object's interface
- Servants run within servers, which are containers for any number of implementations

CORBA

- CORBA distributed objects meld the client-server paradigm with the distributed object paradigm
  - A client only knows a CORBA object by its *interface*
  - The location of the implementation of a CORBA interface is irrelevant, and may not be known to the client
  - CORBA distributed objects allow for dynamic runtime composition and delegation of other objects
  - CORBA hides the networking details just like RPC does
  - Unlike RPC, CORBA preserves the OO paradigm

CORBA Technical Objectives (a partial list)

- Distributed Transaction support
- Robustness and high availability
- Versioning of objects
- Notification of events (messages)
- Provide relationship semantics between classes (inheritance and polymorphism)

CORBA Technical Objectives (a partial list)

- Type conformance for interfaces (interface inheritance)
- Distribution transparency
- Performance of both local and remote operations
- Extensible and dynamic behavior
- Naming Service for object implementation location (osagent, COS Naming)
- Concurrency control of remote references

Chicken Parts

- The Object Request Broker
  - “CORBA bus”
- The Client
  - The Stub
    - The client side adapter to the ORB
- The Server
  - The Skeleton
    - The server side adapter to the ORB
- The Object Adapter

“My ORB, My ORB, they’ve stolen my ORB” - Michelet

- What and Where, exactly, is the ORB?
  - The “CORBA Magical Mystery Bus”
    - Transportation?
    - Conduction?
    - The metaphysical ambiance: “ORB” in Plotinus and Spinoza
- In the end, the “ORB” is nothing more than the little fat man behind the screen

The Dirty Little Secret

- The ORB is simply a set of vendor provided services, interfaces, and libraries (dynamic and static)
  - These might include:
    - static and dynamic libraries
    - an activation daemon
    - a listener agent (osagent)
    - the client stub and its dependent libraries
    - the server skeleton and its dependent libraries
    - the Object Adapter (Basic, Portable) (from another library)
  - And also might include as well:
    - the Dynamic Invocation Interface (DII)
    - the Dynamic Skeleton Interface (DSI)
    - the Interface Repository
    - the Implementation Repository

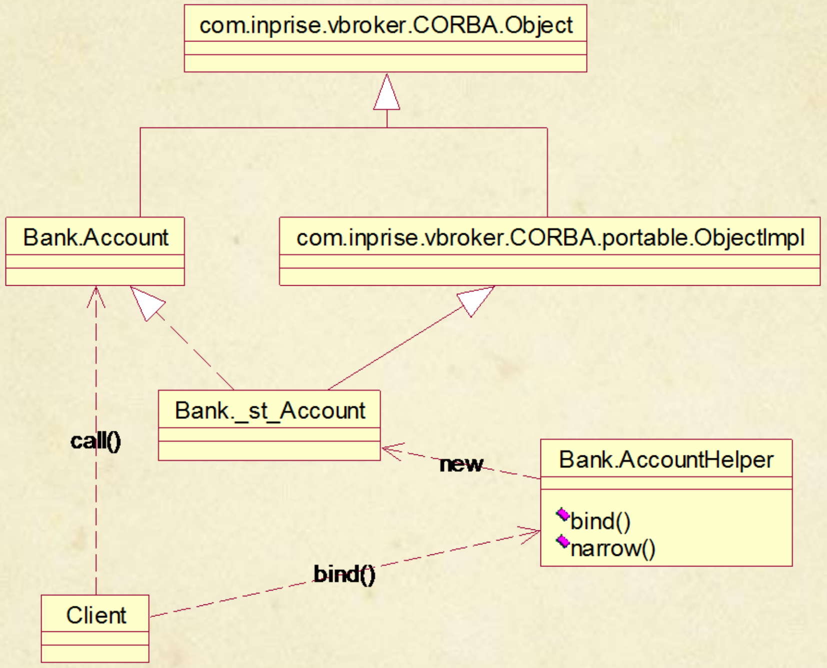
The Stub

- Generated by the idl compiler in the native language and *instantiated* by the narrowing function of a Helper class.
- The stub is local to the client and linked into the client
- The stub performs the role of a *proxy* for the client, and contains a full *delegatory* implementation of the remote interface
- Therefore, you do not have to write the stub, the idl compiler will generate one for you
- The stub’s implementation is one of delegation

The Stub

- The IDL compiler maps the interface into the correct language, and also generates code to locate the proper skeleton for delegation
- The IDL compiler also creates in the stub the marshalling code necessary to handle the communication
- The stub may be created by some other idl compiler than is used to create the skeleton

VisiBroker Stub Implementation

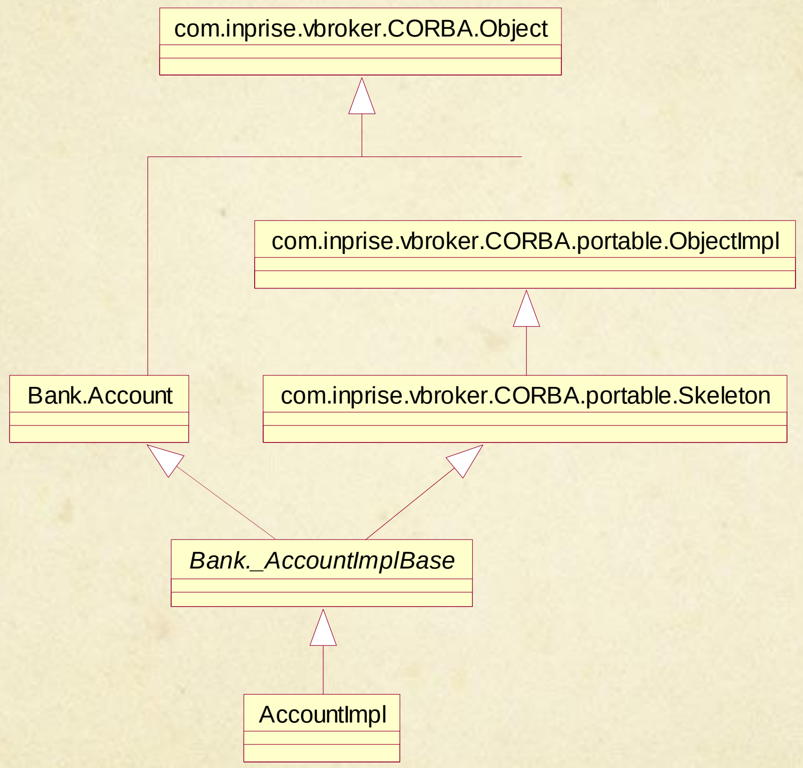


The client calls bind() on the AccountHelper object, which in turn calls narrow(), which creates the Bank._st_Account stub for the client’s use

The Skeleton

- The skeleton is the server-side counterpart of the stub
- The IDL compiler generates the skeleton, so you don’t have to code it
- Skeletons are abstract classes which define method calls that have no inherent implementation of the idl interface (no flesh), but they do implement helper functions such as execute(), dispatch(), and provide marshaling and demarshaling.
- The “execute” method uses a method dispatch table to make the appropriate method call on your implementation
- Although you don’t have to code the skeleton, you do have to code the implementation of the idl interface; the skeleton is going to be calling your implementation
- Coordinates communication with the Object Adapter

VisiBroker Skeleton Implementation


com.inprise.vbroker.CORBA.Object
com.inprise.vbroker.CORBA.portable.ObjectImpl
Bank.Account
com.inprise.vbroker.CORBA.portable.Skeleton
Bank._AccountImplBase
AccountImpl

The Object Adapter

- Resides on the server side and receives calls from the server-side ORB
- Instantiates object implementations if they are not already present in memory
- Assigns newly created objects an IOR
- Registers all objects created
- Activates and deactivates object implementations
- Dispatches incoming client calls from the ORB core to the skeleton and thereby on to the object implementation
- [Visibroker]: Registers object implementations with the Smart Agent

The Portable Object Adapter (POA)

- Provides a set of interfaces for managing:
  - Object References
  - Servants
- Code written to the POA is now portable at the source code level (not so with BOA).
- The POA exists to map (delegate, route, dispatch) incoming client requests to the correct servant implementation of a given CORBA Object.

POA

- The POA does the following:
  - maps object references to servant implementations
  - enables dynamic transparent activation of servant implementations
  - Associates Policy metadata with CORBA objects
  - Persists CORBA object states over multiple servant instances

POA

- Multiple POAs can be active in a given server.
- However, all POAs in a server must be derived from a ROOT POA.
- The Root POA is used to create Policy objects that are then passed to create_POA()
- You can think of a POA as a “Policy Domain”, which specifies policy metadata for all servants within its “domain”

Lifetime Management

- The POA is responsible for activating servant implementations
- The POA can create Object References and or servant implementations on an as-needed or policy-defined basis.

The VisiBroker Smart Agent

- The Smart Agent is a native language object location daemon
- At least one SA must be running on each network (local subnet, ie. 192.168.3.0)
- When an SA starts up, it makes a UDP call across the subnet and registers itself as a participating SA with other SAs that are already running. It broadcasts to port OSAGENT_PORT (14000)
- When the Object Adapter registers a new object implementation, the OA communicates this to its SA

The VisiBroker Smart Agent

- The SA records the name and the location (socket) of the object implementation
- When a client call comes through the stub for a new object binding, the stub contacts the SA to see if it knows where to find the object. If the SA knows, it provides that information to the client via the stub and all further communication with the object implementation is direct socket.

The VisiBroker Smart Agent

- If the SA doesn't know about the object, it contacts the other SAs it knows about on the subnet, and hopefully one of them will know about the object. If so, that SA returns the location information to the client, and all further communication with the object implementation is direct socket.
- VB object implementations ping (UDP message) their bound SA every 2 minutes. If they don't receive a sanity check back, the object tries to find another SA.
- If an SA doesn't receive a ping from a registered object every 2 minutes, it marks the object as suspect in its hash table, and then the SA itself pings the object 2 minutes later. If it doesn't receive a sanity check the second time, the SA marks the object as MIA and removes the object's IOR entry from its hash table.
- The SA also provides round-robin load balancing of object implementations automatically. Remember, there can be more than one registered implementation of a given interface, thus distributing the processing load.

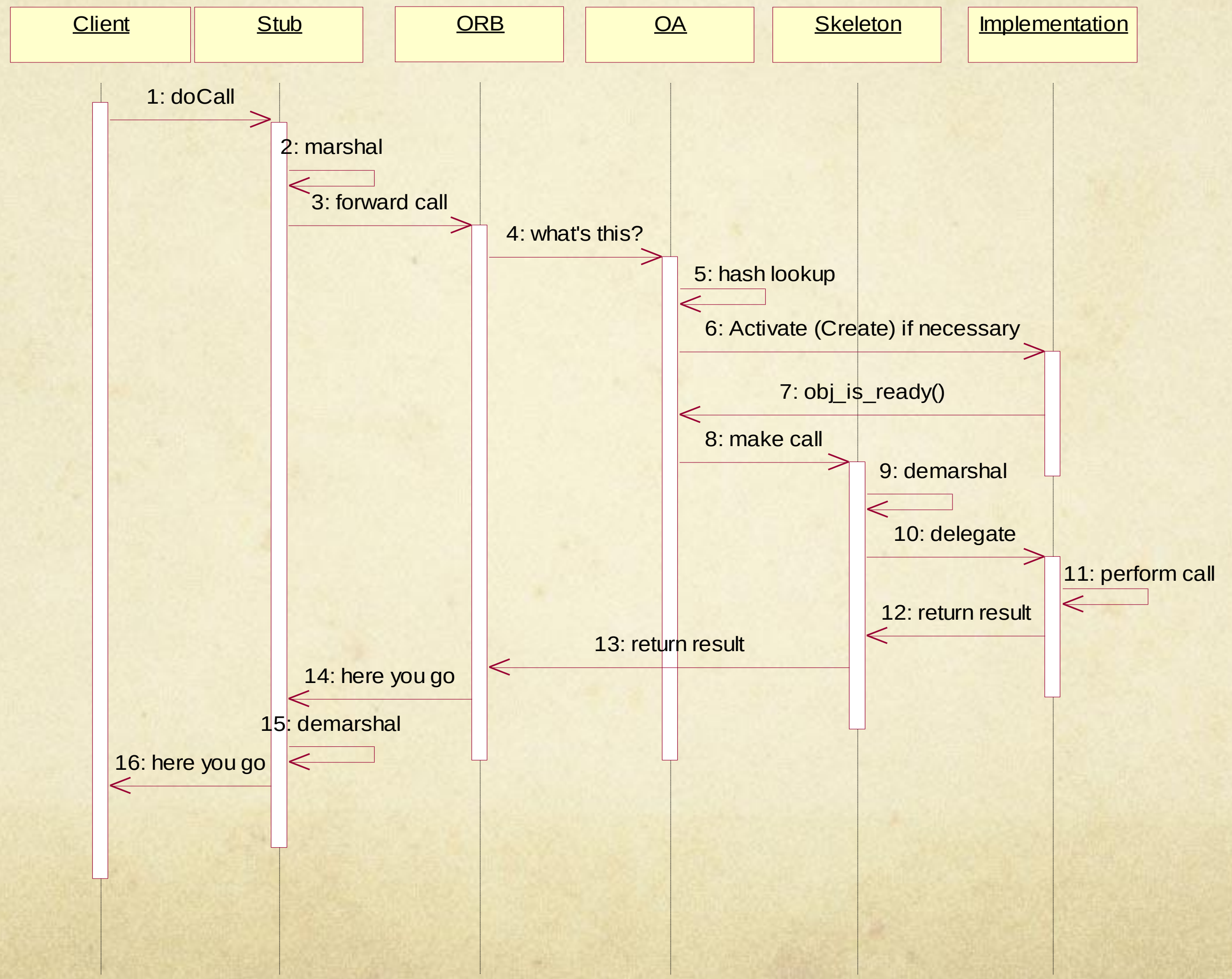
The CORBA Request Lifecycle

- A client locates a CORBA object and binds to it and thereby gets the CORBA object's remote reference (handle)
- The client then makes a request on a method defined in the interface of the CORBA object
- The *stub*, or *client-side* part of the ORB (dynamically linked in to the client application) *marshals* that request and sends it [over-the-wire] to the *skeleton*, or *server-side* ORB core linked to the server application which contains the CORBA object's implementation (servant)
- The server-side ORB dispatches the request to the Object Adapter, which *instantiates* an object implementation if one doesn't already exist or it needs more for load balancing purposes

The CORBA Request Lifecycle

- The Object Adapter looks up the method and target identity and dispatches the request to the appropriate object's *skeleton*
- The skeleton *demarshals* the call and *delegates* the method call to the appropriate method in the servant implementation
- The servant executes the method referenced in the request
- The servant returns the result through the skeleton which marshals the result for [over-the-wire] transmission over a socket to the client-side ORB linked to the client application
- The client-side ORB receives the result and forwards the result to the stub which demarshals the result and delivers that result to the client application, and the call is concluded.

The Call Sequence



References and Interoperable Object References

- An IOR references a *single* object implementation
- Multiple IORs can reference the *same* object implementation
- References can be null
- References are opaque (encapsulated, clients can't mess with their interiors)
- References can be persisted (IORs can be stored in a file or URL)
- References support late binding
- References are interoperable

References and Interoperable Object References

- IORs:
  - Repository ID
    - String that identifies the most derived type of the IOR at the time of creation (used for narrowing)
  - End Point Information
    - Complete connection information on how to contact the [remote] server, including, for IIOP, the IP address and port the server (or OAD) is listening on
    - information on multiply-supported protocols and transports
  - Object key
    - Complete connection information the target *object implementation*.
    - The Object key is created by the server-side ORB when the object reference is bound to the client
- Object References are created by the ORB's idl compiler

IOR Details

- An IOR looks like this to the ORB:

```
IOR:010000001000000049444c3a4163636f756e743a312e30000002000000000000003400000001010000 1900
0000636865657461682e6c732e616e6f6d616c6f75732e6e65740000050d0c000000424f41c0a803100000201e
0301000000240000000100000001000000010000001400000001000000010001000000000009010100000000
000
```

- Using the VisiBroker utility printIOR, we can see a better picture of this IOR:

```
Interoperable Object Reference:
Type ID: IDL:Account:1.0                [shows the Repository ID for this object]
Contains 2 profiles.
  Profile 0-IIOP Profile:               [profile is IIOP-based]
    version: 1.0
    host: cheetah.ls.anomalous.net       [object is hosted on cheetah]
    port: 3333                            [
    Object Key: ForeignId[object_key={12 bytes: [B][O][A](192)(168)(3)(16)(0)(0)[ ](30)(3)}]
  Profile 1-Unknown profile:
struct TaggedProfile{unsigned long tag=1;sequence<octet> profile_data={36 bytes:
(1)(0)(0)(0)(1)(0)(0)(0)(1)(0)(0)(0)(20)(0)(0)(0)(1)(0)(0)(0)(1)(0)(1)(0)(0)(0)(0)(0)(9)(1)(1)(0)(0)(0)
(0)(0)};}
```

Problem 1: How to guarantee transparency of implementation language

- The CORBA object interface could be written in C or as a Java Interface, but then you've just statically defined the language and binding characteristics of the implementation of the interface
- The OMG defined the Interface Definition Language in 1991 as a solution to this problem
  - IDL is the OMG's "object contract language" -- Pope
  - IDL is not a complete programming language.
  - Most significantly, you cannot write a class implementation using IDL, *only its interface*
  - Thus, IDL has no flow control, no if-then-else syntax, no while/for loops, no variables, etc.
  - Even "attributes" eventually get translated into *methods*.
  - IDL thus can only define *types* and those types semantics
  - IDL is the language used to define an interface that is *compiled* into a programming language by the *idl compiler*

Interface Definition Language

- IDL:

```
typedef float cash;
struct bigBucks {
    double amt;
};
interface Account {
    bigBucks withdrawal(in cash amount, inout cash balance);
    bigBucks deposit(in cash amount, inout cash balance);
};
```

- Supports signatures like most languages:
  - boolean account deposit(in money amount);
- Supports attribute definition:
  - attribute string myName;
    - translates to (in C++):
  - virtual char * myName() = 0;
  - virtual void myName(const char * value) = 0;
- First Class CORBA Objects and Second Class CORBA objects
  - First class CORBA objects are interface objects with implementations, passed by ObjRef
  - Second class CORBA objects are structs, etc., that are pure data representations, passed by value.

Native Language Bindings

IDL Type	Java Mapping	C++ Typedef	C++ Mapping
boolean	Boolean	CORBA::Boolean	bool or 0/1
char	char	CORBA::Char	char
octet	byte	CORBA::Octet	unsigned char
string	String	CORBA::String_var	char *
short	short	CORBA::Short	short (16bit)
unsigned short	short	CORBA::UShort	unsigned short
long	int	CORBA::Long	long (32bit)
unsigned long	int	CORBA::ULong	unsigned long
float	float	CORBA::Float	float
double	double	CORBA::Double	double
enum	class w/ final vars	N/A	enum
struct	final class	N/A	struct
typedef	Helper class	N/A	typedef
sequence	array	N/A	array (_var)
array	array	N/A	array (_var)

The VisiBroker idl2java compilation process: File Generation

- *You write the IDL file:*

```
module Bank {             //modules provide namespace
protection in idl
    interface Account { … };
    interface … { … };
};
```

- Compile it with idl2java:
  - idl2java myInterfaces.idl

The VisiBroker idl2java compilation process: File Generation

- idl2java generates a number of files (in a subdirectory if you use module):
  - Bank._AccountImplBase: implements the server-side skeleton interface, provides marshaling
  - Bank._st_Account: implements the client-side stub (for the Java language)
  - Bank.AccountHelper: helper functions such as narrow(), proprietary bind()
  - Bank.AccountHolder: holds a public instance variable of type Account. Used by clients in passing *out* and *inout* parameters. Temporary staging for return value.
  - Bank.Account: The Java Interface for the idl. You must implement this interface.
  - Bank._example_Count: an example implementation for the Bank.Account interface
- Classes *you must provide*:
  - AccountImpl.java: The implementation of Bank.Account (the Account interface)
  - AccountServer.java: The server application that creates and publishes a CountImpl with the ORB
  - AccountClient.java: The client application that uses the Account interface

Performance Issues for Remote Operations

- Scalability: linear scalability of the method processing time as client demand increases
- Response Time: Response Time should not be unreasonably slow
- Parallelism: Clients should have to do nothing special to achieve parallel execution--this should be the activity of the ORB
- Throughput: The total amount of work an object can handle should remain consistent