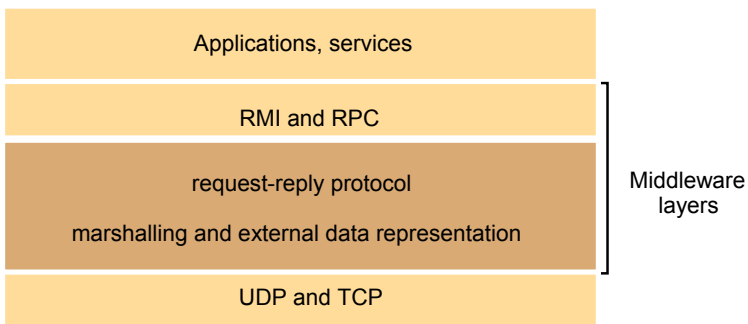
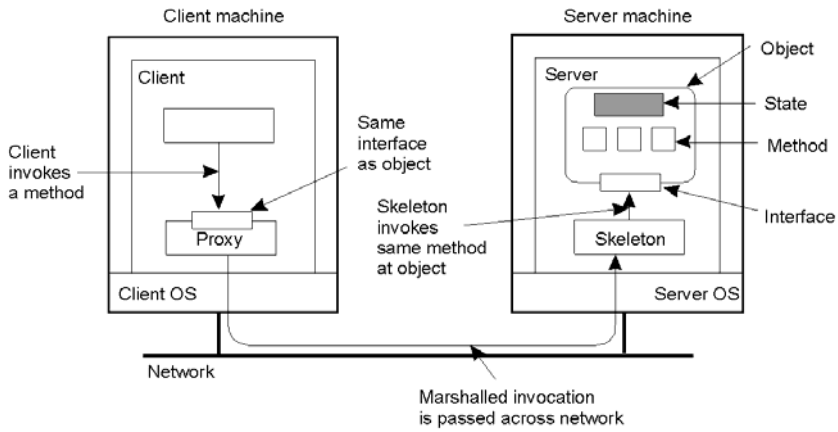
Remote Object Invocation

Distributed Software Systems

Middleware layers



Distributed Objects



Compile-time vs run-time objects

- Objects can be implemented in many different ways
  - compile-time objects, i.e. instances of classes written in object-oriented languages like Java, C++
  - database objects
  - procedural languages like C, with a appropriate "wrapper code" that gives it the appearance of an object
- Systems like Java RMI support compile-time objects
- Not possible or difficult in language-independent RMI middleware such as CORBA
  - these systems use object adapters
  - implementations of object interfaces are registered at an object adapter, which acts as an intermediary between the client and object implementation

Persistent vs transient objects

- Persistent objects continue to exist even if they are not contained in the address space of a server process
- The "state" of a persistent object has to be stored on a persistent store, i.e., secondary storage
- Invocation requests result in an instance of the object being created in the address space of a running process
  - many policies possible for object instantiation and (de)instantiation
- Transient objects only exist as long as their container server processes are running

Static vs dynamic remote method invocations

- Typical way for writing code that uses RMI is similar to the process for writing RPC
  - declare the interface in IDL, compile the IDL file to generate client and server stubs, link them with client and server side code to generate the client and the server executables
  - referred to as static invocation
  - requires the object interface to be known when the client is being developed
- Dynamic invocation
  - the method invocation is composed at run-time

    invoke(object, method, input_parameters, output_parameters)
  - useful for applications where object interfaces are discovered at run-time, e.g. object browser, batch processing systems for object invocations, "agents"

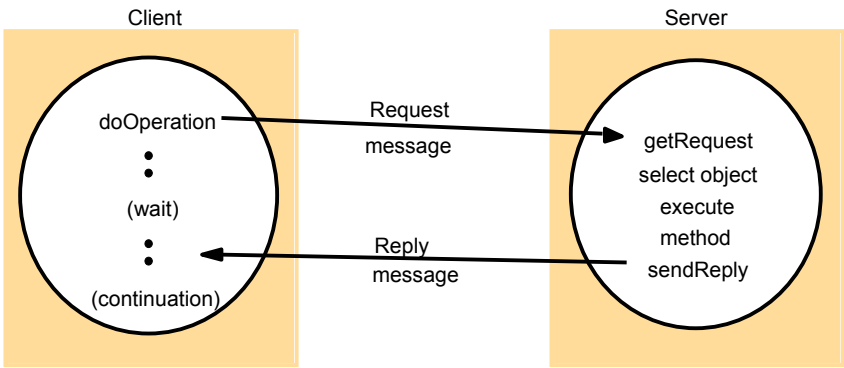
Design Issues for RMI

- RMI Invocation Semantics
  - Invocation semantics depend upon implementation of Request-Reply protocol used by R MI
  - Maybe, At-least-once, At-most-once
- Transparency
  - Should remote invocations be transparent to the programmer?
    - Partial failure, higher latency
    - Different semantics for remote objects, e.g. difficult to implement "cloning" in the same way for local and remote objects or to support synchronization operations, e.g. wait/notify
  - Current consensus: remote invocations should be made transparent in the sense that syntax of a remote invocation is the same as the syntax of local invocation (access transparency) but programmers should be able to distinguish between remote and local objects by looking at their interfaces, e.g. in Java RMI, remote objects implement the *Remote* interface

Issues in implementing RMI

- Parameter Passing
- Request-Reply Protocol
  - Handling failures at client and/or server
- Supporting persistent objects, object adapters, dynamic invocations, etc.

Request-reply communication



Operations of the request-reply protocol

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)
sends a request message to the remote object and returns the reply.
The arguments specify the remote object, the method to be invoked and the arguments of that method.

public byte[] getRequest ();
acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
sends the reply message reply to the client at its Internet address and port.

Request-reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*array of bytes*

Representation of a remote object reference

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

CORBA interoperable object references

IOR format

IDL interface type name	Protocol and address details			Object key	
interface repository identifier	IIOP	host domain name	port number	adapter name	object name

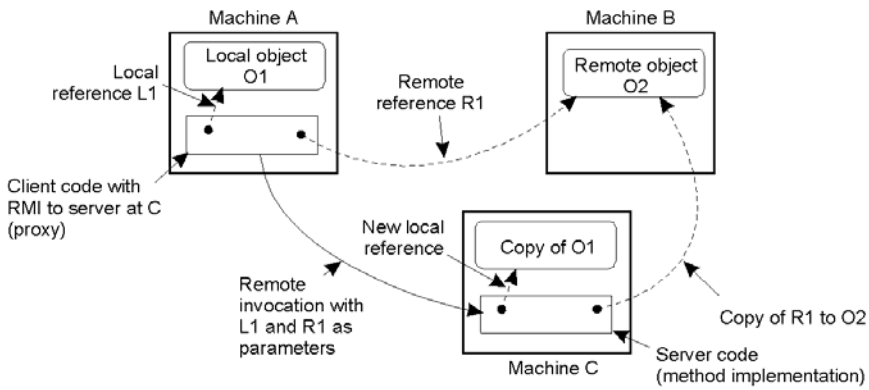
Request-Reply protocol

- Issues in marshaling of parameters and results
  - Input, output, Inout parameters
  - Data representation
  - Handling reference parameters
- Distributed object references
- Handling failures in request-reply protocol
  - Partial failure
    - Client, Server, Network

Marshalling

- Pack method arguments and results into a flat array of bytes
- Use a canonical representation of data types, e.g. integers, characters, doubles
- Examples
  - SUN XDR
  - CORBA CDR
  - Java serialization

Parameter Passing: local vs remote objects



Remote object references are passed by reference where local object references are passed by value

CORBA CDR for constructed types

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

CORBA CDR message

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20-23	"on__"	
24–27	1934	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

Indication of Java serialized form

Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h0 and h1 are handles

RPC exchange protocols

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Handling failures

- Types of failure
  - Client unable to locate server
  - Request message lost
  - Reply message lost
  - Server crashes after receiving a request
  - Client crashes after sending a request

Handling failures

- Client cannot locate server
  - Reasons
    - Server has crashed
    - Server has moved
    - (RPC systems) client compiled using old version of service interface
  - System must report error (remote exception) to client
    - Loss of transparency

Handling failures

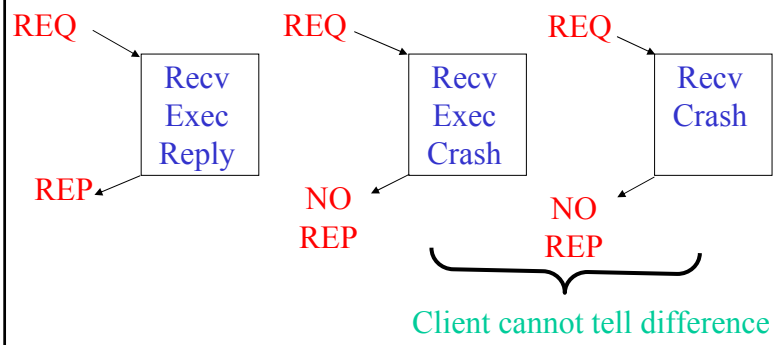
- Lost request message
  - Retransmit a fixed number of times before throwing an exception
- Lost reply message
  - Client resubmits request
  - Server choices
    - Re-execute procedure ➔ service should be idempotent so that it can be repeated safely
    - Filter duplicates ➔ server should hold on to results until acknowledged

Invocation semantics

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Handling failures

- Server crashes



Handling failures

- Server crashes
  - At least once (keep trying till server comes up again)
  - At most once (return immediately)
  - *Exactly once impossible to achieve*
- SUN RPC
  - At least once semantics on successful call and maybe semantics if unsuccessful call
- CORBA, Java RMI
  - at most once semantics

Handling failures

- Implementing the request-reply protocol on top of TCP
  - Does it provide applications with different invocation semantics?
    - NO!
      - TCP does not help with server crashes
      - If a connection is broken, the end pts do not have any guarantees about the delivery of messages that may have been in transit

Handling failures

- Client crashes
  - If client crashes before RPC returns, we have an "orphan" computation at server
    - Wastes resources, could also start other computations
  - Orphan detection
    - Reincarnation (client broadcasts new "epoch" when it comes up again)
    - Expiration (RPC has fixed amount of time T to do work)

RMI Software Components

- Communication module
  - Implements the request-reply protocol
- Remote reference module
  - Responsible for translating between local and remote object references and for creating remote object references
    - Maintains remote object table that maintains a mapping between local & remote object references
    - E.g. Object Adaptor in CORBA

RMI – Object Activation

- Activation of remote objects
  - Some applications require that information survive for long periods of times
  - However, objects not in use all the time, so keeping them in running processes is a potential waste of resources
  - Object can be activated on demand
    - E.g. standard TCP services such as FTP on UNIX machines are activated by inetd

Object Activation

- Active and passive objects
  - Active object = instantiated in a running process
  - Passive object = not currently active but can be made active
    - Implementation of its methods, and marshalled state stored on disk
- Activator responsible for
  - Registering passive objects that are available for activation
  - Starting named server processes and activating remote objects in them
  - Keeping track of locations of servers for remote objects that it has already activated
- Examples: CORBA implementation repository, JAVA RMI has one activator on each server computer

RMI – Other topics

- Persistent object stores
  - An object that is guaranteed to live between activations of processes is called a persistent object
  - Stores the state of an object in a marshalled (serialized) form on disk
- Location service
  - Objects can migrate from one system to another during their lifetime
  - Maintains mapping between object references and the location of an object

RMI – Other topics

- Distributed Garbage Collection
  - Needed for reclaiming space on servers
- Passing "behavior"
  - Java allows objects (data + code) to be passed by value
    - If the class for an object passed by value is not present in a JVM, its code is downloaded automatically
  - See Java RMI tutorial example
- Use of Reflection in Java RMI
  - Allows construction of generic dispatcher and skeleton

Distributed Garbage Collection

- Java approach based on reference counting
  - Each server process maintains a list of remote processes that hold remote object references for its remote objects
  - When a client first removes a remote reference to an object, it makes an addRef() invocation to server before creating a proxy
  - When a clients local garbage collector notices that a proxy is no longer reachable, it makes a removeRef() invocation to the server before deleting the proxy
  - When the local garbage collector on the server notices that the list of client processes that have a remote reference to an object is empty, it will delete the object (unless there are any local objects that have a reference to the object)
- Other approaches
  - "Evictor" pattern
  - Leases