Distributed Objects and Remote Invocation

Programming Models for Distributed Applications
Extending Conventional Techniques

- The **remote procedure call** model is an extension of the conventional procedure call.
- The **remote method invocation** (RMI) uses an OO-based model to allow objects in one process space invoke methods in another process space.
- The **distributed event-based** model extends event-driven paradigms.
Relationship to Middleware

Applications

RMI, RPC and events

Request reply protocol

External data representation

Operating System

Middleware layers
Interfaces

- The interface specifies the procedures and the variables that can be accessed from other processes.
- So long as its interface remains the same, the implementation may be changed without affecting the users of the module.
Interfaces in Distributed Systems

- Modules run in separate processes
- Direct access to variables is not possible
- Call-by-value and call-by-reference are not suitable when the caller and the procedure are in different processes
- Pointers cannot be passed as arguments nor returned as results of calls to remote modules
Input and Output Parameters

- Input Parameters are **passed** to the remote module
- Output Parameters are **returned** in the reply message
More on Interfaces

- **Service Interfaces** refer to the specification of the procedures offered by a server, defining the types of the input and output arguments of each of the procedures.
- **Remote Interfaces** specify the methods of an object that are available for invocation by objects in other processes, defining the types of the input and output arguments of each.
Neither service interfaces nor remote interfaces may specify direct access to variables
Interface Definition Languages (IDLs)

- We need an adequate notation for defining interfaces
- IDLs are designed to allow objects implemented in different languages to invoke one another
Example IDL from CORBA

// In file Person.idl
struct Person {
    string name;
    string place;
    long year;
};

interface PersonList {
    readonly attribute string listname;
    void addPerson(in Person p);
    void getPerson(in string name, out Person p);
    long number();
};
IDL Technologies

- CORBA's IDL
- Sun's XDR
- WSDL (web services description language)
- OSF's DCE (RPC)
- Microsoft's DCOM IDL
Communication Between Distributed Objects

- The Object Model
- Distributed Objects
- The Distributed Object Model
- Design Issues
- Implementation
- Distributed Garbage Collection
Requirements of the Object Model

- Within a distributed system, an object's data should only be accessible via its methods.
- Objects can be accessed via object references.
- Interfaces provide a definition of the signatures of a set of methods.
- An action is initiated by an object invoking a method in another object.
Invoking a Method

- The state of the receiver may be changed
- A new object may be instantiated
- Further invocations on methods in other objects may take place (resulting in a chain of related method invocations)
More on the Object Model

- Exceptions provide a clean way to deal with error conditions without complicating code
- Each method heading explicitly lists as exceptions the error conditions it might encounter
- Garbage Collection - automatically detecting when an object is no longer needed and freeing its space
Distributed Objects

- The physical distribution of objects into different processes or computers in a distributed system is a natural extension of the logical partitioning of object-based programs.
- This enforces encapsulation.
- Concurrency controls are an important consideration.
The Distributed Object Model

- The goal is to extend the object model to make it applicable to distributed objects
- **Local** method invocations occur between objects in the same process
- **Remote** method invocations occur between objects in different (possible remote) processes
Local and Remote Method Invocations
Fundamental Concepts

- The **Remote Object Reference** - objects can invoke the methods of a remote object if they have access to its remote object reference (identifier).
- The **Remote Interface** - a facility whereby an object specifies which of its methods can be invoked remotely.
The Remote Object Reference

- An identifier that can be used throughout a distributed system to refer to a particular unique remote object
- The remote object reference is specified by the invoker
- Remote object references can be passed as arguments and returned as results
The Remote Interface

- The remote interface specifies the methods that the remote object implements
- In Java, we can use public instance methods
- In CORBA, any language can be used as long as there's an IDL compiler available
Remote Object and Interfaces
Design Issues for RMI

- RMI is a natural extension of local method invocation
- Design Issue - the choice of invocation semantics - local invocations are executed exactly once, but this may not hold for remote method invocations
- Design Issue - how much transparency is desirable?
Invocation Semantic Design Decisions

- Do we retransmit the message until either a reply is received, or is the server assumed to have failed?
- When retransmissions are used, are duplicate requests filtered at the server?
- Is a “results history” maintained on the server to avoid duplicated effort whenever a retransmission is encountered?
- **Note**: local method invocation - exactly once semantics
## Reliability as seen by Invoker

<table>
<thead>
<tr>
<th>Retransmit request message</th>
<th>Duplicate filtering</th>
<th>Re-execute procedure or retransmit reply</th>
<th>Invocation semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td>Maybe</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Re-execute procedure</td>
<td>At-least-once</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Retransmit reply</td>
<td>At-most-once</td>
</tr>
</tbody>
</table>
Maybe Invocation Semantics

- Executed once or not at all
- Can suffer from omission failures within network
- Can suffer from crash failures on server side
- Useful only when applications can withstand occasional failed invocations
At-Least-Once Invocation Semantics

- Executed one or more times
- The invoker either receives a result or an exception is generated (no result received)
- Retransmissions masks omission failures
- Can suffer from remote server failures (crashes)
- Can suffer from arbitrary failures (resulting in a retransmission causing the server to execute a method more than once)
- Useful when applications are made up of exclusively idempotent operations
At-Most-Once Semantics

- Executed once or not at all
- A result informs the invoker that the method was executed exactly once
- An exception informs the invoker that the method was not executed at all
- At-most-once semantics is achieved by using all available fault tolerant mechanisms
Transparency and RPC/RMI

- Important issue
- RPC tries to make the remote procedure call look exactly like a local one
- RPC marshalling and message-passing procedures are hidden
- The notion of transparency has been extended to apply to distributed objects, but it involves hiding not only marshalling and message passing but also the task of locating and contacting the remote object
Remote invocations are more vulnerable to failure than local ones.

It is impossible to distinguish between failure of the network and failure of the remote server process.

Objects using RMI need to be able to recover from such situations.

Latency of an RMI is several orders of magnitude greater than that of a local one.
The current consensus seems to be that remote invocations should be made transparent in the sense that the syntax of a remote invocation is the same as that of a local invocation, but that the differences between local and remote objects should be expressed in their interfaces.
Concurrency - Key Point

The knowledge that an object is intended to be accessed by remote invocation has another implication for its designer - it should be able to keep its state consistent in the presence of concurrent accesses from multiple clients.
Implementation of RMI
RMI Component Terminology

- **Communication module** - carries out the request-reply protocol, providing a specified invocation semantics
- **Remote Reference module** - translates between local and remote object references (as well as creating remote object references)
- **Servant** - an instance of a class that handles the remote requests
The RMI Software Components

- **Proxy** - makes the remote method invocation transparent, marshals arguments and unmarshals results, and sends and receives messages.
- **Dispatcher** - receives the request message from the communication module and selects the appropriate method.
- **Skeleton** - implements the remote interface, unmarshals the arguments in the request message and invokes the corresponding method in the servant. It then waits for the method to complete, then marshals the results (and maybe an exception) in a reply message to the sending proxy's method.
Generating Classes

- The classes for the proxy, dispatcher and skeleton are generated automatically by an interface compiler.
- The Orbix technology works with CORBA and interfaces are defined in CORBA IDL.
- The Java RMI compiler generates the proxy, dispatcher and skeleton classes from the class of the remote object.
Dynamic Invocation

- An alternative to proxies
- A client program receives a remote reference to an object whose remote interface was not available at compile time
- CORBA provides the Interface Repository
- Java RMI allows for the dynamic downloading of classes to clients
The RMI Binder Service

• A binder on a distributed system is a separate process that maintains a table containing mappings from textual names to remote object references
• It is used by servers to register their remote object by name and by clients to look them up
• CORBA has a dedicated Naming Service
• Java has a binder called RMIregistry
Activators

- Processes that start server processes to host remote objects are called **activators**
- Registers object that are available for activation by recording the names of the servers against the URLs or filenames of the corresponding object
- Starts named server processes and activates remote object in them
- Keeps track of the locations of the servers for any remote objects that are already activated
Persistent Object Stores

- We want a guarantee that objects live between activations of processes
- CORBA provides the **persistent state service**
- Java provides **Java Data Objects** and **Persistent Java**
- (Re-)Activation of persistent objects is generally designed to be transparent
Locating Objects

- A **location service** helps clients to locate remote objects from their remote object references.
- It uses a database that maps remote object references to their probable current locations.
- The locations are "probable" because an object may have migrated since it was last heard of.
The aim of a distributed garbage collector is to ensure that if a local or remote reference to an object is still held anywhere in a set of distributed objects, then the object itself will continue to exist, but as soon as no object any longer holds a reference to it, the object will be collected and the memory it uses recovered.
Designing Distributed Garbage Collection

- Typically based on a **reference counting scheme**
- Whenever a remote object reference enters a process, a proxy will be created and will stay there for as long as it is needed
- The server should be informed of the (new) proxy at the client
- When the proxy ceases to exist at the client, the server should be informed
Implementing Distributed Garbage Collection

- Each server process maintains a set of the names of the processes that hold remote object references for each of its remote objects.
- When a client first receives a remote reference to a particular remote object, it informs the server of that remote object and creates a local proxy.
- The server adds the client to the list of referrers to that object.
- When the client's garbage collector notices that the proxy is no longer needed, it informs the corresponding server and deletes the proxy.
- The server removes the client from the list of referrers to that object.
- When the list of referrers is empty, the server's garbage collection technology reclaims the space used by the object.
Distributed Garbage Collection Semantics

- Pairwise request-reply communication
- At-most-once invocation semantics
- Java's distributed garbage collection can tolerate communications failures by ensuring that the operations on the server which add/remove a referrer are idempotent
- Java's "leases" also allow failures on the client-side to be tolerated
Remote Procedure Call (RPC)

- Like RMI, in that the client program calls a procedure in another program running in a server process
- However, it lacks the ability to create a new instance of objects and therefore does not support remote object references (Note: most RPCs are based on the C programming language)
- At-least-once and at-most-once invocation semantics are the most common for RPC
- Generally, RPC is implemented over a request-reply protocol
RPC in Action

Client Request → Reply Communication

Client process

Client stub procedure

Communication module

Server process

Server stub procedure

Communication module

Dispatcher

Service procedure
Implementing RPC

- The client that accesses a service includes one **stub procedure** for each procedure in the service interface (that is, as provided "statically" by the server)
- The stub procedure is similar to RMI's proxy
- Servers implement **stub procedures** and **skeletons**
Case Study: Sun RPC

- Sun RPC was designed in the early 90s to support client-server communication within Sun's NFS technology (network file system)
- Sun RPC is essentially identical to ONC RPC (Open Network Computing) and is documented in RFC 1831
- Implementors can choose between RPCs over UDP or TCP
- Sun RPC uses at-least-once semantics
Sun RPC's Interfaces

- An interface definition language called XDR is used (very C-like in syntax)
- XDR is rather primitive in comparison to CORBA's IDL or Java
- The RPC interface compiler is called rpcgen
When rpcgen Generates

- Client stub procedures (to be called as if local procedures)
- The server's main procedure, dispatcher, and server stub procedures
- The XDR marshalling and unmarshalling procedures for use by the dispatcher and the client/server stub procedures
Example XDR

```
const MAX = 1000;

struct Data {
    int length;
    char buffer[ MAX ];
};

struct writeargs {
    int f;
    int pos;
    Data data;
};

struct readargs {
    int f;
    int pos;
    int length;
};

program SAMPLEREADWRITE {
    version VERSION {
        void WRITE( writeargs ) = 1;
        Data READ( readargs ) = 2;
    } = 2;
} = 9999;
```
rpcgen Generated Header file

sample.h (see PDF file)
rpcgen Generated Client Code

sample_clnt.c (see PDF file)
rpcgen Generated Server Code

sample_svc.c (see PDF file)
rpcgen Generated XDR Code

sample_xdr.c (see PDF file)
Sun RPC runs a local binding service called the **port mapper** at a well-known protocol port number on each server computer.

- When a server starts up, it registers with the binding service (port mapper).
- When the client starts up, it requests the port number of the remove service by requesting the necessary data from the port mapper.
Authentication in RPC

- Request and reply messages provide additional fields to support authentication
- The server program is responsible for enforcing access control by checking the information in the authentication fields
- Several different authentication protocols are supported: none, UNIX-style (uid/gid), shared-key signing and Kerberos-style
Events and Notifications

- The idea behind the use of events is that one object can react to a change occurring in another object.
- Distributed event-based systems extend the local event model by allowing multiple objects at different locations to be notified of events taking place at an object.
The publish-subscribe paradigm is used

An object publishes the type of events that it will make available

Other objects can subscribe to the types of events that are of interest to them

Objects that represent events are called notifications

Subscribing to an event is called "registering interest"
Main Characteristics of Distributed Event-based Systems

- **Heterogeneous** - components within the distributed system that were not (initially) designed to inter-operate can be made to work together
- **Asynchronous** - notifications are sent asynchronously by event-generating objects to all the objects that have subscribed to them to prevent publishers needing to synchronize with subscribers
- Publishers and subscribers are de-coupled
The Participants in Distributed Event Notification

1. Object of interest is notified to the observer.
2. The observer notifies the subscriber.
3. The subscriber is notified of the event.
The Participants Explained

- The main component is an **event service** that maintains a database of published events and of subscribers' interests
- The **object of interest** - an object that experiences changes in state as a result of its operations being invoked, resulting in the transmission of event notifications
- The **event** - occurs at the result of the completion of a method execution
- **Notification** - an object that contains information about an event
- **Subscriber** - receives notifications about events that it has subscribed to
- **Observer objects** - decouples an object of interest from its subscribers, as it can have many different subscribers with many different needs
- **Publisher** - an object that declares that it will generate notifications of particular types of event (and it may be an object of interest or an observer)
Case Study: Java's Jini

- The Jini distributed event specification allows a potential subscriber in one JVM to subscribe to and receive notifications of events in an object of interest in another JVM (usually running on another computer).
- A chain of observers may be inserted between the object of interest and the subscriber.
Jini's Objects

- **Event Generator** - allows other objects to subscribe to its events and generates notifications
- **Remote Event Listeners** - an object that receives notifications
- **Remote Events** - an object that is passed by value to remote event listeners (this is equivalent to a notification)
- **Third-party Agents** - may be interposed between an object of interest and a subscriber (this is equivalent to an observer)
How Jini Works

- An object subscribes to events by informing the event generator about the type of event and specifying a remote event listener as the target for the notifications.
- Not surprisingly, Jini is built on top of Java's RMI technology.
Case Study: Java RMI

- Extends the Java object model to provide support for distributed objects in the Java language
- It allows objects to invoke methods on remote objects using the same syntax as for local invocations
- Objects make remote invocations are aware that the object is remote as they have to handle RemoteException events
- The implementor of the remote object must implement the Remote interface
Programming distributed applications in Java is relatively simple due to the fact that a single-language system is used.

However, the programmer is responsible for considering the behaviour of the developed object within a concurrent environment.
Interfaces and Parameters

- Typically remote objects must implement the `Serializable` interface
- A serializable object can be passed as an argument or result in Java RMI
- Parameters of a method are assumed to be input parameters, the result of the invocation of the method is a single output parameter
As Java is designed to allow classes to be downloaded from one virtual machine to another, there is no need to keep the complete set of classes within every working environment (as they can be downloaded on demand).

Both client and server programs can make transparent use of instances of new classes whenever they are added.

The RMIregistry is the binder for Java RMI.

An instance of RMIregistry must run on every server computer that hosts remote objects.

This is not a system-wide binding service - clients must direct their lookup enquiries to particular hosts.
Paradigms for distributed programming:

RPC, RMI and event-based systems