Lecture 8
CORBA, Java RMI, .NET Remoting
(Review) Distributed Objects

• Model a service as an object (with state and methods), whose interface and implementation reside on different machines
  – Clients access object using a proxy that presents the same interface

Advantages
• Hiding of differences between local and remote interactions
  – Call site looks the same in local and remote case
    • All necessary information encapsulated in the object reference
• Better support for state management at client and server
  – Object reference identifies necessary state
  – Transmission of object references avoids (expensive) state transfer
• Closer integration of type system with distributed architecture
  – Compile-time checking of message format, other errors
  – Run-time inspection of object reference to determine appropriate format

Example systems: CORBA, DCOM, Java RMI, .NET Remoting
(Review) Programming with Distributed Objects

**Client host**
- Client program
- Proxy
- Client OS
- Client invokes a method
- Proxy generation tightly integrated into type system

**Server host**
- Server program
- Server Stub
- Server OS
- Stub invokes method
- Marshalled invocation
- State
- Methods
- Interface

**Association of state with request**
- Network
- Message can contain remote object references

Differences from RPC shown in green font

10/2/2003
(Review) Common Issues for Distributed Objects

• Parameter passing
  – RPC-like passing of structures (by value)
  – Plus, passing of (opaque) object references (by reference)

• Object implementation
  – Compile-time objects: Implementation is instance of a class declared at compile-time in an object-oriented language (e.g., Java, .NET languages)
  – Run-time objects: Object adaptor creates illusion of object

• Object binding
  – Implicit: Binding happens when a remote object reference is accessed
  – Explicit: Binding requires explicit call

• Representation of object references
  – Name server, server identifier, object ID

• Object persistence and activation
  – Is there a notion of object state, even when object not in memory?
CORBA

Common Object Request Broker Architecture
• A language-neutral distributed object architecture
• Standardized by the Object Management Group (OMG)
  – Same folks who standardized DCE RPC
  – Microsoft’s answer to CORBA: DCOM
    • Built on top of Microsoft RPC and COM (Common Object Model)
    • Efficient support for interactions between objects located on the same machine

Features
• Parameter passing: Uses own wire format, called CDR
• Object implementation: Associated with IDL interface at run time
• Object binding: Explicit, via separate function calls
• Object references: Refer to a name server, support dynamic invocation
  – Permit queries about implemented interfaces
• Object persistence: Persistent, Activatable
CORBA Architecture

Client Host

- Client application
  - Static IDL stub (proxy)
  - Dynamic Invocation Interface
- Object Request Broker (ORB)
- OS

Server Host

- Object implementation
  - Server stub (skeleton)
- Dynamic Interface
  - Object adapter
- ORB
  - interfaces implementations
- OS
CORBA Object Services

A standard set of services (also implemented as CORBA objects) that provide commonly-required low-level and basic functionality

- Collection Service
- Concurrency Service
- Event Service
- Externalization Service
- Licensing Service
- Life Cycle Service
- Naming Service
- Persistence Service
- Properties Service
- Query Service
- Relationship Service
- Security Service
- Time Service
- Trader Service
- Transaction Service
- …
Java **Remote Method Invocation (RMI)**

Java language-level support for remote objects

- **Parameter passing**: Uses own wire format
  - Parameters to RMI methods extend a predefined interface (*Serializable*)
    - “serializing” = marshalling
  - Possible to define custom serialization routines

- **Object implementation**: Compile-time definition
  - RMI interfaces extend a predefined interface (*java.rmi.Remote*)
  - Implementation class implements RMI interface
    - Typically by extending a predefined class
      - *java.rmi.Activatable*, *java.rmi.UnicastRemoteObject*
  - Client and server stub code automatically generated by invoking *rmic*
  - Additional exceptions defined for remote interactions
Java RMI (cont’d)

• **Object binding**: Both explicit and implicit
  – **Explicit**: Using static methods in the `java.rmi.Naming` class
    • Server binds/rebinds name (string) with a nameserver (`rmiregistry`)
    • Client looks up name with the nameserver
  – **Implicit**: Whenever a client receives a remote object reference
    • Can invoke methods on it as if it were a local interface

• **Object references** (`RemoteRef`)
  – Stores information about the server name, unique object ID, codebase
  – Distributed GC using reference counts

• **Object persistence**: Both transient and persistent objects supported
.NET Remoting

- Provides (almost all of) the functionality of Java RMI …
  - Except for activation of a persistent object

- … with the following additional benefits
  - Remoting framework works at the level of the CLR
    - Hence, able to interoperate across all .NET languages
  - Extensible wire-formats and transports
    - Comes with Binary Encoding/TCP and SOAP/HTTP
    - Application developers can build their own
  - Multiple “activation” modes
    - Activation here refers to creation of a new object instance (different from passivating/activating a persistent object)
    - Server Activation
    - Client Activation
  - Lifetime management (GC) of remote objects using “leases”
    - Involve less traffic as compared to reference counting-based schemes
.NET Remoting: Architecture

Functionality

- Server makes available a type at a well-known end-point
  - An operation against this type results in an instance being created (unless one already exists)
- Client makes a request for the type
  - Obtains a proxy that provides the same interface as the type
    - Method invocations against the proxy are forwarded to the server object
  - Can forward the proxy on to other objects

Details

- What kind of types can be made available using .NET remoting?
- What does an end-point look like?
  How does a client make a request against an end-point?
- How are parameters/return values passed in method invocations?
- What information is contained in a proxy?
- When are new object instances created at the server? When do these die?
.NET Remoting: Remotable Types

- Any type that extends the class `MarshalByRefObject`
- Type can have fields …
  - Define the state of the object
- … and methods
  - Object references that are not remote should be passed by value
    - Require that the object implement the `ISerializable` interface or have the `[serializable]` attribute
    - A new copy gets created at the destination
  - Remote object references are passed by reference
    - Results in an `ObjRef` instance: the representation of the object reference
      - Contains all of the information required to locate and access the object from anywhere on the network
      - Class hierarchy, interfaces it implements, object URI, …
.NET Remoting: End-Points

• End-point = Channel + Name

• **Channel**: Transport for transferring messages to/from the remote object
  – .NET framework provides the following three: **TCP, HTTP, SMTP**
    • In each case, a unique port number is required
  – Application developer can build their own
  – At least one channel must be registered for remotable objects
  – Can have many channels per application

• **Name**: A Universal Resource Identifier (URI) that identifies the type being exported
  – Must be consistent with a registered channel
  
  • `tcp://...`, `http://...`, `smtp://...`
.NET Remoting: More About Channels

• In general, a channel is built up out of a chain of processes
  – “channel sink chain”

• **Formatter** sinks
  – Serialize messages into streams of bytes (wire-format)
  – Built-in: *SOAP* and **Binary Formatter**
  – Custom Formatters allow talking to any endpoint
    • E.g., IIOP.NET allows interoperability with CORBA

• **Transport** Sinks
  – Establish a connection to the transport sink on the client/server
  – Forward the formatted message to another transport sink
  – Built-in: **TCP** and **HTTP**

• Custom sinks can be added to the chain
  – Logging, encryption, …
.NET Remoting: Proxies

- Implementation consists of two parts
  - **Real** proxies: The (generic) communication layer
  - **Transparent** proxies: Provide the same interface as the remote object
    - Built dynamically by real proxy
.NET Remoting: Object Activation

- Type available only as long as there is an **active listener**
  - With registered channel
  - Different from COM, Java RMI

- Server exposes well known object for clients to connect
  - Bound to known channels with known name

  Two kinds:
  - **single call**: Object instance is created for each call on channel
    - Implements the stateless model of the web.
  - **singleton**: One shared instance provided for all clients
    - Serves as "gateway" into stateful application

  - Object instances are created on-demand by the server (**server activation**),
    or can be managed by the client (**client activation**)
.NET Remoting: Example 1

- Namespaces
  - `System.Runtime.Remoting.RemotingServices`
    - Provides fundamental remoting infrastructure
      - Remoting configuration, Connecting to remote object instances, Exposing "well known objects"
  - `System.Runtime.Remoting.ChannelServices`
    - Channel registration and management
  - `System.Runtime.Remoting.LifetimeServices`
    - Lease-based lifecycle management for objects
  - `System.Runtime.Remoting.TrackingServices`
    - Universal hooks for tracking remoting activities

- [Code walkthrough of a simple string-reverser application using .NET Remoting]