Object-based distributed systems

INF 5040/9040 autumn 2009

Lecturer: Frank Eliassen

Plan

- Request-response protocols
- Characteristics of distributed objects
- Communication between distributed objects (RMI)
- Object-servers
  - Multi-threaded object servers
- CORBA middleware
- Java RMI
Layering of middleware

- Applications
- Middleware services
- RMI og RPC
- Request-response protocols
- Marshalling and external data representation
- UDP or TCP

Request-reponse protocols
Basis for client-server communication

- Usually based on UDP or TCP

Client

- DoOperation
- wait
  (continue)

Server

- getRequest
- select object
- execute method
- sendReply

Message structure
RMI-like protocol

- message type
- request id
- ROR
- method name
- arguments

Frank Eliassen, SRL & Ifi/UiO
Failure model for request-response protocols

- Protocol can be exposed to
  - omission failure
  - process crash failure
  - message order not guaranteed (UDP)
- Failure is detected as *timeout* in the primitive DoOperation:
  - recovery actions depend on the offered delivery guarantee

Failure and recovery for request-response protocols (I)

- Timeout DoOperation
  - Send request message repeatedly until
    - response is available, or
    - assume server has failed (max no of retrans.)
- Duplicate request messages
  - occur when request message is sent more than once
  - can lead to operations being executed more than once for the same request!
  - => must be able to filter duplicate requests (role of request id)
- Lost response messages
  - server has already sent response message when it receives a duplicate request message
  - => may have to execute the operation again to get the right response
    - OK for operations that are "idempotent"
Failure and recovery for request-response protocols (II)

- Logs (histories):
  - used by servers offering operations that are not “idempotent”
  - contains response messages already sent
- Disadvantage of logs:
  - storage requirement
- if a client is allowed to do only one request at a time to the same server, the log can be limited in size (bounded by the number of concurrent clients)
- at reception of the next request message from the same client the server may delete the last response message for that client from the log

Classification of request-response protocols

- Classification after (Spector, 1982):
  - basis for implementing different types of RMI and RPC (with different levels of delivery guarantees)
- Request (R) protocol
  - Only Request-message. No response message from server
  - No confirmation that operation has been performed (one-way operation)
- Request-Reply (RR) protocol
  - Reply-message confirms that the Request-message has been performed
  - A new request from the client confirms reception of Reply-message
- Request-Reply-Acknowledge (RRA) protocol
  - separate message from client to confirm reception of Reply-message
  - tolerates loss of Ack-message
    - Ack with a given request id confirms all lower requests ids
A note on request-response protocols: TCP vs UDP

- UDP has limited packet size
  - => need for fragmentation/defragmentation protocols
- request-response protocols over TCP avoids this problem
  - TCP ensures reliable delivery of byte streams
- Problem:
  - much overhead if the connection has to be created at each request
  - => need for optimization (leave connection open for later reuse)
  - upper bound on number of concurrent TCP-connections could cause problems

Layering of middleware

Middleware

- Applications
- Middleware services
- RMI og RPC
- request-response protocols
- marshalling and external data representation
- UDP og TCP
Message data representation

- Data structures must be flattened before transmission and rebuilt on arrival
- Issue: the representation of data structures and primitive data types can be different between systems
- Two methods for exchanging binary data values
  - Use external format
    - sender converts data values to an agreed external format
    - recipient converts to local form
  - Use sender’s format
    - recipient converts the values if necessary
    - message carries indication of format used

Marshalling

- “marshalling”
  - serialize data structures to messages (sequence of data values)
  - translate sequence of data values to an external representation
- “unmarshalling”
  - inverse of “marshalling”
Some external data representation formats

- Sun XDR (representation of most used data types)
- ASN.1/BER (ISO standard, based on “type-tags”, open)
- NDR (used in DCE RPC)
- CDR (used in CORBA RMI, binary layout of IDL types)
- Java Object Serialization (JOS)
- XML (used in SOAP: “RMI” protocol for Web Services)

Plan

- Request-response protocols
- Characteristics of distributed objects
- Communication between distributed objects (RMI)
- Object-servers
  - Multi-threaded object servers
- CORBA middleware
- Java RMI
Characteristics of distributed objects - I

- Distributed objects execute in different processes.
  - Each object has a **remote interface** for controlling access to its methods and attributes that can be accessed from other objects in other processes located on the same or other machines
    - Declared via an “Interface Definition Language” (IDL)
  - Remote Method Invocation (RMI)
    - Method call from an object in one process to a (remote) object in another process

Characteristics of distributed objects - II

- Distributed objects have a unique identity referred to as a **Remote Object Reference (ROR)**
- Other objects that want to invoke methods of a remote object needs access to its ROR
- RORs are “first class values”
  - Can occur as arguments and results in RMI
  - Can be assigned to variables
- Distributed objects are encapsulated by interfaces
- Distributed objects can raise “exceptions” as a result of method invocations
- Distributed objects have a set of named attributes that can be assigned values
The type of a distributed object

- Attributes, methods and exceptions are properties objects can export to other objects
- These properties determine the type of an object
- Several objects can export the same properties (same type of objects)
- The type is defined once
- The object type is defined by the interface specification of the object

Declaration of remote methods

- A remote method is declared by its signature
- In CORBA the signature consists of
  - a name
  - a list of in, out, and inout parameters
  - a return value type
  - a list of exceptions that the method can raise

  ```
  void select (in Date d) raises (AlreadySelected);
  ```
RPC/RMI invocation semantics: design choices

- Reliability semantics of RPC/RMI under partial failures

<table>
<thead>
<tr>
<th>Fault tolerance measures</th>
<th>Invocation semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retransmit request message on timeout?</td>
<td>Duplication filtering</td>
</tr>
<tr>
<td>No (R)</td>
<td>____</td>
</tr>
<tr>
<td>Yes (RR)</td>
<td>No</td>
</tr>
<tr>
<td>Yes (RR)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

RMI invocation semantics in object and RPC middlewares

- RMI in CORBA and Java have “at-most-once” invocation semantics under partial failures
  - referred to as synchronous requests
- CORBA allows other forms of synchronization that provides other invocation semantics
  - One-way operations: maybe-semantics
  - Deferred synchronous RMI
- SUN RPC: at-least-once semantics
Communication between distributed objects

Middleware

Applications
Middleware services
RMI og RPC
request-response protocols
marshalling and external data representation
UDP og TCP

Remote method invocations

- A client object can request the execution of a method of a distributed, remote object
- Remote methods are invoked by sending a message (including method name and arguments) to the remote object
- The remote object is identified and located using the remote object reference (ROR)
- Clients must be able to handle exceptions that the method can raise
Remote object with remote interface

- Three main tasks:
  - Interface processing
    - Integration of the RMI mechanism into a programming language.
    - Basis for realizing access transparency
  - Communication
    - Message exchange (request-reply protocol)
  - Object location, binding and activation
    - Locate the server process that hosts the remote object and bind to the server
    - Activate an object-implementation
    - Basis for realizing location transparency

Frank Eliassen, SRL & IfI/UiO
RMI interface processing: role of proxy and skeleton

Elements of the RMI software (I)

- RMI interface processing: Client proxy
  - Local “proxy” object for each remote object a client holds a ROR (“stand-in” for remote object).
  - The class of the proxy-object has the same interface as the class of the remote object. Can perform type checking on arguments.
  - Performs marshalling of requests and unmarshalling of responses.
  - Transmits request-messages to the server and receive response messages.
Elements of the RMI software (II)

- RMI interface processing: Dispatcher
  - A server has one dispatcher for each class representing a remote object.
  - Receives requests messages
  - Uses method id in the request message to select the appropriate method in the skeleton (provides the methods of the class) and passes on the request message.

Elements of the RMI software (III)

- RMI interface processing: Skeleton
  - A server has one skeleton for each class representing a remote object.
  - Provides the methods of the remote interface.
  - A skeleton method unmarshals the arguments in the request message and invokes the corresponding method in the remote object.
  - It waits for the invocation to complete and then marshals the result, together with any exceptions, in a reply message to the sending proxy's method.
RMI interface processing: role of remote object reference module

Client ROR module
remote object table: MAP(ROR,proxy)

Communication module

Server ROR module
remote object table: MAP(ROR,servant)

Skeleton & dispatcher for B’s class

Generation of proxies, dispatchers and skeletons

Design
IDL definitions
Precompile

Skeletons
Add server code

Compile
client implementation

Stubs

client code

Compile

Uses

INF 5040 autumn 2009
Server and client programs

- Server program contains
  - the classes for the dispatchers and skeletons
  - the implementation classes of all the servants that it supports
  - an initialization section: creates and initializes at least one servant
    - additional servants (objects) may be created in response to client requests
  - register zero or more servants with a binder
  - potentially one or more factory methods that allow clients to request creation of additional servants (objects)

- Client program contains
  - the classes and proxies for all the remote objects that it will invoke

Source: S. Krakowiak, Middleware Architecture with Patterns and Frameworks,
http://sardes.inrialpes.fr/~krakowia/MW-Book/

Factory pattern for creating additional objects

source: S. Krakowiak, Middleware Architecture with Patterns and Frameworks,
Deferred synchronous RPC (also applicable to RMI)

RMI name resolution, binding, and activation

- **Name resolution**
  - corresponds to mapping a symbolic object name to an ROR
  - performed by a name service (or similar)
- **Binding in RMI**
  - corresponds to locating the server holding a remote object based on the ROR of the object and placing a proxy in the client process's address space
- **Activation in RMI**
  - corresponds to creating an active object from a corresponding passive object (e.g., on request). Performed by an *activator*
    - register passive objects that are available for activation
    - activate server processes (and activate remote object within them)
Locating the server of a remote object

- Corresponds to mapping an ROR to a communication identifier.
  - integrated in ROR
    - Address can be extracted directly from the object reference
  - location service
    - A location service is used by the client proxy at each request
  - cache/broadcast
    - Each client has cache of bindings (ROR, comm. identifier)
    - If ROR not in cache, perform broadcast with ROR
    - Servers that host the object respond with comm.identifier
  - forward pointers or address hint (to e.g., location service)
    - Used at object migration
  - Combinations of the above

Remote method invocation

Implicit and explicit binding

Distr_object* obj_ref;
// Declare a system wide object reference
obj_ref = lookup(obj_name);
// Initialize the reference to a distrb. obj
obj_ref->do_something();
// Implicit bind and invoke method

Distr_object* obj_ref;
// Declare a system wide object reference
Local_object* obj_ptr
// Declare a pointer to a local object
obj_ref = lookup(obj_name);
// Initialize the reference to a distrb. obj
obj_ptr = bind(obj_ref);
// Explicitly bind and get pointer to local proxy
obj_ptr->do_something();
// Invoke a method on the local proxy

Plan

- Request-response protocols
- Characteristics of distributed objects
- Communication between distributed objects (RMI)
- Object-servers
  - Multi-threaded object servers
- CORBA middleware
- Java RMI
Object-server:
Server tailored to support
distributed objects

- Services realized as objects that the server encapsulates
  - Services can be added or removed by creating and removing remote objects
- Object servers act as places where objects can live
- Object servers activate remote objects on demand
  - Several ways to activate an object

Object servers must assign processing resources to objects when they are activated

- When an object is activated, which processing resources should be assigned to the implementation?
- Activation policy
  - A particular way of activating an object
  - Different dimensions
    - How to translate between ROR and local implementation?
    - Should the server be single-threaded or multi-threaded?
    - If multi-threaded, how to assign threads to objects and requests?
      - One thread per object? One per request?
      - Transient vs persistent objects, etc
- No single activation policy that fits all needs
  - Object servers should support several concurrent activation policies
  - Objects can be grouped according to which activation policy they are governed by
Organization of object servers that support different activation policies

- **Object-adapter**: software that implements a specific activation policy (supported by CORBA Portable Object Adapter (POA))

Object references

- **Remote-object-reference (ROR)**
  - Identifier for remote objects that is valid in a distributed system
  - Must be generated in a way that ensures uniqueness over time and space (=> a ROR can not be reused)
  - Example:

<table>
<thead>
<tr>
<th>Internet address</th>
<th>port number</th>
<th>adapter name</th>
<th>object key</th>
<th>interface of remote object</th>
</tr>
</thead>
</table>
Plan

- Request-response protocols
- Characteristics of distributed objects
- Communication between distributed objects (RMI)
- Object-servers
  - Multi-threaded object servers
- CORBA middleware
- Java RMI

Object-servers must assign processing resources to objects when objects are activated

- When an object is activated, which processing resources should be assigned to its implementation?
  - Create a new process or thread?
  - Are there several ways this can be done?
  - Is there a best way (cf. activation policies)?
Multi-thread server

<table>
<thead>
<tr>
<th># processors</th>
<th># threads</th>
<th>Max. calls/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>no disk caching</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>no disk caching</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>disk caching</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>disk caching</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>disk caching</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Alternative threading-policies for object activation

a. Thread-per-request

b. Thread-per-connection

c. Thread-per-object
Plan

- Principles for realising remote methods invocations (RMI)
- Object-servers
- Multi-threaded object servers
- CORBA RMI
- Java RMI

CORBA middleware

- Offers mechanisms that allow objects to invoke remote methods and receive responses in a transparent way
  - location transparency
  - access transparency
- The core of the architecture is the Object Request Broker (ORB)
- Specification developed by members of the Object Management Group (www.omg.org)
CORBA RMI

Clients may invoke methods of remote objects without worrying about:
- object location, programming language,
- operating system platform, communication
- protocols or hardware.

CORBA supports language heterogeneity

- CORBA allows interacting objects to be implemented in different programming languages
- Interoperability based on a common object model provided by the middleware
- Need for advanced mappings (language bindings) between different object implementation languages and the common object model
Elements of the CORBA common object model

- Metalevel model for the type system of the middleware
- Defines the meaning of e.g.,
  - object identity
  - object type (interface)
  - operation (method)
  - attribute
  - method invocation
  - exception
  - subtyping/inheritance
- Must be general enough to be mappable to common programming languages
- CORBA Interface Definition Language (IDL)

CORBA IDL

- Language for specifying CORBA object types (i.e. object interfaces)
- Can express all concepts in the CORBA common object model
- CORBA IDL is
  - not dependent on a specific programming language
  - syntactically oriented towards C++
  - not computationally complete
- Different bindings to programming languages available
A CORBA IDL specification

Implicit supertype Object

Supertype

Object type

Interface Organisation {
  readonly attribute string name;
}

Supertype

Parameter-direction

Interface Club : Organisation {
  exception NotInClub();
  readonly attribute short noofMembers;
  readonly attribute Address addr;
  attribute sequence<Trainer> trainers;
  attribute sequence<Players> team;
  void sale(in Players)
  raises NotInClub;
};

Return value

Exception

Language bindings for CORBA IDL

Common object model

IDL

Java

Smalltalk

C++

Ada-95

Cobol

C

Frank Eliassen, SRL & Ifi/UiO
CORBA architecture

Description of registered types:
- methods
  -- parameters
  - ...

Interface repository
Impl. repository

Run time catalogue:
- classes that a server impls.
- activation info for server and object impl.
  - ...

Client
Client program
ORB Core
proxy for A

request

server
ORB Core
Object adapter
Skeleton

Servant B

reply

IIOP

POA: manages and controls object impls:
- registering object impl.
- activating object impl.
  - ...

CORBA ROR format (IOR) reflects the organization of object servers

CORBA Interoperable Object Reference (IOR)

Type name (repository id)
Protocol and Address details
Object Key (Adapter and Object name)

CORBA ROR format (IOR) reflects the organization of object servers

Frank Eliassen, SRL & Ifi/UiO 55

Frank Eliassen, SRL & Ifi/UiO 56
CORBA RMI binding

- Binding in RMI corresponds to mapping object references (ROR) to “servants”
  - servant: implementation of one or more CORBA objects
- ROR in CORBA: Interoperable Object Reference (IOR)
- Location process:
  - based on information encoded in the object reference

CORBA Interoperable Object Reference (IOR)

Layering of middleware

Middleware

Applications
Middleware services
RMI og RPC
request-response protocols
marshalling and external data representation
UDP og TCP
Plan

- Principles for realising remote methods invocations (RMI)
- Object-servers
- Multi-threaded object servers
- CORBA RMI
- Java RMI
Java Remote Method Invocation (RMI)

Java RMI

- Remote Method Invocation (RMI) supports communication between different Java Virtual Machines (VM), and possibly over a network
- Provides tight integration with Java
- Minimizes changes in the Java language/VM
- Works for homogeneous environments (Java)
- Clients can be implemented as *applet* or *application*
Java Object Model

- Interfaces and Remote Objects
- Classes
- Attributes
- Operations/methods
- Exceptions
- Inheritance

Java interfaces to remote objects

- Based on the ordinary Java interface concept
- RMI does not have a separate language (IDL) for defining remote interfaces
- Pre-defined interface `Remote`
- All RMI communication is based on interfaces that extends `java.rmi.Remote`
- Remote classes implement `java.rmi.Remote`
- Remote objects are instances of remote class
Java remote interface:
Example

interface Team extends Remote {
    public:
    String nama() throws RemoteException;
    Trainer[] trained_by() throws RemoteException;
    Club club() throws RemoteException;
    Player[] player() throws RemoteException;
    void chooseKeeper(Date d) throws RemoteException;
    void print() throws RemoteException;
};

Java RMI parameter passing

- Atomic types transferred by value
- Remote objects transferred by reference
- None-remote objects transferred by value

class Address {
    public:
    String street;
    String zipcode;
    String town;
};

interface Club extends Organisation, Remote {
    public:
    Address addr() throws RemoteException;
    ...
};

Returns a copy of the Address-object
Architecture of Java RMI

Summary - I

- Request-response protocols
- Distributed objects executes in different processes.
  - remote interfaces allow an object in one process to invoke methods of objects in other processes located on the same or on other machines
- Object-based distribution middleware:
  - middleware that models a distributed application as a collection of interacting distributed objects (e.g., CORBA, Java RMI)
Summary - II

- Implementation of RMI
  - proxies, skeletons, dispatcher
  - interface processing, binding, location, activation
- Invocation semantics (under partial failure)
  - maybe, at-least-once, at-most-once
  - Reliability of RMI is at best “at-most-once”
- Multi-threaded servers
  - can in some cases be used to increase the throughput (method calls/time unit) if, e.g., I/O is the bottleneck
- Principles of CORBA
  - Clients may invoke methods of remote objects without worrying about: object location, programming language, operating system platform, communication protocols or hardware.
- Principles of Java RMI
  - Similar to CORBA but limited to a Java environment

Extra slides
CORBA
Portable object adapter (POA)
- Enables portability of object implementations across different ORBs
- Supports lightweight transient object and persistent object identifiers (e.g., for objects stored in databases)
- Supports transparent object activations
- Extensible mechanism for activation policies
- Several POAs in one single server

CORBA RMI binding (II)
- Transient IOR
  - Valid only as long the corresponding server process is available
  - After the server process has terminated, the IOR will never be valid again
  - The location of the server process is encoded into the IOR of the object.
- Persistent IOR
  - continue to function (denote same CORBA object) even when the server process terminates and later starts up again
  - An activator (implementation repository) can automatically start a server process when a client is using a persistent object reference and terminate the server again after a certain idle time
  - The location of the activator is encoded into the IOR of the object IOR. The actual location of the server process must be resolved via the activator.
  - A persistent POA must be registered at an activator.
  - A persistent POA creates persistent IORs and knows how to activate persistent objects that it manages
Locating transient IORs over IIOP

- Client
  - Object reference (IOR)
    - IDL:MyObj bobo:1805 OA9.obj_979

- Server on bobo:1805
  - OA9
    - obj_971
    - obj_979
    - obj_983

Locating persistent IOR over IIOP

- Uses Implementation Repository (IR) as activator.
- IR:
  - Handles process/thread-creation and -termination, a.m.
  - Is not portable (specific to an ORB implementation)
  - Not standardized
    - Tailored to specific environments
    - Not possible to write specifications that cover all environments
    - Communication between an ORB and its IR is not visible to the client
  - Object migration, scaling, performance, and fault tolerance are dependent on IR
  - Implemented usually as a process at a fixed address
    - A set of host machines that is configured under the same IR is denoted a "location domain"
Locating persistent IOR over IIOP

Client

Object reference
IDL:MyObj MyIR:1801 Ole, obj_979

(1) op()[Ole, obj_979]
(2) fork/exec(rsh bobo "'/usr/local/bin/Ole -x"")

(3) My_adr(bobo:1799)

Server table

ObjAdapt Start_up command address
Ole rsh bobo "'/usr/local/bin/Ole -x"
Frank /usr/local/bin/Frank bobo:1799
OA2

(4) location_forward(bobo:1799)

(5) op()[Ole, obj_979]
(7) reply to op()

bobo:1799

Java RMI development process

Define remote interface
Implement the interface

Client stub

Server class

Start RMI Registry
Start server objects
Register remote objects

Implement client

Start client

uses