Distributed Internet Applications - DIA

Principles of Object-Oriented Middleware
Why Middleware?

Distributed system construction directly on top of a transport layer is rather difficult.

- The communication will often need to send complex parameters.
- Different encodings of data types in memory.
- Parameters and return values might refer to other distributed system components.
- Developers and administrators would have to implement component activation.
- Type safety is an important concern. Achieving type safety manually is quite error prone.
- Implementing synchronization is rather tedious and error prone.
Application engineers could use network operating systems to exchange information based on sending fixed-length messages or writing into a byte stream.
Middleware simplifies distributed system construction by implementing the presentation and session layers.
Types of Middleware

1. Transaction-Oriented Middleware
2. Message-Oriented Middleware
Transaction-Oriented Middleware is often used with distributed database applications.

Object-oriented middleware has transaction-oriented middleware capabilities.
Message-Oriented Middleware is used when reliable, asynchronous communication is the dominant form of distributed system interaction.

- IBM’s MQSeries, Sun’s ToolTalk and NCR’s TopEnd

This paradigm supports:

- Asynchronous Message Delivery
- Multi-Casting

The degree of fault-tolerance is very high.

- Message queues store messages temporarily on persistent storage.

Object-oriented middleware is starting to be integrated with message-oriented middleware.
Remote Procedure Calls (RPCs)

An *Remote Procedure Call* (RPC) is a procedure call across host boundaries.

It is invented by Sun Microsystems as part of their *Open Network Computing (ONC)* platform.

The origin of object-oriented middleware is *RPC*.
The server components that execute RPCs are called **RPC programs**.

RPCs are clustered in RPC programs.

An RPC program has an **interface definition** that defines procedures that can be called remotely.

Interface definitions also define data types of arguments that can be passed to remote procedures.

The DCE (Distributed Computing Environment) includes an interface definition language that is used to define these exports in a way that is independent of programming languages.
Example: An interface definition in Sun’s RPC interface definition language.

```c
const NL = 64;
Struct Player {
    struct DoB { int day; int month; int year; }
    struct name<NL>;
};
program PLAYERPROG {
    version PLAYERVERSION {
        void PRINT (Player) = 0;
        int STORE (Player) = 1;
        Player LOAD (int) = 2;
    } = 0;
} = 105040;
```
RPC – Presentation Layer

RPC presentation layer integration maps application data structures onto a homogeneous and transmissible form.

The mappings between application and transport data representations are called *marshalling* and *unmarshalling*.

Tasks Achieved by the Presentation Layer Implementation

Resolution of data heterogeneity

- Common data representation
- Transmission of data declaration

Marshalling and Unmarshalling

- static
- dynamic
An example of marshalling and unmarshalling (generated by the IDL compiler):

```c
char * marshal() {
    char * msg;
    msg = new char[4*(sizeof(int) + 1) + strlen(name) + 1];
    sprintf(msg, "%d %d %d %d %s", dob.day, dob.month, dob.year, 
            strlen(name), name);
    return (msg);
};
void unmarshal(char * msg) {
    int name_len;
    sscanf(msg, "%d %d %d %d", &dob.day, &dob.month, &dob.year, &name_len);
    name = new char[name_len + 1];
    sscanf(msg, "%d %d %d %d", &dob.day, &dob.month, &dob.year, &name_len, 
           name);
};
```
Marshalling is specific to the particular remote procedures.

Marshalling can be implemented statically at compile time or determined dynamically at run-time.

*Stubs* are generated by remote procedure call systems.

Client and server stubs are static implementations of marshalling and unmarshalling.

RPC systems on Unix provide an interface definition language compiler, called *rpcgen*, that generates a *client stub* and a *server stub*.
The session layer implementation needs to enable clients to locate an RPC server (statically or dynamically).

```c
print_person(char * host, Player * pers) {
    CLIENT * clnt;

    clnt = clnt_create(host, 105040, 0, "udp");
    if (clnt == (CLIENT *) NULL) exit (1);
    if (print_0(pers, clnt) == NULL)
        clnt_perror(clnt, "call failed");
    clnt_destroy(clnt);
}
```

The above C code creates a client stub for the RPC program in previous example. If the creation succeeds, it calls the stub with print_0.

Is this location transparent?
Dynamic Binding

Each host of a Unix network that supports RPCs runs a daemon called **portmap**.

Any RPC server registers its programs with the **portmap** daemon that runs on the host.

- Clients can contact a single **portmap** daemon to get a list of all programs that reside on the server.

- Clients can also broadcast a search for a program to all **portmap** daemons of a network and the **portmap** daemons with which the program is registered will respond.
Object-Oriented Middleware evolved more or less directly from the idea of remote procedure calls.

The first of these systems was OMG’s Common Object Request Broker Architecture (CORBA).

Microsoft added distribution capabilities to its Component Object Model (COM).

Sun provided a mechanism for Remote Method Invocation (RMI).

The idea here is to make object-oriented principles available for the development of distributed systems.
Every object-oriented middleware has an *interface definition language* (IDL).

IDLs support the concept of object types as parameters; failure handling; and inheritance.

A disadvantage of RPCs is that they are not reflexive. This means that procedures exported by one RPC program cannot return another RPC program.

A server object that implements an object type can return other server objects.
interface Player: Object {
    typedef struct Date {
        short day; short month; short year;
    };
    attribute string name;
    readonly attribute Date DoB;
};
interface PlayerStore: Object {
    exception IDNotFound{ }
    short save ( in Player p);
    Player load ( in short id) raises (IDNotFound);
    void print ( in Player p);
};

Objects of type PlayerStore return a reference to another server object in operation load.
The presentation layer implementation of object-oriented middleware is very similar to that of RPCs.

Object-oriented middleware also supports client and server stubs.

Object-oriented middleware presentation layers need to map object references to the transport format (marshalling and unmarshalling).
Session Layer Implementation

The session layer implementation of object-oriented middleware is more complex than that of RPCs.

Transport layer implementations, such as TCP and UDP, use addressing schemes based on host name and port numbers.

RPC systems use the portmap daemon in order to map between service names and host names.

The session layer of OO middleware needs to map object references to hosts.

1. Addressing of server objects is based on object references.
2. The object references are generated by the middleware.
3. The session layer implementation of object-oriented middleware maps object references to hosts in a fully transparent way.
Session Layer Implementation

Session layer implements object activation policies in the object adapter.

- The object adapter implements the mapping of object references to active object implementations.
- Object adapters need to be able to start up servers, which register in an implementation repository or registry.

Session layer needs to implement operation dispatch.

- The object needs to invoke the requested operation. This is referred to as operation dispatch.

Session layer needs to implement synchronization.

- Object requests are, by default, synchronous forms of communication.
Developing with Object-Oriented Middleware

Design and Implementation Process

Design

Interface Definition

Server Stub Generation

Server Coding

Server Registration

Client Stub Generation

Client Coding
Design

Design the server objects based on requirements that existing and future client objects may have.

Use an CASE tool that supports an object-oriented notation (UML).

- Class diagrams
- Sequence diagrams
- State diagrams
Interface Definition

IDLs provide language constructs for all concepts of their underlying object model.

Every object-oriented middleware has an object model.

Interface definitions add considerable detail to class diagrams.

Interfaces can be seen as contracts that govern the interaction between client and server objects.

Interfaces are also the basis for distributing type information.

Client and server stubs are automatically derived from interfaces.
Stub Generation

Client and server stubs are proxies for servers and clients.

Stubs are incarnations of the Proxy Design Pattern [Gamma et al., 1995].

A proxy is placeholder that hides a certain degree of complication.

Client and server stubs hide the fact that an operation execution is being requested in a distributed fashion.

Client (server) stub resides on the same host as the client (server) object.
Stub generation

Method Calls versus Object Request

- Caller
- Called

- Stub

Transport Layer (TCP or UDP)
Stub Generation Process

Stubs are generated by the IDL compiler that is provided by the middleware.
Distributed Ruby

DRb - a library module - is a distributed object system for Ruby.

It is written in pure Ruby and uses its own protocol.

It does not rely on or interoperate with other distributed object system such as CORBA, DCOM, RMI, or .NET.
Distributed Ruby

DRb allows methods to be called remotely.

References to objects can be passed between processes.

Method arguments and return values are dumped and loaded in marshalled format.

An object in a remote process is locally represented by a `DRb::DRbObject` instance.
Example: server code

```ruby
require 'drb/drb'
URI = "druby://localhost:8787"
class TimeServer
  def get_current_time
    return Time.now
  end
end
$server_object = TimeServer.new
$SAFE = 1

DRb.start_service(URI, $server_object)
DRb.thread.join
```

[~/snb] $ ruby dservver.rb
require 'drb/drb'
SERVER_URI = "druby://localhost:8787"
DRb.start_service
timeserver = DRbObject.new_with_uri(SERVER_URI)
puts timeserver.get_current_time

[~/snb] $ ruby dclient.rb
Fri Nov 12 17:30:32 CET 2004
An instance of `DRb::DRbObject` acts as a sort proxy for the remote object.

Methods called upon this `DRb::DRbObject`, for example `timeserver` in the cline code, are forwarded to its remote object, for example `$server_object` in the server code.
Components of a DRb Application

Server:

- Start a TCP server socket and listen.
- Bind an object to the drb server instance.
- Accept connections from clients and respond to messages they send.
- Optionally provide Access Control services

Client:

- Establish a connection to a DRb server process.
- Bind a local object to the remote DRb object
- Send messages to the server object and receive its messages.
Example: Use of Factory Pattern

```ruby
require "drb/drb"
URI = "druby://localhost:8787"

class Logger
  # Make DRb send Logger instances as
  # DRb reference, not copies.
  include DRb::DRbUndumped

  def initialize(name, fname)
    @name = name
    @file_name = fname
  end

  def log(msg) # method
    f = File.new(@file_name,"a+")
    f.puts("#{Time.now}: #{@name}: #{msg}")
    f.close
  end
end
```
class LoggerFactory
  def initialize(base_dir)
    @base_directory = base_dir
    @loggers = {}
  end
  
  def get_logger(name)
    if !@loggers.has_key? name
      fname = name.gsub(/\.[\/]\/, "_").untaint
      @loggers[name] = Logger.new(name, @base_directory + "\" + fname)
    end
    return @loggers[name]
  end
end
Example: Use of Factory Pattern

SERVER_OBJECT = LoggerFactory.new("/Users/mortezanahrwar/tmp/dlog")

$SAFE = 1
DRb.start_service(URI, SERVER_OBJECT)
DRb.thread.join

THIS IS ALL
Example: Use of Factory Pattern

```ruby
require 'drb/drb'
SERVER_URI = "druby://localhost:8787"
DRb.start_service
log_service = DRbObject.new_with_uri(SERVER_URI)
```
Example: Use of Factory Pattern

```ruby
# Client code

t1 = Thread.new{
  logger = log_service.get_logger("logA")
  logger.log("Hello, World!")
  logger.log("Goodbye, World!")
  logger.log("=== EOT ===")
}

t2 = Thread.new{
  logger = log_service.get_logger("logB")
  logger.log("Hallo, beste mensen!")
  logger.log("Dag beste mensen!")
  logger.log("=== EOT ===")
}

t3 = Thread.new{
  logger = log_service.get_logger("logC")
  logger.log("123456789")
  logger.log("123456789".reverse)
  logger.log("=== EOT ===")
}
```
Example: Use of Factory Pattern

t1.join
t2.join
t3.join
puts "Bye!"

Fri Nov 12 22:30:56 CET 2004: logB: Hallo, beste mensen!
Fri Nov 12 22:30:56 CET 2004: logB: Dag beste mensen!
Fri Nov 12 22:30:56 CET 2004: logB: === EOT ===

Fri Nov 12 22:30:56 CET 2004: logC: 123456789
Fri Nov 12 22:30:56 CET 2004: logC: 987654321
Fri Nov 12 22:30:56 CET 2004: logC: === EOT ===