Adaptable Service Oriented Architectures

Krzysztof Zieliński
Department of Computer Science
AGH-UST
Krakow Poland
Agenda

• DCS
• SOA
• WS
• MDA
• OCL
• ODA
• AC
• SOKU
Introduction to DCS
Definition of a Distributed System

Tanenbaum & Van Renesse (1985)

“A distributed operating system is one that looks to its users like an ordinary centralized operation system, but runs on multiple, independent CPUs. The key concept here is transparency, in other words, the use of multiple processors should be invisible (transparent) to the user. Another way of expressing the same idea is to say that the user views the system as a virtual uniprocessor, not as a collection of distinct machines.”
## Transparency in a Distributed System

<table>
<thead>
<tr>
<th>Transparency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Hide differences in data representation and how a resource is accessed</td>
</tr>
<tr>
<td>Location</td>
<td>Hide where a resource is located</td>
</tr>
<tr>
<td>Migration</td>
<td>Hide that a resource may move to another location</td>
</tr>
<tr>
<td>Relocation</td>
<td>Hide that a resource may be moved to another location while in use</td>
</tr>
<tr>
<td>Replication</td>
<td>Hide that a resource may be shared by several competitive users</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Hide that a resource may be shared by several competitive users</td>
</tr>
<tr>
<td>Failure</td>
<td>Hide the failure and recovery of a resource</td>
</tr>
<tr>
<td>Persistence</td>
<td>Hide whether a (software) resource is in memory or on disk</td>
</tr>
</tbody>
</table>

**Different forms of transparency in a distributed system.**
Multicomputer Operating Systems

Machine A

Machine B

Machine C

Distributed applications

Distributed operating system services

Kernel

Kernel

Kernel

Network
Network Operating System

Machine A
- Network OS services
- Kernel

Machine B
- Network OS services
- Kernel

Machine C
- Network OS services
- Kernel

Distributed applications

Network
Network Operating System

- Employs a client-server model
  - Minimal OS kernel
  - Additional functionality as user processes
Middleware-based Systems

- General structure of a distributed system as middleware.
Communication Protocols

- Protocols are agreements/rules on communication
- Protocols could be connection-oriented or connectionless
Communication Issues

• Message-oriented communication
  – Persistence and synchronicity
Persistence

- Persistent communication
  - Messages are stored until (next) receiver is ready
  - Examples: email, pony express

Mail stored and sorted, to be sent out depending on destination and when pony and rider available
Persistence

• Transient communication
  – Message is stored only so long as sending/receiving application are executing
  – Discard message if it can’t be delivered to next server/receiver
  – Example: transport-level communication services offer transient communication
  – Example: Typical network router – discard message if it can’t be delivered next router or destination
Synchronicity

• Asynchronous communication
  – Sender continues immediately after it has submitted the message
  – Need a local buffer at the sending host

• Synchronous communication
  – Sender blocks until message is stored in a local buffer at the receiving host or actually delivered to receiver
  – Variant: block until receiver processes the message

• Six combinations of persistence and synchronicity
Persistence and Synchronicity Combinations

a) Persistent asynchronous communication (e.g., email)
b) Persistent synchronous communication
Persistence and Synchronicity Combinations

c) Transient asynchronous communication (e.g., UDP)
d) Receipt-based transient synchronous communication
Persistence and Synchronicity Combinations

e) Delivery-based transient synchronous communication at message delivery (e.g., asynchronous RCP)
f) Response-based transient synchronous communication (RPC)
Issues in Client-Server Communication

- Addressing
- Blocking versus non-blocking
- Buffered versus unbuffered
- Reliable versus unreliable
- Server architecture: concurrent versus sequential
- Scalability
Addressing Issues

• *Question:* how is the server located?

• Hard-wired address
  – Machine address and process address are known a priori

• Broadcast-based
  – Server chooses address from a sparse address space
  – Client broadcasts request
  – Can cache response for future

• Locate address via name server
Blocking versus Non-blocking

• Blocking communication (synchronous)
  – Send blocks until message is actually sent
  – Receive blocks until message is actually received

• Non-blocking communication (asynchronous)
  – Send returns immediately
  – Return does not block either

• Examples:
Buffering Issues

• Unbuffered communication
  – Server must call receive before client can call send

• Buffered communication
  – Client send to a mailbox
  – Server receives from a mailbox
Reliability

• Unreliable channel
  – Need acknowledgements (ACKs)
  – Applications handle ACKs
  – ACKs for both request and reply

• Reliable channel
  – Reply acts as ACK for request
  – Explicit ACK for response

• Reliable communication on unreliable channels
  – Transport protocol handles lost messages
Remote Procedure Calls

• Goal: Make distributed computing look like centralized computing
• Allow remote services to be called as procedures
  – Transparency with regard to location, implementation, language
• Issues
  – How to pass parameters
  – Bindings
  – Semantics in face of errors
• Two classes: integrated into prog, language and separate
Conventional Procedure Call

a) Parameter passing in a local procedure call: the stack before the call to read

b) The stack while the called procedure is active

```
Stack pointer

<table>
<thead>
<tr>
<th>Main program's local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes</td>
</tr>
<tr>
<td>buf</td>
</tr>
<tr>
<td>fd</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>read's local variables</td>
</tr>
</tbody>
</table>

(a) (b)
```
Parameter Passing

• Local procedure parameter passing
  – Call-by-value
  – Call-by-reference: arrays, complex data structures

• Remote procedure calls simulate this through:
  – Stubs – proxies
  – Flattening – marshalling

• Related issue: global variables are not allowed in RPCs
Client and Server Stubs

- Principle of RPC between a client and server program.

![Diagram showing the process of RPC between a client and server program.](image-url)
Stubs

• Client makes procedure call (just like a local procedure call) to the client stub
• Server is written as a standard procedure
• Stubs take care of packaging arguments and sending messages
• Packaging is called *marshalling*
• Stub compiler generates stub automatically from specs in an Interface Definition Language (IDL)
  – Simplifies programmer task
Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client
Example of an RPC

1. Client call to procedure
2. Stub builds message
3. Message is sent across the network
4. Server OS hands message to server stub
5. Stub unpacks message
6. Stub makes local call to "add"
Marshalling

• Problem: different machines have different data formats
  – Intel: little endian, SPARC: big endian
• Solution: use a standard representation
  – Example: external data representation (XDR)
• Problem: how do we pass pointers?
  – If it points to a well-defined data structure, pass a copy and the server stub passes a pointer to the local copy
• What about data structures containing pointers?
  – Prohibit
  – Chase pointers over network
• Marshalling: transform parameters/results into a byte stream
Binding

• Problem: how does a client locate a server?
  – Use Bindings

• Server
  – Export server interface during initialization
  – Send name, version no, unique identifier, handle (address) to binder

• Client
  – First RPC: send message to binder to import server interface
  – Binder: check to see if server has exported interface
    • Return handle and unique identifier to client
Binding: Comments

- Exporting and importing incurs overheads
- Binder can be a bottleneck
  - Use multiple binders
- Binder can do load balancing
Failure Semantics

• *Client unable to locate server:* return error
• *Lost request messages:* simple timeout mechanisms
• *Lost replies:* timeout mechanisms
  – Make operation idempotent
  – Use sequence numbers, mark retransmissions
• *Server failures:* did failure occur before or after operation?
  – At least once semantics (SUNRPC)
  – At most once
  – No guarantee
  – Exactly once: desirable but difficult to achieve
Failure Semantics

- **Client failure**: what happens to the server computation?
  - Referred to as an *orphan*
  - *Extermination*: log at client stub and explicitly kill orphans
    - Overhead of maintaining disk logs
  - *Reincarnation*: Divide time into epochs between failures and delete computations from old epochs
  - *Gentle reincarnation*: upon a new epoch broadcast, try to locate owner first (delete only if no owner)
  - *Expiration*: give each RPC a fixed quantum $T$; explicitly request extensions
    - Periodic checks with client during long computations
Asynchronous RPC

a) The interconnection between client and server in a traditional RPC
b) The interaction using asynchronous RPC
Deferred Synchronous RPC

- A client and server interacting through two asynchronous RPCs.

```
Client
Call remote procedure
  Wait for acceptance
  Request
  Accept request
  Call local procedure
  Return from call
  Return results

Server
Call client with one-way RPC
  Acknowledge
  Time
```
Remote Method Invocation (RMI)

- RPCs applied to *objects*, i.e., instances of a class
  - *Class*: object-oriented abstraction; module with data and operations
  - Separation between interface and implementation
  - Interface resides on one machine, implementation on another

- RMIs support system-wide object references
  - Parameters can be object references
• When a client binds to a distributed object, load the interface ("proxy") into client address space
  – Proxy analogous to stubs
• Server stub is referred to as a skeleton
Proxies and Skeletons

• Proxy: client stub
  – Maintains server ID, endpoint, object ID
  – Sets up and tears down connection with the server
  – [Java:] does serialization of local object parameters
  – In practice, can be downloaded/constructed on the fly
    (why can’t this be done for RPCs in general?)

• Skeleton: server stub
  – Does deserialization and passes parameters to server
    and sends result to proxy
Binding a Client to an Object

(a) Example with implicit binding using only global references

Distr_object* obj_ref;
obj_ref = ...;
obj_ref-> do_something();

// Declare a systemwide object reference
// Initialize the reference to a distributed object
// Implicitly bind and invoke a method

(b) Example with explicit binding using global and local references

Distr_object obj_ref;
Local_object* obj_ptr;
obj_ref = ...;
obj_ptr = bind(obj_ref);
obj_ptr-> do_something();

// Declare a systemwide object reference
// Declare a pointer to local objects
// Initialize the reference to a distributed object
// Explicitly bind and obtain a pointer to the local proxy
// Invoke a method on the local proxy

a) (a) Example with implicit binding using only global references
b) (b) Example with explicit binding using global and local references
Parameter Passing

- RMI is less restrictive than RPCs.
  - Supports system-wide object references
  - [Java] pass local objects by value, pass remote objects
Code and Process Migration

• Motivation
• How does migration occur?
• Resource migration
• Agent-based system
• Details of process migration
Motivation

• Key reasons: performance and flexibility
• Process migration (aka strong mobility)
  – Improved system-wide performance – better utilization of system-wide resources
  – Examples: Condor, DQS
• Code migration (aka weak mobility)
  – Shipment of server code to client – filling forms (reduce communication, no need to pre-link stubs with client)
  – Ship parts of client application to server instead of data from server to client (e.g., databases)
  – Improve parallelism – agent-based web searches
Motivation

• Flexibility
  – Dynamic configuration of distributed system
  – Clients don’t need preinstalled software – download on demand
Migration models

- Process = code seg + resource seg + execution seg
- Weak versus strong mobility
  - Weak => transferred program starts from initial state
- Sender-initiated versus receiver-initiated
- Sender-initiated (code is with sender)
  - Client sending a query to database server
  - Client should be pre-registered
- Receiver-initiated
  - Java applets
  - Receiver can be anonymous
Who executed migrated entity?

- **Code migration:**
  - Execute in a separate process
  - [Applets] Execute in target process

- **Process migration**
  - Remote cloning
  - Migrate the process
Models for Code Migration

- Alternatives for code migration.
Do Resources Migrate?

• Depends on resource to process binding
  – By identifier: specific web site, ftp server
  – By value: Java libraries
  – By type: printers, local devices

• Depends on type of “attachments”
  – Unattached to any node: data files
  – Fastened resources (can be moved only at high cost)
    • Database, web sites
  – Fixed resources
    • Local devices, communication end points
## Resource Migration Actions

### Resource-to machine binding

<table>
<thead>
<tr>
<th>Process-to-resource binding</th>
<th>Unattached</th>
<th>Fastened</th>
<th>Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>By identifier</td>
<td>MV (or GR)</td>
<td>GR (or MV)</td>
<td>GR</td>
</tr>
<tr>
<td>By value</td>
<td>CP (or MV, GR)</td>
<td>GR (or CP)</td>
<td>GR</td>
</tr>
<tr>
<td>By type</td>
<td>RB (or GR, CP)</td>
<td>RB (or GR, CP)</td>
<td>RB (or GR)</td>
</tr>
</tbody>
</table>

- Actions to be taken with respect to the references to local resources when migrating code to another machine.
- GR: establish global system-wide reference
- MV: move the resources
- CP: copy the resource
- RB: rebind process to locally available resource
Migration in Heterogeneous Systems

• Systems can be heterogeneous (different architecture, OS)
  – Support only weak mobility: recompile code, no run time information
  – Strong mobility: recompile code segment, transfer execution segment [migration stack]
  – Virtual machines - interpret source (scripts) or intermediate code [Java]
END
Lecture 1