Distributed Computing and Object Systems

José M. Vidal

Thu Feb 12 11:47:57 EST 2004

This talk is based, in part, on:

- Jim Farley, Java Distributed Computing[1], Chapter 3.

1 Computing

- **Monolithic computing:** computer is alone and only uses its own resources.
  - Multiple users allowed via **timesharing**.
- **Distributed computing:** multiple computers, each with its own CPU and memory, connected via a network.
  - **Cooperative computing:** a form of distributed computing where a program is split among computers. (SETI, distributed.net).
  - **Grid Computing**[2]: buzzword duJour.

    > "Grid computing can be differentiated from almost all distributed computing paradigms by this defining characteristic: The essence of grid computing lies in the efficient and optimal utilization of a wide range of heterogeneous, loosely coupled resources in an organization tied to sophisticated workload management capabilities or information virtualization."

    also, see the Globus toolkit[3].
- **Parallel computing:** more than one CPU executes the same program.

1.1 Why Distributed Computing

Arguments for:

- Computers are cheap, network is available.
- Architecture mirrors that of organization and resource availability.
- It can, sometimes, scale well by adding more machines.
- If a machine breaks we can replace it with a new one.

Arguments against:

- A distributed system is one in which the failure of a computer you didn't know existed can render your own computer unusable. – Lamport
- It adds a lot of new security headaches.

1.2 Interprocess Communications

- At their root, all IPC systems contain:

1. Send:
2. Receive:
3. Connect: for connection-oriented systems, TCP requires connection. UDP does not.
4. Disconnect:

- These have to be properly interleaved or they could lead to a lot of synchronization problems.
- You use either **synchronous** (blocking) operations or **asynchronous** (nonblocking) operations.

1.3 Synchronization Techniques

- *Synchronous send and receive*: if receive is expecting more data than was sent, it can block, which can then lead to deadlock.
- *Asynchronous send and synchronous receive*: good if sender does not care about reception.
- *Synchronous send and asynchronous receive*: the receive can be implemented in various ways
  1. Receive returns immediately with either data or null.
  2. Receipt triggers a call-back.
- *Asynchronous send and receive*: requires the IPC to implement a queue.
- Even under synchronous we don't wait forever. There is often a **timeout**.

1.4 Distributed Computing Paradigms

- In **message passing** data is sent between two processes via send/receive.
- In the **client-server model** the server provides services and waits for clients to connect to it. Clients issue requests. (Web)
  - **Network services** extend the idea by providing location transparency. (Jini[4]).
- In **peer-to-peer** all play equivalent roles (both client and server). (JXTA[5], Jabber[6]).
- In **message-oriented middleware** an intermediary messaging system handles all messages. It can do so either
  - **point-to-point** using a message repository, or
  - with **publish-subscribe** where receivers subscribe to the type of messages they want.
- The **remote procedure call** (RPC) is a function call that gets executed at some other machine. (DCE RPC[7], SOAP).
- The distributed objects[8] paradigm. (Java RMI, CORBA)
- In **object spaces** the objects are located in a common object space that all can access. (Linda[9], JavaSpaces[10] part of Jini).
  - **Groupware** and **Blackboard** applications extend this idea by providing added shared functionality.

- The **mobile agents** idea is to write code that moves from host to host. (Tracy[11]).

Note:

Message-passing is the simplest of all models as it is essentially just a socket abstraction. Client-server models are by far the most popular application model due in large part to the fact that most applications seem well suited to the restrictions of this model. That is, it is often the case that a service needs to be provided by one machine to other machines because the machine in question either has a lot more CPU power than the rest, or a lot more hard drive space, or access to some special data, etc. Client-server is also easier to implement since all we need to do is make the server do a synchronous receive. The idea of making these services location-independent has been floating around for decades and implemented in various systems, most notably Jini. It is not easy to achieve location independence, especially in the presence of faults (lost messages, broken servers, etc.), so every implementation needs to make certain concessions on how often the service search will work versus how long and how many resources it will take.

Peer-to-peer systems hold the most promise for the development of robust and agile distributed systems. However, we do not yet have algorithms that will support their growth. The problem lies in open p2p systems where all the peers are selfish agents. These systems require mechanisms that can align the individual selfishness of the peers with the global needs of the system. These algorithms are being developed by researchers in multiagent systems.

Message-oriented middleware adds some often-needed functionality to the basic message-passing paradigm. These systems are usually small utility libraries. RPC was the first departure from message passing. It was the first step in integrating distributed computation into the programming language. Unfortunately, the libraries that exist mostly don't work well with each other so you can only RPC to a similar host.

The idea of shared tuple spaces was popularized by the Linda language. Implementing these spaces presents several problems. The tuple-space needs to be physically distributed and yet available in its entirety to all the agents, even in the presence of failures. The searches for a particular tuple need to be performed by some agent, but we also do not want to overload any one agent. Finally, read-write permissions might need to be implemented in these objects. The Linda papers explain how these were implemented in their systems, but there exist many algorithms for achieving these goals, all of them, of course, have different strengths and weaknesses.

Mobile agents have been characterized as a solution looking for a problem. They do, however, have a place in systems were communication costs are very high and a lot of computation is required using the local data. Unfortunately, these type of constraints do not seem to appear often (or, at all?) in the real world, so mobile agent systems are still only studied by academic researchers.

2 Distributed Objects

- The idea: Let the object reside anywhere, transparently.
- To distribute applications easily. No need to develop a new protocol.
  - To distribute resources: data, CPU power, physical constraints, etc.
  - Redundancy.
  - To be near user for faster access.
- No need to add new messages or design a new language.
- No need to deal with data conversion.
- Make re-distribution easier.
- Make it easier to change where the split (between what's local and what's remote) happens at a later time.
- Its easier to implement fine-level security and access restrictions.

2.1 Isn't this easy to do?

Our goal is something like:

```
//w resides on some other machine
Widget w = new Widget();

//this function is executed on that other machine.
w.calculate();
```

- In order to do achieve this small miracle we need many things.
- We need a protocol for creating new objects remotely, invoking methods on them, getting the results back, and deleting unused objects.
- In order to do this we must send
  - Class references
  - Object references.
  - Method references.
  - Method arguments and return values.

2.2 Using the ClassLoader

- We can use the `java.lang.ClassLoader` to start building a DOS.
- The `ClassLoader` is an abstract interface to loading classes. To implement it we must implement:
  - `loadClass(String classPath, boolean resolve)`
  - `readClass(String classPath)`
- We could define a new `ClassLoader` which can load classes over the network (Applets already do this).
- We can decide that class references will be of the form `machine-name:classname`.

2.3 Object References

- First, remember the difference between objects and classes! An object does not contain the methods.
- We can send an object by using the `Serializable` interface and writing it on a socket. Easy!
- But, an **object reference** somehow needs to point back to the machine:object so that a method call on that object reference will call the appropriate machine.
- We can achieve this by building an object lookup table on the client.

Client				m.sc.edu (Server)	
Object	Machine	Obj ID		Obj ID	Object
o	m.sc.edu	101	–new o()→ ←m.sc.edu:01– –101:calc()→	101	o

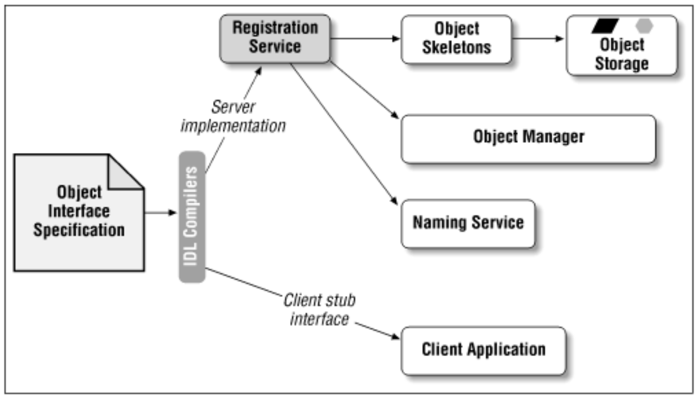
2.4 Methods and Arguments

- We can handle **method references** by simply sending the method number, rather than the name as in the previous slide.
- **Method arguments** and return values can be handled by serializing them and sending them along with the method call.

Client				m.sc.edu (Server)	
Object	Machine	Obj ID		Obj ID	Object
o	m.sc.edu	101	– 101:calc(F f)→	101	o

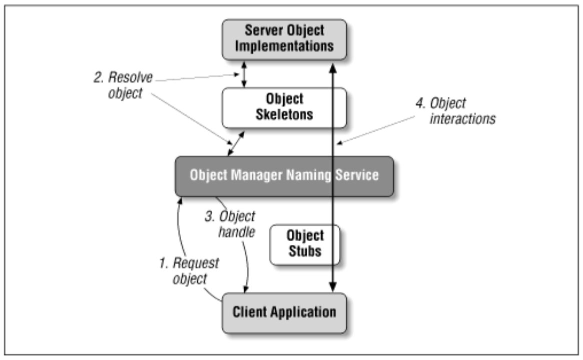
- Upon receipt of this method call, the Server must determine if `f` is a local or remote object. If its remote, it must go back to the client (which is now a `Foo` server) and to access `f`'s data and method members.
- This is getting very hard!
  - Both end up being client and server.
  - How does one program find the machines that are server for a particular object? We should have some kind of naming service.
  - In our scheme, a return-by-value forces the client to create a new object, which the server also did (double work).
- Luckily, Java RMI, CORBA, and DCOM, are all implemented solutions to this problem. So, use them!

2.5 General Features of DOS



- All **Distributed Object Systems** (DOS) include RMI, CORBA, DCOM, but not SOAP. They all share some features.
- Object specification is used to generate
  1. **skeleton**: an interface between object implementation and the communications code provided by the DOS.
  2. **stub**: an interface between the client and the communications code. The client thinks the stub is the object.
- The main idea is that we want only one class interface definition.

2.6 Runtime Transactions



2.7 Example

- We start with one definition of our widget interface.

```
//This is a pretend example
interface Widgets {
  double getStrenth();
  void tickle(int times);
  String getWisdom();
}
```

- From this code, we run a program that generates both a stub and a skeleton.
- The skeleton is placed in the object server, which stands ready to serve and service copies.
- The stub is placed in the name service and, somehow, finds its way to the client.
- In CORBA the programs can be written in different languages and platforms.

2.8 Object Manager

- The **object manager** is the heart of the server since it manages object skeletons and references (ORB, registry service).
  - It handles creation of new object using skeleton, store it, send back referral.
  - Handles method calls routing to object.
  - Handles deletion.
- It might handle dynamic object (de)activation and persistent objects.
- It might reside on the host serving the objects, partitioned between clients and server, or on a third host.

2.9 Naming Services

- The **naming service** maintains the mapping between interfaces and servers.
- It uses this information to answer questions from clients.
- At its simplest, it does name matching. New, more semantically complex, description languages are being developed which will provide near-matches.

- At its simplest, it returns the name of a host. New technologies hope to provide redundancy and choice.
- The **object communication protocol** handles the object and method references, as well as the marshaling of basic data types.

2.10 Security

- The client must be authenticated (use passwords and PKE).
- Is the client allowed to get a new reference to this object?
- Is the client allowed to access this method?
- If the client passes the object's reference to another client, does the new client inherit the old one's privileges?
- Can a stub somehow compromise the client's security by executing illegal code locally? The stubs need to be verified.
- Are man in the middle attacks possible? Can someone pretend to be someone else and make client calls on a server? What about someone pretending to be the server?
- Each of the architectures addresses, or fails to address, these issues in a different way.

Notes

[1]http://www.oreilly.com/catalog/javadc/chapter/ch03.html
[2]http://www-106.ibm.com/developerworks/grid/library/gr-heritage/?ca=dgr-lnxw01GridNextGen
[3]http://www.globus.org/
[4]http://wwws.sun.com/software/jini/
[5]http://www.jxta.org
[6]http://www.jabber.org
[7]http://www.opengroup.org/publications/catalog/c706.htm
[8]http://jmvidal.cse.sc.edu/talks/distobjects/dos.xml
[9]http://www.cs.yale.edu/Linda/linda.html
[10]http://java.sun.com/developer/Books/JavaSpaces/introduction.html
[11]http://mobile-agents.org/

This talk is available at `http://jmvidal.cse.sc.edu/talks/distobjects`