Center for Pervasive Computing Report Series

Publication CfPC-2002-PB-?

Issues in Implementing Distributed Object Systems

Author(s): Henrik Bærbak Christensen

Version: 0.2

Status: Open

Abstract

This paper presents a checklist of issues that must be considered when one attempts at implementing a distributed object system based upon remote method invocation. The check-list is in its early stage and does not attempt at being complete. Also the check-list only summerises issues but does not provide any solutions.

Contents

1 Introduction

The object oriented programming paradigm seems appealing for implementing distributed systems. Objects can be viewed as small computational devices encapsulating state and behaviour and interacting through message passing. The step from having objects interacting within the same address space to having objects interacting in *different* address spaces seems fairly straight forward and seems like a technological rather than a conceptual challenge. And indeed there is an abundance of technologies that take this approach: OMG's CORBA, Microsoft's DCOM, Sun's Java RMI, etc.

The basic premise of these technologies is to provide *location transparency* through *remote method invocation*, i.e. objects can invoke methods on other objects regardless of the location of these objects—local or present on another computer in the network. Of course, there is a fundamental flaw in this approach as a remote invocation is fundamentally subject to other concerns than a local invocation: networks are much slower, objects may suddenly disappear if their host environment is shut down, communication lines are faulty, etc. Some authors have pointed out that these problems show that the basic premise is fundamentally wrong [2] and argue in favour of radically different solutions. However, as the mainstream technologies have achieved some degree of success and acceptance the checklist provided assumes the use of remote method invocation.

2 Checklist

2.1 Core technology: Broker

At the center of any remote invocation scheme is the abstraction that handle method invocation (more or less) transparently over a wire. This abstraction is well-studied and present knowledge has been documented in the form of a design pattern: the *Broker* pattern [1, p. 99]. *This pattern should be carefully studied.* Besides a description of the participating objects' roles and responsibilities, Buschmann also provides a lengthy list of implementation issues that have to be considered. Also of interest is the *Forwarder-Receiver*- and *Client-Dispatcher-Server*-patterns; these patterns describe more lightweight approaches to remote data exchange.

Buschmann's implementation check-list is given in headlines below, for a discussion of each issue, consult the book.

1. *Define an object model, or use an existing.*
2. *Decide which kind of component-interoperability the system should offer.*
3. *Specify the APIs the broker component provides for collaborating with clients and servers.*
4. *Use proxy objects to hide implementation details from clients and servers.*
5. *Design the broker component.*

   This includes considering:

   (a) *Specify on-the-wire protocol.*
   (b) *Is there need for bridges to bridge different networks?*
   (c) *Direct or indirect communication variant?*
   (d) *Responsibility for marschalling: proxies or broker?*
   (e) *Support asynchronous communication?*
   (f) *Provide directory service.* For mapping local object identities with the physical location of server objects on the server node.
   (g) *Naming service required?*
   (h) *Support dynamic invocation?*
   (i) *Exception handling.*
6. *IDL compiler / proxy generation.*

2.2 Services

In addition to the core technology there are some issues that must be considered:

1. *Fault tolerance.* What is the semantics of the broker in case some metod cannot be invoked (the server object is not present, the server node is down, the network is down, etc.)? Is it flagged as an error, an exception thrown, or is the request buffered (relevant for asynchronous requests)? Should the broker start up server objects in case they are not present?
2. *Quality of Service (QoS).* Should server objects (services) be replicated and the broker make sure requests are forwarded to available server objects?

3. *Realtime requirements.* Are there any realtime requirements for the system in terms of timeliness and predictability?
4. *Security.* Is it possible that hostile objects can compromise the operation of the system?
5. *Transactions.* Is it necessary to group requests into transactions whose effect can be undone (roll-back) in case one request in the group fails? Is it necessary for transactions to be nested?
6. *Garbage Collection.* Is it the responsiblity of the programmers or must some distributed GC algoritm be implemented?
7. *Persistence.* Must objects survive between sessions?
8. *Interoperability with legacy systems?* Are there legacy systems based upon technologies like CORBA, DCOM, or propriatery that the system need to interface?

2.3 Advanced functionality

In addition you have to consider whether your solution may be required to handle advanced features such as:

1. *Mobile objects/migration.* Is it required that objects can migrate from one address space to another? For instance that some vital objects/services on a given node will automatically move to another node before the former node is shut down.
2. *Code mobility.* Is it required that a given node can accept code dynamically (as opposed to the traditional scenario that require a manual software installation step.)
3. *Single point of failure.* Is it allowable that the system has central services that can be assumed to be operational at all times? For instance the ORB is a single point of failure as is a centralised lookup/binding service.
4. *Broadcast / multicast.* The design patterns above have an underlying assumption of peer-to-peer communication but in many embedded settings a broad- or multicast paradigm (publish/subscribe) is more appropriate. Broadcast means that a publisher sends data with no dedicated receiver, all nodes on the bus receives the message. Multicast means that a set of subscribers receive a given message. (Note that multicast can be simulated on a broadcast bus like CAN and the reverse is also true). Is a publish/subscribe paradigm important or necessary for the given domain?
5. *Dynamic code loading.* Is it important that software can be uploaded and made to run in the software environment at runtime? (In constrast to for instance change a ROM on a board.)
6. *Predictablility and realtime analysis.* In case of dynamic upload of code, which models can be used to analyse and predict if given real-time constraints such as quality of service, end-to-end timing, etc. are satisfied?
7. *Asynchroneous messages ('oneway').* Is it required or desirable that the system supports asynchroneous messages, where methods are called but the caller proceeds its computation instead of blocking until a reply is returned? Or maybe no reply is needed?

8. *Bindingtypes.* What kind of bindings between objects are the most appropriate? There are *name-binding* (CORBA/DCOM/RMI-way) where some unique name is bound to a remove object reference and *inferface-binding* (Jini) where a request for a service that satifies a certain interface returns a proxy that then acts as mediator of communication.

9. *Connection- or connection-less communication:* Traditional TCP/IP communication is connection-oriented in that you open a peer-to-peer connection using Sockets. I.e. there is quite some overhead in establishing the link between the participants of the communication. In connectionless communication on the other hand you more or less is able to post anything to anyone at anytime without further effort.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture*. John Wiley and Sons, 1996.

[2] R. Guerraoui and M. E. Fayad. OO Distributed Programming is not OO Programming. *Communications of the ACM*, 42(4), 1999.