Center for Pervasive Computing Report Series

Publication CfPC-2002-PB-?

Distributed Object Systems A Terminology Document

Author(s): Henrik Bærbak Christensen

Version: 1.0

Status: Closed

Abstract

Middleware technologies for supporting distributed object oriented programming are widespread. Many of these employ variations over the same core technology but often use different terminology. The purpose of this report is to provide a common terminology that describes key abstractions. Our focus is embedded systems, and the presentation owes much to the fact that our underlying bus system is the CAN-bus.

Contents

1 Scope

Our viewpoint is that of *object-oriented* distributed systems though many of the abstractions are the same in for instance systems that allow remote procedure calls (RPC).

2 Purpose

To provide a common terminology for key abstractions in the design of distributed object-oriented systems.

3 Basic abstractions

3.1 Computational unit

We use the term **computational unit** broadly to cover any unit in a software system that is capable of performing some software controlled functionality. Typically it is an object in an object oriented system, and we will use the word object and computational unit interchangeably.

3.2 Node

A computational unit or object is an entity in the *logical view* of the software design [1, 3]. At the *deployment level* we have **nodes** (in an embedded system typically microcontroller cards with downloaded/ROM software) that may contain one or several computational units.

3.3 Abstraction levels

In our discussion we operate at at least two levels of abstraction with respect to the data that is transferred between computational units.

> At the **datagram level** computational units exchange data over some transport medium in the form of simple finite bit-sequences. Example is the CAN-datagram that has a 11/29 bit message header and 8 bytes of payload data.
>
> At the **method invocation level** computational units exchange data over some transport medium in the form of object-oriented messages. Example is RMI where as remote message invocation has an identity of the receiver object, a identity/name of the wanted method, and a list of parameter values.

These levels can be considered the bottom- and top levels from our perspective and our mission is to connect these two levels in an operational software solution.

4 Communication

4.1 Data dispatch

When a computational unit decides to send data to another unit we speak of data dispatch:

> **Data dispatch** is the process of sending data from one computational unit to another.

At the datagram level the data dispatch may be as simple as putting datagrams onto the transport medium. The method invocation level can be viewed in the same light if we relax the ordinary semantics of method invocation. If we broaden the scope of method invocation to include asynchronous invocation with no blocking of the caller and no return value, then method invocation is identical to data dispatch except that the data is organised according to some well defined semantics (to unambiguously identify parameters, types, etc.)

4.2 Roles during data dispatch

Data dispatch is inherently asymmetric as one object is the sender/activator and the other object is receiver/activated.

We will use the terms sender and receiver to denote the object that activates and is activated respectively:

> An sending/invoking/activating computational unit (object) is denoted a **sender**.

> An receiving/invoked/activated computational unit (object) is denoted a **receiver**.

Note that both receiver and sender are *roles* that a given object takes on *during the time of a data dispatch*. Thus a given object may act as a receiver at time t_1 and act as sender at time t_2. This is different from the traditional client-server paradigm for macro-structuring for instance world wide web where the server and client roles are fixed for given applications like web-server and web-browser.

4.3 Dispatch patterns

Senders dispatch data to receivers. The multiplicity in the dispatch-relation is interesting as it supports different communication styles and each of these styles have their benefits and weaknesses.

Here we will classify dispatching based upon whether there are exactly one, several, or all receivers in the system involved.

	One receiver	*N receivers*	*All receivers*
One sender	one-to-one	one-to-many	one-to-all

The matrix of dispatch patterns.

The *one-to-one* or peer-to-peer dispatch is the underlying pattern of normal method invocation where one object invokes a method in exactly one other object (decided by the polymorphic nature of the receiver's actual type).

The *one-to-all* or broadcast pattern is the reality of the CAN-bus where a datagram put onto the CAN-bus is received and interpreted by every node in the system (however, there may be several objects running on each node so from the application's perspective it may not be a one-to-all pattern).

The *one-to-many* or multi-cast pattern is often very interesting in embedded systems where many controllers may 'listen' and react upon a single source of data. This dispatch pattern is underlying the group communication (or publisher/subscriber or producer/consumer) paradigm. Group communication is normally approximated in standard OO languages by using the observer design pattern.

Note that one may be tempted to fill out the matrix with *many-to-one*, *many-to-many*, etc. patterns but recall that the viewpoint is that of a single dispatch operation. Thus for instance the many-to-one pattern means that many senders at exactly the same instance in time dispatch data to a single receiver—which is clearly not feasible.

4.4 Dispatch synchronisation

Basically a data dispatch operation can be either **asynchronous** or **synchronous**.

In the synchronous case a sender dispatch data to receiver(s) and blocks until the receiver has acknowledge the reception of the data. We may also term this an acknowledge data dispatch.

In the asynchronous case a sender dispatch data to receiver(s) and simply proceeds its processing. It does not block nor expect any acknowledgement of data reception from the receiver.

4.5 Rendezvous synchronisation

There is also the **rendezvous** or request/reply synchronisation. Rendezvous is normally used in the one-to-one dispatch pattern. It is not an atomic synchronisation style but relies on a handshake protocol that includes at least two data dispatches. In the first data dispatch the sender dispatch data to a receiver (request phase). Next data is dispatched in the opposite direction (reply phase).

A rendezvous can be either blocking and non-blocking for the sender. In a blocking rendezvous it is the well-known remote message invocation scheme: The sender invokes a method on a receiver and blocks until the method return value is supplied. In a non-blocking rendezvous the sender proceeds as the request phase typically returns a *future value* that is later changed with the real returned value upon reception of the value of the reply phase.

5 Terminology for middleware abstractions

From the discussions it should be clear that any middleware must decide upon and support at least the following concepts:

- *Identity*: This includes identity of single objects and/or groups of objects. This means both assigning identity as well as providing identity of receivers to senders upon request.
- *Marshalling*: This includes techniques for coding datatypes and identities in a form where they can be transported over a wire.
- *Synchronisation*: How are data dispatch synchronised? Is the sender blocked until receiver acknowledges the dispatched data (synchronous) or is it unblocked (asynchronous), or does it support both?
- *Rendezvous protocol*: If rendezvous is needed (for instance to support synchronous remote method invocation with blocking of the caller) a suitable request/reply handshake protocol must be defined.
- *Data dispatch*: This includes deciding dispatch patterns and synchronisations to support and how to support them. In the likely case that rendezvous is supported a proper handshake protocol must be designed.
- *Error handling*: Distributed systems are vulnerable to network errors and must detect and signal errors to the application layer.
- *Real-time*: If real-time constraints are essential one must ensure that suitable abstractions exist to allow specifying deadlines or priorities and suitable methods exist for analysing system performance.

5.1 Directory service

To dispatch data to some receiver object the sender needs to know the identity of the receiver at run-time. Monolithic applications traditionally use memory references for object identity but this is clearly not feasible in the distributed case.

Usually this is solved by designing and programming using *abstract identities* at the design level, and then mapping these to *physical identities* at run-time.

> A **directory service** is a run-time component that maintains a directory over (abstract identity, physical identity) mappings in a distributed system. A new entry is added to/changed in the directory service in a process denoted *receiver registration*, and the directory is read in a process denoted *directory lookup*.

5.1.1 Abstract identity

Please note that the abstract identity can be just about anything. In CORBA the abstract identity is a hierarchical name like e.g. `/au/daimi/oo-dist/anton`, in Jini it is a Java interface.

5.1.2 Physical identity

The physical identity part of the mapping must be interpreted in a broad sense. It does not necessarily map to a specific connection to the receiver object. For instance in most Broker pattern based systems the returned physical identity is a reference to a proxy object that takes care of data dispatch. In Jini the physical identity is a Java class file that implements the requested interface and that takes care of data dispatch.

The directory service is not restricted to one-to-one dispatch patterns. The physical identity may actually be a list of receivers and the the abstract identity must then be interpreted as the identify a group of receivers.

5.1.3 Receiver registration

To create the proper (abstract identity, physical identity) entry in the directory, the only object that really knows the relation must tell it to the directory service—and this is the receiver object itself.

So the receiver object registers at the directory its abstract identity and the physical identity that it has been assigned at runtime.

5.1.4 Directory lookup

When a sender wants to dispatch data to some receiver for the first time it needs the physical identity of the receiver. It can then query the directory service for the physical identity associated with the abstract identity of the receiver it needs to communicate with.

5.1.5 Discussion

In simple systems the physical identities can be decided at design time and there is no need for abstract identities nor a directory service.

Note that the directory service can become a single-point-of-failure in a system.

The directory service corresponds closely to the 'Dispatcher' component of the 'Client-Dispatcher-Server' pattern [2, p. 323].

In the discussion above the directory service is only involved in establishing the value of the physical identity. In a system that allows run-time reconfiguration, the physical identity of a server may change over the course of the system's lifetime. If this is the case, data cannot be dispatched directly to the server but has to go through some component that can dynamically reroute data (we will term this component the "dispatcher" in the next section).

5.2 Data dispatch

Once a sender has established the physical/run-time identity of a (set of) receiver(s) it needs to dispatch data. Data dispatching can be abstracted in a number of ways depending on what qualities (such as location transparency, distribution transparency, protocol encapsulation, etc.) one needs to achieve in the design.

Below we list some known techniques, setting the terminology, and outline the qualities that they have. We do not claim the list to be exhaustive.

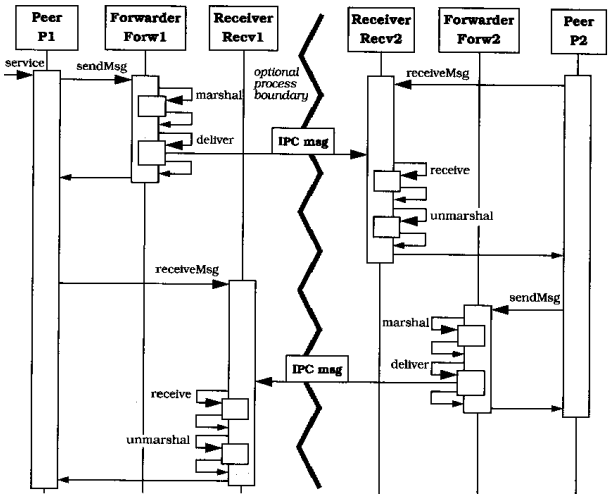

Figure 1: Dynamics of the forwarder/receiver pattern.

5.2.1 Direct data dispatch

This is the raw datagram level where the call site itself deals with the underlying IPC (inter process communication) mechanism, marshaling data, synchronisation, and identify receivers by their physical identity.

The primary quality of this approach is performance and very detailed control. The approach generally lacks portability, modifiability, and as it is prone to errors (it require very detailed understanding of details across the full development team) it also lacks maintainability.

5.2.2 Forwarder/Receiver

Buschmann lists the forwarder/receiver design pattern [2, p. 307] as a way to *decouple communication from the underlying interprocess communication protocol*: Object that dispatch data using forwarders and receiver should be portable to a new underlying network structure as only the forwarders and receivers needs to be rewritten.

The basic dynamic behaviour (Fig. 1) is that object A sends a message to object B by calling `sendMessage` on its forwarder denoting B's abstract identity as receiver. The forwarder does the marshaling and maps B's abstract identity to a receiver representing B on the receiver node. B's receiver accepts the datagram and holds it until B can accept it calling a *receiveMessage* on the receiver. The pattern is symmetric in the sense that both A and B have forwarders and receivers.

Note that forwarder and receiver engage in peer-to-peer communication without any intervening components.

The responsibility of the forwarder is to

1. Provide a `sendMessage` method
2. Marshal and dispatch data to receiver
3. Map abstract identities to physical identities

The responsibility of the receiver is to

1. Provide a `receiveMessage` method

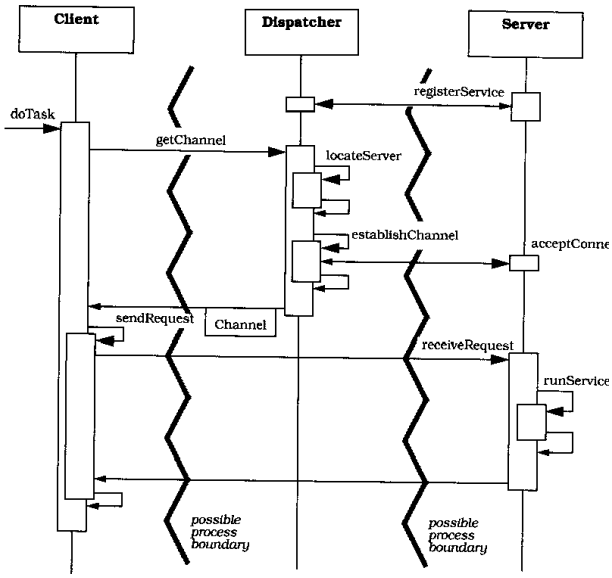

Figure 2: Dynamics of the client/dispatcher/server pattern.

2. Receive and unmarshall data from forwarders

This pattern is basically an asynchronous dispatch pattern where the forwarder just dispatches data without blocking for acknowledgement of the reception of it by the receiver. Receivers can choose to block or not on the `receiveMessage` call. As outlined in Buschmann you can of course build a rendezvous protocol on top of the pattern.

The pattern does not dictate peer-to-peer communication. The abstract identity of the receiver may be a group identity instead. The forwarder will then dispatch data to all receivers in the group.

This pattern is vulnerable to reconfigurations in communication paths at run-time as the (abstract identity to physical identity) mapping is stored distributed over all forwarders in the system.

The basic quality that this pattern achieves is portability and modifiability by decoupling objects from the underlying IPC mechanism.

5.2.3 Client/Dispatcher/Server

The main purpose of this pattern is to provide location transparency by using a directory service (part of the dispatcher component).

In the pattern the client wants to use functionality in a server. It does so by asking the dispatcher to establish a connection to the server, identified by the server's abstract identity. Beforehand, the server has registered its physical identity to the dispatcher. Thus the dispatcher can establish a connection to the given server and return this to the client. From then on the communication between client and server is one-to-one. Note that this pattern assumes a connection-oriented communication like sockets.

The dynamics is sketched in Fig. 2.

Thus the purpose of the dispatcher is both to provide a directory service in our terminology as well as to establish connections.

This pattern does not shield clients nor servers from the underlying IPC mechanics. Thus it is often combined with the forwarder/receiver pattern to handle that aspect.

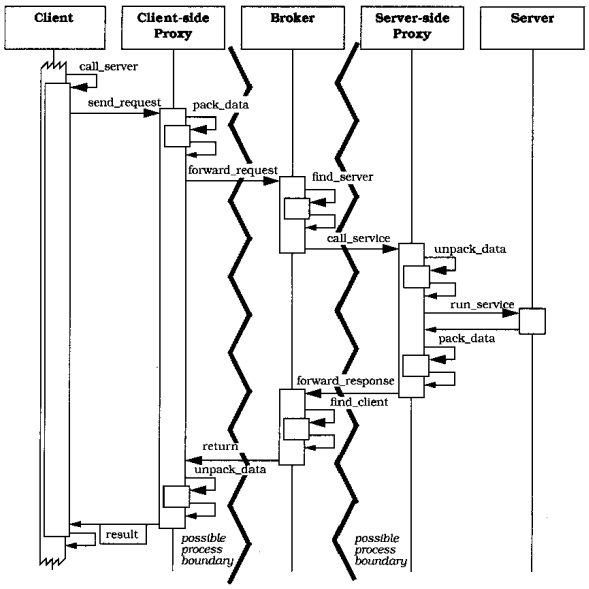

Figure 3: Dynamics of the broker pattern.

It does also not comment on marshaling.

The basic quality that this pattern achieves is modifiability and ease of programming by providing location transparency.

5.2.4 Broker

Basically this pattern combines the low-level forwarder/receiver and client/dispatcher/server pattern and adds rules for marshaling, rendezvous synchronisation, error handling, automatic code generation, and naming to form an architectural pattern for remote method invocation in a distributed object system. The broker pattern is the underlying architectural pattern in well-known middleware systems like Java RMI, OMG CORBA, and Microsoft DCOM.

The broker pattern describes five components. A Client and a Server are objects on two distinct nodes. A Client-side proxy is a proxy that to the client acts exactly as the server object but resides in the address space of the client. The client-side proxy forwards method calls to- and receives method results from the actual server object via the broker. It is responsible for the marshaling of method calls. A Server-side proxy is a proxy for the server object. It receives method calls from the broker, call the proper server method, and dispatches return values back to the client-side proxy. Before it can be invoked remotely it must register itself with the local broker. The Broker is conceptually a single component that permeates the distributed system but at the deployment level there is a broker running in each application. It is the messenger that is responsible for the data dispatching between clients and servers through the proxies.

The dynamics is sketched in Fig. 3.

5.3 Fundamental abstractions

To support *decoupling from IPC* we will use the term **forwarder/receivers**. A forwarder is responsible for marshaling data and forwarding (data dis-

> patching) to a server's receiver. A receiver is responsible for receiving data, unmarshaling it, and invoking the server object.

In the forwarder/receiver pattern these just provides 'sendMessage' and 'receiveMessage' methods to be used by their clients. In the broker pattern the forwarder is a proxy that implements the server objects interface but still its purpose is the same: to forward/dispatch data to the server. In the forwarder/receiver pattern they communicate directly. In the broker pattern they communicate via the broker. In both patterns the forwarder/receiver is responsible for the marshaling/unmarshaling. Note that the proxies of the broker pattern contains both the forwarder and receiver as e.g. a client-side proxy is also responsible for receiving the invocation return value.

> To support *location transparency* we will use the term **dispatcher**. The dispatcher is responsible for dispatching data from a forwarder to the proper receiver (or set of receivers) either as a local data dispatch or as a remote data dispatch. To the forwarder the receiver is identified using some abstract identity. A receiver must register itself with a dispatcher. The dispatcher must define a strategy for error handling (like raising exceptions in case of network failure). The dispatcher must guaranty timely deliveries in case real-time deadlines are important.

In the forwarder/receiver pattern the dispatcher is part of the forwarder component. In the broker pattern it is identical to the broker component. It is somewhat similar to the dispatcher in the client/dispatcher/server pattern (in that it establishes channels of communication) but note that we expect there to be one dispatcher per application (and a central directory service) whereas Buschmann's dispatcher is a centralised component that is more similar to our directory service.

If we want group communication it is also the dispatcher that is responsible for the one-to-many dispatch pattern.

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1997.

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

[3] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley Object Technology Series, 2000.