A Framework for Distributed Interaction

Stephen Crane

October 11, 1996

Abstract

In this position paper we argue that the implementation of abstractions which support interaction between entities in a distributed system is usually *ad hoc* and that there is a need for a rigorous framework which can accomodate different types of interaction subject to an agreed set of requirements. We present these requirements and a framework which meets them and argue that the quality of service supported by the underlying transport system should have no bearing on the semantics of the interaction.

1 Introduction

Distributed programming environments have commonly restricted programmers to a single way of expressing interactions between their programs' components. Remote Procedure Call [1] is popular because it generalises the centralised, single-threaded procedure call into one between address spaces.

However, distributed programs are quite unlike centralised programs. Without describing all of the ways in which distributed semantics differ, we will concentrate on one: concurrency. A centralised program, being constrained to a single address space, almost invariably possesses a single thread of control. When it *is* multi-threaded, communication between threads is via shared data, supported by monitor conditions or semaphores to allow synchronisation and mutual exclusion.

A distributed program on the other hand *always* has multiple threads of control. Whether more than one is simultaneously active depends on the reasons for distributing the components of the program in the first instance:

Access to Remote Data In this case, the 'distributed program' is a collection of loosely coupled components which are potentially located in different enterprises. In an open distributed system, long-running servers act only in response to requests from relatively short-lived clients. In this model, a client performs some computation, requests information from a server and blocks until it receives a reply. For such needs, RPC is perfectly suited.

Exploitation of Parallelism In this case, the motivation for distribution is that the problem at hand is susceptible to sub-division and the sub-problems are capable of being solved simultaneously. Where communication between the components is required (and for maximum speed-up, it should of course be minimal) it takes the form of asynchronous notification of information. Rarely, if ever, is RPC's request-reply paradigm the best solution.

Of course, in practice few distributed programs fall neatly into either category. If more than one interaction style is not supported, programmers are forced to model the required ones in terms of the dominant paradigm. While such modelling is perfectly possible [7] it does not follow that it is a pleasant task and, programmers

being what they are, the choice of winning paradigm has been the cause of many holy wars.

If more than one interaction style is supported, there are many arguments in favour of treating them all equally, apart from an end to the RPC versus message-passing debate. We outline some of these arguments in the next section. The third section describes the components of the framework while the fourth demonstrates its flexibility by example. We conclude with a complete list of currently implemented interactions.

2 Requirements on the Framework

The requirements which motivated development of our interaction framework stem from many years of development of distributed programming environments for both instructional and industrial applications [4, 9, 10]. We distinguish three requirements as crucial: support for multiple binding idioms; separation of transport and interaction semantics; and selection over a set of guarded endpoints.

2.1 Multiple Binding Idioms

Our model of distributed computation imposes a strict separation between program structure and its algorithmic aspects [8]. This has been referred to as 'programming in the large' or viewing components as 'software integrated circuits'. In traditional models, binding is established by the first party to the transaction — the client. Our experience has led us to the belief that, while essential, client-originated binding is inappropriate for parallel programming and can obscure application structure, and that clarity is more easily obtained by locating the responsibility for establishing initial application structure elsewhere.

2.1.1 First-party Binding

The essence of first-party binding is that the client uses a reference to initialise one of its interfaces. How the client obtained the reference is immaterial, it could have got it from a nameserver, as a result of a previous transaction, or even from its execution environment.

First-party binding is extremely common in modern distributed systems. In Orbix for example, [6] all binding is performed by the first-party and the result of a bind action is a proxy. This is normally implicit: endpoint and reference are indistinguishable to the programmer with automatically-generated marshalling code performing the necessary transformations.

2.1.2 Third-party Binding

In the same manner, we define third-party binding as a binding between client and server which is established by 'something else' which is neither client nor server.

Third-party binding is relatively rare in modern distributed systems. Systems which transform a configuration description into an initial application structure commonly use third-party binding to do so [3]. Management utilities which perform online application reconfiguration are also in the role of the third party when they connect a client to a server [2, 5].

2.2 Varying QoS

Quality of Service is a term which describes the demands of a distributed application on the transport system interconnecting its components [13]. It introduces a need

to replace traditionally common transport semantics such as 'best effort' or 'reliable delivery' by a spectrum of communication protocols one of which is best suited to the requirements of a particular application.

Interaction and QoS are orthogonal concepts. While programmers should be free to specify the quality of service which they require from the transport supporting a binding, the style of the interaction should have no bearing on the quality of service provided by it. Programmers should *not* be misled into considering RPC inherently reliable, but message-passing best effort.

2.3 Selection

The Unix device-independent I/O model is a powerful abstraction which, among other things, allows servers to wait for (`select`) an I/O event from one of a number of heterogeneous devices. In a similar fashion, we wish to allow programmers of server components to allow services to be implemented by any combination of interaction styles. This leads to a requirement that all interactions share a common ancestor class which allows servers to wait for an event on one of several services.

Events of interest are not confined to incoming requests for service although this is the common case. An application may also wish to know when the QoS required on a binding drops below some threshold or when a connection to the remote service breaks altogether.

3 The Framework

The previous section motivated the need for a framework by describing some of the properties which we wish all interactions to possess. In the light of these requirements, we outline the participating elements of the framework.

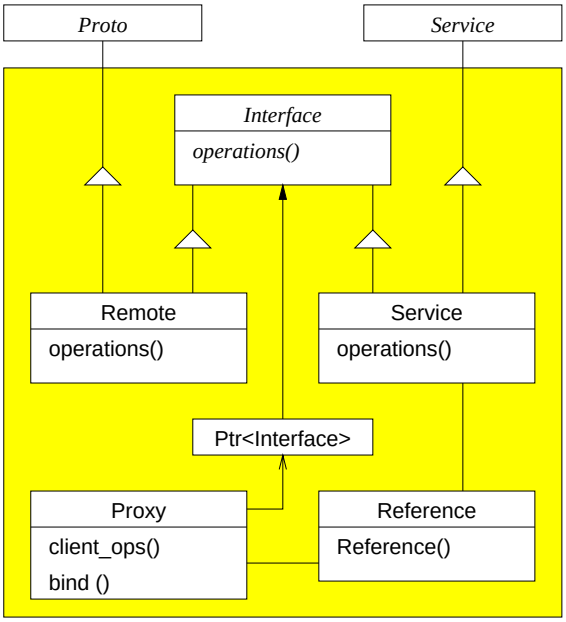

Figure 1: Class relationships between elements of an interaction.

3.1 Participants

The shaded area of figure 1 shows the elements of an interaction required by the framework. Classes outside of this area, `Proto` and `Service`, are in supporting roles and are not directly relevant to the discussion.

Service This class defines the service which an interaction provides. Instances of this class are destinations of messages sent to servers, e.g., ports, remotely-invokable objects, etc.

Proxy is a placeholder which is bound to the interaction endpoint. The semantics of attempted use before binding depends on the binder associated with it. A proxy which is published in a configuration description is required to block a task accessing it until a third-party has bound it. On the other hand, use of a private unbound proxy should cause an exception.

Reference is the intermediary between Service and Proxy. It is initialised from a service and a QoS description and passed as a parameter to a compatible Proxy's `bind` operation where it causes creation of the desired communications subsystem.

Interface defines the abstract signature of the target class which supports the interaction. This facilitates provision of location transparency.

Remote defines the interface to the communications subsystem described by the above QoS description. Location transparency is achieved by making its interface indistinguishable from that of the Service itself.

The latter two elements are more implementation-oriented and as such less mandatory. However they are a convenient way of fulfilling the requirement of location transparency. Figure 2 shows how instances of these classes interact to establish a binding between a co-located proxy and service, while figure 3 shows the establishment of a binding to a remote service.

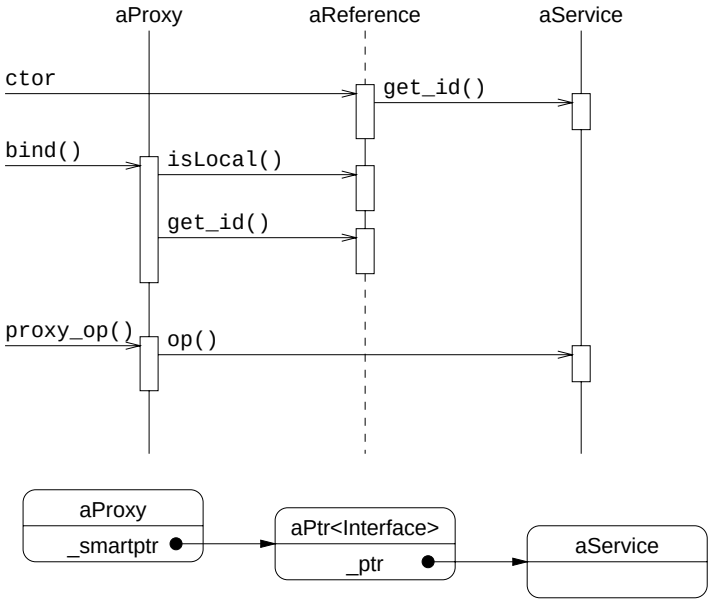

Figure 2: Binding to a local service.

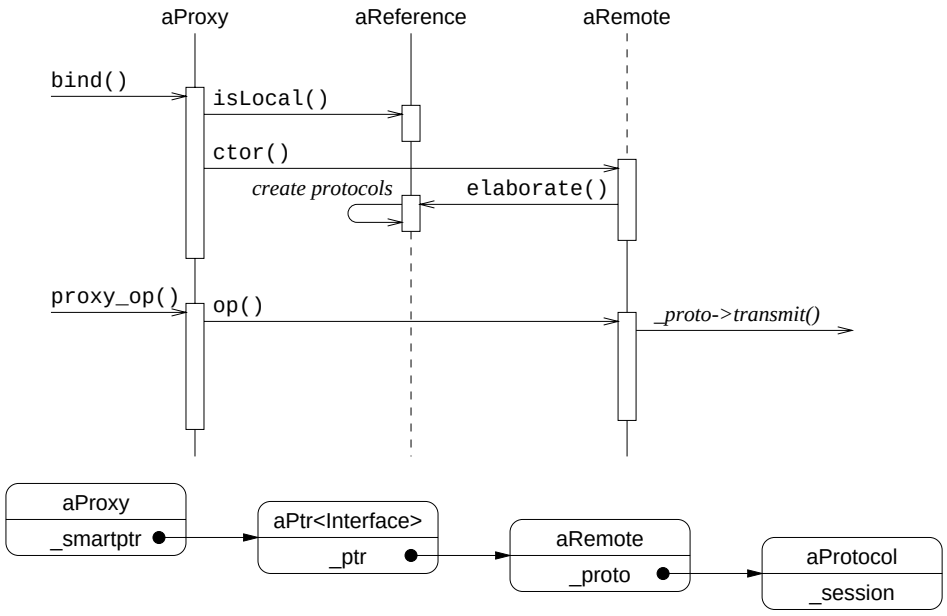

Figure 3: Binding to a remote service.

3.2 Binders

A binder implements the desired binding style by permitting controlled binding of the proxy which it manages. Different binders implement different behaviours in response to attempted use of an unbound proxy. A binder is often (but not always) transient, existing only to manage the initial binding action before removing itself. Currently identified binders are:

First-party binders require their proxies to be bound before use. When binding occurs, they remove themselves from the invocation sequence. They trap attempts at invocation before binding, throwing an exception or terminating the program.

Third-party binders block their invokers until an externally-initiated binding action occurs. Like first-party binders, they are transient.

Import binders intercept the first invocation and trigger the importation of a reference to the required service from a nameserver. They too are transient.

Reconfigurable binders behave like persistent third-party binders. They export a rebind service to a nameserver to which online management tools connect in order to break and replace the binding. These binders are responsible for obtaining the consent of the protocol layers underpinning the binding before carrying out the reconfiguration.

Relocating binders are persistent objects which support transparent binding replacement. This is useful for maintaining bindings to mobile servers. Possible semantics of these binders can be found in [11]. Their behaviour is closely related to that of reconfigurable binders.

Figure 4 shows the relationships between these binder classes. Interaction class proxies have no direct dependency on any of them and contain merely a reference to the smart pointer class, from which binders descend and by which they replace themselves when binding has taken place.

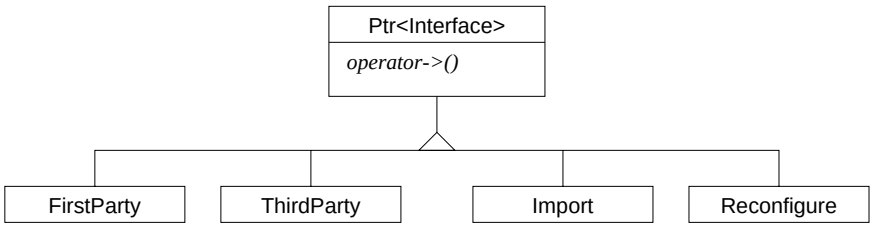

Figure 4: Binders: class relationships.

4 Populating the Framework

The validity of the framework is demonstrated by example. As extrema we consider event dissemination and remote object invocation.

4.1 Event Dissemination

Event dissemination is not easily accomodated in a pure client-server distributed system because it is not immediately clear which entity in the interaction is the client and which is the server.

In our model, clients are entities desiring event notification, while servers disseminate notifications of events. The interaction is typed by the structure of the notification data. The interaction comprises two distinct bindings: the primary carries control messages from the client to the server while the secondary, back-binding, carries event notifications from the server to the client. The back-binding is established by an `enable` control message carrying a reference to an 'event-sink service' private to the client-side of the interaction. When it receives a control message, the event service binds a new 'event sink proxy' using the reference to the event sink, and adds this proxy to its list of clients. `C++` classes which implement this interaction are:

Event implements the event service. It contains a list of proxies to client-side notification services. It supports the `announce` operation by which a server can transmit a notification using every proxy in this list.

Event::Proxy implements a requirement for a compatible event service. It possesses control members `enable` and `disable` by which a client indicates to the bound server its desire for event notifications. It supports blocking and selectable wait operations to allow the client to synchronise with the arrival of data.

Event::Reference is a typed, transmissible object which facilitates construction of the primary (control) binding from the client to the server.

Event::Interface is a typed abstract base class of **Event** and **Event::Remote**, supporting location transparency.

In our implementation, the event interaction comprises two instances of the asynchronous `Notify` class of which one carries the control data to the server and the other the event data to the client. The bindings between these endpoints may be heterogeneous, offering differing transport semantics for the two types of data, for instance reliable control messages and timely event messages to support group multimedia transmission.

4.2 Remote Object Invocation

In a CORBA-compliant environment [12] interaction between client and server entities is achieved by specifying their interfaces in an *Interface Definition Language*, IDL. An example of IDL is,

interface *file* {
    int *open* (**in** string name, **in** unsigned perm);
    int *close* (**in** int des);
    int *read* (**out** string buf, **in** int len);
    int *write* (**in** string buf, **in** int len);
}

The IDL compiler translates this definition into a representation which is useable in the implementation language of client and server components, typically C++.

In terms of our interaction model, the IDL compiler is a factory of user-defined invocation-oriented interaction classes. The back-end of our IDL compiler emits classes reflecting this view, figure 5,

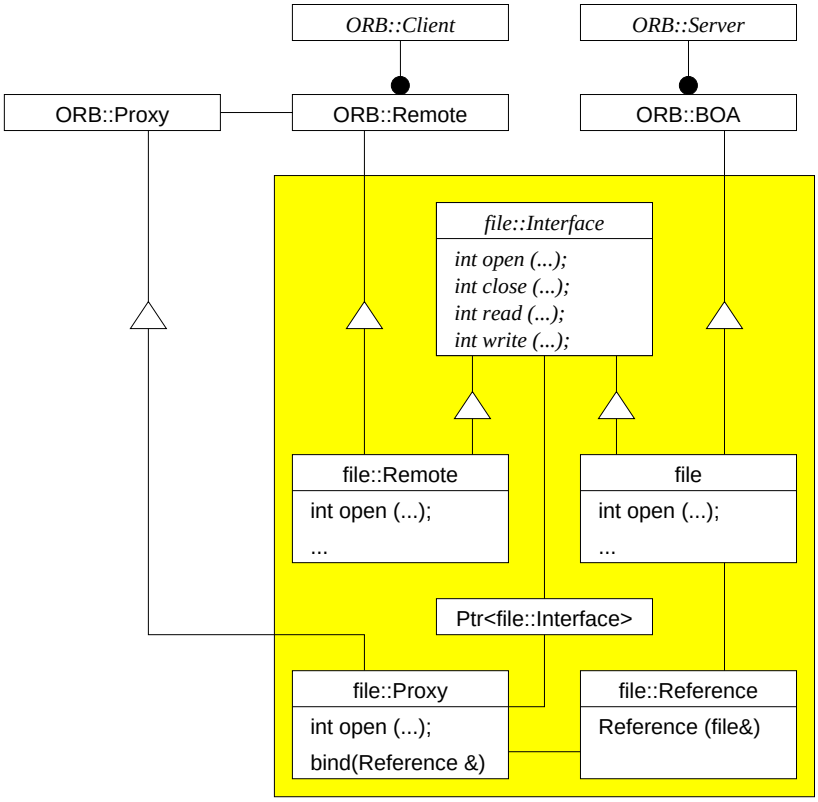

Figure 5: An IDL-generated interaction: class structure.

file is the object which implements the file-system service,

file::Proxy is a compatible, client-side entity,

Ptr<file::Interface> provides bindable access to either a local `file` or a `file::Remote`,

file::Remote provides transparent access to a non-local `file`, marshalling request parameters and unmarshalling returns.

Functionality common to all object-interaction classes is located in the ORB namespace:

BOA or *Basic Object Adaptor* implements a remotely-invokable server-side endpoint. CORBA further defines certain activation and authorisation responsibilities for BOAs which we do not implement at this level.

Remote stores the identity of its peer *BOA* and provides operations such as `send` and `wait` to send marshalled buffers and wait for the arrival of unmarshalled buffers via the *Client*.

Client provides an interface to the top of the protocol stack tailored to remote invocations. It is an abstract class; its functionality is supplied by a concrete client-protocol class which will be described in the next section.

Server like *Client*, is an abstract interface whose concretion is provided by a server-side protocol.

4.3 Pseudo-Interactions

Pseudo-interactions are degenerate cases of the interaction framework illustrating other aspects of its flexibility which were not envisaged when the framework was designed. We illustrate two cases: distributed program initialisation and operating system signals.

4.3.1 Distributed Program Initialisation

Distributed programs in Regis [9] comprise multiple copies of an executable image running on different machines. We refer to a single executing image as a *node*. Nodes exhibit different behaviours depending on their command-line arguments.

The first node of a Regis program to run always executes the anonymous singleton component at the root of the tree of components. When it wishes to instantiate a component at a node which does not yet exist, it contacts an execution service requesting it to run a new copy of the image with different arguments on another host on the network. These arguments inform the new node of (a) its logical node number and (b) the location of the root's configuration service. The new node binds by first party to the root configuration service and notifies it of *its* configuration service. It is now ready to accept instantiation and binding instructions.

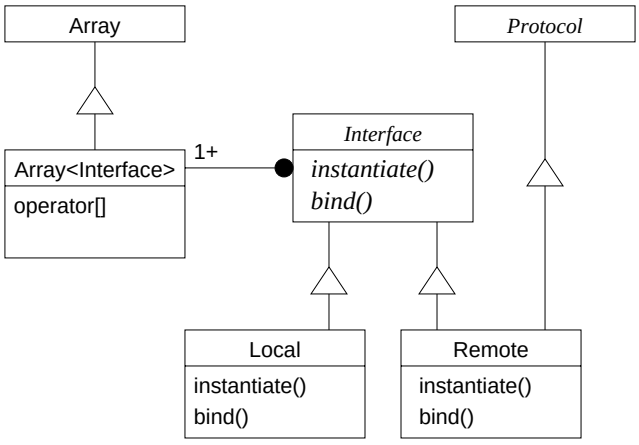

Figure 6: Configuration service: class relationships.

The interaction between nodes' configuration services is a degenerate case because, in order to provide a third-party binding service, the participants can only use first-party binding. Distributed node services are accessed through a membership map (of which a copy is present at each node but only the root's must be complete) indexed by logical node number. All entries in this map are accessed through a common abstract interface to provide location transparency to higher levels of the configuration service, figure 6.

4.3.2 Signals

A signal is an example of a process interacting with its operating system environment. It is desirable that a component be able to accept notification of a signal as easily as to participate in any other kind of interaction. This is achieved using the `Signal` interaction. It is degenerate because it cannot be bound to: the `Service` is accessed from the kernel managing the server's address space.

5 Conclusion

Our framework for construction of distributed interactions has been found to be flexible enough to accomodate any interaction which we have discovered to date. Currently implemented interactions are,

Interaction	*Purpose*
`Sync`	Distributed Synchronisation (Semaphores)
`Notify`	Asynchronous Notification
`Port`	Typed Message Ports
`Entry`	Ada-style entry-points
`Event`	Event dissemination
`Signal`	Operating System Signal
Object	Remote method invocation

Figure 7: Useful interactions.

In contrast to the approach described here, an earlier incarnation of the Regis system encouraged construction of user-defined interactions by exploiting the properties of the `Port`, either through inheritance or containment. Where ports were found wanting, the desired behaviour was obtained by allowing their default behaviour to be overridden. Such patching demonstrated that a deeper abstraction lay beneath that of the port and what was required was a framework in which to exploit it.

In summary, our framework provides the following abstractions,

- Location transparency is guaranteed by requiring the `Remote` participant to possess the same interface as the `Service` itself.
- The `Proxy`'s desired runtime binding semantics are obtained by the *binder* which controls access to the smart pointer to the service.
- Type and role compatibility is enforced by the `Proxy`'s typed `bind` member function which ensures that a particular interaction's `Proxy` can only be bound to that interaction's `Service` and further, that the type of data transmitted must be compatible with that expected by the receiver.

Acknowledgements

The author acknowledges the contribution of many stimulating discussions with his colleagues in the Distributed Software Engineering section at Imperial College, especially Jeff Magee, Naranker Dulay and Nat Pryce.

References

[1] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[2] S. Crane, N. Dulay, H. Fosså, J. Kramer, J. Magee, M. Sloman, and K. Twidle. Configuration management for distributed software services. In Y. Raynaud A. Sethi and F. Faure-Vincent, editors, *Integrated Network Management IV*, pages 29–42. Chapman and Hall, 1995.

[3] N. Dulay. Darwin language reference manual. Technical report, Department of Computing, Imperial College, 1992. ftp://dse.doc.ic.ac.uk/regis/darwin.ps.gz.

[4] N. Dulay, J. Kramer, J. Magee, M. Sloman, and K. Twidle. Distributed system construction: Experience with the conic toolkit. In J. Nehmer, editor, *Experiences with Distributed Systems*, pages 187–212. Springer-Verlag, 1987.

[5] S. Friedberg. Transparent reconfiguration requires a third-party connect. Technical Report 220, Department of Computer Science, University of Rochester, November 1987.

[6] Iona Technologies Ltd., 8-34 Percy Place, Dublin 4, Ireland. *The Orbix Architecture*, January 1995.

[7] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, April 1979.

[8] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference*, Barcelona, September 1995.

[9] J. Magee, N. Dulay, and J. Kramer. A constructive development environment for parallel and distributed programs. *IEE/IOP/BCS Distributed Systems Engineering*, 1(5):304–312, September 1994.

[10] J. Magee, J. Kramer, M. Sloman, and N. Dulay. An overview of the rex software architecture. In *Proceedings of the Second IEEE DCS Conference*, Cairo, October 1990.

[11] S. Menon and R. J. LeBlanc Jr. Object replacement using dynamic proxy updates. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 1(5):271–279, September 1994. Special issue on configurable distributed systems.

[12] The Object Management Group, OMG Headquarters, 492 Old Connecticut Path, Framington, MA 01701, USA. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, July 1995.

[13] R. Steinmetz and K. Nahrstedt. *Multimedia: Computing, Communications and Applications*. Prentice-Hall Innovative Technology Series, New Jersey, 1995.