Technical Report

UCAM-CL-TR-332
ISSN 1476-2986

Number 332

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Distributed computing with objects

David Martin Evers

March 1994

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

© 1994 David Martin Evers

This technical report is based on a dissertation submitted September 1993 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Queens' College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Summary

Distributed systems and object-based programming are now beginning to enter the mainstream of computing practice. These developments have the potential to simplify the distributed application programmer's task considerably, but current systems impose unnecessary burdens. Distributed operating systems provide palatable message passing between remote processes but leave the preparation and interpretation of messages to application code. Remote procedure call systems use familiar language-level concepts to hide distribution, but the awkwardness of service creation and binding discourages the use of transient objects. Finally, object-based programming languages which support distribution often ignore the possibility of failures and do not efficiently accommodate heterogeneity.

This dissertation discusses the design, implementation and evaluation of a practical system for *network objects* which addresses these problems for a representative programming language (Modula-3) and distributed computing environment (the ANSA testbench). We propose that language-level objects should explicitly represent *bindings* to potentially remote service access points (*interfaces*), which are sufficiently lightweight that they can be used as transient handles for shared state. Our system uses local objects to stand for remote services and local method call to cause remote operation invocation. Within a process, concurrency control is provided by familiar language-level facilities. The local programming language's object type system is made to represent the global service type system in a natural way. We support dynamic creation of service interfaces and the transmission of network object references in invocations. We allow the dynamic types of network object references to propagate between separate programs. Finally, we provide automatic, fault-tolerant and efficient distributed garbage collection of network objects. In each case, we discuss the requirements of a useful design and the tradeoffs necessary in a real implementation. Our implementation runs on stock systems connected by standard local and wide area networks and internetworking protocols. We believe our approach would support additional library-level tools for security, stable storage, distributed transactions and transparent service replication, though we have not pursued this.

The dissertation demonstrates that it is practical to retain many important amenities of modern programming languages when providing support for the construction of applications in a heterogeneous and evolving distributed system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed systems and object-based programming are now beginning to enter the mainstream of computing practice. These developments have the potential to simplify the programmer's task to the point where widespread construction of distributed applications (as well as the facilities traditionally associated with operating systems) is economically feasible.

Established tools for dealing with the problems of distribution in a heterogeneous environment have been collected into sophisticated systems which are now becoming widely available. Many of these systems are based on *objects*, in one form or another. Likewise, programming languages which support objects as first class citizens are now in common use. The topic of this dissertation is how the relationship between objects at these two levels can best be used to support distributed application programming.

Current systems impose unnecessary burdens on programmers. Distributed operating systems provide palatable message passing but leave the preparation and interpretation of messages to application code. Remote procedure call systems use familiar language-level concepts to hide distribution, but the awkwardness of service creation and binding discourages the use of transient objects. Finally, those object-based programming languages which support distribution often ignore the possibility of failures and do not efficiently accommodate heterogeneity.

This dissertation discusses the design, implementation and evaluation of a practical system for *network objects* which addresses these problems. We have not designed a universal distributed programming language integrated with a distributed computing environment.

Instead, our approach is to take a representative single-system programming language and investigate how its support for objects may most naturally be used to present the facilities of a distributed computing environment to the application programmer.

We propose that a language-level object in a client should explicitly represent a *binding* to a potentially remote service, and that a language-level object in a server should explicitly represent the *interface*, or service access point, which is the target of such a binding. Service creation should be sufficiently lightweight that service interfaces can be used as handles for transient state shared between clients and servers. To achieve these goals in a heterogeneous environment, we use a new technique: we automatically generate network object type declarations in the target language, as well as their associated stubs, from free-standing interface specifications.

Our system uses local *surrogate* objects to stand for remote services and uses local method call to cause remote operation invocation. Within a process, concurrency control is provided by familiar language-level facilities. The local programming language type system is made to represent the global service type system in a natural way. To hide RPC binding, we support the dynamic creation of service interfaces from language-level objects and the transmission of network object references in invocations. We allow the dynamic types of network object references to propagate between separate programs. Finally, we provide automatic, fault-tolerant and efficient distributed garbage collection of network objects.

For each of these features, we discuss the requirements of a useful design and the trade-offs necessary in a real implementation. The representative programming language in our implementation is Modula-3, and the representative environment for distributed computing is the ANSA testbench. Our implementation runs on stock systems connected by standard local and wide area networks and internetworking protocols. As far as we are aware, it is the first system in which a language other than C has been integrated with the ANSA testbench. Although our implementation work is based on Modula-3 and ANSA, we believe that much of our design and experience is applicable in other languages and environments.

Our distributed garbage collector was designed with the other co-authors of [Birrell 93a] while the author was a student intern at DEC SRC. Everything else described in this dissertation is the sole work of the author, who has implemented the collector twice: first for the SRC Network Objects system, and in a different form in the system presented in this dissertation.

In this work, we have concentrated on the language-level presentation of the management

of, and communication via, two-party network object bindings. Persistence of service state, secure communication between authenticated principals and mobility of services are outside the scope of the dissertation, as are techniques for achieving increased reliability through transparent replication, automatic fail-over, and atomic transaction protocols. We believe that it is possible to integrate established solutions to these problems with our approach.

In summary, the dissertation demonstrates that it is practical to retain many important amenities of modern programming languages when providing support for the construction of applications in a heterogeneous and evolving distributed system.

1.1 Outline

The structure of the remainder of the dissertation is as follows:

Chapter 2 introduces the topics of distributed computing and objects. We briefly review previous work in distributed operating systems, and the evolution of language-level support for distributed application programming. We then identify the problems we intend to address and state the goals and non-goals of our system's design. Finally, we present an outline of the environment in which our implementation work has been carried out.

In Chapter 3, we introduce the network object concept. We describe how the Modula-3 language can be combined with the ANSA testbench to produce a system which presents ANSA services to the programmer as language-level network objects. We then describe in more detail the relationship in our design between ANSA service definitions and the network object types which are their Modula-3 counterparts. Finally, we present an implementation of a stub compiler and runtime system to support the design features described so far.

Conventional programming languages have supported programmers with increasingly sophisticated type systems. Their purpose has been to allow mistakes to be detected as early as possible and to support information hiding. Distributed applications can also benefit from the use of types to govern binding between separately compiled programs. In Chapter 4, we discuss the uses of types in distributed systems and present the corresponding features of our system.

With the freedom to create remotely-accessible services shared by reference as simply as

objects within a single program comes the burden of resource management. Chapter 5 explains the problem, and presents a distributed garbage collector which considerably reduces its impact on application programmers. First we describe the collector's design in the abstract, then we explain how we have implemented it in practice.

We have implemented the system presented in the previous chapters in its entirety. Chapter 6 reports experience of distributed application programming with our system, describes some of the tests we made of it and presents some simple measurements of its performance.

In Chapter 7 we place our system in the context of recent work on distributed object-based systems and distributed garbage collection.

Finally, in Chapter 8 we present concluding remarks and directions for further research.

You may already know what a blow to the ego it can be to have to read over anything you wrote 20 years ago, even cancelled checks . . . It is only fair to warn even the most kindly disposed of readers that there are some mighty tiresome passages here, juvenile and delinquent too.

— THOMAS PYNCHON, *Slow Learner* (1984)

Chapter 2

Background

2.1 Distributed Computing

In the Cambridge Distributed Computing System [Needham 82], and others like it, services which had previously been associated with timesharing operating systems were scattered among small computers whose main I/O device was the network. This was made possible by the advent of cheap microcomputers and local area networks with low error rates and high bandwidths. Developments in internetworking and wide area network technology have now made distributed computing feasible on a continental scale.

In distributed computing, we have many computers and many networks which we wish to have work together to some end. Sometimes it is appropriate to move the program to the data, as is common for databases. Sometimes both should move to the computer—perhaps it is a PostScript printer. Sometimes logically shared memory fits the problem, and can be used if appropriate hardware support is available.

However, in an environment in which cooperating processors (or their abstractions, processes) can fail independently, and in which communication between processors is significantly slower and less reliable than internal communication, acceptable performance can only be achieved by keeping programs and the fine-grained data they manipulate together. Current local and wide area networks of conventional computers fit this description. While the bandwidth and error properties of networks may in the future approach those of traditional multiprocessor busses, their communication latencies will not: the communicating parties are physically further apart in a distributed system. In a distributed system, the

entities which are shared between cooperating parties are not bytes but *services*: packages of state and program which can perform computation and storage *on behalf of* others.

We will see later that service creation need not be a heavyweight operation: service instances can be created on the fly, as long as enough is known about their types (the specifications of the programs) ahead of time. Also, as in the case of a database, it is often possible to improve performance and make services more flexible by transmitting additional programs to them. These programs are written usually in a small special-purpose interpreted language, but occasionally in a bigger one such as CLU, Scheme or PostScript [Stamos 90, Bartlett 91, Gosling 86].

In this dissertation, we will be concerned with flexible language-level tools for the use and management of services in a distributed system. The central bridge between programming languages and distributed systems is the concept of an object.

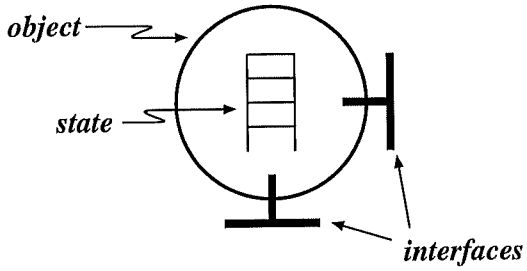
2.2 Objects

What is an object? The details of the answer to this question depend on who one is talking to and which system (or model) one is discussing. We will shortly discuss a number of systems which support objects in some form, but we should first explain our general view of the ideas involved. Our terminology, and in particular the distinction between objects and interfaces, is based on that of the ANSA architecture for distributed systems [ANSA 89]. It is illustrated in figure 2.1.

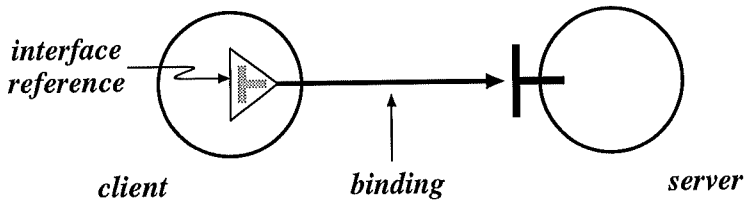
An *object* is something which has an *identity* (in some context) and *encapsulates* some *state*, which it may manipulate on behalf of the outside world in response to *invocations* of *operations* in its *interfaces*. The number of objects in a system may vary with time.

State is data—"the bits". An item of state is associated with exactly one object: objects cannot overlap or contain other objects. The outside world can only read or write an item of state *indirectly*, by making an invocation on the object which contains it. This is the sense in which objects encapsulate (as well as contain) state.

An interface is a service access point: a collection of operations, on exactly one object, with its own identity. One object may have (or *export*) many interfaces, for which it is said to be the *server*. The number of interfaces on an object may vary with time. We will often use the word "service" as a synonym for "interface".

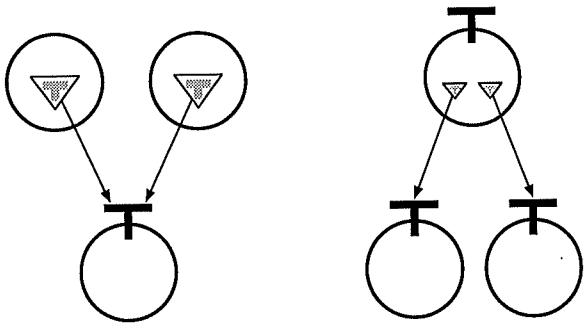


An object encapsulates state, which is accessible only via its interfaces.



An object may hold references to interfaces on other objects.

The association between an interface reference and the interface it denotes is called a binding.



An interface on an object is potentially the target of many bindings.

An object may be a server of some interfaces and a client of others.

Figure 2.1: Terminology for objects and interfaces.

An object may hold a *reference* to an interface on another object as part of its state. It is then said to be a *client* of that interface. Only interfaces may be named by objects in this way: objects cannot name other objects directly. Interface references held by distinct objects are regarded as distinct *as references*. The number of interface references in a system may vary with time. A collection of objects in some context may be designed such that it appears to be a single object, in that some of the interfaces on the objects in the collection cannot ever be referred to outside that collection. Objects are therefore in this sense the *smallest* units of encapsulation.

We will call the association between an interface reference and the unique interface it denotes a *client-server binding*, or a binding for short. An interface reference is half of exactly one binding. An interface is potentially the target of many bindings. We will often abuse this definition by using the word "binding" when referring to information about a given association held at the client.

An invocation is a dynamic process which starts at a client of an interface. It is possession of an interface reference which allows the holder to invoke operations over a binding. The invocation proceeds to the object which is the server of the denoted interface, possibly carrying some data called *arguments*. At the server, a computation is performed which may produce some *results*. The invocation, carrying any results with it, then returns to the client, where it finishes. The computation performed as a result of an operation invocation depends on *both* the operation *and* the object on which it is invoked. This is often called "dynamic binding" (of operation names to procedures) in the context of object-oriented programming languages. However, we will use the phrase slightly differently. By "dynamic binding", we mean the creation, as computation proceeds in an active system, of client-server bindings in the sense given above.

These definitions fall far short of capturing everything of interest in a distributed system, but will be useful nonetheless. Neither are they absolutely watertight (in how many interfaces may a given operation appear?). Slightly different terminology is widely used in discussions of programming languages: objects and interfaces are seldom distinguished, and the word "interface" often means an interface *specification*. Through force of habit, or for consistency with established terminology, we will sometimes use the words "object" and "interface" in that way. Whenever the reader encounters them, he or she is invited to consider whether they are being used of concepts at the level of a programming language or of a distributed system.

With these preliminaries over, we will now briefly review previous work in distributed operating systems and programming languages.

2.3 Distributed Operating Systems

Several operating systems have been designed to support distributed computing by regarding more-or-less traditional processes as the providers of services. Examples include CHORUS [Rozier 88], Mach [Jones 86], and Amoeba [Mullender 86]. In each of these systems, a (theoretically) small and efficient kernel supports a few minimal abstractions of the hardware, and everything else is built on these.

In these systems, a *process* is essentially a virtual address space on some machine, protected by memory management hardware from the activities of other processes on that machine. A *thread* is an independent locus of control within a process, with its own program counter and CPU state. Threads in the same process communicate and synchronise with each other via shared data. Communication between processes (possibly on different machines connected by a network) is by message passing. Operating system services such as filing systems, directory services, process and paged virtual memory management and network communication itself are all implemented as processes, communicating with clients and each other by inter-process communication (IPC).

IPC in these systems is based on ports, capabilities and messages. A *port* is a message queue protected by the kernel. A *capability* is a port identifier accompanied by some access rights and possibly other information. A *message* is a collection of (possibly typed) data items, which may include capabilities for ports. A thread enqueues a message at a port by calling a *send* primitive with the message and a capability for the port. To service a request, a thread calls a primitive *receive* operation on a port bound to its process. If there are no messages waiting, the thread is blocked. Otherwise, a message is returned and the thread acts on it. Since many interactions follow a request-response pattern, it is generally possible to send a *reply* to a message by supplying another primitive with an identifier for the message and the response. In this case, the *send* primitive either blocks the calling thread until a reply is received or returns a voucher which can be used in a later call to a synchronising *receive-reply* primitive. Thus, in the terminology of section 2.2, processes, ports, capabilities and messages loosely correspond to objects, interfaces, interface references and invocation arguments or results, respectively.

In Mach, ports bound to processes on remote machines are represented by surrogate ports held by a local *message server* process. This communicates via network protocols with the message server at the target machine, which forwards messages to the appropriate local ports. The right to receive on a port can be handed off to another process dynamically; this can be used to short-cut expensive general computation once the existence of a special

case has been established.

Similarly, CHORUS ports can migrate between processes. They can also be placed in collections called *groups* which have their own identities and capabilities. It is possible to choose whether a message sent to a group is enqueued at all or one of the ports it contains. Process segments in CHORUS are managed by associated *mapper* processes and can be cached in physical memory at more than one node. The mappers can implement coherence protocols between these local caches to provide shared memory between machines.

Capabilities in Amoeba include both a port identifier and an *object number*; the former identifies a server process and message queue, while the latter is interpreted by the server itself. A *rights* field contains a bit mask denoting which operations the holder of the capability may invoke on the object to which it refers. To prevent forgery, the rights field is cryptographically mixed with a random bit string when the capability is created. Amoeba's message passing primitives are designed to be used in the request-response style. A request header identifies the operation to be performed, while a reply header returns an error code and an optional capability. Parameters or results which will not fit into the headers are placed in an associated buffer.

These operating systems provide a palatable abstraction of message passing in a distributed environment. The main benefit they provide is to hide from the programmer the resolution of port identifiers to network-level addresses which underlies the use of capabilities. They achieve this by use of protocols generally based on broadcast in local area networks. As a result, dynamic binding of clients to server processes by passing capabilities is straightforward within a LAN.

However, such systems leave a significant amount of work to the programmer. Little help is provided to deal with the possibilities of network or process failure and server congestion. A major burden is that messages must be prepared and interpreted by application code. Even with the help of library routines to insert and extract values of common types such as integers and character strings, this is tedious and error prone, since it is not subject to automatic checking.

2.4 Language-Level Support

Some of the problems mentioned in the previous section are not essential features of *distributed* computing. Various attempts have been been made to address these problems

by mechanisms at the level of the programming languages used by application writers. We will briefly discuss three approaches: language-level abstractions of shared address spaces, remote procedure call, and the use of object-based languages.

2.4.1 Linda

Linda [Carriero 86] provides language-level support for an elegant model of distributed computing based on *distributed algorithms* [Carriero 89] rather than services. Despite our emphasis on service-based systems, we will sketch Linda for completeness.

Linda programs operate on distributed data structures in a global content-addressable *tuple space* whose elements consist of typed data records called tuples. Linda augments a standard programming language with three simple operations to manipulate the shared data. The `out()` operation adds a tuple to tuple space. The `in()` operation takes a combination of literal values and formal parameters which specify a template for tuples. If there is a matching tuple in tuple space, it is removed and exactly one `in()` operation returns, having bound the supplied formals; otherwise `in()` blocks until a matching tuple is present. Finally, the `read()` operation behaves exactly like `in()` but does not remove the matching tuple from tuple space. A complete Linda program terminates when all its constituent processes terminate or block for input.

Binding in Linda is by convention: the first element of every tuple is a character string name chosen by the programmer. The contents of one tuple may be used to address another by using the matching mechanism. Concurrency control is provided by the atomic removal of tuples from tuple space by `in()`. Linda is designed to be implemented on multiprocessors with a fast broadcast interconnect, so in practical implementations the contention implicit in the `in()` primitive becomes contention on the hardware bus, and is resolved by hardware arbitration mechanisms. Tuples implicitly contain a unique identifier to make the deletion from tuple space required by `in()` an idempotent operation.

The Linda model is suitable for parallel programming applications which are naturally expressed in terms of a shared data structure. However, because it relies on fast broadcast communication, the model does not scale to larger distributed systems. Neither does it address the possibility of failures. For these reasons we will not take our discussion further, except to note that the Linda primitives use language-level formal and actual parameters to relieve the programmer of the burden of creating and interpreting messages.

2.4.2 Remote Procedure Call

Many interactions between clients and servers in a service-based system take the form of *pairs* of request and response messages. The idea of remote procedure call [Birrell 84] is to use the familiar abstraction of calling a procedure which takes arguments and returns results to cover the distributed situation as well. In both the local and remote cases, control and data pass from caller to callee and back again. Three features distinguish RPC from the message passing facilities mentioned above: The creation and interpretation of messages is hidden behind the procedure call mechanism of the programming language. Concurrency is generated and controlled exclusively by local threads. Finally, there is an explicit attempt to make remote calls as reliable as local ones, as far as possible, even in the face of transient communication problems.

In a classic RPC system, the programmer is relieved of the task of *marshalling* data to and from message buffers by *stubs* generated automatically from the specifications of the procedures to be invoked. This automation, together with the typechecking provided by the programming language compiler, eliminates much of the drudgery of remote communication.

Procedural and message passing systems are essentially equivalent [Lauer 79] from the point of view of concurrency management, provided threads are sufficiently cheap to create. Therefore it is unnecessary to confuse the programmer by providing both. RPC systems choose procedure call as the dominant model because of its familiarity and the presence of hardware support for the local case. Because a thread which makes an RPC blocks until the call returns, all asynchrony is (in the absence of failures) apparently *local* asynchrony between different threads. Importantly, all synchronisation is between threads in the same address space, for which hardware support and well developed abstractions such as semaphores, monitors, critical regions, event counts and sequencers *etc.* are available [Bacon 93]. Having said this, the RPC concurrency model is not a licence to ignore the fact of distribution. It is generally inadvisable for a thread to hold a local mutex on some data structure while making a remote call, just as it would be a bad idea for a process in the message passing model to refuse to service new requests while awaiting the reply to a message it sent as a result of a previous request. In either case, bad performance or even deadlock can easily result.

In a classic RPC system, if an RPC returns successfully, it is guaranteed that the server procedure has executed exactly once. If the call returns an error indication, because of process or communication failure, it is known that the server procedure will execute at most once. Successful calls made from the same client thread are guaranteed to execute

in the same order at the server. These requirements are met by use of a specialised communication protocol implemented by a runtime library.

Early RPC designs tried to hide as many as possible of the differences between local and remote calls. Later designs placed less emphasis on this goal of *transparency*. We will mention three differences which undermine complete transparency: binding requirements, unreliability and heterogeneity.

In the single-process case, there is no ambiguity about the target of a particular procedure call. The call site has been bound to the callee by link editing at or before load time. In the remote case, an explicit mechanism must be made available to allow a client program to select and locate the service instance it requires. This is an awkward feature of traditional RPC: usually, each remote procedure is required to take an additional argument representing a *binding* to the required remote server. A binding generally denotes a *group* of remote procedures called an *interface*. Bindings contain interface identifiers and the addresses and port numbers required for the underlying communication protocols. They can be obtained in various ways. In the usual mechanism, servers register themselves with some name service with an explicit *export* operation, and clients obtain bindings from this service with an *import* operation. Binding is expected to be a much less frequent activity than making RPCs and may be correspondingly more expensive. The number of RPC interfaces on a program instance is fixed when the program is linked. In this respect, classical RPC interfaces are different from the interfaces of section 2.2.

The Concurrent CLU RPC system [Hamilton 84] retained the convenience of language-level support but did not attempt to hide the inevitable differences in performance and failure properties between local and remote calls. CCLU RPC used a distinct syntax for declaring and calling remote procedures. Calls on remote procedures could provide the remote binding explicitly in a distinguished clause of the syntax. They did not attempt to recover from network failures unless told to do so. The motivation for this was that end-to-end checks required at the application level often make expensive reliable communication techniques redundant [Saltzer 84]. Rather than using stub procedures for remote calls, the CCLU compiler itself generated calls to runtime marshalling code. This code interpreted information maintained by the local garbage collector to linearise entire heap structures for transmission, preserving pointer loops and sharing as required by the semantics of values in CLU [Liskov 81]. CCLU also extended CLU with concurrency control primitives which were implemented on top of the Mayflower kernel.

Finally, a distributed system has the opportunity (and sometimes the requirement) to accommodate *heterogeneity*. Different parts of an evolving system may be written in

different programming languages, or run on distinct hardware architectures or operating systems. An RPC mechanism can act as the glue which connects and coordinates the disparate components of a distributed system, but must trade off transparency to do so. RPC for a single programming language and architecture can be tailored for its particular type system and its values' representations on that architecture. A mechanism for communication between different languages must define an explicit external representation for values and restrict the transmittable types to those which can be interpreted in every language. Similarly, interface specifications must be written in an external *interface definition language* or IDL. IDL interfaces contain type and procedure specifications; from these, stubs for each programming language are generated.

Mach and Amoeba (among others) have been extended with IDLs (Matchmaker and AIL, respectively) to support heterogeneous RPC over their message passing protocols. Stubs are generated mainly in C, though support for other languages is planned for both. The Heterogeneous Computer Systems project at the University of Washington took this approach much further in the HRPC facility [Bershad 87]. HRPC allowed a collection of existing RPC mechanisms to interwork by defining clean abstractions of the components of any RPC system. Five components were identified: the stubs, the binding protocol by which clients locate servers, the data representation on the wire, the transport protocol which moves bits between hosts, and the control protocol which tracks the state of each call. Each of the last four components could be selected independently. This was done dynamically when establishing a binding via the HCS Name Service. The components communicate with each other via procedure variables in the binding structure passed to each stub.

2.4.3 Object-Based Languages

RPC tries to present distribution to the programmer in familiar terms, but as we saw in section 2.4.2, binding to services in remote programs is different from binding to procedures in the same program. In the terminology of section 2.2, a same-program procedure name denotes both an operation and a *fixed, implicit* object: the program instance in which the procedure is called. In order to allow the same operation to be invoked at varying remote objects, RPC procedures must be given the extra binding argument. With only a slight shift of emphasis, from the procedure to the binding, this style becomes that of object-based programming languages. It is therefore attractive to use a programming language which supports objects for programming in a distributed system. For the rest of this section, we will be using the word "object" in the senses defined by the various

languages under discussion.

One possibility is to start from an existing object-oriented programming language and attempt to add as much distribution transparency as possible. This was the approach taken in Distributed Smalltalk [Bennett 87], which allowed object references to be passed between nodes and provided remote method invocation. Smalltalk objects are not self-contained. They acquire behaviours from definitions in classes. Classes themselves share behaviour by *reactive inheritance*, in which changes to a class definition immediately affect the behaviours of all its dependent objects and classes. Because of these mechanisms, Distributed Smalltalk faced severe problems in implementing object mobility. Since everything (even an integer) is a full-blown object in Smalltalk, object mobility is a prerequisite for efficient remote communication. Distributed Smalltalk also suffered from the mismatch between distribution and the traditional Smalltalk approach to type errors. A programmer only discovers a type problem when a method call fails with a "message not understood" error displayed in a debugger window. The problem lies in deciding on which machine to pop up this debugger.

Another possibility is to design an object-based programming language whose semantics are intended from the outset to model distributed systems. This is the approach taken in Emerald [Black 87, Jul 88, Raj 91]. Emerald attempts to provide the programmer of a distributed system with both the facilities of modern single-system programming languages and the tools required to deal with distribution, all in a uniform manner. The Emerald programmer is supported by objects, monitors, a sophisticated type system and object mobility.

Everything in Emerald (from booleans to compilers and abstract machines) is in theory an object. Every object has a network-wide identity, some private data in the form of references to other objects, a set of operations and, optionally, its own background thread. An object's data may be accessed only by its own operations. Operation invocation provides the only means of interaction between objects. Many threads may be active in an object; concurrency control is provided by monitors. These are strongly coupled to objects: some of a monitored object's operations are designated as entry points to its monitor.

The signatures of an object's public operations define its *abstract type*. We will say more about abstract types in Chapter 4; for now we simply mention that Emerald defines a *type conformance* rule, based on the notion of substitutability of instances of abstract types, which governs binding between objects. Type checking is performed statically where possible and at bind time otherwise. Among other things, the information hiding

provided by abstract types allows the Emerald compiler to generate code to implement the uniform object invocation semantics by varying mechanisms. These trade off efficiency against generality according to what the compiler can deduce from the program text about how a particular object will be used. This relieves programmers of the burden of making these decisions for themselves.

An RPC system hides the work of locating and communicating with a remote server, but not the fact of its remote location: stubs must be generated, and bindings obtained. In contrast, the Emerald programmer need not choose *a priori* which objects can be remotely invoked. All communication between objects is notionally via a runtime kernel, which locates remote objects and implements remote operation invocation. (It is important that the compiler optimise this where it can.) Emerald allows programmers to ignore or control the relative location of objects.

Because objects encapsulate both code and data, they are self-sufficient. They can therefore be made mobile, provided that the code can be interpreted everywhere. It is possible to move objects from node to node explicitly, and to co-locate one object with another. This allows decisions about object location to be deferred until run time and to change dynamically with circumstances, in order to minimise communication costs. For instance, when objects are passed as parameters to operations on other objects, the programmer may specify that the parameters themselves should move to the callee. This style of object mobility contrasts with the value transmission method for abstract data types described in [Herlihy 82]. In Emerald, code and data are packaged in an object and are (notionally) transmitted together. Herlihy's scheme allows separate implementations of a given abstract type to communicate *values* of that type among themselves by agreeing on a common external representation.

Like all pure object-based languages, Emerald defines parameter passing with call-by-sharing semantics, even in the remote case. As usual, it is necessary to provide hints (possibly implicit in type information) which cause the use of more efficient implementation mechanisms where appropriate, such as in the case of small immutable objects like integers. With these semantics, garbage collection becomes a virtual necessity as well as an important tool; we will discuss Emerald's distributed garbage collector in section 7.2.3.

It is instructive to compare, as is done in [Levy 91], the styles of distributed application programming encouraged by RPC systems and object-based systems like Emerald. Applications built with RPC systems tend in practice to follow a pattern in which client programs bind to an entire server program as a unit. Operations which involve referring to small parts of the state of the server are implemented by ad hoc mechanisms,

and callbacks from server to client are rare. Neither problem is an essential property of RPC. However, the awkwardness of service creation and binding, which in many systems involves communication with a separate name service, encourages the creation of fewer, longer-lived server programs. By contrast, applications built in languages like Emerald create transient objects freely and bind to them dynamically. Though mobile objects must in theory be located afresh at every invocation, hints are usually cached with bindings.

Emerald's uniform object model comes at a price. In order to achieve acceptable performance (especially with object mobility), Emerald is implemented in a homogeneous environment. Every node is of the same hardware architecture and is attached to the same local area network. Each node is a single address space. To protect objects from each other, Emerald relies on the compiler's implementation of the language semantics rather than hardware memory protection. Because the Emerald approach requires a tight coupling between language and runtime, it is difficult to integrate with existing systems. Most importantly, to provide uniform semantics for all objects, Emerald simply ignores the possibility of non-maskable node or communication failure: the alternative is to force programmers to deal with the possibility of failure every time their programs read an integer.

The Emerald approach has the great advantage of conceptual economy: one simple, uniform model describes the whole system. Its main disadvantage is that it fails to capture the essential complexities of a real distributed environment: one simple, uniform model describes the whole system.

2.5 Design Goals

As we have seen, there are a number of possible granularities at which the object structure of section 2.2 can be applied in a distributed system, from the processes of the operating systems mentioned above to small data such as integers in Emerald. In the work presented in this dissertation, we have attempted to retain the best of both worlds: both the sophisticated linguistic features of object systems like Emerald and the support for realistic distributed environments provided by heterogeneous RPC systems like Matchmaker. Our goals are as follows:

- Services should be as simple to create and bind to as language-level objects, in order that they may be used as handles for transient state shared between clients and servers.

- Services should be as effectively and flexibly typed (both statically and dynamically) as language-level objects.
- Tools for local concurrency control and distributed garbage collection of transient services should be available to distributed applications.
- Multiple languages and architectures should be supported.
- The programmer should have enough information to exercise control over the handling of failures in a manner appropriate to his or her application.
- An efficient implementation should be possible on stock hardware using standard compilers, operating systems and language runtime environments.

As a result of these goals, we are prepared to make some sacrifices:

- Identification of service types *a priori* is acceptable. We aim to reduce the incidental drudgery of distribution, not to avoid the need to design applications for distribution from the start.
- Mobility of service instances is not a priority, though the ability to pass service references freely is. If it is simple enough to create and bind to services dynamically, applications can implement their own location policies at service creation.

Implementing complete distribution transparency for all objects in the face of partial failure, communications problems and heterogeneity is prohibitively expensive. System designers therefore make different tradeoffs between transparency, reliability, linguistic support, implementation complexity and performance. Many object-based distributed systems exist [Chin 91]. Ours balances these factors in a manner which we have found very useful in practice.

Rather than defining a single, completely uniform object model, we have investigated what happens when we start from a separate distributed computing environment and programming language. The environment provides typed, invokable services independently of particular programming languages, operating systems and hardware. A programming language provides its own facilities for objects, threads, exceptions and so on. We have attempted to integrate the two so that language-level object identity and type in a single program can *stand for* identity and type in the distributed environment. We also investigate how the tools (such as type checking and garbage collection) associated with

local programming languages can be extended in a language-independent way to realistic distributed systems.

In order to prepare for the presentation (in Chapters 3, 4 and 5) of our work in pursuit of these goals, we will now outline the two systems on which we have based our experiments. Our representative distributed computing environment is the ANSA testbench, and our representative programming language is Modula-3.

2.6 The ANSA Testbench

The Advanced Networked Systems Architecture [ANSA 89] is an object-based framework for open distributed processing in a heterogeneous environment. The ANSA testbench [ANSA 92c] is a particular implementation of the ANSA engineering model, running on top of UNIX and other operating systems, and is in regular use in the Laboratory.

The features of the ANSA testbench closely follow the definitions in section 2.2: services are provided by *objects* at *interfaces*. Objects are the smallest units of encapsulation, distribution and failure. Interfaces are the units of binding and service provision. An object can provide services at many interfaces, and be a client of many interfaces.

Interfaces are typed. An interface's type is described by an *interface specification* written in the testbench's IDL, a descendant of the Xerox XNS Courier language [Xerox 81] from which stubs are generated in the usual way. An interface specification consists of a set of data type and operation declarations. An interface specification can be declared to be *compatible* with a set of ancestor interface specifications, whose types and operations it inherits. IDL's data types are fairly standard. Although there are no pointer types (which would support the transmission of graph-structured data), there is a reference type for interfaces, which we will discuss below. IDL is independent of any particular programming language: a mapping to and from IDL types must be defined for each. A standard external representation is defined for IDL data types.

2.6.1 Interface References

Interfaces exported by objects are named by *interface references*. Unlike the bindings of standard RPC systems, and more like capabilities in Mach, CHORUS and Amoeba, interface references may be passed freely across interfaces without the mediation of a log-

ically central name service. Interface references contain address hints for the underlying transport services, though these are not visible to the application programmer. IDL has both a generic `INTERFACEREF` data type and a type constructor which declares that an `INTERFACEREF` is `OFTYPE Foo`, for some interface type `Foo`, though such values contain no type information themselves. Possession of an interface reference allows an object to *invoke* an operation in an interface exported by another object. Two alternative invocation styles are distinguished in interface specifications: *interrogation* operations have exactly-once semantics and cause the calling thread to block until a reply is received, while *announcement* operations have at-most-once semantics and do not block the caller. Announcement operations do not return results.

2.6.2 The Trader

Offers of service by objects are published in a well-known service called the *trader*. An offer consists of an interface reference associated with a collection of name-value properties. Offers are placed in a hierarchical space of *contexts* named by path names. A string property called 'Type', whose value is the name of the corresponding interface specification, is associated with every offer. The trader can search a requested context and its descendants for offers which are compatible with a requested service type (according to a directed acyclic compatibility relation stored as part of the trader's state) and whose properties obey a set of requested constraints, returning acceptable interface references to the client. Separate trader interfaces allow the management of the context and type spaces. Traders can be *federated* to mount a portion of one trader's context space in another's.

2.6.3 DPL

Application programs that wish to use testbench facilities do so through commands written in a language called DPL, which provides some convenient syntactic sugar for calls to stub and runtime procedures. DPL statements are embedded in the application source, and are expanded into appropriate application language code by a preprocessor. A DPL preprocessor for C is the only one currently available. Since C lacks linguistic features for concurrency control and invocation of dynamically bound operations, these must be provided by the testbench runtime.

On the client side, a C identifier can be `DECLARE`d to hold an interface reference to a service

of a given type. Once this interface reference has been obtained, via the trader or through other operations, invocations can be made on it with an object-like syntax:

```
! DECLARE {ir} : Foo CLIENT
  ...
! {results} <- ir$Operation (args)
```

Service interfaces of a given type can be created by a DPL pseudo-operation which returns a reference to the new service suitable for transmission to other services like the trader:

```
! DECLARE {svc} : Foo SERVER
! {svc} :: Foo$Create(concurrency, initialisation args)
! {} <- traderRef$Export (... , svc)
```

A pointer to per-interface state is handed in an *interface attributes* argument to the server routines supplied by the programmer. If the optional initialisation arguments are given to `Create`, the programmer must supply a function which initialises this state appropriately. Server functions are passed pointers through which to write their main results, and return an error code which is interpreted by the server stubs.

2.7 Modula-3

Modula-3 [Nelson 91b] is a recent addition to the long line of programming languages descended from Algol-60. It was designed for use in large scale systems programming, and is particularly well adapted for building distributed systems. It is strongly typed, and provides separately-compiled modules specified by interfaces, type-safe pointers with garbage collection, exception handling, lightweight threads, objects with inheritance and the isolation of unsafe code. These features are combined in a sufficiently coherent manner that the language's defining report [Cardelli 91] is only fifty pages long.

The next few sections describe in more detail the aspects of Modula-3 that are most important for distributed systems work.

2.7.1 Objects

Objects in Modula-3 are quite closely related to those in Simula 67 [Dahl 70], where the idea originated. An object is a record of data fields paired with a record of procedure values called methods. Objects are classified into object types; instances of a given type share the same method suite and data record type, but have distinct data records.

The values in the method suite refer to ordinary top-level procedures. The method call `o.m (arg, ..)` causes the procedure bound to `m` in the method suite of the object referred to by `o` to be invoked with its first argument bound to `o` and the remaining arguments bound to the values specified by `(arg, ..)`. The method procedure then has access to `o`'s data record via its first argument in the usual way; there is no reserved "self" identifier or implicit interpretation of unqualified identifiers as fields of the current object. In C++ terminology, all methods are "virtual".

New object types may be defined by extending old ones, specifying extra fields for the data record or extra method signatures for the method suite. A new object type may also be defined by specifying a new procedure to be bound to an existing method of an old object type. For example, if `A` is an object type with a method `m1`, the declaration

```
TYPE
  AB = A OBJECT
    f: F;
  METHODS
    m2 () := P2;
  OVERRIDES
    m1 := P1;
  END;
```

introduces a new type `AB` whose data record has all the fields of `A` plus a new field `f` of type `F`. Its method suite is a copy of `A`'s updated so that `m1` is bound to procedure `P1` and extended with a new method `m2` bound to `P2`. It is possible to introduce methods without binding procedures to them.

Initial values for fields and overrides for methods may be supplied when an object is created with the `NEW` pseudo-procedure. Storage for objects is always heap allocated, and objects are always assigned by reference. Thus object identity is pointer equality, and object arguments to procedures are passed in the call-by-sharing style of CLU [Liskov 81].

2.7.2 Exceptions

Modula-3 provides facilities which allow the programmer to deal separately with normal and exceptional outcomes of statements. If the execution of a statement RAISEs an EXCEPTION, the currently active control scopes are left one by one until a handler for the exception is encountered, when control passes to the handler. Values can be associated with exceptions and made available to the activated handler.

Handlers are introduced into the control stack by TRY-EXCEPT statements. There may thus be more than one control scope per procedure activation. Special code can be attached with a TRY-FINALLY statement to a control scope which is activated as the scope is left. Such code is generally used to ensure that some invariant is restored irrespective of how control leaves the protected block.

As an example, if Zero is an exception which has been declared with an integer argument, then after the statement sequence

```
i := 0; j := 0; k := 0;
TRY
  TRY
    k := 1;
    IF i = 0 THEN RAISE Zero (2) END;
    j := 3
  FINALLY
    k := 0
  END
EXCEPT
  Zero (n) => j := n
END
```

we have i = 0, j = 2 and k = 0. In this example, k is intended to suggest some state internal to the abstraction represented by the inner TRY-FINALLY, j some state shared with its caller (the outer TRY-EXCEPT), and i some property of a lower-level abstraction.

Unlike Mesa [Mitchell 79] (one of its ancestors), Modula-3 does not allow a handler to resume the control scope which raised its exception. In this and other respects, exceptions in Modula-3 resemble those of CLU.

Procedures are abstractions of statement sequences; thus their specifications need to include information about exceptional outcomes as well as normal ones. A `RAISES` clause in a procedure signature can be used to make guarantees about which exceptions may propagate out of a call. These guarantees are enforced by checks generated by the compiler. One can give either an explicit (possibly empty) set of exception identifiers, or the special set `ANY`, which places no restrictions on which exceptions the procedure may raise.

As well as allowing normal-case and abnormal-case code to be separated, exceptions have an analogous effect on procedure (or block) specifications. A block can raise an exception to indicate that it is unable to establish its normal postcondition, whether because of the failure of a lower-level abstraction or because the caller failed to satisfy the block's precondition. The latter situation is strictly a programming error in the caller, but the best response depends on the circumstances. In a single program it would be acceptable to crash (for instance by raising an exception which is absent from the `RAISES` clause of the enclosing procedure), but when a client program in a distributed system fails to meet the precondition of a server procedure, it is the client, not the server, which should crash. This can be arranged by having the server raise an exception which is passed back to the client, though this exception should be distinguished from the others to prevent the client's failure from being masked by code which recovers from failures of abstraction.

2.7.3 Threads

A Modula-3 program can contain multiple lightweight processes called threads, which share the same address space and communicate via shared data. Access to shared data can be serialized by acquiring and releasing objects of type `MUTEX`; the language provides some syntactic sugar for this in the form of the `LOCK` statement. To implement more complicated scheduling policies, threads may be blocked on objects called condition variables. These are associated by convention with a `MUTEX` which is used to protect the data used to make the scheduling decision. `Signal`ling a condition variable makes at least one (and possibly more) of the threads blocked on it runnable, whilst `Broadcast`ing unblocks all such threads.

These facilities are a practical variant of Hoare's monitors [Hoare 74], based on experience with Mesa [Lampson 80] and Modula-2+ [Rovner 85]. They are used in a style best captured by [Birrell 91a], from which the simple example in figure 2.2 (an unbounded LIFO buffer) is adapted. The `LOCK` statements ensure that `mu` is acquired before the enclosed statement block is entered, and released no matter how control leaves it. In fact, each `LOCK` is simply sugar for an obvious `TRY-FINALLY` statement, and the `RETURN`

```
VAR
  mu        := NEW (MUTEX);
  nonEmpty  := NEW (Thread.Condition);
  l: List.T := NIL; (* protected by mu *)

PROCEDURE Consume (): REFANY =
  BEGIN
    LOCK mu DO
      WHILE l = NIL DO Thread.Wait (mu, nonEmpty) END;
      RETURN List.Pop (l)
    END
  END Consume;

PROCEDURE Produce (item: REFANY) =
  BEGIN
    LOCK mu DO List.Push (l, item) (* now l # NIL *) END;
    Thread.Signal (nonEmpty)
  END Produce;
```

Figure 2.2: Concurrency control in Modula-3.

in Consume is semantically equivalent to the raising of a special return-exception. Here mu protects the trivial monitor invariant (l = NIL) $\Leftrightarrow$ the buffer is empty. This is true whenever mu is not held, and hence immediately after mu is acquired.

The call Thread.Wait (mu, nonEmpty) atomically both releases mu and blocks the current thread on nonEmpty. This atomicity prevents the loss of an intervening signal on nonEmpty, thus avoiding the "wake-up waiting" race. When the thread is unblocked, Thread.Wait will reacquire mu then return. Note that the semantics of Thread.Signal require the condition l = NIL to be rechecked on return from the wait in Consume: more threads might have been unblocked than items placed in l by Produce, and in any case, another thread might enter Consume first. This style, in which spurious wake-ups are benign, also

makes it safe to place the `Thread.Signal` in `Produce` outside the `LOCK` statement, which may reduce contention on `mu`.

It is possible to send an asynchronous interrupt to a thread by `Alerting` it. Alerts are used primarily to abort long-running computations when it is unknown on which condition, if any, the target thread is waiting. There is no guarantee that an alert will be delivered unless the target thread is eventually in certain scheduling procedures, when the exception `Thread.Alerted` is raised in it.

The threads facilities in Modula-3 are intended to allow formal reasoning about the concurrency properties of programs using them. Their semantics are formally specified in [Birrell 91b].

2.7.4 Type System

Modula-3 has a traditional Pascal-like set of scalar types, including programmer-defined enumerations. The constructed types include fixed and open arrays, sets of ordinals, procedure types, records and both "traced" and "untraced" references and objects. There are no unions, discriminated or otherwise: these can be simulated with objects. Storage for traced references is automatically garbage collected, while that for untraced references must be reclaimed by the programmer with `DISPOSE`.

Assignability of an expression to a variable is partly governed by a syntactically determined subtype relation between types. Runtime checks ensure that the value of the expression is a member of the value set of the type of the variable when this cannot be deduced statically. If a type `T` is a subtype of `U`, written `T <: U`, then every value of type `T` is also a value of type `U`. The relation `<:` is reflexive and transitive. It includes the facts that every traced reference type `<:` the predefined universal reference type `REFANY`, and that every traced object type `<: ROOT <: REFANY`. An object type `<:` its unique parent type, and thus all of its ancestors by transitivity; this is the main way for the programmer to extend the `<:` relation.

Procedure types are related quite conservatively: if `P1 <: P2`, `P1` and `P2` have the same number and types of arguments and results, though `P2` may allow more exceptions in its `RAISES` clause. This contrasts with the contravariant conformance rule of such languages as Trellis/Owl [Schaffert 86], Emerald or the many polymorphically typed lambda calculi.

As well as supporting conventional static typechecking of program texts, Modula-3 allows

running programs access to dynamic type information. Every value allocated in the traced heap (that is, every possible referent of a traced reference) is tagged with an integer code which describes the allocated type of the value. Values have the same typecode if and only if they have the same allocated type, though the mapping from typecodes to types may change between program runs. The built-in function `TYPECODE` can be used to inspect the typecodes of types and the referents of traced references. `ISTYPE` can be used to determine whether or not a referent is a member of a given type, `NARROW` changes an expression's view of a referent if this is valid, and the `TYPECASE` statement abbreviates a common idiom in which the *first* of a collection of statements guarded by the dynamic type of a referent is executed.

Figure 2.3 gives some examples in which the object types `A` and `AB` are as in section 2.7.1 above. After these declarations, the assignment `rab := raba` would succeed, `rab := ra` would produce a runtime error, and `rab := ri` would be rejected as an error at compile time. Notice that `Kind (ri)` requires a runtime range check: `CARDINAL` is the non-negative subrange of `INTEGER`.

Types can be given identifiers, usable in type expressions, in the usual way. Two type expressions in Modula-3 denote the same type if they become identical after evaluating constant expressions and replacing type identifiers by their values—a process which produces an infinite expansion for recursive types. Thus Modula-3 uses structural rather than name equivalence for its concrete types. This decision makes it simpler to decide when two types in different programs are the same, though other factors also influenced the language designers [Anon 91].

Abstract types are provided by a combination of language features. A type name can be introduced without revealing its complete structure by means of a partially opaque declaration, such as `TYPE T <: Public`. This asserts that there exists in the final complete program a declaration which binds to the identifier `T` a type which `<:` the type `Public`. More constraints can be placed on `T` by declarations of the form `REVEAL T <: U`, provided these revelations are consistent. Each complete program must contain exactly one definitive revelation `REVEAL T = Tconcrete`. Judicious use of the language's scope rules, interfaces and modules to control the visibility of these declarations provides a flexible way to distribute knowledge of a type at different levels of abstraction amongst a program's components.

Partially opaque type declarations aren't quite enough to provide abstract types: abstraction can be broken by defining a new type structurally equivalent to that in the hidden definitive revelation, and assigning the abstract value to a variable of the new type.

After the declarations:

```
VAR
  r : REFANY; ri := NEW (REF INTEGER);
  ra   : A        := NEW (A);
  rab  : AB       := NEW (AB);
  raba : A        := NEW (AB);

PROCEDURE Kind (rr: REFANY): CARDINAL =
  BEGIN
    TYPECASE rr OF
    | NULL             => RETURN 0
    | REF INTEGER (i)  => RETURN i (* sic *)
    | AB               => RETURN 2 (* note order *)
    | A                => RETURN 3
    ELSE                  RETURN 4
    END
  END Kind;

BEGIN ri^ := 1; r := ri END;
```

The following identities hold:

```
TYPECODE (rab) = TYPECODE (raba) = TYPECODE (AB)

ISTYPE (rab, A) = ISTYPE (raba, AB) = TRUE
ISTYPE (ra, AB) = FALSE

(NARROW (r, REF INTEGER))^ + 1 = 2

Kind (NIL)          = 0     Kind (ri)     = 1
Kind (ra)           = 3     Kind (rab)    = Kind (raba) = 2
Kind (NEW (MUTEX)) = 4      (rab = raba) = FALSE
```

Figure 2.3: Dynamic types in Modula-3.

This is avoided by a device which distinguishes otherwise equivalent types, as follows. Every application of a reference type constructor (`REF` or `OBJECT`) contains an optional `BRANDED "text"` clause which makes `"text"` part of the structure of the constructed type, in much the same way as, for instance, the name of a field. A given `"text"` can be used at most once as a brand in a complete program, which prevents the new type above from being legally declared. If `"text"` is omitted, it defaults to a value guaranteed distinct from all other brands, implicit or explicit, in the program. All definitive revelations must have a branded outermost type constructor.

By these means, knowledge of the concrete structure of a given type can be confined to a limited scope within a single program. When a data item is to be shared between different programs, knowledge of its type's structure, including any explicit brands, confers access to its representation. This raises the problem of checking that communicating programs agree on the type of a piece of data; we will address this in Chapter 4.

and once the edges have been put together, the detail pieces put in place ... and the bulk of the background pieces parcelled out according to their shade of grey, brown, white or sky blue, then solving the puzzle consists simply of trying all the plausible combinations one by one

— GEORGES PEREC (tr. David Bellos), *Life A User's Manual* (1978)

Chapter 3

Network Objects over ANSA

This chapter presents the basic features of our system [Evers 92]. These features allow the services available in the ANSA testbench distributed computing environment to be presented to the application programmer in terms of language-level *network objects* in Modula-3. The features described in this chapter are directed mainly towards those of our design goals (section 2.5) which are concerned with *binding* and *communication* in the remote operation invocation style. Chapters 4 and 5 will discuss distributed typechecking and garbage collection in more detail.

Because they have identities and types and support operation invocation, the objects provided by some programming languages are good candidates to represent services in a distributed system. As we saw in section 2.4.3, it is possible to take a language like Smalltalk and extend its notion of object identity beyond a single program instance. A means to do something similar for Modula-3 was proposed by Greg Nelson in [Nelson 91a]. The way the design of our system presents distribution to the Modula-3 programmer has been strongly influenced by this proposal. We will therefore begin by sketching it.

Nelson's proposal does not address heterogeneous systems with multiple programming languages. The ANSA testbench does, but its embedded-DPL technique (section 2.6.3) is awkward to apply to languages like Modula-3. We present our solution to the problem of integrating the two approaches. We then describe in more detail the relationship in our design between ANSA service definitions and the network object types which are their Modula-3 counterparts. Finally, we will present an implementation of a stub compiler and runtime system to support the design features described in this chapter.

3.1 Network Objects

3.1.1 Remote Objects

The basic intention of Nelson's proposal is to allow a Modula-3 program to invoke methods on objects in other Modula-3 programs. We will outline the proposal by discussing an artificially simple example: a service which echoes text strings.

First, we define an object type to represent the echo service. This service supports a single operation, "`say()`", which just returns its argument.

```
INTERFACE Echo;
  TYPE T = OBJECT METHODS
    say (txt: TEXT): TEXT;
  END;
END Echo.
```

The `Echo` interface defines what Nelson calls a *pure* object type `Echo.T`, which means that no data fields are declared, and no implementations are provided for its methods. Such types are often called abstract types elsewhere. It is a design decision to restrict a client's view of a potentially remote object to such an abstract type.

A program becomes an echo server by supplying an implementation for the type `Echo.T`, and presenting an instance of its implementation to a supporting runtime system:

```
MODULE Server EXPORTS Main;
IMPORT Echo, NetObj;

PROCEDURE Say (self: Echo.T; txt: TEXT): TEXT =
  BEGIN RETURN txt END Say;

BEGIN
  NetObj.Export (NEW (Echo.T, say := Say), "echo 1");
  LOOP END
END Server.
```

In the module Server, the call to NEW creates an instance of a subtype of Echo.T which provides an implementation override for the say() method. This object is then explicitly made available to other programs—we imagine for now that NetObj manages a global textual name space in which the new server object is bound to the string "echo 1". Modula-3 programs terminate when their main bodies complete; this is indefinitely postponed by the final loop.

A client program obtains a binding to the newly available service by presenting the appropriate name to the runtime:

```
MODULE Client EXPORTS Main;
IMPORT Echo, NetObj, Text; EXCEPTION FatalError;

VAR e: Echo.T := NetObj.Import ("echo 1");
BEGIN
  IF NOT Text.Equal (e.say ("Hello, world"), "Hello, world")
    THEN RAISE FatalError END
END Client.
```

In the module Client, this binding takes the form of the *surrogate* object e which is created as result of the call of NetObj.Import. The allocated type of e is a subtype of Echo.T. Its implementation of say() performs marshalling and makes an RPC to the server to which it is bound.

The Modula-3 type system and object invocation have been used here to provide some distribution transparency. The specifications of local and remote services look the same, inasmuch as they are both expressed by the type Echo.T. The remote method call mechanism hides the location of the server object and the work of communicating with it. Such transparency is untenable in the long run: remote calls have different semantics because they may fail. (Our response to this will be to weaken the specification of Echo.T, by adding exceptions to its methods' RAISES clauses, until it *can* be honoured by an RPC implementation.)

Because the client-side stubs and server-side implementation are embodied in two subtypes of the object type Echo.T, they can coexist: a single program may be both a client and a server of the echo service. RPC systems which represent services as *module interfaces* have more difficulty achieving this.

3.1.2 Implicit Export

Network objects' methods can accept arguments and return results which are themselves network objects. These are passed by reference.

To illustrate this, consider how the notional global textual name space managed by `NetObj` above might be implemented, for the special case of `Echo.T` objects, by a service of type `EchoRegistry.T`. This is intended as an illustrative example, not as a serious design for a service trading facility!

```
INTERFACE EchoRegistry;

IMPORT Echo;

TYPE
  T = OBJECT METHODS
    register (svr: Echo.T; name: TEXT);
    lookup   (name: TEXT): Echo.T;
  END;

END EchoRegistry.
```

Assuming an echo server has somehow obtained a `registry: EchoRegistry.T`, it can publish its server object `es` with the invocation `registry.register (es, "echo 1")`. A client program using the same registry can then bind to the server object with the invocation `ec: Echo.T := registry.lookup ("echo 1")`.

The implementation of an `EchoRegistry.T` server looks exactly the same as if it were dealing entirely with local objects. It would probably be a simple wrapper around a table mapping `TEXT`s to `Echo.T`s. For this to work, the RPC runtime at the `EchoRegistry.T` server must, if necessary, create a surrogate for the `svr` argument of `register()`. The `svr` object must be implicitly exported from the echo server and implicitly imported into the registry server, without having been explicitly given a name by the programmer. The result of `lookup()` must be dealt with similarly. However, in this case it is likely that the returned value at the registry server is already a surrogate. The receiving client of the registry service must then obtain another surrogate, bound to the same echo server as the surrogate held at the registry.

One of the strengths of the network object proposal over traditional RPC systems is that it allows programmers to reconfigure a running distributed system simply by passing what look like ordinary object references. Underlying mechanisms hide the necessary remote binding activity from the programmer. With this flexibility of dynamic configuration comes the responsibility of managing the resources denoted by the references, just as with pointers in a single address space. Passing references across interfaces within a single program leads to the use of automatically garbage-collected heaps. Passing network object references between programs will lead to a requirement for distributed garbage collection. Chapter 5 will investigate how this can be automated.

3.2 Network Objects over ANSA

The Modula-3 network objects proposal presents remote services to programmers as ordinary language-level objects. The proposal was particularly appealing to the author because it closely resembled an unimplemented design sketch presented in his Diploma dissertation [Evers 89]. The transparency of network objects allows the same mechanisms to be used in both the single-program and distributed cases: object types to define services, objects for service instances, threads for concurrency, and exceptions to deal with failure. The proposal can be implemented by a fairly conventional stub generation technique. Code for surrogates and server stubs would be produced by an automatic stub generator, whose input would be a network object type declaration in Modula-3. The drawback of the proposal (apart from the fact that no implementation existed at the time of the work reported here) is that it only allows remote binding and communication between Modula-3 programs, rather than the components of a wider heterogeneous system.

The ANSA testbench, by contrast, is intended to accommodate heterogeneity. It defines the notions of service and invocation *independently* of any programming language, operating system or communication facility. It also provides a sophisticated trading service. These features supply a rich infrastructure on which to construct heterogeneous distributed systems.

To shield the clients and the server of an interface from these details of each others' implementations, the testbench requires that remotely accessible service interfaces are specified in IDL (section 2.6). IDL's data types and the defined syntax of their external representation provide a *lingua franca* for communication between disparate objects. Stubs are automatically generated in the appropriate programming language. The input to the stub generator is a language-independent IDL interface.

These features are currently implemented only for the C programming language. C does not support abstract types, concurrency or exceptions. The testbench therefore supplies these features in a runtime library. It uses embedded DPL (section 2.6.3) to make interface references first class citizens in the hybrid C/DPL language. Interfaces themselves have no supporting syntax: as we mentioned in section 2.6.3, the programmer must unpick an interface attributes pointer in server routines.

Embedding DPL in Modula-3 would be awkward and confusing. There would be two sorts of "object": ordinary Modula-3 objects, and testbench interfaces. Each would have its own invocation syntax and type system. In the light of the network objects proposal, this distinction seems unnecessary. Therefore our design identifies network objects with ANSA service interfaces, and surrogates with interface references. This gives us a uniform invocation syntax, and (by using exceptions) semantics.

What about types? In order to hide heterogeneity, we want to specify remotely accessible services (network objects) in IDL, not Modula-3. However, we must obtain Modula-3 abstract object type definitions from somewhere; otherwise it will be impossible to provide the transparency of the network objects proposal. The solution is straightforward: since we have identified interfaces with network objects, we must also identify interface specifications (in IDL) with network object specifications (abstract object types in Modula-3). We must devise a mapping from IDL specifications to Modula-3 type declarations. Once such a mapping is defined, the actual Modula-3 types can be generated automatically from the IDL during stub generation. As in Nelson's proposal, this also produces code to implement surrogates, server stubs and call-by-reference for remote network objects.

Table 3.1 summarises our basic design. Since, in ANSA, it is only possible to refer to other objects indirectly (by naming *interfaces*), we do not need to choose any language-level structure to correspond to an ANSA object. In our system, the boundaries of an object in the ANSA sense are up to the programmer to decide.

Before we specify our mapping from IDL to Modula-3 in more detail, we will expand slightly on the properties of our design from the point of view of a programmer using our system.

ANSA	Modula-3
Interface Type —as defined by some IDL	Abstract `OBJECT` type; abstract types define only method signatures, not state
Interface Instance —unit of service as provided by some ANSA object	Server-side concrete instance of abstract type above
InterfaceRef —a binding as passed to a client	Client-side surrogate for concrete server object above
Object —unit of mobility and encapsulation of state	No direct analogue in Modula-3; a similar rôle may be played by an object instance connected to one or more of the server objects above, or by a complete module, or even a program
No direct analogue in ANSA	`INTERFACE` —a scoping mechanism for type-safe separate compilation of modules

Table 3.1: Dictionary of corresponding ANSA and Modula-3 terms.

3.2.1 Remote Interfaces

We have chosen to regard our network objects as representations at the language level of ANSA interfaces. Thus an IDL interface specification

```
Echo: INTERFACE =
BEGIN
  say: OPERATION [ txt : STRING ]
       RETURNS   [ STRING ];
END.
```

defines an abstract network object type `Echo.T` just like the one in section 3.1.1. Similarly, Modula-3 servers and clients of this interface work with server and surrogate subtypes of `Echo.T`, as before.

This choice restricts Modula-3 programmers. They can only provide or be clients of services which can be defined in IDL, whose type system is less rich in some respects than that of Modula-3. We make this choice because we wish to interwork with the existing ANSA testbench. ANSA relies on a common service specification model to hide the differences between heterogeneous implementations.

In any RPC system, increasing transparency brings corresponding costs in performance and implementation complexity. The best tradeoff depends on the weight given to the benefits of each transparency. Since we give a high weight to the ability to interwork with a heterogeneous world, we can accept the restrictions of IDL, whose facilities represent a fair balance of convenience and cost in such an environment. In practice restrictions such as the lack of recursive types are not too irksome; some could easily be removed (section 3.3.2), though for compatibility, we have not attempted this.

3.2.2 Binding

In the testbench, references to ANSA service interfaces may be passed to remote programs (section 2.6.1) in invocations. In our design, concrete server objects are identified with ANSA interface instances, and are named and located by the same mechanism. The IDL

interface specification

```
EchoRegistry: INTERFACE =
NEEDS Echo;
BEGIN
  register: OPERATION [  svr: INTERFACEREF OFTYPE Echo;
                        name: STRING ]
              RETURNS [];

  lookup:   OPERATION [ name: STRING ]
              RETURNS [ INTERFACEREF OFTYPE Echo ]
END.
```

defines a type almost identical to `EchoRegistry.T` in section 3.1.2. It can be implemented with a Modula-3 object type and used just as explained there. The `svr` argument of `register` and the result of `lookup` are both network objects. Their types are surrogate or server subtypes of the `Echo.T` type generated from the `Echo` IDL interface. On the wire, network object references are transmitted as IDL interface references.

3.2.3 Trading

We have seen how the network object and ANSA binding models can be made consistent. We must now deal with the presentation of the testbench trader (section 2.6.2) to the Modula-3 programmer. The trader exports well-known ANSA interfaces, so it is straightforward for every Modula-3 program to contain readily accessible surrogates for these interfaces. The interfaces themselves can be located by the usual bootstrap mechanisms.

The trader's offer registration and lookup operations deal in values of the generic interface reference type, appearing to the Modula-3 programmer as methods essentially of the form

```
register (serviceTypeName: TEXT; ref: AnsaNetObj.T; .. ) .. ;
lookup   (serviceTypeName: TEXT; .. ): AnsaNetObj.T;
```

where `AnsaNetObj.T` is defined in the libraries of our system as the common supertype of all ANSA network objects. It is up to the programmer to ensure that the dynamic types of the `ref` argument to `register` and the result of `lookup` agree with the static types

denoted by the `serviceTypeName` arguments of these operations. We will have more to say about this in Chapter 4.

To present a slightly simpler interface to the application programmer, and to provide slightly stronger static type checking, we also associate explicit `Import` and `Export` procedures with the Modula-3 translation of each IDL interface, whose network object arguments and results have the appropriate static type. Using the interface name as the service type, these procedures supply appropriate (overridable) defaults and invoke the trader's lookup and registration operations, respectively.

For example, to export an instance of the IDL `Echo` service in the context `"/m3/test"`, a Modula-3 server program uses

```
EchoSRPC.Export (NEW (Echo.T, say := Say), context := "/m3/test");
```

and a Modula-3 client program might bind to such an instance with

```
e := EchoCRPC.Import (context := "/m3/test");
```

Programmers can also supply a property list on export and constraints on import. They must still register service types with the trader separately, through the interfaces mentioned in section 2.6.2. We will discuss the interaction of the trader's type management facilities with our network objects type system in section 4.6.

3.3 Interface Translation

Having decided on the view of network objects presented in the previous section, we now examine in more detail how an IDL service definition is represented in Modula-3. To do this, we must specify translations for IDL's base types, type constructors, operations, exceptions and interfaces.

To provide an overview of this mapping, figures 3.1 and 3.2 give an example of a complete IDL interface specification and its Modula-3 translation. This interface is for a much simplified version of the Active Badge [Want 90] service available in the Laboratory. Workers in the Laboratory can wear small badges which periodically emit an identifying infra-red

```
SimpleBadge: INTERFACE =
NEEDS Callback;
BEGIN
  BadgeID  : TYPE = CARDINAL;
  Sighting : TYPE = RECORD [ loc: STRING, id: BadgeID, qlty: CARDINAL ];
  Sightings: TYPE = SEQUENCE OF Sighting;

  queryBadge : OPERATION [ id: BadgeID ] RETURNS [ Sightings ];
  trackBadge : OPERATION [ id: BadgeID; cb: CallbackRef ] RETURNS [];
END.
```

Figure 3.1: An IDL interface.

signal which is sensed by a network of detectors around the buildings. Information about badge sightings (time, location *etc.*) is collected by an existing testbench service. The simplified interface we present allows a client to obtain a list of recent sightings for a given badge, or arrange to be called back on one of its own interfaces whenever a badge is sighted.

3.3.1 Base Types

Almost all IDL base types have natural exact equivalents in Modula-3. We define these in a standard Modula-3 `INTERFACE` called `Ansa`. Some IDL ordinal types have long and short forms with different ranges: these can be represented using Modula-3 subranges and the `BITS n FOR` construction.

IDL `CARDINAL`s and `STRING`s are slightly trickier. IDL `CARDINAL`s are 32-bit unsigned integers, but Modula-3's `CARDINAL` type is just the non-negative subrange of `INTEGER`, which has only 31 significant bits on a typical 32-bit implementation. Fortunately, Modula-3 defines a required `Word` interface which provides all the usual operations by interpreting

```
(* Generated by stubm3 on Tue Jul 27 19:27:28 1993 *)

INTERFACE SimpleBadge;
IMPORT AnsaNetObj, Ansa, Management, BaseTypes;
IMPORT Callback;

TYPE
  T = Management.T OBJECT METHODS
    queryBadge (id: BadgeID): Sightings      RAISES {Ansa.Failure};
    trackBadge (id: BadgeID; cb: Callback.T) RAISES {Ansa.Failure};
  END;

  BadgeID   = Ansa.Cardinal;
  Sighting  = RECORD loc: TEXT; id: BadgeID; qlty: Ansa.Cardinal; END;
  Sightings = REF ARRAY OF Sighting;
END SimpleBadge.
```

Figure 3.2: The Modula-3 translation of figure 3.1.

INTEGERs as unsigned quantities, so an IDL CARDINAL becomes a Modula-3 Ansa.Cardinal, which is identified with a Word.T with no loss of information.

IDL has two "character" types: 7-bit CHARs and 8-bit OCTETs. Modula-3's CHAR type is an enumeration with at least 256 elements, and can thus contain both. IDL STRINGs are sequences of 7-bit characters, but are translated to Modula-3 8-bit TEXTs for convenience. This requires the programmer to be aware of the possible loss of information when a TEXT or a CHAR is marshalled. In practice this has caused no trouble; more inconvenient has been the tendency of some existing IDL interfaces to use fixed-length arrays of characters with optional NUL terminators instead of STRINGs for text. This sits more easily with C's programming style than with Modula-3.

IDL	Modula-3
`enum : TYPE = { a , b };`	`TYPE enum = {a, b};`
`arr : TYPE = ARRAY n OF elt;`	`TYPE arr = ARRAY [0..n-1] OF elt;`
`seq : TYPE = SEQUENCE OF elt;`	`TYPE seq = REF ARRAY OF elt;`
`rec : TYPE = RECORD [f: ft, ..]`	`TYPE rec = RECORD f: ft; .. END;`
`iref : INTERFACEREF OFTYPE Foo;`	`TYPE iref = Foo.T;`
`chce : TYPE = CHOICE enum OF { a => at, b => bt }`	`TYPE chce = BRANDED OBJECT END; chce_a = chce BRANDED OBJECT f: at END; chce_b = chce BRANDED OBJECT f: bt END;`

Table 3.2: IDL and Modula-3 type constructors.

3.3.2 Constructed Types

IDL's type constructors all have equivalents in Modula-3, as displayed in Table 3.2. Note that in Modula-3, sequences and choices are heap allocated. Since Modula-3 lacks the usual discriminated unions, we use a branded object type to represent a choice. Subtypes of this basic type represent the union's summands, and can be discriminated by the Modula-3 TYPECASE statement (section 2.7.4).

Unlike Modula-3, IDL has neither recursive types, necessary to define lists and trees directly, nor pointers for directed acyclic and cyclic graphs. However, it does have variable-length sequences. Programmers must themselves convert tree- or graph-structured data to and from values of marshallable types. In practice, on the rare occasions when we have needed to pass such data structures across a network object interface, the corresponding application data structures have contained additional information that should not be transmitted across the abstraction boundary. Thus, gathering the information to be transmitted has imposed no extra burden on the application programmer.

Having said this, it should be convenient to construct values of the appropriate marshallable types. In this respect, our representation of IDL sequences as Modula-3 open arrays is not ideal, primarily because their sizes must be known when they are allocated. Although Modula-3 does not have built-in array types with the ability to grow dynamically (such as CLU's), we could have supplied our own. Indexing and iteration over these would be only slightly more expensive than for standard open arrays. The main reason we did not introduce dynamic arrays is that their types are awkward to specify in Modula-3: a generic interface [Nelson 91b, p. 45] would be cumbersome to use since it must be explicitly instantiated and given a name for every argument type; dynamic arrays of `REFANY` avoid this problem, but require runtime type checks.

If IDL allowed recursive types, one could define a list by using a recursive boolean choice. The corresponding objects would be reasonably convenient to construct in Modula-3, but would require careful marshalling to avoid overrunning small thread stacks.

IDL interface reference types like `CallbackRef` in figure 3.1 become network object types like `Callback.T` in figure 3.2. Here, the callback reference is being transmitted in the expectation that invocations will be made on it by the receiver, as its name suggests. In cases where all that is required is to pass an identifier for some local data rather than the data itself, the data can be associated with a network object. Then a reference to this object may be used as a token for the data—unlike CCLU RPC [Hamilton 84, p. 46], no special mechanism is required for this. This lightweight service creation and binding is a key characteristic of the network objects style.

3.3.3 Operations and Exceptions

Operations in an IDL interface `Foo` map to methods of a network object type `Foo.T`.

IDL operations may return multiple results, but Modula-3 procedures cannot. However,

Modula-3 procedures *can* return structured values, including records. Thus if an IDL operation returns multiple results, an appropriate record type is automatically generated for the return type of its Modula-3 translation. This design is a mistake: the corresponding idiom in Modula-3 is to assign multiple results to `VAR (*out*)` parameters. This avoids allocating extra space for a result record from which results will simply be copied to their eventual destination.

If an operation takes a large argument, such as a fixed-size array or a record, the corresponding Modula-3 argument is given the mode `READONLY` to avoid a redundant copy.

IDL's synchronous (interrogation) and asynchronous (announcement) operations cannot be syntactically distinguished in Modula-3: all local method calls are synchronous. However, their different semantics in the remote case *are* implemented in client stubs. To reflect this, during stub generation we append the comment `(* ANNOUNCEMENT *)` to network object method signatures where necessary .

We give each operation method a `RAISES {Ansa.Failure}` clause in its signature; the `Ansa.Failure` exception takes a single argument enumerating possible lower-level failures, such as RPC timeouts. At some point, it is likely that IDL will allow user-defined exceptions (or "named terminations" in ANSA terminology) to be declared and raised by operations. When that happens, the Modula-3 translation should be straightforward.

3.3.4 Interface Dependencies

Under our mapping, a complete IDL interface becomes a complete Modula-3 interface defining some ordinary types and a network object type. After a `NEEDS Bar` directive, an IDL interface has access to types in the interface `Bar` under the unqualified names bound to them in `Bar`. This usage translates to `IMPORT Bar` in Modula-3, which gives access to `Bar`'s types under names qualified by `Bar`. This is what has happened to the `Callback` interface mentioned in figure 3.1.

IDL interfaces can be declared `COMPATIBLE WITH` others. Since this bears on our later discussion of subtyping in Chapter 4, we say no more about it here.

3.4 Implementation

The previous two sections have described the design of our network objects system from the programmer's point of view. The rest of this chapter presents our implementation of this design.

In outline, the system consists of a stub compiler and a runtime library (figure 3.3). The stub compiler maps service definitions written in ANSA IDL into Modula-3 network object interfaces and stub implementations. The runtime library manages network object binding and provides an ANSA-compatible RPC transport.

In an attempt to reduce implementation effort, we used the existing C ANSA testbench capsule library over UNIX for much of the transport. With some customisation, this can simply be linked into a Modula-3 program. This strategy increased our confidence in our ability to interwork with the installed testbench environment, but as we will discuss in sections 3.4.6 and 6.3, interference between the Modula-3 and capsule runtimes seriously compromised performance and maintainability.

3.4.1 Stub Compiler

The stub compiler, `stubm3`, takes an IDL interface as input. It produces its Modula-3 translation and client-side and server-side stub modules. It is internally organised around an abstract syntax tree, or AST, representing the input interface.

The front end of the compiler reads the main IDL input and the interfaces on which it depends. It consists of a `yacc` parser whose reduce actions are written in Modula-3. The parser's action procedures create AST nodes and maintain the symbol tables for interfaces and types. Because IDL enforces declaration before use and allows no recursive types, it is simple to use the symbol tables to perform binding during the parse. Because Modula-3's copying garbage collector might move AST nodes during the parse, the `yacc` parser stack holds integer node identifiers rather than references into the Modula-3 traced heap. In retrospect, simply inhibiting garbage collection during the parse would almost certainly have sufficed to avoid this. Later versions of SRC Modula-3 have added means to lock designated heap nodes in place when it is unacceptable to stop the garbage collector.

The compiler contains a generic tree walker which the subsequent passes share. Given a closure object and an initial AST node, the tree walker visits its descendants in depth-first

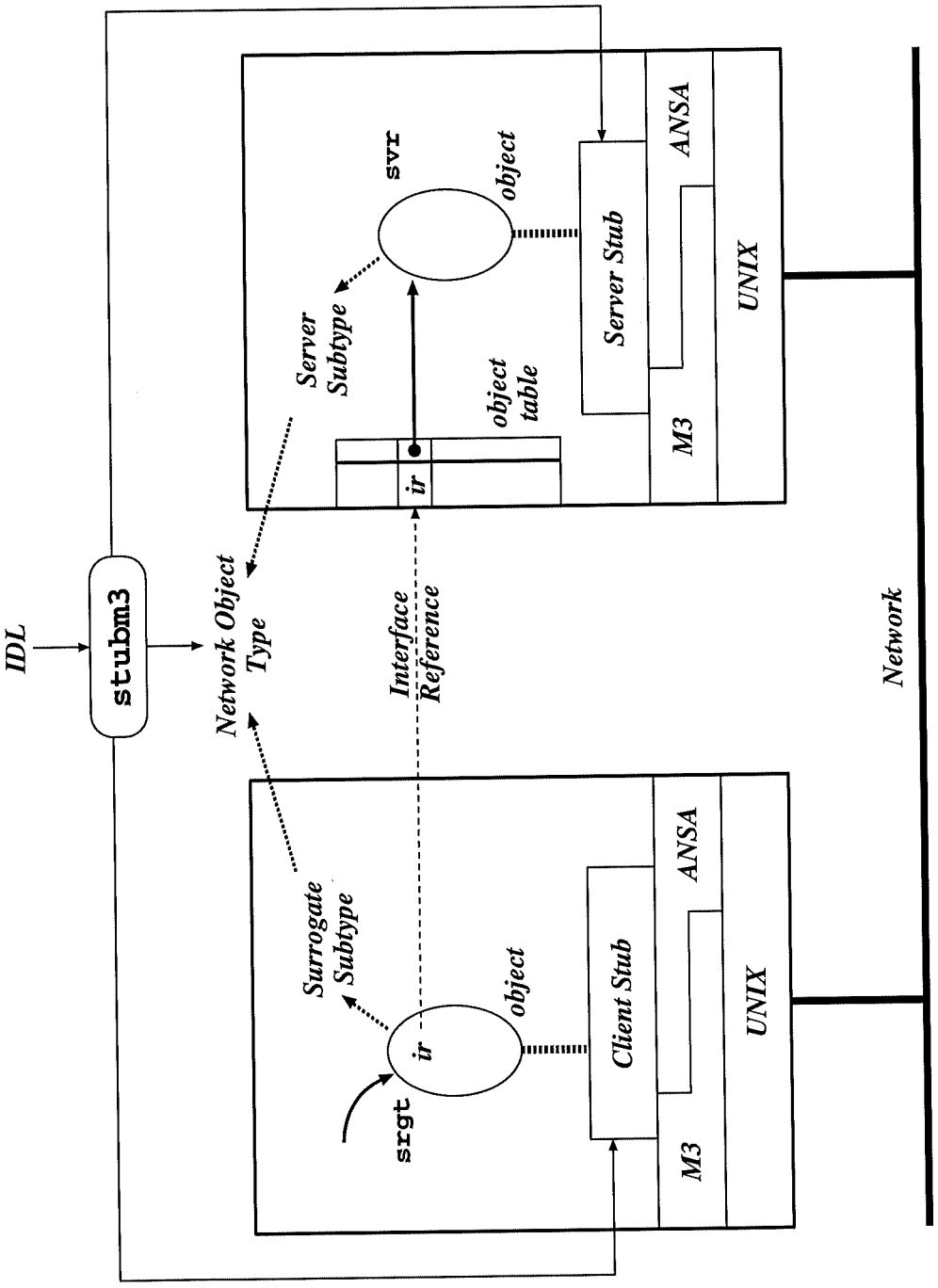


Figure 3.3: Network objects over ANSA: implementation overview.

order. As each node is entered and exited, it is passed to a callback method on the closure object. The callback can prune or terminate the walk by raising appropriate exceptions.

Each output file from the compiler is generated by its own back end. The back ends register themselves with the top level control module, which interprets command line options and invokes the required compiler passes. To produce output, a back end passes a closure object with appropriate methods to the generic tree-walker.

Before any back ends proper are executed, the IDL AST is decorated with information specific to its Modula-3 translation, such as the additional types which must be invented for multiple operation results and choice variants. Much of this work could probably be done on the fly during parsing; it is done in a separate pass in order to preserve the independence of the front end from the target language.

The back ends which generate the stub modules make heavy use of a shared Modula-3 type declaration generator for output such as stub procedure signatures, temporary variables used in marshalling and the target types for enumerations and choices. These back ends also share a module which generates the appropriate marshalling or unmarshalling code for an IDL type from the corresponding AST node. Most context-dependent output is handled by the back ends themselves.

Back ends are able both to contribute public fields and add their own private decorations (such as unparsing methods) to AST node types without interfering with each other. This is achieved by using Modula-3's partially opaque type mechanism and successive revelations [Jordan 90], as illustrated in figure 3.4. The full structure of each AST node type and the visibilities of its fields are only fixed at link time. The runtime cost of accessing a node attribute is comparable to that of a record field access, rather than the table lookup which would be required by a property list mechanism.

In hindsight, the stub compiler design presented here suffers from an excess of abstraction. The generic facilities such as the tree walker and the successive revelation of AST node attributes were motivated by the desire to make it easy to reuse the front end for other target languages, and to add or remove whole back ends as the need arose. In practice, however, these facilities serve mainly to obscure the simple underlying structure of the compiler. The tree walker is no easier to use than straightforward recursive descent procedures; on occasion, its genericity is actually obstructive. Whilst back ends have been added, removed and split during the compiler's evolution, these have largely used their access to the AST node types purely to add specific unparsing methods—and these methods are themselves only a by-product of the genericity of the tree walker!

```
INTERFACE AST;
  TYPE Node <: NodeP;
    NodeP = OBJECT (* public attributes of this node type *) END;
```

```
INTERFACE BackEnd1;
  IMPORT AST;
  REVEAL AST.Node <: Node;
  TYPE Node <: NodeP;
    NodeP = AST.NodeP OBJECT
      (* extra public attributes *)
    END;

                ...

INTERFACE BackEndi;
  IMPORT AST, BackEnd(i-1);
  REVEAL AST.Node <: Node;
  TYPE   Node <:
           BackEnd(i-1).Node;

                ...
```

```
MODULE BackEnd1;
  IMPORT AST;

  REVEAL Node = NodeP
    BRANDED OBJECT
      (* private attributes *)
    END;

                ...

MODULE BackEndi;
  IMPORT BackEnd(i-1);
  REVEAL
    Node = BackEnd(i-1).Node
      BRANDED OBJECT
        (* private attributes *)
      END;

                ...
```

```
INTERFACE ASTAll; IMPORT AST, BackEndN;
  REVEAL AST.Node = BackEndN.Node BRANDED OBJECT END;
```

Figure 3.4: Successive revelation of AST node attributes in `stubm3`.

3.4.2 Stubs

The function of the stub modules generated by `stubm3` is the traditional one: to act as local proxies for potentially remote programs. Their implementations also follow the traditional pattern. The use of network objects for bindings and service implementations simplifies both the stub modules themselves and the interfaces they present to higher levels in comparison with systems like Sun RPC or the ANSA testbench which are written in C. Modula-3's automatic garbage collection and support for threads and exceptions also contribute significantly to this simplification.

Client- and server-side stubs for the interface `Foo` are implemented by separate modules named `FooCRPC` and `FooSRPC`. Whilst this means that programs need not be linked with unnecessary stub code, it is debatable whether the resulting size savings are worth the additional burden on programmers, who must keep track of exactly which stubs are required. This task is partially automated by makefile macros. In any case, the linker is in principle perfectly capable of removing unused code from executables, and the decision to generate marshalling code inline (see below) had far more impact on executable size.

A client stub module defines a *surrogate subtype* of its network object type. This overrides the methods of the abstract service type with stub procedures. When invoked, an operation's stub procedure acquires a call control block and a buffer from the runtime in which to place call data, commencing with the target interface of the current surrogate object and an identifier for the operation. The underlying runtime has the opportunity to perform some session setup at this stage. Having marshalled the call arguments into the buffer, the stub procedure passes it with the control block to the blocking RPC transport or the asynchronous one-shot mechanism, as appropriate. If this succeeds without itself raising an exception (such as a timeout), any RPC results can be unmarshalled from the buffer, commencing with a status code which is converted into the corresponding `Ansa.Failure` exception if necessary. The buffer is returned to the free pool by a `TRY-FINALLY` statement which protects the whole stub procedure.

On the server side of an invocation, the runtime makes an upcall on a worker thread into an interface-type-specific dispatch procedure in a server stub module. We will deal with the registration of dispatchers in section 4.8 below. Having located the target server object, the dispatcher calls the server stub procedure for the operation in hand. The stub unmarshals the call arguments, makes the appropriate local method invocation on the target object, and marshals the results with a status code indicating a successful call. If there were a way for IDL operations to specify application-defined exceptions, the corresponding Modula-3 exceptions would be caught and marshalled at this point. Generic `Ansa.Failures` are

caught and marshalled by a `TRY-EXCEPT` statement in the dispatcher.

Marshalling (that is, the conversion of Modula-3 values to and from the IDL presentation syntax) is performed by inline code in the stub procedures generated by `stubm3`. Attempts to transmit values without IDL equivalents crash the sending program, if they cannot be detected by compile time type checks. By contrast, syntax errors in the IDL value stream at the receiver cause an `Ansa.Failure` exception.

Marshalling IDL base types is unproblematic, though it may involve some byte-swapping. IDL presentation syntax defines a canonical byte ordering rather than, for instance, allowing transmission in sender-native order. Enumerations are marshalled as integers; Modula-3's `VAL` and `ORD` inject and project these to and from the enumeration type, and a range check is required at the receiver. Network object references are marshalled as the corresponding interface references; this will be dealt with more fully in section 3.4.3.

The constructed types are similarly straightforward to marshal, IDL's type constructors having presumably been selected with this in mind. The only slightly unusual case is a choice. At the sender, a `TYPECASE` discriminates the variants on the basis of their dynamic types (section 3.3.2). It cannot marshal spurious subtypes of the parent choice object type, and crashes instead. At the receiver, the branches of an ordinary `CASE` statement either allocate an instance of the appropriate subtype or raise an exception. Stubs for operations with multiple results unmarshal them into a record.

Constructed types are currently marshalled entirely by inline code rather than recursive descent procedures. This choice is a mistake. It was made mainly in order to avoid having to generate interfaces and declarations for the marshalling procedures. Inline marshalling is feasible because true recursive types (such as lists and trees) cannot be defined directly in IDL. To generate inline code, `stubm3` maintains a slave stack in the closure object used to walk the AST for the type to be marshalled. In retrospect, recursive descent marshalling procedures would have been slightly simpler to generate, more compact and probably almost as efficient at runtime. Perhaps more importantly, they would open the way to extensions of IDL with limited recursive types (section 3.3.2).

If IDL had reference types which allowed arbitrary graph structures to be transmitted, they could be marshalled from the Modula-3 traced heap by a technique used in the SRC Modula-3 system, which we now explain:

Because of its copying garbage collector, the SRC Modula-3 runtime has all the information required to traverse arbitrary rooted graph structures in the traced heap. Dynamic

typecode information (section 2.7.4) in a heap node's header word is used to locate a compiler-generated *map procedure* for the node's type. The map procedure applies a callback procedure to each reference field in the heap node. The callback then recursively calls the descendent nodes' map procedures.

SRC Modula-3 uses these facilities to implement its *pickles* facility [Birrell 88], which defines a byte stream presentation syntax for heap structures and provides procedures to convert between the two. Loops and sharing are preserved in the usual way by maintaining a table of visited node identifiers during marshalling; the garbage collector uses forwarding pointers and mark bits in the node header word for a similar purpose.

Programmers could, if absolutely necessary, use this facility to transmit arbitrary picklable data between Modula-3 programs by pickling into a sequence of IDL octets. The extra cost of doing this, including redundant copying, then provides a healthy incentive for the programmer to reconsider the design of a remote interface that requires the transmission of such data!

3.4.3 Object Table

Client and server stubs require some runtime support to function as proxies for remote programs. We first discuss support for the network object binding model, then turn our attention to issues of transport and concurrency.

The network object binding model discussed in section 3.1.2 is straightforward to implement once programs can name potentially remote objects unambiguously. Each program simply maintains a table of object names known to it. For each name, this *object table* records whether the corresponding network object is local to this program (its *owner*), or is in another address space. The table holds a reference either to the local *concrete* object or the unique local *surrogate* for the remote object, as appropriate. Object tables will, of course, play a central rôle in distributed garbage collection, and this is simplified if each program holds at most one surrogate for a given concrete object. Most of what is interesting about the management of the object table is concerned with distributed garbage collection, and will therefore be discussed in Chapter 5. Here we will just sketch the mechanism.

How are object names to be implemented? Under the assumption (valid in all the versions of the ANSA testbench used in our implementations) that an interface reference contains a unique identifier for its referent, it can be used as the key for the object table. To represent

the inverse mapping (required when marshalling a network object), we bind an interface reference to every concrete and surrogate network object as part of some state inherited from the common supertype `AnsaNetObj.T`. Modula-3's partial revelation mechanism is used to restrict the visibility of this state to the module which maintains the object table.

To unmarshal (implicitly import) an incoming network object reference, the runtime looks up the transmitted name in its object table. If present, the table entry points to the correct local object, whether that is concrete or a surrogate. Otherwise, the incoming name refers to an unknown remote object, and the runtime must create a new surrogate. We will discuss how the runtime knows which type of surrogate to create in section 4.7.

To marshal (implicitly export) an outgoing network object reference, the runtime first inspects the local object. If it is a surrogate, the name of the corresponding remote object is already known. Otherwise, it is a concrete object. Either it has never previously been exported, in which case a fresh name must be generated and entered in the object table, or the object has already been assigned a name.

A property of the mechanism described is that when a network object reference is passed between programs and returns to its owner, its imported translation is a direct reference to the local concrete object. Invocations on this reference automatically bypass the RPC transport. The difference in performance between the local and remote mechanisms justifies varying the implementation of invocation in this way; as in Emerald [Black 87], the fact that network objects are abstractly typed allows us to do so without concerning the programmer.

3.4.4 Transport

Given a buffer of call data and the name of the callee network object, the job of the caller's RPC transport is to locate its peer at the callee and engage in a request-response exchange, with the required reliability, over some underlying message passing service. In order to interwork with the existing ANSA testbench, we must use the ANSA remote execution protocol REX for this purpose.

REX defines both *calls* and *casts*. Based on the standard Birrell-Nelson design [Birrell 84], calls use timeout-triggered retries, call identifiers, sequence numbers and implicit acknowledgements to achieve exactly-once (or at-most-once) semantics in the absence (or presence) of total communication failure. Casts are asynchronous, and are neither retransmitted nor acknowledged. They are, however, sequenced—REX discards casts received out of order.

The reliability of casts is determined by the underlying message passing service. REX performs fragmentation and reassembly of call data buffers which are larger than can be accommodated by the message passing service. Calls and casts are suitable to implement IDL's interrogation and announcement operations, respectively.

It would be perfectly possible to implement a REX transport in Modula-3, but we felt that producing a correct implementation which would interwork with the existing ANSA testbench would take us too far afield from the main line of our research. Instead, we used the existing REX implementation in the capsule library, which runs over same-machine UNIX IPC, unreliable internetworking protocols such as UDP and MSNL [McAuley 90], and more reliable flow-controlled byte stream protocols such as TCP.

Rather than relying exclusively on some mechanism to *resolve* interface identifiers (network object names) to transport addresses, the ANSA testbench defines the structure of interface references to *include* a collection of addresses, tagged with identifiers for the transports which can interpret them. We suppress the details of this mechanism for now because it has changed between major testbench versions; the point is that it suffices for the client stubs generated by `stubm3` to hand a surrogate's interface reference to the capsule REX implementation, which will then choose an appropriate message passing service from the addresses contained in the interface reference.

3.4.5 Concurrency

Classical remote procedure call (section 2.4.2) arises from a model of concurrency very similar to that of Modula-3 (section 2.7.3). A server address space executes multiple incoming calls concurrently, each on its own worker thread, though the number of these is usually subject to some load management policy.

The existing UNIX testbench capsule library therefore includes a lightweight threads package. This schedules potentially concurrent activities called *threads* onto a possibly smaller number of virtual processors called *tasks* which provide the resources necessary for a thread to make progress: a stack and register save area. Capsule tasks play the rôle of the pool of worker threads in a classical RPC implementation.

For performance reasons, the capsule REX implementation is quite tightly coupled to this package's scheduler, synchronisation and I/O primitives. This makes it difficult to port the REX implementation alone. Thus it was necessary to integrate the capsule thread package with that in the Modula-3 runtime. It is clear that they cannot coexist unmodified: both

must intercept some UNIX system calls to prevent one thread from blocking its entire UNIX process, and both wish to receive timer interrupts. Even if these problems could be solved, we would still wish Modula-3 threads to enter capsule RPC code (and vice versa) directly, rather than queuing a message requesting action by a capsule thread, with all the scheduling overhead that implies.

Fortunately, the capsule library has a system-dependent layer which defines a reasonably clear interface to the task scheduler. It was possible to implement this interface with Modula-3 threads. The latter are preemptively scheduled and support the Modula-3 garbage collector and synchronisation primitives.

Unfortunately, the interface between the capsule task scheduler and the rest of the capsule library contains quite a lot of publicly writable state, both per-task and shared. This state is unencumbered by procedural abstractions which we could intercept. To maintain it, it was necessary to modify the SRC Modula-3 thread implementation to provide hooks to be called back when creating, destroying, and switching between Modula-3 threads.

3.4.6 Discussion

For historical and practical reasons, the runtime implementation we have described above is more than a little overweight. There are places where it contains two or three layers where one would do:

- The testbench capsule library maintains a table which binds interface identifiers to transport level sockets, dispatch procedures and associated state, while our object table binds interface identifiers to concrete or surrogate language-level objects.
- The capsule REX implementation provides a reliable, flow controlled transport over TCP in the UNIX kernel. However, since REX also works over lighter-weight protocols and is necessary for interworking, this cannot really be considered redundant.
- ANSA tasks are implemented (though not multiplexed) over Modula-3 threads, which are in turn implemented (and multiplexed) over UNIX processes.

It is worth discussing this last point in a little more detail. The interactions between the ANSA and Modula-3 implementations of pseudo-concurrency were an endless source of maintenance problems and concurrency bugs, which became acutely noticeable when testing race conditions in the distributed garbage collector.

Many of these bugs arose during accesses of the capsule scheduler's writable state mentioned in the previous section. In one version of the testbench, this state was only ever read or written within the capsule runtime, under a global lock. Thus it sufficed to replace the procedures to acquire and release this lock with versions which temporarily inhibited Modula-3 thread preemption and ensured the validity of the global state while the lock was held. This unpleasant trick meant we could leave the Modula-3 thread scheduler untouched. However, a subsequent version of the testbench quite legitimately relaxed the locking discipline on per-task state, not expecting tasks to be re-scheduled under its feet. Thus we were eventually forced to add hooks to the Modula-3 scheduler. This involved some parallel changes to the protection of the ANSA scheduler's shared state, because the hooks run in a critical section of the Modula-3 scheduler during which normal locking of `MUTEX`es is not available.

The conflicting thread systems affect the performance of our system as well as its structure. An incoming network request has to traverse far too many scheduling operations before being acted on by a user thread [Tennenhouse 89]. Section 6.3 will describe the consequent loss of performance in more detail.

These problems are simple to describe in a few words, but absorbed an inordinate amount of time and effort. Their root cause was the difficulty of separating the capsule REX implementation from the underlying task system. If we had simply written our own Modula-3 REX implementation, using Modula-3 threads, we might have had less trouble. We would only have needed to track changes in the specification of REX, rather than in both the Modula-3 scheduler and the capsule REX implementation. However, it seems likely that producing a correct REX implementation would itself have taken significant effort.

3.5 Summary

This chapter has shown how a modern systems programming language like Modula-3 can be used for object-based distributed programming in a system such as the ANSA testbench. Our design uses the familiar and coherent concepts of the language to express service binding, concurrency, synchronisation, operation invocation and the possibility and handling of failure. It is possible to do this uniformly in both the local and remote cases, even in a heterogeneous environment. We have achieved this by representing potentially remote service interfaces as network objects.

We presented the design and implementation of a stub compiler which maps language-independent IDL service specifications into the corresponding Modula-3 network object type definitions. The compiler also generates client and server stub modules for these types. We showed how the existing testbench capsule runtime library can be adapted and augmented so that it may be linked into Modula-3 programs. This provides interworking via RPC with other testbench capsules.

Thus far we have been concerned with mechanisms which support the use of local objects to hide remote service location, local method call to hide remote operation invocation, and the transmission of network object references as operation arguments or results to hide remote RPC binding. The next chapter will consider the relationship between the local type system of a Modula-3 program and the global service type system supported by the ANSA testbench.

The ambassadors from Norway, my good Lord,
are joyfully return'd

— POLONIUS, Hamlet (Act II, Sc. 2, line 40)

Speak. I am bound to hear.

— HAMLET (Act I, Sc. 5, line 6)

Chapter 4

Types

This chapter discusses what it means to ascribe a type to a value in a distributed system and what such attributions can be used for. We start with abstract service types, and the relations of subtyping and inheritance between abstract types and their implementations. We describe one way of using the local object types of a programming language like Modula-3 to represent a global service type system like that of ANSA, thus extending the results of the previous chapter which showed how IDL's concrete types could be represented. Our design imposes minimal management overhead by using *type fingerprints* to name service types. We then show how this design achieves type-safe binding (i.e., the transmission of references to abstractly typed services) without sacrificing the flexibility required in an open distributed system. Finally, we briefly describe the implementation of the design in our network objects system.

4.1 Abstract Types and Subtypes

An *abstract type* is an interface specification. That is, an abstract type is a collection of operations, each with a name, a signature defining the number and types of its arguments and results, and (in theory) a specification of its semantics. The only way to interact with a value of an abstract type is to invoke one of its operations—"an abstract type is ... defined by the specifications of its operations, instead of by the representation of its data" [Nelson 91b, p. 5]. This condition is especially attractive in a distributed system for values which may be held remotely.

An abstract type AB is a *subtype* of abstract type A (written "$AB <: A$") if and only if it is safe to use a value of type AB in any context which expects a value of type A. That is, AB must be a richer type, having at least every operation of A with possibly weaker preconditions and stronger postconditions. The types of an operation's arguments and results can be regarded respectively as conjuncts of the pre- and postconditions of the operation.

We have been deliberately vague in our definition of an abstract type as to whether the "types" of operation arguments and results are abstract or not. In some systems, all values are abstractly typed. In these systems, the natural definition of subtypes is in terms of a contravariant conformance rule [Cardelli 85, Danforth 88] such as that of Trellis/Owl [Schaffert 86] or Emerald [Raj 91]. Since an operation in AB must have a stronger postcondition than its counterpart in A, its results' types must be subtypes of their counterparts in A. Similarly, since an operation in A must have a stronger precondition than its counterpart in AB, its arguments' types must be subtypes of their counterparts in AB. Thus the subtype relation is defined inductively.

Other systems include concretely-typed values; that is, values whose representation structures are accessible without the mediation of operations. Whilst it is possible to define a subtype relation on concrete types (see, for example, [Cardelli 89]) this is currently rarely done. In the absence of such a scheme, concretely-typed arguments and results must match exactly between operations of abstract subtypes.

A given abstract type admits many implementations, hiding their differences. As well as supporting the software engineering technique of information hiding [Parnas 72], this is useful in a distributed system [Black 87] for reasons including the following:

- new implementations of old service types can be introduced without halting the system or changing existing clients;
- different implementations may be required on different server machines;
- different implementations are appropriate for different patterns of use, such as local and remote services.

A valid implementation of an abstract type is also a valid implementation of any of its supertypes. As well as allowing the classification of types (modelling application concepts) in the object-oriented style, this too is useful in a distributed system:

- services can be extended to provide new functions whilst remaining usable by existing clients, and
- common features (such as generic management interfaces) can be given to a set of services in a common supertype.

4.2 Subtypes in IDL and Modula-3

In the absence of formally specified semantics for operations, more conservative definitions of "subtype" than that of the previous section have to be used in practical programming languages. We now describe how subtypes are embodied in IDL and Modula-3, and how one may be used to represent the other.

As mentioned in section 3.3.4, an IDL interface `AB` can contain a sequence of declarations that it `IS COMPATIBLE WITH` A_i. These declarations include all the type and operation definitions from the A_i in the definition of `AB`, along with any extra definitions from the body of `AB` itself. Thus (the type denoted by) `AB` is a subtype of each of the A_i. There is no way to refine the types of arguments or results of operations from the A_i, so to decide whether two IDL interfaces are subtypes one simply checks whether they are in the reflexive transitive closure of the is-compatible-with relation.

This differential style is the commonest method of specifying abstract types in current object-oriented programming languages. One mentions the *names* of the supertypes (in some environment), then augments the resulting union. Whilst this simplifies subtype checking, it does introduce some complications. Composing pre-existing specifications requires a policy for resolving clashes of operation and type names; existing IDL compilers (including `stubm3`) ignore this issue. Indirecting through the supertype names has the usual effects: changes to a type definition automatically propagate to its subtypes, and the meaning of an interface depends on the environment in which it is compiled. Often this behaviour is useful; sometimes it is not: for instance, changes to a supertype can introduce name clashes into subtype definitions. Having mentioned these caveats, it should be said that they have never arisen in our practical experience of using IDL.

Modula-3's object types support a form of subtyping (section 2.7.1). Abstract types can be defined by object types with no fields, method procedure bindings or overrides. Subtypes can be defined by extending a single parent type with new methods.

As explained in Chapter 3, our system translates IDL interface specifications into abstract

Modula-3 object types. IDL supports multiple subtyping, but Modula-3 does not. So what happens to IDL interfaces with more than one parent type? The answer to this question is a design choice which affects how faithfully the Modula-3 type system can model that of IDL. Suppose the translations of two compatible IDL interfaces `AB` and A_i are object types `AB.T` and A_i`.T`. Then there are two alternatives: `AB.T` and A_i`.T` are either related or unrelated by Modula-3's `<:` hierarchy.

If we insist that `AB.T <:` A_i`.T`, there will be valid IDL interfaces that cannot be translated to Modula-3. Alternatively, we could flatten `AB` to generate a definition for `AB.T` which includes *all* the operations of `AB`, A_i and their ancestors. This would allow a valid translation of all IDL interfaces, but has some drawbacks. A little care would be needed to avoid duplicating stub code unnecessarily, and it would not be possible to implement `AB.T` by inheriting implementations of A_i`.T`. These are minor problems, and it turns out that the second also applies to the `AB.T <:` A_i`.T` case.

More serious is the following difficulty: consider two variables `ab: AB.T` and `a:` A_i`.T` in some Modula-3 program. If `AB.T` and A_i`.T` are unrelated, neither variable can be assigned to the other without breaking the static type rules of Modula-3, despite the fact that the assignment `a := ab` is statically safe from the IDL perspective. It is no great hardship to forgo such assignments within a single program, but when the target of the "assignment" is an argument in the invocation of a network object operation this is a bigger problem: it prevents the subtype relationship from being used to capture common interfaces (the second of the two uses of subtyping mentioned in the previous section).

Since we would not wish to abandon the network object binding model, we would be forced to attribute the common ancestor type `AnsaNetObj.T` to network object arguments in operation signatures. During an invocation, we would perform our own runtime type checks based on the IDL subtype graph and the dynamic type of the argument on the client side. This is unattractive, partly because of the work involved, but mostly because we lose the static checking of the Modula-3 compiler. This is all the more unpleasant because the programmer must consult the IDL interface (rather than the Modula-3 type) of the target operation to discover the correct argument type, thus increasing the possibility of just the sort of confusion that leads to this kind of static type error in the first place.

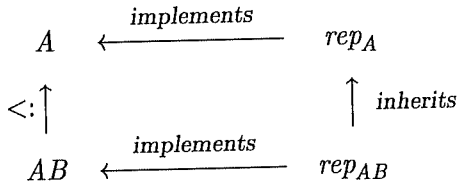
All these problems only arise for interfaces whose ancestors cannot be totally ordered in the is-compatible-with relation. In the ANSA testbench as delivered and used in our experimental environment, there are no such interfaces. Thus it seems tenable in practice to adopt the first policy (`AB.T <: A.T`), and allow `stubm3` to reject interfaces it cannot handle. This is what we have done.

4.3 Inheritance

An implementation of an abstract type defines a procedure for each of its operations. These procedures act on shared state in a *representation* type whose values correspond, via an *abstraction function*, to the possible states of values of the abstract type. The domain of the abstraction function is defined by a *representation invariant* maintained by the operation procedures [Liskov 86a].

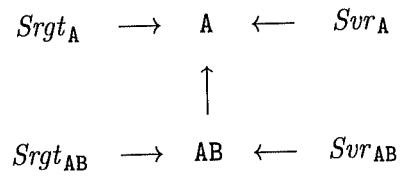
Sometimes it is convenient to define one implementation by describing how it differs from another—typically by adding fields to the representation and adding or replacing operation procedures. The base implementation is then said to be *inherited*. This process has much the same advantages and disadvantages as the differential method for specifying abstract types mentioned above [Liskov 88a].

Given two abstract types $AB <: A$ and an implementation rep_A of A, a natural way to produce an implementation of AB is by inheritance:



Subtyping and inheritance are (in this view) separate concepts: abstract subtypes need not have implementations related by inheritance, and inheriting implementations need not represent subtypes. Unfortunately, many programming languages, including the object-oriented fragment of Modula-3, do not distinguish between the two. Instead, a single relation based on the differential specification mechanism encompasses both.

This can cause problems; one that arises in the context of our network objects system is as follows. Suppose two conforming IDL interfaces give rise to two Modula-3 network object types `A` and `AB`. Then according to the previous section, `AB <: A`. Let $Srgt_{\mathtt{T}}$ and $Svr_{\mathtt{T}}$ be the types of surrogate and server implementations of `T` respectively. Then these six types are related by `<:` thus:



Because a Modula-3 object type can have only a single parent, the client-side surrogate implementation $Srgt_{\mathtt{AB}}$ cannot simply inherit from $Srgt_{\mathtt{A}}$. More seriously, the programmer cannot implement $Svr_{\mathtt{AB}}$ by inheriting from $Svr_{\mathtt{A}}$.

As generated by `stubm3`, $Srgt_{\mathtt{AB}}$ and $Srgt_{\mathtt{A}}$ do in fact share stub procedures for the operations of `A`. The procedures are exported from the $Srgt_{\mathtt{A}}$ client stub module in the form

```
PROCEDURE OpStub (self: A; ... ) ...
```

Notice that the type of the `self` argument for the surrogate stub method is declared as `A`, not $Srgt_{\mathtt{A}}$, as is more natural. This is necessary to allow `OpStub` to be used to override the appropriate methods of $Srgt_{\mathtt{AB}}$, where the `self` argument must be a supertype of $Srgt_{\mathtt{AB}}$. The trouble we have avoided is that $Srgt_{\mathtt{A}}$ and $Srgt_{\mathtt{AB}}$ are *unrelated.*

This solution works because the stub procedures need no state beyond what they inherit from `AnsaNetObj.T`. The operations of $Svr_{\mathtt{A}}$, by contrast, are likely to need access to their own state. The only way to share this state and associated procedures between $Svr_{\mathtt{A}}$ and $Svr_{\mathtt{AB}}$ is to hive them off into their own (possibly abstract) type. Though our design supports the introduction of enhanced service types to existing systems by subtyping, it seems to discourage unpremeditated inheritance of implementations; this may well not be a bad thing [Snyder 86, Raj 91], but it does conflict with a common use of inheritance.

It is possible to use Modula-3's partially opaque types (section 2.7.4) to wedge an implementation into the abstract type chain between `AB` and `A`, so that it can be inherited from `AB`. However, the mechanics of this are so baroque in the general case that we use the technique only to provide system-defined implementations of certain management interfaces to which all network objects conform.

The reader may be wondering whether the server-side dispatch procedures for `A` and `AB` (section 3.4.2) share operation stubs. They do. No inheritance is involved.

4.4 Typechecking

If we attribute types from some semantic domain to values, we can use them to avoid type errors. We distinguish between two kinds of value: concretely typed data and abstractly typed services. These are associated with different sorts of type error:

- Concretely typed data may be misinterpreted (e.g., an integer as a character, an array as a record, or a record of one type as a record of another type), which may cause subsequent computation to be arbitrary.
- An attempt may be made to invoke an operation which is not in the abstract type of the target service. This may crash the caller, but does not affect the callee.

Typechecking prevents type errors by comparing the expected type of a value with the type received when the value is transmitted from one part of a system to another (e.g., from the right to the left hand side of an assignment, or from a client to a server).

Typechecking can be performed at various stages in the life of a program fragment. Sometimes it is possible to prove from the static program text and assumptions about its eventual environment that no type errors in some class can occur at runtime. These proofs may be carried out automatically by the compiler, possibly with help from explicit assertions by the programmer in the form of type declarations. Wider classes of type error may be ruled out by later checks when separately compiled modules are linked, when a client program binds to a service or when a server receives some data or an invocation.

The aim of typechecking is to avoid confusion caused by programmers' mistakes; it is not necessarily to eliminate all runtime checks. The point of trying to prove the absence of some type errors before runtime is to warn the programmer of possible mistakes as early as possible. The runtime checks needed by a server to detect the remaining errors and to protect itself from incorrect or malicious clients can generally also catch the sort of errors eliminated by early typechecking. If this is the case, early typechecking is superfluous for correctness, but its cost may be repaid in time saved by programmers.

The argument that residual runtime checks may be made more efficient by ruling out some cases statically depends on how much can be assumed about the communicating parties. This raises issues of authentication [Lampson 92] which are beyond the scope of this dissertation. We will describe some experience of what can go wrong later (section 6.1).

4.5 Type Fingerprints

In order to "compare the expected and received types" (section 4.4) of a transmitted value, we must be able to identify the received type. Thus the value must (at least notionally) be accompanied by some form of type tag. Such tags should be cheap to transmit and compare.

If types can be given names by programmers, and if the communicating parties *share the environment* in which these names are bound to their referents, a canonical name for a type may be used as the tag. This sharing is readily arranged within a single linked program, but may be more costly to achieve between distant parts of a distributed system. In this approach, two types are equal when two *programmers* have agreed that they are by using the same name. This is the mechanism used by the ANSA testbench.

Another approach is to automate the management of the space of type names to some extent. In languages like CLU and Modula-3 with structural type equivalence (section 2.7.4), one can maintain a library of bindings between encoded type structures and automatically allocated names, unique with respect to that library. Again, the library must be shared between the communicating parties. Hamilton's CCLU RPC system adopted this technique [Hamilton 84, p. 61].

A third approach is to use the "encoded type structures" themselves as the type tags. This is unattractive because such encodings are likely to be bulky and expensive to transmit and interpret. All that the communicating parties need agree on, however, is the mapping from type structures to their encodings.

While this third approach may be too expensive as it stands, it can be approximated more cheaply as follows. A *type fingerprint* is a compact, fixed-length, probabilistically unique encoding of a type structure, and thus combines most of the advantages of the previous two approaches. One way to think about fingerprints is as hash values. Consider how a compiler might attempt to look up a type structure in the library maintained in the second approach, in order to discover the corresponding tag (if one exists). To make this process reasonably efficient, the library might be organised as a hash table whose buckets contained (*name*, *encoded type*) pairs. But suppose the hash function were strong enough that collisions never occurred, making it unnecessary to compare the type being looked up with the type in the table entry. Then the hash value *itself* would be a perfectly good identifier for the type.

Such a strong hash function would be rather hard to come by, but fortunately, there *are*

hash functions that reduce the probability of collisions to a negligible level [Rabin 81]. Strictly, fingerprints are only hints, but distinct types are so likely to have different fingerprints that they can be used as if they were the truth.

The SRC Modula-3 system provides 60-bit fingerprints based on the arithmetic of polynomials over the boolean field, modulo a fixed irreducible polynomial of degree 61. This can be done with cheap bit operations and some tables. These fingerprints are used for separate compilation and to ensure the type safety of persistent storage of concrete data in pickles (section 3.4.2).

We have investigated the use of fingerprints as type tags in our network objects system, as described in the following sections.

4.6 Static Types and Trading

The ANSA trader (section 2.6.2) maintains a database of service type names and the compatibility relationships between them. As mentioned above (section 4.5), manual intervention is required to update this state: new types must be registered before service offers can be traded.

When a C/DPL program exports a service to the trader, it supplies the name of the service type:

```
! DECLARE {svc} : AB SERVER
...
! {} <- traderRef$Export ("AB", ... , svc)
```

and a client specifies the required type name (among other constraints) when it imports the service from the trader:

```
! DECLARE {ir} : A CLIENT
...
! {ir} <- traderRef$Import ("A", ... )
```

The DPL preprocessor checks that the declared types of `svc` and `ir` are identical to the types supplied to the trader's import and export operations. Since the trader knows that

the type AB is compatible with A, the svc offer might be supplied to the client if it satisfies the client's other constraints. The trader does not check types, but uses them as a means to support a particular kind of property and constraint specification.

We can improve the type safety of bindings established via the trader. We do this by using the fingerprints of the abstract types implemented by a server object. Suppose the fingerprints of the types above are $fp_{\mathtt{A}}$ and $fp_{\mathtt{AB}}$. Then when the server exports its AB to the trader, we automatically include the pair

FPSet $\{ fp_{\mathtt{A}}\ fp_{\mathtt{AB}} \}$

in the property list for the offer. Similarly, when the client imports the service, we automatically include the conjunct

and ' $fp_{\mathtt{A}}$ ' in FPSet

in the constraints.

To what extent is this an improvement on the existing situation? The Modula-3 stubs for A and AB could supply the correct type names automatically, by providing import and export procedures with the appropriate signatures (as in section 3.2.3):

```
INTERFACE ABSRPC; (* server stub for AB.T *)
  PROCEDURE Export (ref: AB.T; ... ) ...
```

The implementation of ABSRPC.Export would supply the string "AB" as an argument in a call to the trader. Then the Modula-3 typechecking of the ref argument would be sufficient, with no need for fingerprints. This is essentially equivalent to the situation in C/DPL, except that a Modula-3 capsule may contain more than one implementation of AB.

The real benefit of using fingerprints is that, by identifying an abstract type unambiguously, they avoid the cost of managing the shared space of type names within a trading domain. Note that this is not simply the cost of maintaining the trader's type graph: there must also be agreement among *programmers* on the mapping from type names to service types, a mapping which is not represented in the trader. The worst case is when an old type name is reused for an incompatible new type, for instance when a new version of an

evolving service is installed[1]. Since, unlike the space of offer contexts, the space of type names is not hierarchical, its management cannot be delegated except by ad hoc schemes. It is not clear what happens to type name spaces when formerly disjoint trading domains are federated (section 2.6.2).

The fingerprint of a service type is (in theory) calculated from its defining IDL interface after replacing identifiers for both concrete data types and other service interfaces by their values, similarly expanded. In this scheme fingerprint equality is (probabilistically) equivalent to structural type equality, which is all that can be checked by receiving stubs: IDL's presentation syntax on the wire includes no name tags apart from operation names. Conveniently, this agrees with Modula-3 typechecking, though it would be simple to define fingerprints under name equivalence. In practice, the distinction has never mattered.

Fingerprints are not quite as useful for subtype checking as full structures. The fingerprint set $\{fp_{\mathtt{A}}, fp_{\mathtt{AB}}\}$ for the type `AB` above records that `AB` was *constructed by extension* of `A`. An equivalent free-standing type would have the fingerprint set $\{fp_{\mathtt{AB}}\}$; an attempt to pass such a server to a client requesting an `A` would not typecheck, despite being perfectly safe.

Having presented our fingerprint mechanism, we should say that we know of no occasions when the current ANSA testbench facilities have resulted in type-incorrect bindings. As mentioned above, such erroneous bindings would probably be detected at invocation time anyway. We cannot yet claim any experimental evidence for the superiority of fingerprints over the existing facilities, but we would expect the benefits to increase with the number of programmers in a trading domain.

4.7 Dynamic Types and Binding

The previous section described how type fingerprints can be used when establishing bindings via the trader. We now turn our attention to bindings established by passing network object references in arguments and results of invocations.

When a program first receives a network object reference, it must create a surrogate (section 3.4.3). What should the type of this surrogate be? The possible choices are fixed by the client stub modules linked into the program, each of which defines a surrogate type for a particular abstract service type.

[1]This has in fact happened in the Laboratory's Active Badge system.

The signature of the operation in which the network object reference is transmitted will have attributed an abstract type to the appropriate parameter or result. An obvious answer is to use the surrogate type for that abstract type. Suppose our program has a server object with an operation

```
register (ref: A.T; name: TEXT) ...
```

Then the server stub for `register` would call a procedure in `A`'s client stub module to obtain a $Srgt_{\mathtt{A}}$ for use as the `ref` argument in its call on the server object. Notice that some information about the type of `ref` may have been lost: the invocation at the client might have been

```
registry.register (NEW (ABSvr.T), "enhanced server") ...
```

where `ABSvr.T` is a server implementation of the abstract type $AB <: A$. The loss of information is unlikely to matter to the registry, but it might to another client which later retrieves the `"enhanced server"`.

This problem is a failure of transparency from the Modula-3 point of view. If `registry` were in the same address space as its registering and retrieving clients, the latter could (implicitly or explicitly) `NARROW` the returned object reference to the type `AB.T`, making use of the dynamic type information in the Modula-3 heap (section 2.7.4).

The fact that, in this scheme, a program which receives a network object reference has no access to the dynamic type of the referent has other unpleasant consequences. Suppose the program containing the `registry` server receives the same `ABSvr.T` over two different interfaces: once with static type `A.T` and *later* as an `AB.T`. Then the first surrogate will be created with type $Srgt_{\mathtt{A}}$, and be entered in the object table. When the second reference arrives, there is already a surrogate for the remote `ABSvr.T`, but it has the wrong type. We could simply create a second surrogate, of type $Srgt_{\mathtt{AB}}$, but this complicates the management of the state of the distributed garbage collector (section 5.3.6). Since all references to the same remote object are no longer necessarily equal, some transparency is also lost, but this is rather less serious.

The solution we have adopted, following [Nelson 92], allows dynamic type information to propagate between address spaces. We arrange that a surrogate is created for the richest (<:-least) abstract type for which the receiver and the server (which is not necessarily the sender) *both* have appropriate stubs.

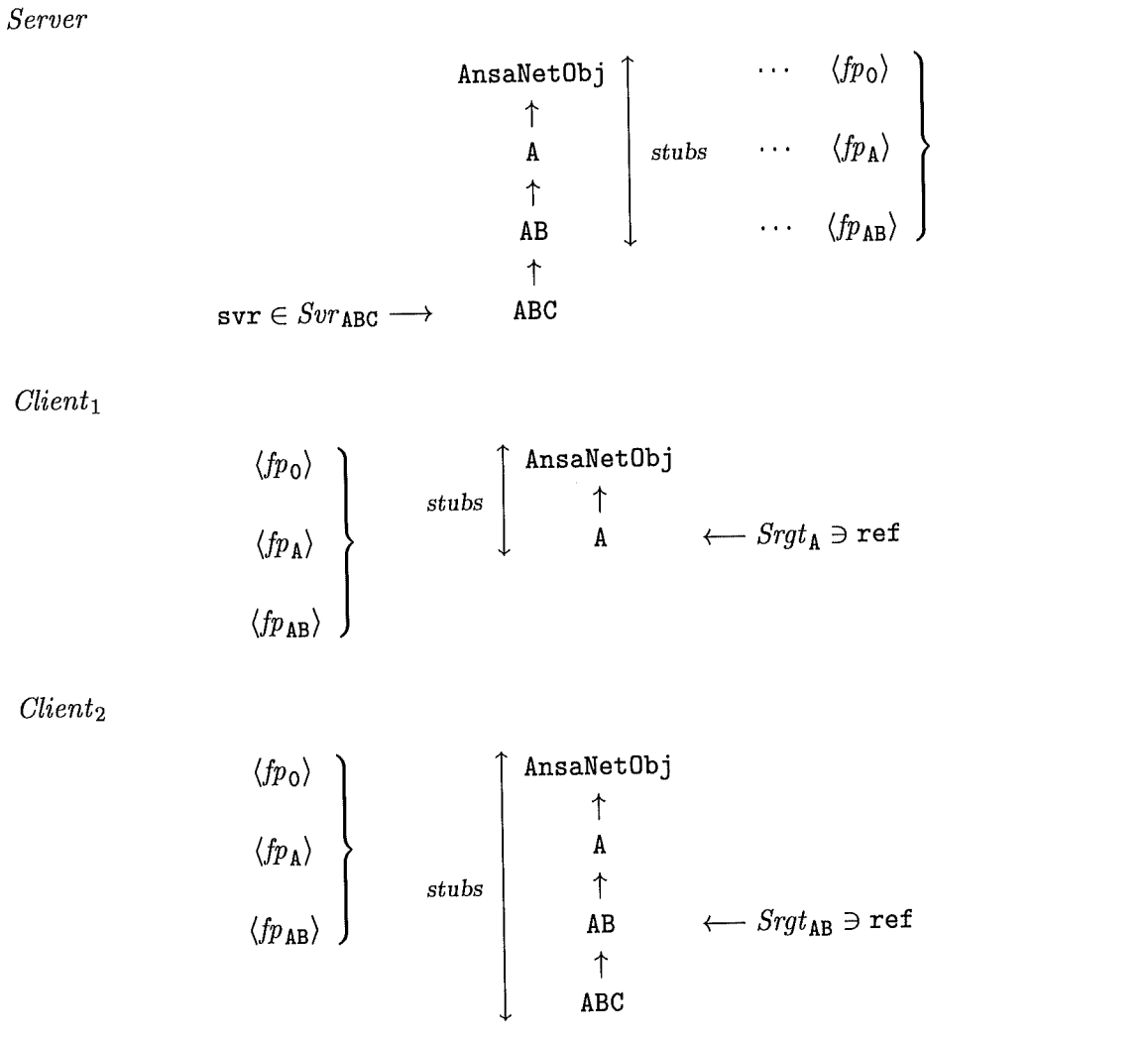


Figure 4.1: Choosing the type of a surrogate.

As in the previous section, the set of abstract types implemented by the server object and for which the server program has stubs can be represented by a set of fingerprints. This set is now regarded as ordered by `<:`. It could either be passed between programs with the corresponding interface reference, or could be discovered by a call from the receiver to the server before a surrogate is created. Since our distributed garbage collector needs to make such a call anyway, we take the second course.

In figure 4.1, when `svr` is passed to $Client_1$, an old program which expects a reference of static type `A` and has client stubs for that type only, the surrogate `ref` is created with type $Srgt_{\texttt{A}}$. When $Client_1$ passes `ref` to $Client_2$, a newer program which has client stubs for the extended service `ABC` but only expects an `A` from its interface with $Client_1$, the new surrogate is richer. However, it still only gets the type $Srgt_{\texttt{AB}}$: although the owner of `svr` implements a $Svr_{\texttt{ABC}}$ subtype of `ABC`, it has for some reason only been linked with server stubs for `AB`. All programs have stubs for `AnsaNetObj.T`.

Notice that this strategy requires the receiver to check that the newly created surrogate is actually as rich as statically expected. Suppose in figure 4.1 that the server transmits `svr` to $Client_2$ as an operation argument "`ref`" of static type `ABC`. Then since `ref` is received with type $Srgt_{\texttt{AB}}$, the invocation cannot proceed in $Client_2$; instead, we raise an exception, which will propagate back to the caller. Receiving stubs must perform this check in any case (to protect themselves from being spoofed by rogue transmitters), so we would gain nothing by performing the check at the sender.

Some existing ANSA services, such as the trader, deal in generic untyped interface references. These IDL analogues of Modula-3 `REFANY`s are given the type `AnsaNetObj.T` in Modula-3 signatures by `stubm3`. Since such services do not support our distributed garbage collector either, we allow the programmer of a receiving program to convert a surrogate for one type to a surrogate for a richer type by passing the required typecode to an explicit `Narrow ()` procedure. No guarantees can be made about the validity of the type of a fresh surrogate created this way. We will mention how this compatibility wart interacts with the distributed garbage collector in section 5.3.5.

4.8 Stub Type Registry

The mechanisms of the previous two sections are implemented in our system by a run-time module called the *stub type registry*. This module communicates with client and server stub modules via an interface defined in terms of typecodes, which have a unique

meaning only in a single running program (section 2.7.4), and fingerprints, which are valid everywhere.

When the client stub module for the abstract type AB is initialized, it registers its surrogate type with the call

```
AnsaStubs.RegisterSrgt (TYPECODE (AB.T), TYPECODE (Srgt_AB))
```

When the AB server stub module is initialized, it registers its dispatch procedure (section 3.4.2) similarly:

```
AnsaStubs.RegisterDispatcher (TYPECODE (AB.T), Dispatch_AB)
```

When `svr:` Svr_{ABC} in figure 4.1 is first exported, it is given a name and entered in the object table (section 3.4.3) with the dispatch procedure returned by

```
AnsaStubs.Dispatcher (TYPECODE (svr))
```

which is $Dispatch_{AB}$, the one for the <:-least supertype of Svr_{ABC} for which a dispatcher has been Registered. The corresponding vector of fingerprints is generated with

```
AnsaStubs.TcToFpTower (TYPECODE (svr))
```

The fingerprint vector is only calculated once: subsequent calls to TcToFpTower for the same typecode or one of its supertypes use an appropriate subarray which is stored with the dispatch procedure for the typecode in question. Similarly, towers for subtypes are constructed using all the fingerprints stored so far.

A client receiving a reference to svr for the first time obtains its fingerprint vector (fps) as a result of a call to the server. It then uses the stub type registry to discover the local typecode of the appropriate surrogate type. Finally, it passes this typecode to the SRC Modula-3 runtime heap allocator to create the surrogate:

```
RTHeap.Allocate (AnsaStubs.FpTowerToSrgtTc (fps))
```

The implementation of `FpTowerToSrgtTc` converts the fingerprints in `fps` into the corresponding local typecodes (if they exist) and returns the surrogate typecode for the first (`<:`-least) with a `RegisterSrgt` entry. The result of this calculation is cached in a table, indexed by fingerprints, to speed up future queries. Since every program contains stubs for `AnsaNetObj.T`, the search is guaranteed to succeed for well-formed `fps`. An ill-formed `fps` causes an exception to be raised in the caller of the operation which transmitted the corresponding network object reference.

Maintaining the bindings between abstract service types and client and server stubs in a runtime registry, rather than relying on static information, incurs some runtime overhead. However, the resulting flexibility seems worth the cost. References to network objects of new types can pass through old programs without any loss of information or type safety. In addition, stubs which marshal and unmarshal network object references are rather simpler, since they need only call generic import and export routines. Calling routines specialised for each service type involves a surprisingly large amount of work in the stub generator to keep track of the dependencies between stub modules.

we here dispatch
You, good Cornelius, and you, Voltemand,
For bearers of this greeting to old Norway,
Giving to you no further personal power
To business with the king, more than the scope
of these delated articles allow.

— CLAUDIUS, Hamlet (Act I, Sc. 2, line 33)

Chapter 5

Distributed Garbage Collection

5.1 Introduction

Different parts of a computing system often share state or immutable data with each other by naming its container [Saltzer 79]. Such state sometimes represents other resources. For instance, different routines or modules in a program may keep pointers to some shared object allocated from the (finite) system heap; an editor and an incremental compiler running in separate processes on the same machine might keep handles on nodes in a shared parse tree; a program running on a cycle server might interact with its user(s) via windows on other machines; a client program might hold a temporary lock on a file stored at a remote server, a descriptor containing a cursor at which to read or write the file, a binding to the file service, and an identifier for the file itself; a file on one file server might contain a link to a file on another file server.

Whenever such sharing occurs, the problem arises of managing the resources required to represent the containers themselves. When a container's contents are no longer of interest, usually because they can no longer affect the subsequent computation, the container can be reused (possibly under a different name, if the names at that level are not themselves a scarce resource). Reuse is a necessary evil: resources are rarely unlimited in the long run. However, when names are passed across the interfaces between different parts of the system, no one component necessarily has enough knowledge to decide when reuse is safe, particularly when each container can hold the names of others. Under these circumstances, the familiar problems of dangling references and resource leaks are difficult to avoid without careful programming of *all* the components.

For any given collection of components sharing some given type of state, it is always possible to modify the interfaces between them to require the transmission of enough information to allow containers to be reused safely. Typically, one component will be deemed to own the containers of some type, and the others must explicitly inform it when they are no longer interested in a particular container. However, this strategy has some disadvantages. For each instance of the problem, a separate scheme must be designed, implemented and established to be correct. Such schemes often significantly increase the complexity of both the interfaces and the implementations of the components. Both these problems are multiplied when there are indirect dependencies crossing multiple interfaces. As a result, programmers in such systems are discouraged from creating transient names and containers dynamically.

These problems have led to the use of automatic garbage collectors which provide, for an entire uniform naming structure, a single, correct solution to an abstract version of the problem in which possession of a container's name is regarded as the sole indication of an interest in its contents. The most widely used garbage collectors manage pointers between heap nodes in a single address space. There are many well-established schemes [Wilson 92] for this situation, in which a heap node is reusable when it is unreachable via any chain of pointers from a given set of *root* objects.

As the examples given above imply, similar resource management problems arise in service-based distributed systems. In an object-based system such as we are considering in this work, it is often natural to represent transient, shared, potentially remote resources as network objects providing some service, even when this service has few public operations—or indeed none at all, in which case the network object reference is useful simply as a typed name to be passed to other services with a fuller view (section 3.3.2). As we have mentioned before, lightweight creation and binding of service interfaces is a primary motivation for the network objects style.

An automatic distributed garbage collector for network objects and their references is therefore a valuable tool; collection of garbage objects can then be used in an application-specific way to trigger recovery of associated resources. However, the requirements and environment of such a collector are sufficiently different from the single-program case to make building one a challenging problem. Distributed systems differ from centralised ones because they must continue to be useful in the face of the independent failure of their components and the cost and unreliability of communications between them. They must achieve this despite the fact that no one component has complete knowledge of the global state. This affects most aspects of the design of such systems, and resource management is no exception:

- Our collector must remain safe and live despite communications problems.
- If it is to be used in a long-running server to manage resources associated with network objects that are shared with clients, it must be possible to reclaim them even when clients are unavailable: a file server which has handed a reference representing a file lock to a client which subsequently cannot communicate with the server (either because it has crashed or because of extended communications problems) would like to reclaim that lock despite the client's inability to cooperate.
- The collector should impose low overheads in performance, in restrictions on the freedom of programmers and in the cooperation it requires from conventional local collectors. It should run in parallel with normal computation, require no global synchronisation between processes, allow network object references to be transmitted freely between them and require minimal modification of their local collectors.

In an ordinary program, the performance cost of a generic garbage collector which cannot make use of application-specific knowledge may be too high in some critical areas. Furthermore, even in a centralised system it is not necessarily desirable to use a single collector for all objects regardless of the number of references they can hold or the expected lifetimes of both references and objects.

Likewise, automatic distributed garbage collection is not appropriate for all resource management tasks in a distributed system. It is often possible to regard some objects as merely caching state which can always be rebuilt (at some cost) from information held elsewhere; such objects can simply be discarded unilaterally, regardless of whether outstanding references remain. Client code required to recover from failures of distribution transparency may also be able to recover from dangling references. End-to-end arguments [Saltzer 84] are helpful in striking the right balance here.

A strategy which may be appropriate for long-term persistent naming structures is the *librarian's algorithm*: simply migrate old and infrequently-used objects to permanent bulk storage with unbounded capacity but high retrieval cost (such as tape or optical disc). The cost of retaining some garbage forever may be outweighed by the cost of determining the liveness of all data strictly. It may not be easy even to define an appropriate concept of garbage at this level: the semantics of names may be imprecise [Needham 93] and since the objects involved (e.g., versions of documents) are of interest mainly to humans, one should perhaps be wary of attempting full automation.

We should distinguish between the use of garbage collection in a distributed system, such as in the asynchronous garbage collector of the Cambridge File Server [Garnett 80], from

distributed garbage collection, in which references and their referents may be in different processes. It is the latter we have addressed. (The CFS collector did not guarantee to preserve objects whose only references were held outside the file system: a request to store such a reference in the file server might be rejected. It did guarantee, even in the face of crashes and communication problems, that if such a request succeeded, it would produce a state consistent with the resurrected object having never been garbage.)

The network objects in our system represent services rather than small data items. They are supported on stock operating systems and hardware connected by well-established local and wide area networks and internetworking protocols. Their lifetimes range from the time required to make a few RPCs up to the lifetime of a process in this environment. It is network objects that we wish to reclaim; we have not tackled automatic garbage collection of persistent heaps, whether distributed or otherwise. The rest of this chapter presents the design and implementation of an efficient fault-tolerant distributed garbage collector which comes close to meeting our requirements.

5.2 The Collector

5.2.1 Basic Algorithm

Our garbage collector is a fault-tolerant distributed algorithm which runs concurrently in all processes that transfer network object references. It interacts with the processes' local garbage collectors to achieve two goals: *safety* (collecting only garbage) and *liveness* (collecting all garbage). More precisely, these are

- **Safety.** If a process P has a surrogate for an object o, then o is in its owner's object table and hence protected from its local garbage collector.
- **Liveness.** If it remains true that no process holds a surrogate for o, then eventually it remains true that o is not in its owner's object table.

To meet these conditions, we arrange that the owner of o knows which processes have surrogates for it, and that o is kept in the owner's object table whilst this *dirty set* is non-empty. Maintaining $o.dirtySet$ rather than just a reference count has advantages for robustness in the face of transient communication problems and process crashes, as we shall see.

Introducing *o.dirtySet* splits our safety and liveness conditions into two parts. Safety now requires that if P has a surrogate for o then $P \in o.dirtySet$, and that if *o.dirtySet* is non-empty, o is in the object table. Liveness decomposes similarly. Maintaining the invariant relationship between *o.dirtySet* and its owner's object table is simple, since they are in the same program; maintaining the other conditions is more interesting. The actions of the collector fall into two classes: those that try to maintain safety by inflating *o.dirtySet* (adding processes), and those that try to achieve liveness by deflating *o.dirtySet* (removing processes).

The basic outline of the algorithm is as follows. When a client process receives a reference to o, it adds itself to *o.dirtySet* by making a *dirty call* to o's owner *before* it creates a surrogate. After a process detects that it has no reachable surrogate for o, it removes itself from *o.dirtySet* by making a *clean call*. The algorithm can reclaim an object even if a process crashes while holding a reference to it: when o's owner detects that a client process P has crashed, it removes P from *o.dirtySet* itself.

The details of the algorithm order dirty and clean calls to ensure that *o.dirtySet* is maintained conservatively, without compromising liveness, while requiring only pairwise synchronisation between processes.

The algorithm falls short of the goals above in two ways. It fails to be completely safe in that o's owner may misinterpret extended but impermanent loss of communication with a dirty client process P as its death. If P holds the only surrogate for o, then o might be prematurely removed from its owner's object table; should communication be restored, P will discover the problem when it attempts an invocation on its surrogate. The algorithm fails to collect all garbage: our liveness condition, like those of reference counting schemes, does not rule out distributed cycles of inaccessible objects; responsibility for avoiding or breaking cycles rests with the programmers involved *as a group*.

5.2.2 Object Table

A concrete object o has an associated name *o.id* which uniquely identifies it for all time and is used when transmitting a reference to o. Each process contains an *object table*, which maps identifiers to the corresponding local surrogate or concrete object when unmarshalling a reference or servicing an invocation (section 3.4.3).

At o's owner, the object table entry for *o.id* is said to be *concrete* and contains an ordinary reference to o. While present, this reference protects o from the local garbage collector.

As mentioned in section 3.4.3, we keep at most one reachable surrogate for o at a client process, and the object table entry for $o.id$ contains a *weak reference* to it. Weak refs do not protect their referents from the local collector.

A weak ref for an object is created by presenting the object to a local garbage collector interface together with a cleanup procedure [Hayes 92]. The weak ref returned is a token which can be mapped to the corresponding ordinary reference until the local collector determines that its referent is unreachable. When this happens, the local collector schedules a call to the cleanup procedure before reclaiming the referent's space.

A surrogate's object table entry can be in one of three states. While a dirty call is in progress for o and no surrogate has yet been created, the object table entry for $o.id$ is *nascent* and does not contain a valid weak ref. Once the surrogate has been created, the entry is *alive*, and holds a valid weak ref from which the surrogate can be obtained. Once the local garbage collector determines that the surrogate is unreachable, the entry becomes *dead* and subsequent attempts to convert the weak ref fail. Dead entries are eventually removed by the surrogate cleanup procedure.

To summarise, the object table entry for id at P is alive if and only if there is a unique reachable surrogate with $srgt.id = id$ at P; the entry is concrete if and only if there is a unique local concrete object with $o.id = id \neq \mathbf{nil}$.

5.2.3 Dirty Set and Timestamps

For each concrete object o, we maintain $o.dirtySet$ at its owner as described above. How can we keep $o.dirtySet$ up to date with the state of the client P with respect to o? We could imagine making strictly serialised clean and dirty calls from P such that between each action, the state of the client agrees with the state of the dirty set. Such an approach (as in the first collector presented in [Vestal 87]) would be obviously correct, but is expensive to implement in the presence of process or communication failure and imposes excessive synchronisation at the client.

It turns out that the clean and dirty actions can be broken into separate parts: one to change the client state and one to call the owner to update the dirty set. The four actions

- put P in $o.dirtySet$;
- create and enter the surrogate in P's object table;

- remove the object table entry and reclaim the surrogate; and
- remove P from $o.dirtySet$

can be interleaved less strictly as long as the owner can tell in which order the client states occurred.

To achieve this, we label successive states at P (with respect to o) with a strictly increasing timestamp. Each subaction, including the clean and dirty calls, is then associated with the timestamp of the state to which it refers. We maintain the highest timestamp seen from P for o with $o.dirtySet$ at the owner. The owner ignores any clean or dirty call with an earlier timestamp.

With their timestamps, clean and dirty calls become idempotent; this makes dealing with transient communication failures easier. We will see later that under normal circumstances the owner need not keep timestamps for processes it believes to be clean, reducing the amount of per-client state at the owner.

5.2.4 Transmitting a Reference

Suppose a process P transmits a reference for o to process Q in the arguments or results of some invocation (figure 5.1). If Q already has a reachable surrogate for o, then the entry for $o.id$ in Q's object table is alive, the surrogate can be obtained from its weak ref and no further action is required (*Import* and *CheckPresent*; figure 5.3). Similarly, if Q is o's owner, its object table contains a concrete entry for o.

Otherwise, we must add Q to $o.dirtySet$, create a new surrogate and enter it in Q's object table. If all the state involved were in a single address space, we would make this an atomic action under a shared lock. In the distributed case, however, this is impractical. Instead, we allow transient inconsistency between the states of Q and the owner.

Safety requires that we ensure Q is in $o.dirtySet$ first. The first thread that finds the object table entry for $o.id$ absent or dead makes it nascent, acquires the next timestamp, releases its lock on the object table and makes the dirty call (figures 5.4 and 5.5). Other threads unmarshalling o while this call is outstanding discover the nascent table entry and block on a condition variable (section 2.7.3) until the outcome is known.

If the dirty call succeeds, the original thread creates the new surrogate, obtains a weak

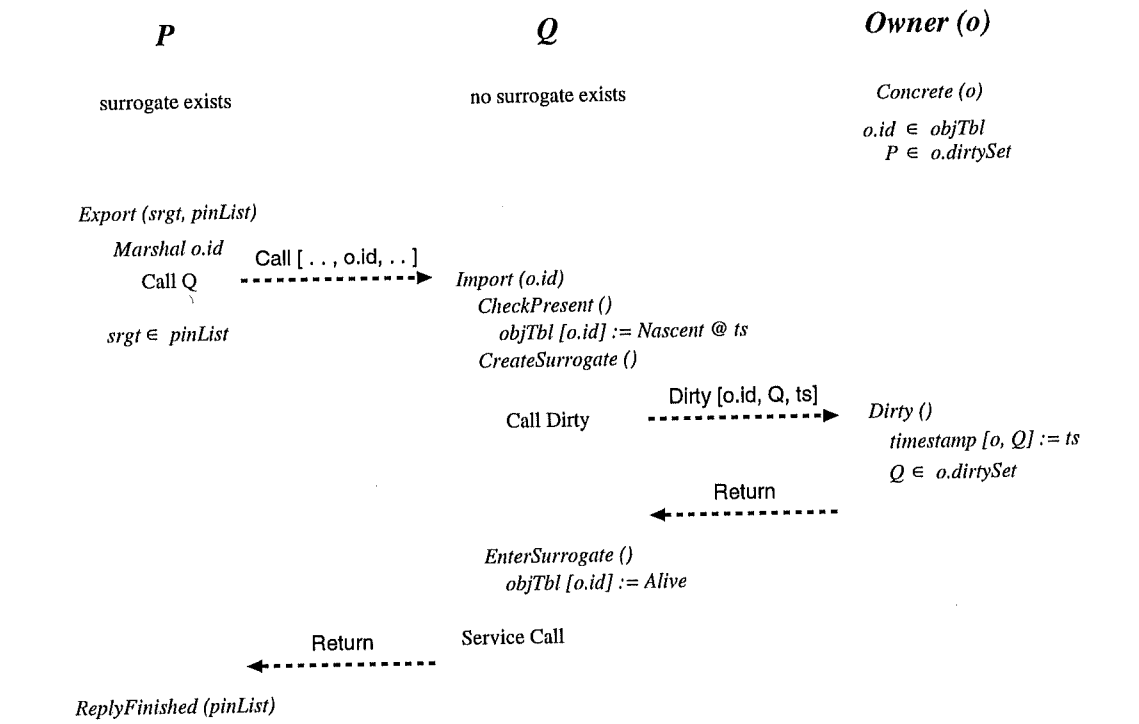


Figure 5.1: Overview of reference transmission.

ref for it with the collector's surrogate cleanup procedure and makes the object table entry alive, before unblocking any waiting threads (*EnterSurrogate*; figure 5.3). While the newly-created surrogate remains reachable, we can be sure that the timestamp recorded for (o, Q) at the owner is that of the state when its object table entry became nascent. We will deal with how to achieve liveness in the face of failed calls in section 5.2.7 below.

Safety also requires some synchronisation between P and Q: we must keep $o.dirtySet$ non-empty until the receiver's dirty call is known to have been processed. For suppose P has the only surrogate for o, and having transmitted it to Q, drops the reference. If the consequent clean call from P is processed first, it will leave $o.dirtySet$ empty, and the dirty call from Q will find that o has been removed from its owner's object table.

If P holds a surrogate, then P will be in $o.dirtySet$ while it remains reachable, as mentioned

above. Therefore we keep the surrogate reachable by placing a strong reference to it on a *pin list* associated with the transmission (*Export*; figure 5.2). On the other hand, if P is o's owner, we must do slightly more work, in case a clean call from some process R makes *o.dirtySet* empty before Q's dirty call arrives. To prevent this, the owner simply places itself in *o.dirtySet* while transmissions are in progress; this is inexpensive because *o.dirtySet* is stored in the owner's address space. Since we allow multiple concurrent transmissions of o from its owner, we must keep track of *how many* are in progress and only remove the protective entry in *o.dirtySet* when this becomes zero. The count, in *o.pins*, is maintained by placing o on the transmission's pin list, just as in the surrogate case. The details of our implementation in fact require no extra space for *o.pins* (section 5.3.3).

Once Q's dirty call is known to have been processed, it is safe (and required for liveness) to release the protection of the objects in the transmission's pin list (*ReplyFinished*; figure 5.2). How does P know when this happens? If the transmission was of arguments in a call from P to Q (as in figure 5.1), the fact that that call has returned is sufficient, since Q's dirty call was nested under it. (This means that *surrogate arguments* need not be placed on pin lists: they are protected by the references in the caller's stack frame.) However, if the transmission was of results of a call from Q to P, we require an acknowledgement from Q. This does not affect the latency of Q's original call since it can be sent asynchronously. The necessary modifications to a standard RPC protocol like REX are quite simple; we discuss them later with other implementation details (section 5.3.4).

The figures on the next few pages collect all the abstract pseudo-code for our algorithm in one place. The reader may wish to skip to the next section (p. 87) at this point.

```
Export (o: Obj; in out ps: PinList) : Id =
  lock mu do
    if o.id = nil then { ¬∃ id. objTbl [id] = o }
      o.id := NewId (); objTbl [o.id] := Concrete (o)
    end
    ps := o :: ps
    if Concrete (o) then
      inc o.pins; o.dirtySet := o.dirtySet + MyProcessId
    end
    return o.id
  end
end Export

ReplyFinished (in out ps: PinList) =
  lock mu do
    foreach o ∈ ps | Concrete (o) do
      dec o.pins
      if o.pins = 0 then UnRef (o, MyProcessId) end
    end
    ps := nil
  end
end ReplyFinished
```

Figure 5.2: Collector transmission actions: Transmitter.

```
Import (id: Id) : Obj =
  case CheckPresent (id) of
    Yes (obj) ⇒ return obj
  | No  (ts)  ⇒ return EnterSurrogate (id, CreateSurrogate (id, ts))
  end
end Import

CheckPresent (id: Id) : Yes (Obj) | No (Timestamp) =
  lock mu do
    while id ∈ objTbl ∧ objTbl [id] ≠ Dead do
      case objTbl [id] of
        Alive ∨ Concrete (obj) ⇒ return Yes (obj)
      | Nascent ⇒ Thread.Wait (mu, srgtReady) (* synch *)
      end
    end
    objTbl [id] := Nascent
    return No (NextTimestamp())
  end
end CheckPresent

EnterSurrogate (id: Id; srgt: Obj) : Obj =
  lock mu do { objTbl [id] = Nascent }
    if srgt ≠ nil then
      objTbl [id] := Alive (WeakRef (srgt, CleanupSrgt))
    else
      objTbl.delete (id) { id ∉ objTbl }
      enqueue CallClean (id, NextTimestamp(), Strong)
    end
  end
  Thread.Broadcast (srgtReady)
  return srgt
end EnterSurrogate
```

Figure 5.3: Collector transmission actions: Receiver.

```
CreateSurrogate (id: Id; ts: Timestamp) : Obj =
  (* mu not held *)
  call Dirty (id, MyProcessId, ts) at Owner (id)
    except RPCFailure ⇒ return nil
  return NewSurrogate (id)
end CreateSurrogate
```

Figure 5.4: Collector transmission actions: Surrogate creation.

```
Dirty (id: Id; p: ProcessId; ts: Timestamp) =
  lock mu do
    if id ∉ objTbl then return (* orphan *) end
    o := objTbl [id] { Concrete (o) }
    if timestamp [o, p] < ts then
      o.dirtySet := o.dirtySet + p; timestamp [o, p] := ts
      OnProcessDeathDo (p, ProcessTerminated)
    else
      skip (* orphan *)
    end
  end
end Dirty
```

Figure 5.5: Collector transmission actions: Owner.

```
CleanupSrgt (srgt: Obj) =
  { srgt is unreachable }
  lock mu do
    if srgt.id ∈ objTbl ∧ objTbl [srgt.id] = Dead then
      objTbl.delete (srgt.id)
      enqueue CallClean (srgt.id, NextTimestamp(), Normal)
    end
  end
end CleanupSrgt

CallClean (id: Id; ts: Timestamp; kind: Normal | Strong) =
  (* mu not held *)
  repeat
    call Clean (id, MyProcessId, ts, kind) at Owner (id)
  until call succeeds or Owner (id) believes MyProcessId has crashed
end CallClean
```

Figure 5.6: Collector cleanup actions: Client.

```
Clean (id: Id; p: ProcessId; ts: Timestamp; kind: Normal | Strong) =
  lock mu do
    if id ∉ objTbl then return (* orphan or late *) end
    o := objTbl [id] { Concrete (o) }
    if timestamp [o, p] < ts then
      UnRef (o, p)
      if kind = Strong then keep [o, p] := true end
      if keep [o, p] then
        timestamp [o, p] := ts
      else
        timestamp [o, p] := −∞
      end
    end
  end
end Clean

ProcessTerminated (p: ProcessId) =
  lock mu do
    foreach o | p ∈ o.dirtySet do UnRef (o, p) end
  end
end ProcessTerminated

UnRef (o: Obj; p: ProcessId) =
  (* mu held *)
  o.dirtySet := o.dirtySet − p
  if o.dirtySet = {} then
    objTbl.delete (o.id); o.id := nil
  end
end UnRef
```

Figure 5.7: Collector cleanup actions: Owner.

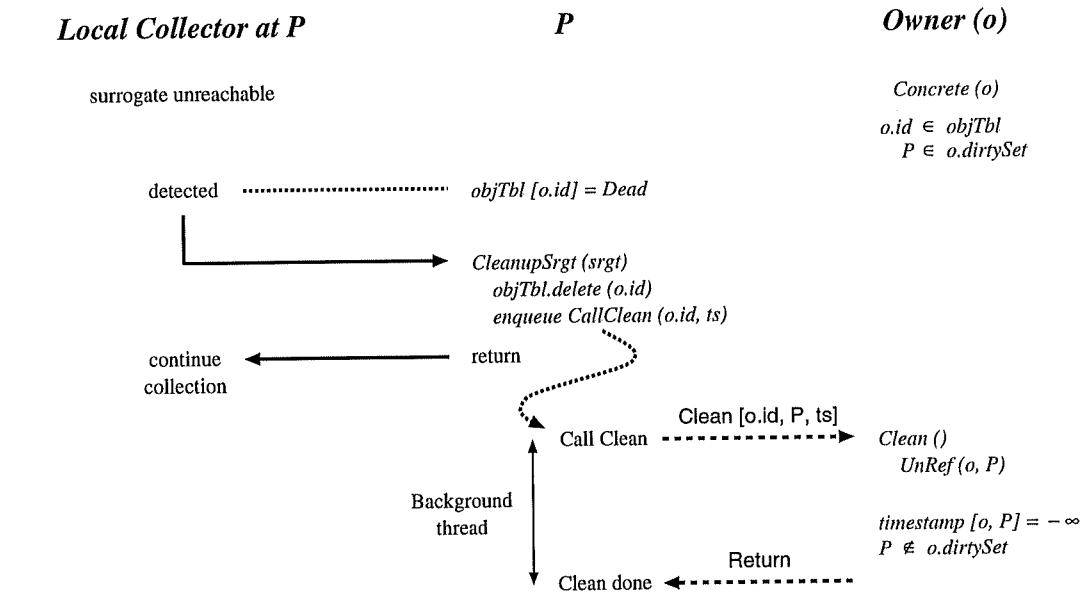


Figure 5.8: Overview of surrogate cleanup.

5.2.5 Cleanup

The previous section was mostly concerned with safety. We now turn our attention to the collector actions whose main goal is liveness. As mentioned earlier, when the local garbage collector at P discovers that a surrogate for o is unreachable, it schedules a call to the surrogate cleanup procedure (figure 5.8). The cleanup executes asynchronously; this implies an arbitrary scheduling delay.

Again, we need to update both P's object table and $o.dirtySet$; again, these subactions can be decoupled using timestamps. The cleanup procedure first inspects the current state of the object table (*CleanupSrgt*; figure 5.6). It must take further action only if there is a dead entry for the surrogate. In this case, it deletes the table entry, acquires the timestamp for the new state, and schedules a clean call to the surrogate's owner. Notice that the other cases can arise:

- the table entry may be nascent or alive if a reference for o has arrived after the death but before the cleanup of the surrogate incarnation (or *avatar*) in hand; or
- there may be no table entry at all if such an intervening surrogate has already died and been cleaned up.

In the second of these cases a clean call would be superfluous, though safe. In the first, a clean call would be unsafe; the fact that none is generated is necessary for the condition stated in the previous section (on the timestamp for (o, P) recorded at the owner) to hold. The fact that the local collector has discovered a particular avatar to be dead has only the force of a *hint* that P may now be clean.

Although pending dirty calls must lock out clean calls for safety, liveness requires only that the contents of *o.dirtySet* should *eventually* reflect the absence of a reachable surrogate at P. Clean calls can therefore be made asynchronously in a background thread (*CallClean*; figure 5.6), and a pending clean call for o from P does not delay dirty calls caused by concurrent unmarshalling. Since there is no synchronisation between outstanding clean and dirty calls from P, the semantics of RPC and the vagaries of scheduling mean they will execute in an indeterminate order at the owner. However, the overall outcome *is* determinate, safe and live: it is that defined by the calls' timestamps.

When the owner receives a clean call from P with a fresh timestamp (*Clean*; figure 5.7), it updates its stored timestamp, removes P from *o.dirtySet* and if necessary removes its concrete object table entry for o. The new timestamp need not be retained unless there have been communication failures between P and the owner; we will be more explicit about this in section 5.2.7.

5.2.6 Process Failure

When a process P terminates, whether normally or abnormally, it should be removed from the dirty sets of the objects for which it holds surrogates. We arrange that the collector at a process is informed when any of its dirty clients terminates (*ProcessTerminated*; figure 5.7), and make the appropriate adjustments to the dirty sets of the objects it owns.

How is the owner informed of P's death? If the underlying operating system has an appropriate mechanism, that can be used: if P and the owner run on the same machine, this is straightforward; a distributed system may provide a shared death-notification service (see, for instance, any of [ANSA 92b, Craft 85, Birman 87b, Mullender 86]). Our liveness

condition will be satisfied as long as P's death cannot go unreported for ever.

In the absence of other mechanisms, we place the burden of detecting P's death on the owners, which periodically ping P. If such attempts continue to fail for some fixed time, P is declared dead. Of course, this technique cannot distinguish an extended communication failure from process failure. As a result, safety can be violated. If P is deemed dead while holding the last surrogate for o, then subsequent attempts to invoke o will be reported failed at P as if there were still communications problems, even if communication has been restored.

Death detection can also interfere with the synchronisation required during transmission of a reference. If P holds the last surrogate for o, transmits it to Q and is reported dead at o's owner before Q's dirty call, that call will be reported failed at Q. Thus the death-detection timeout should be longer than the expected time for the dirty call to arrive. The value of this timeout is a parameter of our collector which affects the balance between safety and liveness. It should probably be set in the light of application-level knowledge. Greater safety could be bought at the cost of significantly increased overheads if P informed o's owner before transmitting the reference, turning reference transmission into a three-way consensus protocol between P, Q and the owner (as in [Lermen 86], though that algorithm can tolerate neither process nor communication failure).

Liveness is not compromised by the death of a receiving process Q during transmission. Any pins at P are always released, either because P's call to Q is reported failed, or because Q's acknowledgement of results from P fails to arrive. As we will see later, the mechanism in the underlying RPC protocol is similar in both these cases (section 5.3.4). There is a potential race between the last dirty call from a failed process Q and the detection of its death at the owner. Should the dirty call be processed second, $o.dirtySet$ will be inflated. However, since this re-arms death detection for Q (*Dirty*; figure 5.5), liveness is ensured in the long run.

5.2.7 Communication Failure

We now consider how the collector reacts when clean or dirty calls from a client P fail because of communication problems. The semantics of the RPC system we depend on are such that if a call returns successfully at P, it has executed exactly once so far at the owner and will never execute there in the future. If the call is reported to have failed at P, we know only that it has executed at most once so far, and may yet execute as an orphan at the owner at some future time.

Suppose a clean call is reported failed at P (*CallClean*; figure 5.6). The timestamp mechanism means that the call is idempotent: it can be repeated safely and any future orphans will be ignored by the owner. For liveness, a failed clean call must be repeated until it succeeds or communication with the owner is deemed permanently lost, at which point we know that the owner has either crashed itself or has decided that P has crashed. Since the owner ignores out-of-date clean calls, we do not trouble to re-check P's object table between retries.

When a dirty call is reported failed at P (*Create-* and *EnterSurrogate*; figures 5.4 and 5.3), we cannot safely create a surrogate; P may not be in $o.dirtySet$. For liveness we must now issue a clean call, in case the failed dirty *has* executed at the owner. The precondition for any clean call is that $o.id$ is not in P's object table in the state labelled by the timestamp of the call. Thus we delete the nascent table entry, acquire the new timestamp and enqueue the clean.

In order to reduce storage requirements in the common case, a normal clean call for o from P (caused by the death of a surrogate) causes the owner to forget the stored timestamp for (o, P); we represent unstored timestamps by $-\infty$ in figure 5.7. This is safe because a subsequent clean (with a later or earlier timestamp than the forgotten one) would have no additional effect at the owner, and a subsequent dirty with a later timestamp should succeed. However, if a dirty call with an earlier timestamp than the forgotten one arrives at the owner, it will erroneously succeed, compromising liveness and creating a storage leak.

What could have caused such a dirty call? It cannot have returned successfully at P: if it had, the clean call with the later forgotten timestamp could not even have been issued at P by the time the dirty arrived at the owner. Therefore it must be an orphan from a dirty call which was reported failed at P earlier than the forgotten timestamp. So after a failed dirty call, P must send a *strong* clean call which causes timestamps for (o, P) to be retained indefinitely, against the day when any dirty orphan arrives. Dirty orphans need strong cleans! We now have a storage leak of one timestamp record; this is likely to be less serious than retaining o (potentially indefinitely).

5.2.8 Correctness

The introduction of dirty sets and object tables splits the safety and liveness invariants of our collector for o into three parts: those at the client only, at the owner only, and shared between client and owner:

- **Client:** P has a reachable surrogate for o if and only if P's object table entry for $o.id$ is *Alive*.
- **Owner:** o is in its owner's object table if and only if $o.dirtySet$ is not empty.
- **Shared (safety):** if the table entry at P for $o.id$ is *Alive* and the owner does not believe P has failed, then $P \in o.dirtySet$.
- **Shared (liveness):** if it eventually remains true that no *Alive* table entries exist for $o.id$ at non-failed processes P then eventually it remains true that $o.dirtySet$ is empty.

The combination of these conditions implies the invariants we gave in section 5.2.1, except that as we mentioned there, maintaining liveness after process failure has forced a weaker safety condition on us.

It is straightforward to prove that the conditions **Client** and **Owner** are maintained by the actions given in the figures above while the respective mutexes *mu* are not held, assuming that weak refs meet the specification in section 5.2.2. The **Shared** invariants are proved via reasoning about timestamps. We just sketch the key areas of the proof here; the interested reader will find more details in [Birrell 93a].

The following lemma will be helpful: the synchronisation over the condition variable *srgtReady* in *CheckPresent* and *EnterSurrogate* (figure 5.3) ensures that there is at most one thread executing in *CreateSurrogate* or *EnterSurrogate* with $id = o.id$, despite the fact that *mu* is not held in *CreateSurrogate*.

The shared safety condition is initially true. To demonstrate that it is maintained, we need two facts:

- when we create an *Alive* entry at P for $o.id$, we must know either that $P \in o.dirtySet$ or that the owner believes P failed.
- when the owner removes P from $o.dirtySet$, we must know either that P has no *Alive* entry for $o.id$ or that P is believed failed.

To show the first, consider the entry's transition in *EnterSurrogate* to *Alive* from the *Nascent* state it was placed in at time *ts* by *CheckPresent*. At this point we know the dirty call in *CreateSurrogate* was successful, and therefore either $ts \leq timestamp\,[o, P]$ or the owner now believes P has failed. Furthermore, no later clean or dirty call has

been generated from P. There was no later clean from *CleanupSrgt* because the guard for enqueuing one was and is false. By the lemma, *Create-* and *EnterSurrogate* have generated no later dirty or clean either. Therefore $timestamp\,[o, P] = ts$ and $P \in o.dirtySet$ at this point, unless P is believed failed. The case analysis in *CheckPresent* ensures that no dirty call with a timestamp greater than ts will be generated while this entry remains *Alive*.

To show the second, consider the removal of P from $o.dirtySet$ by *UnRef* (figure 5.7). If it was *ProcessTerminated* that called *UnRef*, its precondition trivially implies the required fact. If the call came from *ReplyFinished* (figure 5.2), its guard implies that the object table entry is concrete, not *Alive*. Finally, if *Clean* called *UnRef*, the guard in *Clean* implies that ts was fresh, and the guard and deletion in *CleanupSrgt* (figure 5.6) at the client implies that there was no entry at all at ts. There may now be a dirty with a higher timestamp on its way, but it cannot yet have made an *Alive* entry at the client; the entry is at most *Nascent*.

This all suffices to show that $timestamp\,[o, P] = ts$ and $P \in o.dirtySet$ while the entry made *Nascent* at ts is *Alive* and P has not been reported failed at the owner. With the local invariants, the safety of the pinning mechanism now follows.

The shared liveness condition is proved by considering the last transition from *Alive* to *Dead* for an $o.id$ entry at P. We know from the safety proof that the owner's stored timestamp for this entry is from its last transition to *Nascent*. Its final death will eventually cause *CleanupSrgt* to execute. This will enqueue a *CallClean* with a fresh timestamp. Likewise, if the final dirty call from P was reported failed, a fresh clean call was enqueued. The clean will eventually be repeated until it succeeds, the owner has crashed, or there is extended loss of communication, when *ProcessTerminated* will execute at the owner. In either case, P is removed from $o.dirtySet$. Late orphaned dirties do not compromise liveness as we showed in section 5.2.7.

5.3 Implementation

5.3.1 Inter-Process Interfaces

In the description of the collector algorithm above, clean and dirty calls from clients for $o.id$ were directed to the process "*Owner* $(o.id)$". We must now specify in more detail the interface between clients and the collectors at owners for our ANSA implementation.

The collector instance at o's owner manages *o.dirtySet* and its interaction with the local object table. It presents an interface to client collectors consisting of the dirty and clean operations. This is a network object interface specified in IDL in the usual way. How can a client obtain a surrogate for o's collector? If *o.id* identified the owner process and o separately, this would be simple: the collector interface at a process would simply be assigned a well-known object identifier within that process. Making *o.id* an *impure* name [Needham 93] in this way is in fact the technique used in the SRC Network Objects implementation [Birrell 93b].

In the ANSA testbench, interface identifiers cannot (at least in theory) be interpreted like this. However, in version 4, every interface is compatible with `Management`, which provides operations to return one of a fixed set of management interfaces associated with the object. It would be straightforward to add a collector interface to this set, but that would mean another remote call on the receipt of a reference at a clean client, doubling the overhead. Version 4 interface references can also themselves carry additional references to certain other interfaces associated with the object. In an ideal world, its owner's collector would be one of these, but considerations of compatibility prevent us from doing this yet. Instead, we simply make "dirty" and "clean" operations on o itself. To this end, we define an interface type `DGC` (figure 5.9) with dirty and clean operations, and require every automatically collected interface to be compatible with it.

This decision has a number of advantages and disadvantages. The two main advantages are the lower collector overhead and the fact that automatically and manually collected interfaces can be distinguished. This latter is essential if our system is to interwork with the existing ANSA testbench, whose servers and clients know nothing of our garbage collection protocol. Another advantage is that in the implementations of the clean and dirty operations at the owner, which each object inherits from the collector, the concrete object in question is immediately to hand in the "self" argument.

The main disadvantages of the `DGC`-interface approach are that it reveals these operations to the user (who can break the collector invariants by calling or overriding them) and that it interacts with the single-parent subtyping of Modula-3 in a less than ideal way: if we wish to introduce a new type `AB.T <: A.T` and have `AB.T` automatically collected we must already have `A.T <: DGC.T`. We have worked around the first problem with partially opaque types in Modula-3, but this would not extend to other languages. The second problem has not so far arisen in practice.

Various data flow across the `DGC` interface. Client collectors identify themselves by passing references to their local `Capsule`, an ANSA management interface implemented by every

```
DGC : INTERFACE =

NEEDS Capsule;

BEGIN
  Timestamp:   TYPE = ARRAY 2 OF CARDINAL; -- least ... most significant
  Fingerprint: TYPE = ARRAY 2 OF CARDINAL;
  FpTower:     TYPE = SEQUENCE OF Fingerprint;

  dirty: OPERATION [ ts: Timestamp; c: CapsuleRef ]
           RETURNS [ FpTower ];

  clean: OPERATION [ ts: Timestamp; c: CapsuleRef; strong: BOOLEAN ]
           RETURNS [];
END.
```

Figure 5.9: Collector implementation: DGC interface.

process. Since object tables and ANSA capsules are one-to-one, the CapsuleRefs serve as *ProcessIds*. Owners also use calls on client Capsules for death detection. Timestamps for the collection protocol are 64-bit quantities to ensure for all practical purposes that they never overflow. As mentioned in section 4.7, the dirty operation returns an ordered set of type fingerprints.

One other interaction between our collection algorithm and ANSA should be mentioned here. Our requirements for synchronisation between the transmitter and receiver of a reference conflict with the semantics of ANSA announcement (asynchronous) operations. We could maintain the illusion of asynchrony for the programmer who attempts to pass DGCs in announcement arguments, by using a background thread to make a synchronous call, but this conceals overheads for no real benefit. Passing an automatically-collected reference does involve synchronisation, and this should be explicit in interfaces. Attempts to pass references compatible with DGC in announcements are rejected statically by stubm3.

5.3.2 Intra-Process Interfaces

Within every process there are a number of interfaces concerned with the distributed collector: between stub modules and the runtime, and between the runtime and the local collector, the death-detector and the underlying REX RPC protocol implementation.

The interface between client and server stubs and the runtime consists of procedures essentially equivalent to *Export*, *Import* and *ReplyFinished* (figures 5.2 and 5.3). The *Export* procedure has an extra boolean "`result`" argument which is made false in client stubs and true in server stubs. If it is false, the client stub must take responsibility for calling *ReplyFinished*. Otherwise, the runtime makes arrangements with the server-side REX implementation to be informed when it is safe to release the pin list associated with the results' transmission.

Client stubs for operations with `DGC` results must warn the REX implementation to use the modified RPC protocol described below (section 5.3.4) when they initiate their call, and must subsequently inform REX after they have *Import*ed their results, at which point it is known that any dirty calls have returned. The client-side interface to REX allows them to do this.

The local garbage collector in SRC Modula-3 exports a `WeakRef` interface to support object cleanup, which contains `FromRef` and `ToRef` procedures dealing in `WeakRef.Ts`. Attempts to apply `ToRef` to dead weak refs (whose referents have been detected unreachable) return `NIL`. If the cleanup procedure makes a referent reachable again, and a new `WeakRef.T` is created for it with `FromRef`, the new and old `WeakRef.Ts` are unequal, and `ToRef` continues to return `NIL` for the old one. Our collector implementation never attempts to resurrect surrogates in this way, and does not in fact depend on the inequality mentioned. It does depend on the old `WeakRef.T` staying dead.

The last internal interface we mention here is to the death detection mechanism. An `AnsaPinger.T` maintains a set of `Capsules` on which it periodically makes test calls, and provides methods for the runtime to add and remove dirty clients from this set. If a test call continues to fail, the `AnsaPinger.T` removes the failing capsule from its set and makes a callback to the runtime. This method of detecting capsule death has the disadvantage that pings cannot be distinguished at the transport level from ordinary calls, and thus cannot be given a higher priority but lower overhead to help prevent a congested pingee being declared dead prematurely.

5.3.3 Data Structures

How can we represent the state of the collector at a process efficiently? The object table is implemented as a straightforward hash table using the interface reference bound to a network object as its key. Table entries contain both strong and weak reference fields (figure 5.10). There is no need to flag the four possible states of an entry (concrete, nascent, alive and dead) explicitly: they can be encoded by `NIL` entries in the reference fields. Unfortunately, there is no distinguished null `WeakRef.T` in the current implementation. Therefore nascent entries are distinguished by a false "`ready`" bit.

Consider the implementation of *ProcessTerminated* from figure 5.7. It would be unreasonable to search every local dirty set for the dead capsule, so it seems sensible to maintain on each capsule surrogate the set of local objects for which it is dirty. Having made this choice, we notice that whenever we update *o.dirtySet* we in fact have both *o* and *cap* available. The same is true for *timestamp* and *keep*. So we associate a record $\langle\, dirty,\ keep,\ ts \,\rangle$ with the pair (o, cap), and can keep these records in a hash table *cap.exports* indexed by *o*, rather than a table on *o* indexed by *cap*. How can the implementation of *UnRef* (figure 5.7) tell when *o.dirtySet* becomes empty? We simply maintain the size of *o.dirtySet* in an integer *o.refCount*.

A surrogate `Capsule.T` is kept reachable while the client process it represents is not known to be clean by a strong reference from the `AnsaPinger.T` monitoring it. This strong reference can be deleted when *cap.exports* becomes *empty*. It must be retained even if *cap.exports* contains only clean entries (with the *keep* bit set).

These data structures represent those in the collector actions given above as follows:

$$o.dirtySet \;=\; \{\, cap\colon \texttt{Capsule.T} \mid o.ir \in cap.exports \wedge cap.exports[o.ir].dirty \,\}$$

$$timestamp[o, cap] \;=\; \begin{cases} cap.exports[o.ir].ts, & \text{if } o.ir \in cap.exports; \\ -\infty & \text{otherwise.} \end{cases}$$

$$keep[o, cap] \;=\; \begin{cases} cap.exports[o.ir].keep, & \text{if } o.ir \in cap.exports; \\ \textbf{false} & \text{otherwise.} \end{cases}$$

Pin lists are straightforward to represent. There is no need to maintain *o.pins* and *o.refCount* separately: the implementations of *Export* and *ReplyFinished* (figure 5.2) simply increment and decrement *o.refCount* without touching *o.dirtySet*. We can remove *o* from the object table when *o.refCount* becomes zero. The full invariant is

$$o.refCount = |\, o.dirtySet \,| + |\, ClientPins(o) \,| + |\, ServerPins(o) \,|$$

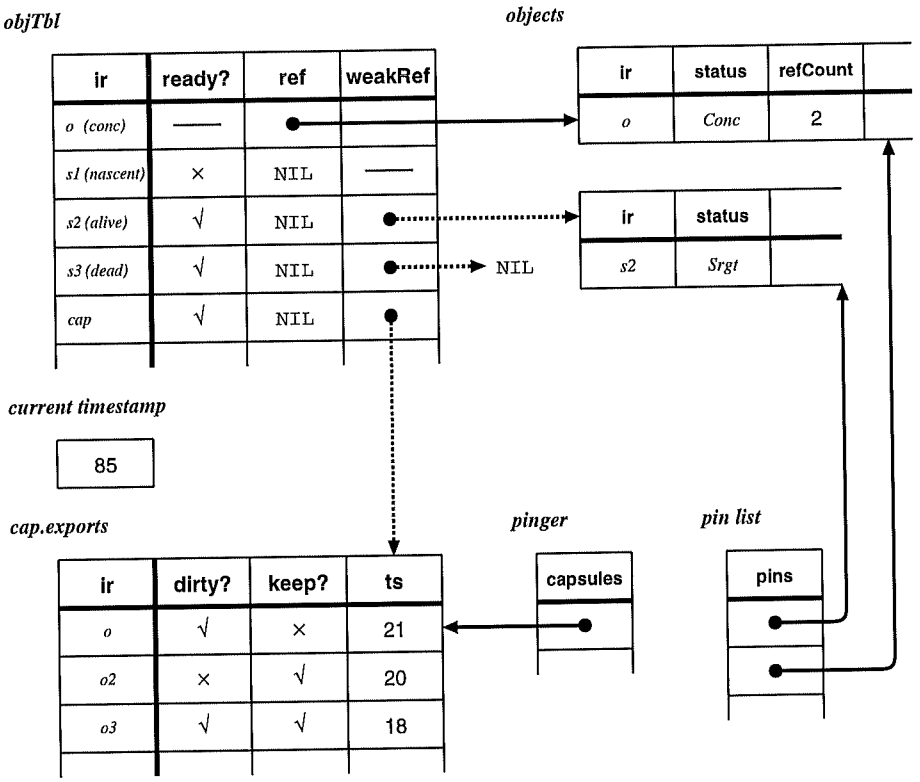


Figure 5.10: Collector implementation: Data structures.

where *ClientPins*(o) and *ServerPins*(o) are the sets of strong references to o from pin lists at its owner, respectively on client-side thread stacks (`result` false) or held by the server-side RPC implementation (`result` true).

5.3.4 RPC Protocol

Suppose a client makes a call to a server that returns network object results. As described above, these must be kept pinned at the server until the server knows that the client has made any dirty calls. We achieve this by a similar mechanism to that used in REX to ensure that the client has received the results in the first place (figure 5.11). In the standard

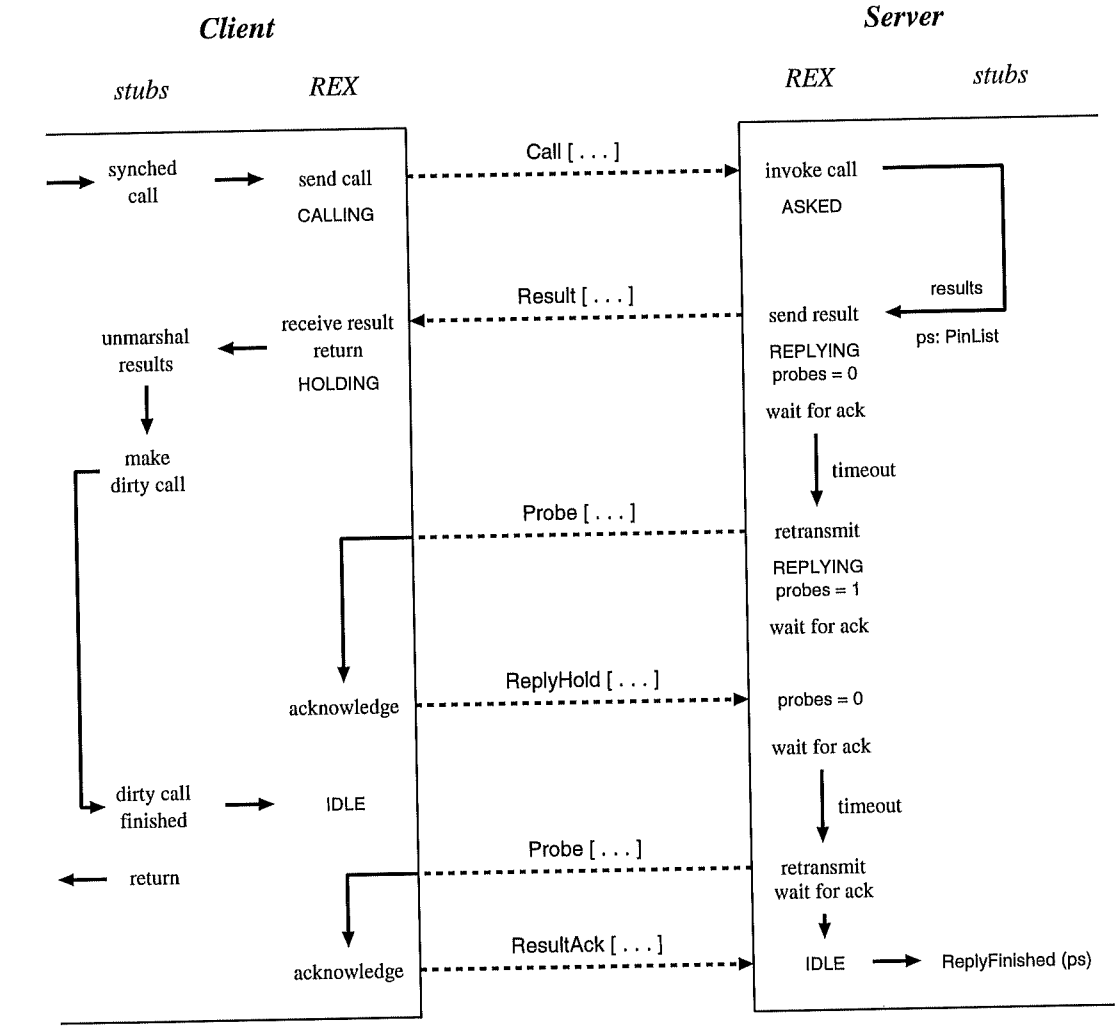


Figure 5.11: Modified REX reply protocol.

mechanism, the server enters a REPLYING state for a call when its results are available. In this state, it sends reply messages to the client (waiting in the CALLING state) until either the reply is acknowledged or some number of retries have gone unacknowledged and the client is deemed unreachable. Replies are acknowledged either explicitly or implicitly by a fresh call message from the client.

All we need do is have the server retry replies indefinitely as long as the client's dirty calls are unfinished, in just the same way as a CALLING client probes indefinitely while the server (in state ASKED) continues to send call acknowledgements as it serves the call. As in that case, the retries need only be single-packet probes rather than full retransmissions, and the retry interval increases to avoid overloading congested clients.

We introduce a new client state HOLDING analogous to the ASKED state at the server. A client call enters the HOLDING state when it receives a reply containing object references, rather than going directly to the IDLE state of the standard protocol. It leaves HOLDING and becomes IDLE when the client stubs have processed the results and know that any dirty calls have finished or failed.

If a HOLDING client receives a current reply probe, it sends a *reply hold* message analogous to the standard call acknowledgement. When a REPLYING server receives such a message, it resets the probe count it normally uses to detect unreachable clients. Thus the server will continue to probe indefinitely as long as the client remains reachable and in the HOLDING state. When an IDLE client receives a reply probe, it sends an ordinary reply acknowledgement which will terminate probing in the standard way.

To maintain the sequence numbers and other protocol state necessary to provide the call semantics of REX interrogations, a client and server share a *session* object and pass identifiers for their respective halves in all messages. A server associates a pin list with its half of a session. When a server session leaves the REPLYING state on receipt of an acknowledgement or a fresh call, it is safe to free the associated pin list. This is initiated by an upcall to the collector. Since the thread making the upcall may be serving the next call from the client, we simply enqueue the pin list for processing by a background thread rather than touching collector state on the new call's critical path. The background thread calls the implementation of *ReplyFinished* (figure 5.2) as dead pin lists become available.

5.3.5 Unsafe Features

We now consider how our collector implementation interworks with the existing ANSA testbench, which does not implement our collection protocol. Programmers dealing with existing services must fall back on manual methods. These are unsafe in the sense that defining and maintaining safety and liveness is the responsibility of the programmer rather than the system. Introducing the DGC interface means that by and large we can keep these worlds separate.

Let us call processes (capsules) which implement our collector *aware*, and processes which do not *naïve*. Then we impose the condition that a process which is a client or server of an IDL interface which uses DGC (either by compatibility or by transmitting data including DGCs) must be aware. This condition can be enforced statically. There are now two sides to the problem: we must decide how aware processes deal with non-DGC surrogate or concrete objects, and how to deal with DGC references which pass through naïve processes.

Aware processes never automatically remove concrete but non-DGC server objects from their object tables. Non-DGC surrogates are still automatically removed from the tables of aware clients after they are detected unreachable by the weak ref machinery. By this time, the programmer has presumably already made arrangements to inform other processes that the surrogate is no longer needed. No clean or dirty calls are generated for non-DGC surrogates.

To allow programmers of aware processes to deal with the collection of non-DGC surrogate and concrete objects, we provide `Discard` and `Withdraw` procedures. These respectively delete surrogate and concrete entries in the local object table. It is a checked runtime error to make an invocation on a `Discard`ed surrogate. `Discard`ing a DGC surrogate triggers cleanup, just as if the surrogate had been detected unreachable by the local collector.

A possible strict safety condition is to require that we never automatically remove a DGC concrete object from its owner's object table if it has ever been passed to a naïve process. We cannot do this in general without such overheads as requiring an aware transmitter to tell the owner before sending a DGC reference to a potentially naïve receiver. Instead, we explicitly require the programmer to be aware that references held by naïve processes will not prevent collection of concrete DGC objects; in this respect, such references are like weak refs. Notice that if this situation arises, the static type of the reference in the interface over which it is transmitted can only be a strict supertype of DGC—that is, it must be `Management` or the generic interface reference type. Otherwise, our awareness condition has been violated.

The prime example of a naïve process with which an aware process must deal is the trader. There is little to be gained by posting DGC service offers to the trader, but programmers might do so to avoid reworking their interfaces when and if it becomes aware. To support this usage, we mark concrete objects which have been exported by their owners to the trader. Marked objects are not removed from the object table when their dirty sets become empty.

As described in section 4.7, we provide a `Narrow` operation to deal with generic non-DGC references. Because of the dynamic type fingerprint mechanism, it is unnecessary (and a checked runtime error) to apply `Narrow` to a DGC, for which the correct "any" type is `DGCRef` rather than the generic interface reference type. In the unlikely event of a DGC passing through a naïve process to an aware one, the receiver can `Narrow` it to a type which `<: DGC.T`. A standard dirty call will be made and the new surrogate, which replaces the narrowee in the object table, will get the richest valid type in the usual way. However, there is no guarantee that the corresponding concrete object is still alive; neither will the non-DGC surrogate that was passed to `Narrow` keep the concrete object alive after the death of the new DGC surrogate. The cleanup for the non-DGC surrogate does not remove the corresponding object table entry in the client, whether that is alive or dead.

5.3.6 Interface Identity

As they stand, our algorithm and implementation rely on interface references to identify interface instances (concrete objects) uniquely and for all time. This allows us to ensure that there is at most one reachable surrogate for a concrete object per capsule. But from version 4 of the testbench, interface references are not (in theory) one-to-one with interface instances [ANSA 92a].

ANSA interface references consist (*inter alia*) of a collection of transport addresses with corresponding protocol identifiers and a 20-byte identifier called a *nonce*. A single interface instance may be accessible at multiple transport addresses, and multiple distinct references to it may exist. Nonces are unique for all time relative to all the addresses at which the interface is ever accessible, but not necessarily globally. The nonces are used in an end-to-end check to detect reused addresses. The testbench regards bitwise interface reference comparison at clients as a hint. References containing equal addresses and nonces denote the same interface instance for all time. References which fail this test may or may not denote distinct interfaces.

To use interface references to index object tables and *cap.exports*, we currently simply hash

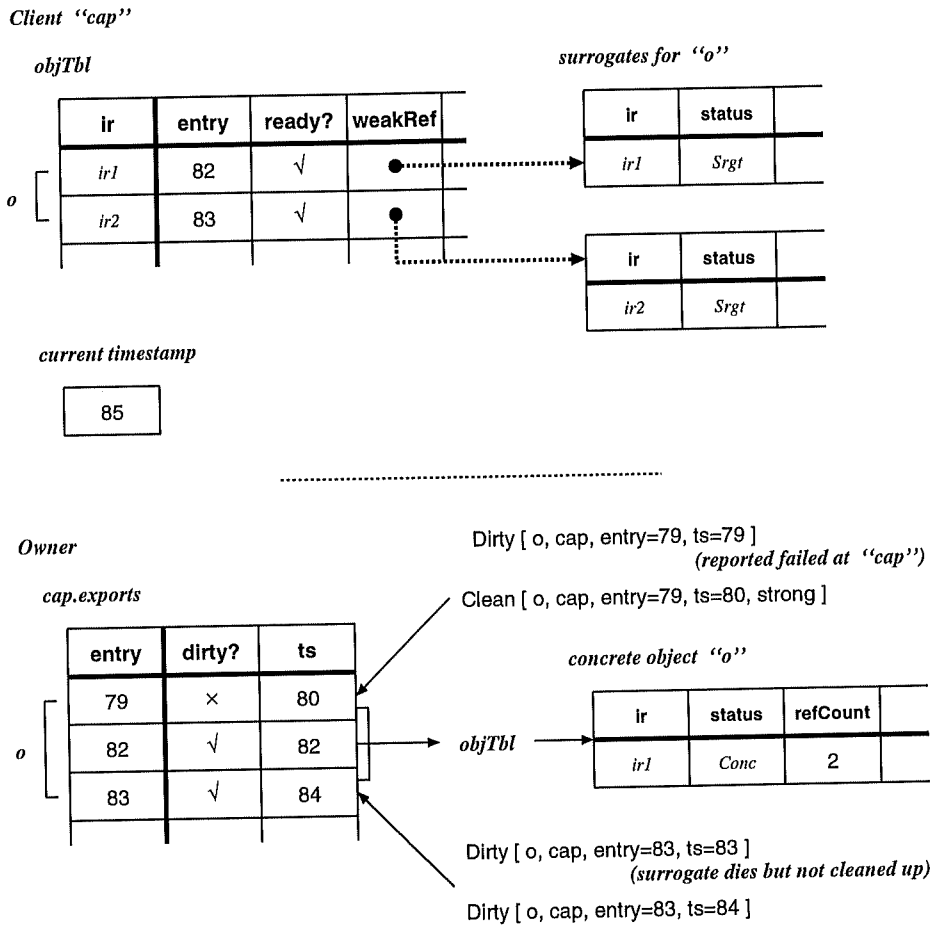


Figure 5.12: Collector data structures without global interface identifiers.

the nonce. This is valid because nonces do in fact currently identify interface instances uniquely and globally, so that interface equality reduces to nonce equality. However, we should describe how to modify our algorithm and implementation to remain correct when interface references have only the properties described above.

Suppose ir_1 and ir_2 are distinct interface references which both denote the concrete object o. The problem arises when ir_2 arrives at a capsule *cap* which already has a surrogate for ir_1. The correctness of our algorithm depends on the fact that the entries in *cap.exports* at o's owner are one-to-one with surrogate object table entries at *cap*, and that these object

table entries are associated with at most one reachable surrogate for o (section 5.2.2). So if we want to make a new surrogate table entry at *cap* (because of the new interface reference), we must simply make a new entry in *cap.exports* at the owner. We have effectively made *o.dirtySet* a multiset.

To carry out this strategy, we must have a means of assigning identifiers to object table entries at *cap*. These identifiers are unique relative to *cap* and are not reused. An identifier is stored in its table entry. All dirty and clean calls for that entry transmit the entry identifier *as well as* the interface reference for o. The object table at the client "*cap*" is still indexed by interface references, but we use the new identifiers rather than o to index *cap.exports* at the owner. That is, the members of *o.dirtySet* are now pairs $(entry, cap)$ rather than capsules alone. The "$|\, o.dirtySet\, |$" part of *o.refCount* is now kept equal to the total number of dirty *exports* entries for o in all *cap*s at the owner.

It is not necessary to store a reference to o in its *cap.exports* entries to maintain this invariant, provided the client can be trusted to associate interface references and entry identifiers consistently: the correct o is already to hand in the dirty or clean call arguments. Also, notice that the *keep* bit is no longer needed in *cap.exports* entries. The next dirty call for ir_1 from *cap* after a strong clean call must create a fresh object table entry at *cap*, and hence a distinct entry in *cap.exports*. A strong clean call merely clears the *dirty* bit in the *cap.exports* entry, without deleting it as a normal clean would.

There remains the question of how *cap* should allocate identifiers for its object table entries. A given entry for *ir* is uniquely determined by the transition at its creation from the state $ir \notin objTbl$ to the state $objTbl\ [ir] = Nascent$. We could keep a separate counter to label these transitions, but of course the timestamp counter at *cap* is perfectly adequate. It is important to emphasise that when the *CheckPresent* action of figure 5.3 finds a *Dead* entry, the *stored entry identifier* already present must be used but a *fresh timestamp* must be generated for the subsequent dirty call. In other words, the two mutually exclusive disjuncts of the postcondition of the **while** in *CheckPresent* require different actions. Timestamps now have a dual function: they order the states of surrogate table entries as before, and additionally label individual entries. This latter function requires a shared counter for the whole object table. Figure 5.12 summarises the modified data structures.

I do not set my life at a pin's fee

— HAMLET (Act I, Sc. 4, line 65)

The ears are senseless that should give us hearing
To tell him his commandment is fulfill'd
That Rosencrantz and Guildenstern are dead

— 1st AMBASSADOR, Hamlet (Act V, Sc. 2, line 75)

Chapter 6

Evaluation and Performance

The author has implemented and tested everything described in Chapters 3, 4 and 5 above, with the exception of the modifications to the distributed garbage collector mentioned in section 5.3.6, which have not yet become necessary. This chapter reports on experiences building applications with our implementation, describes some of the tests we made of it and presents some simple measurements of its performance.

6.1 Applications

The author and others in the Laboratory have constructed various Modula-3 application programs which use our network objects system.

The `m3coffee` program is a client of a pre-existing ANSA service (`GFrame`, by Paul Jardetzky and Quentin Stafford-Fraser) which provides remote access to a video frame grabber attached to a camera which watches a communal coffee pot. It displays the received images with Trestle, a Modula-3 window system [Manasse 91]. Although `m3coffee` was of little value in itself, it provided a thorough test of our system and its ability to interwork with the rest of the ANSA world. The Modula-3 client ran over UNIX on a DECstation and communicated via an implementation of the MSNL internetworking service [McAuley 90] with the C/DPL server capsule running over the Wanda operating system [Dixon 92] on a 68020-based VME crate. Since Trestle was originally designed to run on a multiprocessor, it makes heavy use of Modula-3 threads. The stress this placed on our integration of ANSA tasks with Modula-3 threads (section 3.4.5) revealed many concurrency bugs.

Component		Size	Language
Tools			
	`stubm3`	6,209	M3, C, `yacc`
	`dgcmon`	1,328	M3, IDL, FormsVBT
	DGC logging	256	M3, shell
	total	7,793	
Runtime Library (not including existing ANSA testbench capsule code)			
Network Objects:			
	interfaces	2,708	M3, IDL
	modules	4,579	M3
System-dependent capsule code		4,036	M3, C
	total	11,323	
Tests			
	Misc. tests	1,410	M3, IDL
	Distributed gc	1,187	M3, IDL
	`m3coffee`	219	M3, IDL
	Timing	2,042	M3, IDL, C
	total	4,858	

Figures are lines of source code reported by `wc(1)`.

Table 6.1: Sizes of implementation components.

Another program combining use of our system, Trestle and existing ANSA services is `tab`, Peter Robinson's Trestle Active Badge monitor. Users of this program select from menus the names of people whose current whereabouts they wish to see at a glance. For each such person, `tab` creates and registers a network object with the ANSA server which represents the active badge system. Infra-red signals from the badges [Want 90] then trigger announcement invocations with location data to the callback objects in `tab`, which updates its display accordingly.

When `tab` was written, the code generated by `stubm3` to unmarshal enumeration types (section 3.4.2) performed no range checking, under the (misguided) assumption that static typechecking of the transmitting program would prevent illegal values from being marshalled in the first place. Part of the sighting information transmitted in the callbacks to `tab` from the badge system was an enumeration describing the state of a button on the badge. It turned out that framing errors in the badge network could occasionally cause this value to be corrupted without detection. The badge system servers, written in C, were quite happy to pass these corrupted values on to `tab`, which would then crash when trying to inject them into a Modula-3 enumeration type. Our mistake was to assume too much about the transmitter (section 4.4).

To help test and debug our distributed garbage collector we constructed the `dgcmon` program, whose user interface (which uses the FormsVBT toolkit for Trestle [Brown 92]) is shown in figure 6.1. The program displays the collector state at selected capsules, and allows the user to interact with it in various ways. To inspect the state of a given capsule, the user selects it from a list in `dgcmon`'s *Chooser* window. The chooser collates this list by querying the ANSA trader.

A monitor window for a target capsule has two main areas, labelled *Imports/Exports* and *Capsules*. The first of these contains one line for every object table entry at the target, showing the entry's kind (concrete or surrogate), the nonce component of the corresponding interface reference, the name of the surrogate or concrete object's Modula-3 type and, for a concrete object, its total reference count. The user can instruct the target collector to behave as if a surrogate had died by selecting it and clicking on the *Kill Surrogate* button. The capsules area displays the nonces of each of the capsules for which the target has a surrogate. The *Kill Cap* button can be used to simulate detection by the target of the selected capsule's death.

Entries in both areas are annotated with their status with respect to an object or capsule selected by the user. Capsules are marked with a "`*`", "`c`" or "`o`" if they are dirty or clean for a selected concrete object, or own the referent of a selected surrogate, respectively.

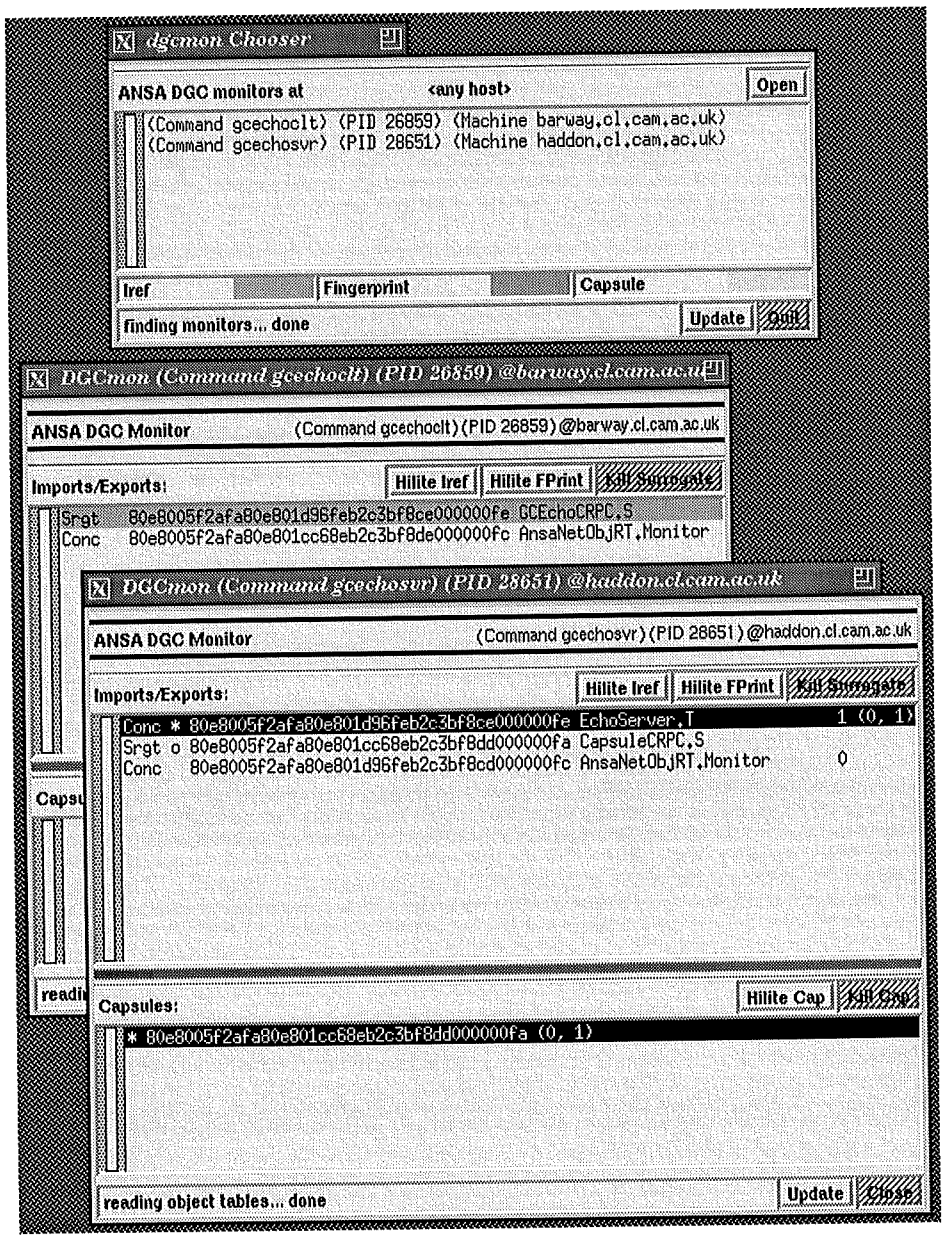


Figure 6.1: DGCmon: a tool to monitor distributed garbage collector state.

The timestamps recorded for a selected concrete object owned by the target are displayed in the form (n, m) at the end of each entry in the capsules area. The object table entries in the imports/exports area are similarly annotated with respect to the currently selected capsule.

Monitor windows obtained from the same chooser cooperate to highlight entries with equal interface references or type fingerprints. The chooser window contains a key for the various highlight colours. Figure 6.1 shows monitors for a simple client/server pair: the client holds a single surrogate of type `GCEchoCRPC.S` for a concrete `EchoServer.T` in the server. The user has chosen to highlight all the surrogates for this server object.

To read and update a collector's state, the `dgcmon` program depends on the `DGCMon` network object interface, for which every collector implements a server. If a special command line argument is present when an aware program starts, it exports its `DGCMon` interface to a trader offer context reserved for this purpose, with enough additional properties (such as host name and process id) to allow the user to identify the program instance from the list presented in `dgcmon`'s chooser window. The `DGCMon` interface provides operations to support the *kill surrogate* and *kill capsule* features and a `dump` operation whose implementation makes a marshallable copy of a consistent state of its local collector, obtained under the collector lock, and returns it to the caller. Notice that this involves redundant copying of this state: once to create the consistent marshallable value, and once to marshal it into the reply buffer. The extra overhead is not a cause for concern in this case, because the operation's performance is not critical and the local garbage collector relieves the programmer of the chore of freeing all the memory used for the copy. However, it is certainly possible to imagine situations where for performance reasons data should be marshalled directly under a user-level lock[1]; customised stubs are then required.

Our system has been used in a number of final-year undergraduate projects in the Laboratory. These have included

- a multi-party "talk" system whose central server manages a collection of multi-way conversations between users' client programs (C. L. Harding);
- a multi-user shared "virtual space" whose users interact via virtual objects in a central server which manages concurrency control for the space (A. Gibson);
- an automatic diary generator for active badge wearers which detects and logs its users' meetings with other badge wearers; users browse their diary with a FormsVBT

[1] I owe this observation to a remark in another context by Sai-Li Lo.

client of their diary-generator service (P. Möckel);

- an adaptation of `stubm3` to generate network object stubs for an implementation of the lightweight MSRPC [Crosby 93] transport (C. T. Charlton); and
- a kernel for a Time Warp distributed simulation system: simulation nodes run in separate processes and communicate with messages tagged with virtual simulation times; optimistic concurrency control is used: a central master maintains global virtual time and sends rollback messages to nodes when conflicts are detected; communication amongst the nodes themselves and between nodes and the master is layered on top of network objects RPC and makes use of the sequencing this implies (J. S. Grewal).

It is noticeable that most of these projects have a simple structure with many clients and a single central server; though some of the others do pass callback references from clients to servers, only the distributed simulator really makes good use of the ability to configure a system by passing network object references around.

6.2 Tests

6.2.1 Basic Functions

It was straightforward to test those areas of our system which are not directly concerned with distributed garbage collection. A client and server of an `Echo` interface exercise interaction with the trader, binding to a service and marshalling of IDL's base and constructed types, possibly between machines of different endian-ness. A client and server of a pair of related interfaces `AB <: A` exercise the use of type fingerprints as offer properties in the trader: the server exports an `AB`, while the client imports first an `AB` and then an `A`, checking the equality of the resulting surrogate references. Modifying `A` and `AB` to make them compatible with `DGC` makes a similar test of the dynamic typing mechanism possible: in this case, the client imports the service as an `A` first, then as an `AB` before checking surrogate equality. The other cases in figure 4.1 can be tested by varying which stubs are linked into the client and server. The `Narrow` mechanism (section 5.3.5) is exercised by the `dgcmon` program described above: the result of searching the trader for registered instances of the `DGCMon` service is a collection of generic `AnsaNetObj.Ts`. The runtime's hybrid implementation of pseudo-concurrency has been best tested by the Trestle/Active

Badge systems mentioned above, some of which have involved servers with lifetimes on the order of a week. That said, concurrency bugs and storage leaks undoubtedly remain.

6.2.2 Distributed Garbage Collector

Testing the distributed garbage collector was less simple. Our strategy was to create, pass and drop many references to a small number of `DGC` instances concurrently between a single owner program and a number of clients. Each program produces logs of its actions during a test run, and checks that its final collector state is correct.

When the owner starts, it allocates two `DGC` objects, the *pushee* and the *pullee*. The pushee will always be passed as an argument of a *push* call from the owner to a client, while the pullee will be passed as a result of a *pull* call from a client to the owner. Once the specified number of clients have contacted the owner, it makes a call to each to start a round of testing and itself forks a specified number of worker threads per client to execute push calls.

When a client starts, it contacts the owner and waits to be triggered. It then forks a number of threads to execute pull calls; the surrogate references these return are immediately dropped on the floor, as are those received when serving push calls. Throughout a test, a background thread in each client periodically forces the local Modula-3 garbage collector to run in order to provoke weak ref cleanup actions. Alternatively, if an appropriate flag is set, the clients explicitly `Discard` (section 5.3.5) surrogate references as they receive them. This is useful to ensure that a large number of races occur.

Once the owner has been informed that all the push and pull calls have been completed, it pauses for some time to allow any pending collector actions to execute. It then declares the round finished; the owner and all the clients inspect their object tables via the local `DGCMon` instance and report a failure if they contain entries for the pushee or pullee objects.

Our collector implementation is instrumented to produce logs of its actions if a special flag is set. The log records indicate the type of event they represent, the time according to the UNIX time of day clock and any other useful parameters, such as timestamp values (local or received) and the identities of the objects and capsules involved. By manually comparing logs from the various parties in a round of testing, it was possible to track down the sequence of events leading to a problem. We would have liked to merge logs from different processes automatically, but this was problematic. Ideally we would have used the timestamp information to recover the true causal partial order of events, but doing this

correctly is of comparable difficulty to constructing a correct garbage collector! Logs from processes on the same machine were merged on the basis of their time of day values; this was useful, but suffered from the fact that several events from different processes could occur in a single clock grain (3.906 ms on a DECstation 3100). We experimented with recovery of clock offsets between different machines by exchanging synchronisation calls, but to no great advantage. Despite these problems, the logs were invaluable debugging tools: `dgcmon` was helpful for general monitoring and to test the capsule death detection machinery, but could give no clue as to the cause of erroneous collector states.

As well as exposing the expected bugs in our threads package, the races between clean and dirty calls provoked by these tests showed up several flaws in earlier versions of our collector implementation. The problems were all caused by a failure to keep the asynchrony of pending clean calls in mind: assertions in our implementations would fail in perfectly legal states, such as when *CleanupSrgt* (figure 5.6) finds no dead object table entry, or when two fresh dirty calls arrive at the owner with no intervening clean call. The collector tests incidentally revealed a Modula-3 compiler bug and an error in the handling of deleted entries in a library hash-table implementation.

Our distributed garbage collector relies on capsules' local collectors to detect that a surrogate has become unreachable. Conservative local collectors, such as that in the SRC Modula-3 runtime, can fail to detect all local garbage. The tests described above demonstrated that this did indeed happen. The local collector's design is based on the mostly-copying collector of [Bartlett 88] and has the great advantage that ordinary C code can be linked into a Modula-3 image, as we have done. However, when this collector encounters a word on a thread stack which might be a pointer into the traced heap, it conservatively retains all heap nodes on the entire heap page apparently referenced. Examination of the stacks and heaps of the participants in the test of the distributed collector showed that the surrogates were allocated on the same pages as other small objects. Some of these, such as the descriptors for the worker threads themselves, had long-lived references from stacks, effectively preventing the surrogates on the same page from ever being collected (unless they were explicitly `Discarded` while still reachable). This problem is a result of what [Demers 90] calls "card pollution". It could be cured by adding to the existing mostly-copying collector a restricted sweep phase which examines only pages promoted because of direct references from the stack, or circumvented by adding application hints in the form of *heap zones* to the current one. In the second case, the idea would be to allocate surrogates on pages in a special heap zone and to avoid keeping direct references from stacks into this zone for significant lengths of time. (The techniques of [Demers 90] and [Boehm 93] for avoiding card pollution apply mainly to non-copying collectors.)

6.3 Performance

6.3.1 Environment

To assess the performance of our implementation, we ran a number of simple timing experiments between a single client and server of a `Null` interface, exercising a different operation in each experiment. The client and server programs ran in separate machines, communicating via REX over UDP over a common Ethernet segment. Each machine was a DECstation 3100, based on the MIPS R2000/2010 processor and FPU clocked at 16.67 MHz, with 16 MB of physical memory and a local disc for swapping and paging, running the Ultrix V4.2A operating system.

During the tests, no other programs apart from the Laboratory's standard daemons were running. No significant swapping or paging of the server or client programs was observed. In Modula-3 tests, the SRC Modula-3 v2.11 environment was used, with thread preemption enabled and the standard VM-synchronised incremental generational local garbage collector running concurrently with the test code. The C code produced by the SRC Modula-3 compiler is compiled with `cc -g`. In C/DPL tests, the standard ANSAware 4.1 environment was used. The C programs produced in this environment are compiled with `cc`, enabling the default optimisations.

The minimum latencies reported below can only be regarded as accurate up to the grain of the UNIX `gettimeofday(2)` clock used for timing, which is 3.906 ms on the machines used. The mean latencies are derived from accumulated times and are correspondingly more accurate. They were reproducible to within 5% or better.

6.3.2 Experiments

Each experiment we carried out had the same general form. The server program starts, exporting an offer for the `Null` service to the trader. The client program then starts, imports this service instance, and iterates some operation a number of times. Each operation consists of measuring the time taken to perform some action, accumulating this time into the running statistics, and optionally restoring the client and server to some state.

Table 6.2 reports the latencies observed for invocations of operations with no arguments or results for a Modula-3 client/server pair with our system, and a C/DPL pair using the same interface. Both interrogation and announcement (asynchronous) operations were

	Interrogation		Announcement	
	M3	C/DPL	M3	C/DPL
minimum	7.8	3.9	—	—
mean	12.2	6.6	2.6	1.7
std. deviation	5.0	3.7	2.5	2.3

Latencies of 10,000 successive invocations of null operations as seen at the caller. Figures are times in milliseconds. For comparison, invocation of a null local Modula-3 method takes around 1.7 microseconds under the same circumstances.

Table 6.2: Performance of remote operation invocation.

tested. In both cases, the time measured is from the call to the return of the operation at the client; there is no "restore" action, so any internal caches remain warm. During null interrogations we observed roughly 33% user, 33% system and 33% idle CPU states at both client and server. During announcements, there were roughly 50% user and 50% system states at both. In the Modula-3 case, the local garbage collector went through 3 collection cycles at the server and none at the client.

Table 6.3 reports the latencies of operations with a single argument (push) or result (pull) of the generic interface reference type which denotes a server with no operations owned by the transmitter. In our Modula-3 system, this is an object reference of type `AnsaNetObj.T`. In C/DPL, it is an `INTERFACEREF`. In the columns headed "Clean Receiver", the "restore" action caused the receiver to discard its reference, removing it from any internal tables. Both client and server went through 1 local collection cycle during each of the Modula-3 tests.

Table 6.4 reports the latencies of push and pull calls with a single argument or result of type `DGC.T` owned by the transmitting Modula-3 program, thus exercising our distributed garbage collector. As above, in the columns headed "Clean Receiver", the "restore" action caused the receiver to discard its reference. In addition, a delay was introduced in the "restore" action to ensure that the resulting clean call generated by a background thread in the receiver was processed by the owner before the next push or pull call was initiated. Inspection of collector logs allowed us to verify that this happened. In this situation, each push or pull call finds the reference to be transmitted absent from the object table at

	Clean Receiver				Dirty Receiver			
	Push		Pull		Push		Pull	
	M3	C	M3	C	M3	C	M3	C
minimum	15.6	7.8	15.6	7.8	15.6	7.8	15.6	7.8
mean	20.1	10.1	20.8	10.3	19.5	10.0	20.0	10.2
std. deviation	4.0	3.1	3.4	4.2	4.8	3.3	3.8	4.8

Round-trip latencies of 1000 transmissions of a Modula-3 `AnsaNetObj.T` network object reference or a C/DPL `INTERFACEREF`. Figures are times in milliseconds.

Table 6.3: Performance of interface reference transmission mechanisms.

	Clean Receiver				Dirty Receiver			
	Push		Pull		Push		Pull	
	DGC	ANO	DGC	ANO	DGC	ANO	DGC	ANO
minimum	46.9	15.6	50.8	15.6	19.5	15.6	19.5	15.6
mean	67.9	20.1	63.2	20.8	22.1	19.5	22.5	20.0
std. deviation	16.4	4.0	15.6	3.4	4.5	4.8	4.2	3.8

Round-trip latencies of 1000 transmissions of a Modula-3 `DGC.T` or `AnsaNetObj.T` network object reference from its owner. Figures are times in milliseconds; the values for `AnsaNetObj.T`s come from Table 6.3.

Table 6.4: Overhead of the distributed garbage collector on reference transmission.

both the owner and the receiver, which must call in dirty. In the columns headed "Dirty Receiver", there is no "restore" action and the transmitted reference is already in both object tables. In all cases, there are 5 ANSA worker threads at each process, more than sufficient to serve all nested calls. The client and server went through 17 and 10 local collection cycles respectively while pushing 1000 `DGC.T` references to a clean receiver.

6.3.3 Discussion

Research into techniques for high performance RPC was not one of our goals: the figures in Tables 6.2 and 6.3 certainly bear this out! For comparison, [Thekkath 93] reports a null call time of 170 *micro*seconds for a high-performance RPC system on an ATM network using DECstation 5000/200 hosts. For null operations and announcements, and when transmitting ordinary interface references, our system has about half the performance of the standard testbench. This is probably caused by a combination of the use of C code as an intermediate language by the Modula-3 compiler, the lack of default optimisations when compiling that C, and the overhead imposed by our merger of the Modula-3 and ANSA capsule runtimes (section 3.4.6).

According to `pixie`, a basic-block profiler, time spent in the Modula-3 scheduler and `MUTEX` locking routines (but excluding transfers between threads and management of masks for `select(2)`, which also occurs in the C/DPL tests using the standard capsule runtime) accounts for 19.8% of instruction cycles in a run of 10,000 null operation invocations, and 36.0% of cycles in a run of 1000 calls pushing a `DGC.T` to a clean receiver. The standard capsule runtime is compiled on the assumption that tasks run to completion; locking calls are unnecessary and are pre-processed out.

The initial performance of our implementation was even worse: we were spending 64% of our cycles in a routine which searches the mask returned by `select(2)` for the next ready file descriptor. On the machines we used, these masks contained 512 bits, only a few of which were ever likely to be set. To ensure remote processes are treated fairly each search starts one bit further along the mask, so on average 256 bits had to be scanned each time. The reported time spent in this routine was reduced to less than 0.43% simply by adding code to count off each ready bit, leaving the scan loop when none remain, and maintaining a high-water mark, at which the scan wraps around.

We would expect transmission of a `DGC.T` reference to a clean receiver to take slightly more than twice as long as to a dirty receiver, allowing for the nested dirty call and the creation of a surrogate of the appropriate type. Table 6.4 shows that the factor is nearer

3 for 1000 iterations. The distributed garbage collector logs show that the problem is not delayed clean calls from the "restore" actions overlapping with later transmissions. Neither is the local garbage collectors' activity (due to the accumulation of dead surrogates) to blame: the timings are not significantly affected by switching them off, and `pixie` profiles do not show significant numbers of cycles spent there. However, varying the number of transmissions in ten steps from 50 to 2000 causes the mean time *per push call* to vary linearly from 53 ms to 80 ms. The extrapolated time for a single push call is 52 ms, which is more in line with our expectations.

On further investigation, we discovered that the main source of the increasing delays was time spent in the ANSA capsule code arming and disarming timers on REX session objects (section 5.3.4). When a client or server session becomes IDLE, a three-minute *decay* timer is armed; if the session subsequently leaves the IDLE state, its decay timer is disarmed. If a session is still IDLE when its decay timer expires, it is reclaimed; the length of the decay timeout is sufficient to ensure that no orphan packets for the session will arrive later. In our `DGC.T` transmission tests, many such decaying sessions accumulate: each iteration exports the reference from its owner afresh, and results in the allocation of new sessions at the owner and receiver for the dirty and clean calls. At worst, there are three minutes' worth of outstanding decay timers. This costs time as well as space because of the nature of the timer queue implementation: a doubly-linked list in order of expiration with insertion by linear search from the head of the queue. Because of its duration, arming a decay timer is very likely to scan the whole queue. One would expect disarming a timer (removing it from the queue) to take constant time in a doubly-linked list implementation, but the capsule code first checks by linear search that the timer to be removed is actually on the queue before deleting it. Time spent disarming and arming the decay timer on the session of the transmitting call itself is on the critical path for our measurements; management of the other sessions' timers also affects us because of increased contention for the global capsule lock.

There are various ways to improve the situation. The length of the decay timeout relative to others (such as for retries) means that when a timer is armed, it is likely to be inserted near either the front or the end of the outstanding timer queue; the implementation could be made more efficient for the second case. When a reference is removed from the object table at either its owner or a client, its IDLE sessions could be reclaimed immediately, rather than waiting for them to decay. We do not currently do this because a session is required at both owner and client for the consequent clean call. Calling in clean on the reference itself (as in our `DGC` interface) is probably a mistake: it would be better for dirty calls to return a separate reference to the owner's collector, on which all clean calls to that owner are made. Then IDLE client sessions for a reference could be discarded in the

action that acquires the timestamp for the following clean call (*CleanupSrgt*; figure 5.6), and IDLE owner sessions in *UnRef* (figure 5.7). This is a question of performance rather than correctness: sessions will eventually be reclaimed by the decay mechanism in any case.

Guil: I think we can say we made some headway.
Ros: You think so? ...
Guil: He might have had the edge.
Ros (roused): Twenty-seven–three, and you think he might have had the edge?! He murdered *us. ...*
Guil: He had six rhetoricals— ... And two repetitions. ...
Ros: Six rhetorical and two repetition, leaving nineteen, of which we answered fifteen. And what did we get in return? He's depressed!
— TOM STOPPARD, *Rosencrantz and Guildenstern Are Dead* (1976)

Chapter 7

Related Work

This chapter places our system in the context of recent work on distributed object-based systems and distributed garbage collection.

7.1 Distributed Systems

7.1.1 SRC Network Objects

Our system was designed and developed concurrently with (and, with the exception of the distributed garbage collector, independently of) the SRC Network Objects system [Birrell 93b]. The starting point for both systems was the proposal in [Nelson 91a] which we reviewed in section 3.1. Both systems use language-level Modula-3 objects to represent service interfaces and their client-side bindings. Both rely on automatically generated stubs. However, in the SRC system, the input to stub generation is a pure object type definition in Modula-3. Thus Modula-3 is a privileged language. This allows simple implementations of some pleasant features: for instance, stream objects suitable for bulk data transfer can be transmitted between programs, and stubs can call on pickling machinery (section 3.4.2) to marshal complex data, including graph structures.

By contrast, our system requires the programmer to define services in an external (and slightly more restrictive) IDL. The benefit we gain from this is the ability to interwork with the existing ANSA testbench and the possibility of extending the network objects

approach to other programming languages. Despite the use of IDL, our system presents the heterogeneous ANSA environment to the Modula-3 programmer very nearly as transparently as the Modula-3-only SRC system. We achieve this by having the stub generator produce *object type declarations* as well as stubs in the target language.

Both systems approach the typechecking problem in similar ways. Since the SRC compiler uses type fingerprints to achieve type-safe separate compilation for Modula-3, the extension of fingerprints to the distributed environment is a natural step. The use of IDL again distinguishes our system from SRC Network Objects: our fingerprints are in theory calculated from expanded IDL definitions, and can therefore be used for cross-language typechecking. However, we must admit that our current implementation simply uses the fingerprints calculated by the SRC Modula-3 compiler from the network object type declarations which `stubm3` generates from IDL specifications.

As we have mentioned, essentially the same distributed garbage collector [Birrell 93a] is used in both systems. The integration of this collection strategy with the heterogeneous ANSA world was the topic of the second half of Chapter 5.

Rustan Leino has implemented a version of the Modula-3 language with extensions for distribution which are based on network objects [Leino 93]. The Modula-3D system runs on the Caltech Mosaic multicomputer, which currently has 256 processing elements, each consisting of a custom processor and 64 KB of memory. This environment allows the possibilities of communication and process failure to be ignored. The `NEW` pseudo-procedure is extended so that objects of type `<:NETWORK` can be created at a designated node, possibly returning a local surrogate. The node on which object `o` resides is returned by the runtime call `Processor.Of(o)`.

7.1.2 Distributed Computing Environments

Chorus/COOL v2 [Lea 92], the CHORUS object-oriented layer, is made of three slices:

- *COOL-base* is a veneer over the CHORUS kernel (section 2.3) which provides *clusters*. A cluster is a distributed (coherent) virtual address space backed by secondary store and shared by a number of closely-coupled homogeneous nodes. These must cooperate in the allocation of ranges of virtual addresses to clusters. Such virtual addresses can be passed in invocations among the processes in which a cluster is mapped without further indirection.

- The *COOL-generic* runtime implements objects in clusters independently of programming languages. It manages dynamic linking of code for classes into clusters. Domain-wide object references name objects uniquely and for all time. They are converted into language references (virtual addresses) by creation of *proxies*, analogous to our surrogates, which perform RPC multiplexed over CHORUS ports and message passing.

- *Language-specific* runtime libraries cooperate with the generic runtime via an upcall mechanism to perform such tasks as pointer swizzling (replacement of object references with proxies) when mapping a cluster in from secondary store.

COOL v2 thus provides operating system support for distributed object based systems with multiple persistent heaps. Its approach to concurrency control, failure handling, stub generation, cross-language typechecking and garbage collection is not discussed in [Lea 92]. Our approach seems quite feasible in the COOL environment, given kernel support for synchronisation of access to memory in clusters.

OSF/DCE [OSF 91], the Open Systems Foundation's Distributed Computing Environment, provides threads, RPC, a directory service, clock synchronisation, secure and authenticated communication and a distributed file system. It is currently implemented on the OSF/1 operating system for the C language using the UDP and TCP Internet protocols. RPC interfaces are specified in an IDL. An IDL specification includes a mechanically generated *universal unique identifier* or UUID and major and minor version numbers. These three values uniquely identify the interface specification in time and space. They thus act as a canonical type name in the sense of section 4.5. The binding between these names and the interface specifications in which they are embedded must be managed by cooperation between programmers. Compatibility between interfaces is also managed by programmers: during binding, interface A is deemed to be compatible with interface B if their UUIDs and major versions are equal and A's minor version is greater; the actual contents of the respective interface definitions are not taken into account.

RPC binding in OSF/DCE is a two stage process. The interface types regularly offered by a given host are registered in a global directory service, whose contents are expected to change infrequently. This allows widespread replication of the directory with relatively weak consistency protocols. Having obtained the identity of a likely host from the directory, a client contacts a port-mapping daemon process at that host to obtain a full binding to a currently available service instance. New service instances cannot be created and bound to without registration at some level in this name service hierarchy. However, a server can issue a client with *context handles*: these are special tokens for portions of

the server state. In order to simplify resource management, such handles can only be used by the original client: they cannot be passed to third parties. In our system, interface references (viewed by programmers as language-level objects) subsume this function. Our network objects are more flexible: they can be passed freely between processes and are supported with distributed garbage collection and a dynamic typechecking mechanism.

OMG CORBA [OMG 91], the Object Management Group's common object request broker architecture, is a framework for object-based systems similar in approach to ANSA. However, it places less emphasis on the practicalities of distribution and communication: these have been abstracted away almost completely. (One has to look quite hard at the CORBA specification to find an occurrence of the acronym "RPC".) Objects (analogous to ANSA interfaces) are named by object references, via which invocations can be made. CORBA IDL is similar to ANSA IDL, though its syntax resembles C++ rather than Mesa. Values transmitted in invocations may be object references. The IDL allows exceptions and `RAISES` clauses to be declared in a style similar to Modula-3, and has an ANY type which can transmit any value.

IDL interface specifications (once compiled) are themselves objects which are stored in the *interface repository*. Client programs can interpret stored specifications and construct requests dynamically without requiring client stubs to be generated ahead of time. Server stubs are still required. Types are described by structured values called TypeCodes. Thus CORBA uses a combination of the second and third techniques mentioned in section 4.5. TypeCodes are essential to the interpretation of values of the ANY type.

To create an object reference, a local object identifier and a reference to the interface specification in the global interface repository is supplied to the runtime. No language-level sugar is defined for this, though it would be possible for an IDL mapping for a language like Modula-3 or C++ to adopt a network object approach like ours. The semantics of type checking are not defined in [OMG 91]. Dynamic typechecking when passing object references would appear to require interaction with the global interface repository to obtain the relevant TypeCodes.

The Esprit Comandos project has a scope and approach broadly similar to that of the OMG CORBA. The **Comandos Type Manager** (TpM) [Campin 91] is intended to act as a repository for elaborate type structures which are created and interpreted by the compilers and runtime libraries of a number of programming languages. Rather than adopting the "lowest common denominator" approach of most IDLs, the TpM type system is intended as a universal type calculus. It supports all the features of the most sophisticated polymorphically-typed lambda calculi, up to and including bounded existentially

and universally quantified recursive dependent types. Type compatibility is defined by the contravariant conformance rules traditional in such calculi. The TpM is intended to perform compatibility checking and conversion of values between languages via a canonical form. This appears to require the cooperation of the logically central TpM service when marshalling; it is not clear how (or whether) values of abstract types are to be transmitted.

The ANSA distributed programming language, DPL [ANSA 93], is a notation for programming in accordance with the ANSA Computational Model [Rees 93]. It is not to be confused with the C/DPL embedded language described in section 2.6.3. DPL is an object-based language with similar facilities to Emerald, but with support for heterogeneity and the handling of failures with language-level exceptions called *named terminations*. DPL, like our system, supports the dynamic creation of services. However, it achieves this in a slightly different way. DPL has language-level constructors for *interfaces* which are analogous to our concrete objects. These capture state in closures formed according to the block structure and scope rules of the language. As in our system, interface references may be freely passed and there is a transparent object-based syntax for invocations, whether local or remote.

Abstract interface types are defined with a `type` constructor which subsumes the rôle of the ANSA testbench IDL in hiding heterogeneity. Conformance between these abstract types is defined by a contravariant rule, as in Emerald. The result of evaluating a `type` constructor is itself an interface which supports the operations necessary to decide type conformance. Conformance is checked at bind time. Thus DPL uses the third approach mentioned in section 4.5.

DPL is implemented by translation to a target language. The bodies of operation implementations may optionally be supplied as code in this language. The interface between such code and the code produced by the DPL translator is specific to each target language. In the current implementation, the target language is C. Since C has no facilities for objects or concurrency, the corresponding DPL features integrate smoothly. However, in a language like Modula-3 or C++, the overlap of features with DPL could result in confusion. When objects are available in the target language, it seems simpler to adopt our approach, which uses them to present potentially remote interfaces to the application programmer.

7.1.3 Reliable Systems

Our distributed garbage collector is fault-tolerant, and our underlying RPC mechanism deals with transient communication problems. However, we provide no other explicit generic support for reliable distributed applications, preferring not to hide the essential difficulties of distribution from our clients. This is a matter of the emphasis of the current work rather than a philosophical position. Various systems have provided programmers with tools to help their applications manage persistent state (which survives crashes) and improve reliability by replication and quorum or consensus mechanisms. The integration of such tools with our system is a topic for further work.

The Argus language and system [Liskov 88b] provides language-level support not only for object-based remote procedure call but also for nested atomic transactions. Like Emerald, Argus is a homogeneous system in which language and runtime are closely coupled. Objects in Argus are called *guardians* and their operations are called *handlers*. Guardians are like ANSA objects, not the ANSA *interfaces* which network objects represent in our system. Guardians are mobile. They can be created dynamically and the names of guardians and handlers can be transmitted in handler calls. However, such dynamically created guardians cannot be nested inside their creators. This prevents guardian names from being used as lightweight handles to transient shared state.

Guardians encapsulate two kinds of state, whose declarations in Argus programs are syntactically distinguished. *Stable* data survives crashes, while *volatile* data does not. After a crash, volatile state is rebuilt from stable state by recovery code specified by the programmer. Argus supports synchronisation and recovery through *atomic actions*. These are *serialisable*: the outcome of actions executed concurrently is the same as if the actions were executed sequentially in some order. They are also *total*: a given action either completes successfully or has no effect, even when the state used by an action resides in more than one guardian. To achieve this, the Argus programmer uses built in atomic data types such as atomic arrays and records. Each instance of an atomic data type has its own readers-and-writers lock. Serialisability is implemented by strict two phase locking: locks are held until an action commits or aborts. Totality is implemented by a versioning mechanism for atomic objects and the use of two-phase commit [Gray 79] for top-level actions.

Despite the language-level support provided by actions and atomic data types, the Argus programmer must still take care to avoid deadlocks and starvation by choosing the scope of actions appropriately. The longer an action lasts, and the more locks it must hold, the greater the potential for poor performance through excessive contention. In our system,

because of the concurrency mechanisms of Modula-3, programmers have complete control over the association of locks and data. However, this does require that programmers make these decisions themselves, implement them correctly, and take responsibility for the consistency of joint states of more than one program.

The Arjuna system [Shrivastava 89] also supports serialisable nested atomic transactions, within which programs invoke operations on objects by RPC. Both transaction handles and objects are instances of C++ classes. Lock management and transaction control, via stable storage, versions and two-phase commit, are provided in high-level C++ classes. Application classes are derived from these by inheritance. They supply appropriate overrides for save and restore operations which must marshal object state to and from buffers. These buffers are used both for transaction undo and for writing and reading state on stable storage. Arjuna's approach of library-level support should also be possible in our system, though we would probably attempt to generate marshalling code for object state automatically from IDL specifications. As our system currently stands, programmers can use SRC Modula-3's pickles facility to save and restore rooted heap structures.

Arjuna has been ported to use ANSAware as the RPC transport. However, stubs are generated from (slightly restricted) C++ class declarations rather than IDL, and Arjuna presentation syntax is used in messages. This prevents straightforward interworking between Arjuna and the rest of the testbench world. In the current implementation, object references cannot be transmitted in invocations. Bindings are obtained by quoting unique identifiers to the runtime.

The *ISIS*$_2$ system [Birman 87b, Birman 87a] is based on *process groups*. Groups are named by text strings chosen by programmers: binding is by convention in *ISIS*$_2$. A process may be a member of more than one group at once. A process communicates with a group by using a message broadcast primitive. The calling process specifies the number of replies it requires before the call returns. Programs in *ISIS*$_2$ can rely on the property of *virtual synchrony*: every process in a group sees events concerning that group (that is, membership changes and message delivery) in the same order, though not necessarily at the same time. Furthermore, a process receiving a message addressed to a group knows that every other process in its current view of that group has also received the message. It may therefore use knowledge of the algorithms employed by the other group members to make (logically) coordinated actions without using additional application-specific consensus protocols.

With these facilities, *ISIS*$_2$ supports the use of replication for both reliability and consistent data sharing. Although *ISIS*$_2$ conceals the details of the protocols used to achieve

consistency from the application programmer, it cannot conceal the extra latency of communication consistency requires. $ISIS_2$ programmers must choose group sizes and consistency requirements carefully to obtain the desired performance for their application.

Facilities for group communication are currently being implemented within the ANSA testbench. Groups are presented to clients as ordinary interfaces. To receive invocations on a group, an ordinary server interface is presented to the group's *management interface*. Clients and servers of an interface group enjoy virtual synchrony properties similar to those of $ISIS_2$. These properties are provided by the interposition of *distributors*, *collators* and *member agents* between clients and servers. Our system should accommodate the ANSA group facilities readily. Though stubs might be different, the application programmer should see little change: the group facilities are all based on standard interfaces specified in IDL.

7.2 Distributed Garbage Collection

Almost all published distributed collectors distinguish between public objects and those which are only ever accessible within their own process, leaving the collection of the latter to a local collector. They vary in the amount of cooperation they require from these local collectors. They also vary in the requirements they place on their environment with respect to the reliability of processes and communication, and in the strength of their safety and liveness guarantees.

The distributed collection schemes fall into two broad classes: those based on reference counting and those based on tracing. Reference counting schemes, including our collector, work by tracking local changes in the reference graph as references are transmitted. They are attractive in a distributed system because they require only limited synchronisation between processes, and can often be implemented with little or no modification of local collectors. However, they are unable to collect garbage in cycles spanning more than one process.

By contrast, tracing schemes propagate reachability information by following references from the root objects, and can thus collect all garbage. Unfortunately, tracing collectors require cooperation (and some amount of synchronisation) among many processes, which is expensive in a distributed system. Almost all distributed tracing schemes are incremental (unlike their stop-the-world local cousins), imposing an overhead on mutator activity during tracing comparable to that of the reference counting collectors.

Because of their complementary properties, many hybrids of tracing and reference counting have been proposed. Hybrids use reference counting for timely and relatively inexpensive collection of non-cyclic garbage, and relatively infrequent tracing for completeness. This technique applies equally to our collector, though we have not yet pursued this. In a hybrid scheme, it is possible for a cheap, safe reference counting collector with relatively weak liveness properties in the face of failures to leave garbage it cannot detect to the tracing collector. We are prepared to pay more for our robust reference counting collector because we view full tracing collection as prohibitively expensive: in our environment, the set of processes which must cooperate in complete tracing is potentially large and difficult to identify.

At the end of their survey of the field, the authors of [Abdullahi 92] note that "about 80% of the distributed garbage collectors reviewed in this paper have not been implemented." We can at least claim not to have added to this set! We will now compare our scheme with representative existing collectors, beginning with cheap reference counting collectors, then considering more robust schemes and finishing with tracing collectors.

7.2.1 Reference Counting Collectors

As we saw in section 5.2.4, naïve reference counting does not extend to a distributed environment: increment (dirty) and decrement (clean) messages must be synchronised, as in our algorithm. There are a number of schemes which can reduce this overhead if it is known that references are never lost as a result of process crashes or communication failures. All these collectors avoid the cost of communicating with the owner of an object o when transmitting an o-reference. They maintain safety by distributing o's reference count among the processes through which o-references are propagated. However, because the owner's knowledge is incomplete, none can reclaim objects in the face of process failure.

Indirect reference counting [Piquer 91], like our algorithm, keeps at most one o-reference per client process by indirecting via records analogous to our surrogates. A list of incoming references protects objects from an independent local collector. Rather than keeping the total reference count with o, Piquer organises o's surrogates into a tree, rooted at o, which records the propagation history of references to o. Each surrogate contains a pointer to its parent in the tree, and a count of the number of its immediate children. Before process P transmits an o-reference to Q, the local count at P is incremented, rather than making a dirty call to the owner from Q as in our scheme. If Q already has a surrogate for o, it atomically clears the surrogate's *deleted* flag and replies with a decrement message to P; otherwise it creates a new o-surrogate with parent P.

When Q's local collector detects that the surrogate is unreachable, it is deleted (and a decrement sent to its parent in P) only if Q is currently on the *fringe* of the propagation tree; otherwise, the surrogate and its count must be *marked* deleted, but retained until its immediate children are deleted and the count at Q falls to zero. If the surrogate is still marked deleted at this time, Q sends a decrement to P as before. When the count at o, the root of the tree, reaches zero, it can be removed from its owner's incoming reference list.

This scheme has been implemented on a network of Transputers. Although it requires a reliable message transport, because duplicated decrements or references lost in transit would compromise safety or liveness, no non-local synchronisation is needed when a reference is transmitted. Unlike our algorithm, which requires both dirty and clean calls, indirect reference counting has an overhead of only one extra message per reference. However, it can retain resources in processes which no longer contain references for o. Further, if a process P in o's reference propagation tree terminates, the portion of o's reference count kept at P will be irrevocably lost, even if P has no non-deleted surrogate at the time. The o-counts in all P's ancestors will never become zero, and o will never be collected. By contrast, our algorithm can reclaim garbage in an environment where processes may terminate independently.

Generational reference counting [Goldberg 89], though it was invented earlier, can be understood as a variation of indirect reference counting in which, rather than maintaining the propagation tree of o-references itself, one merely records in each reference its *generation* (its depth from o in the propagation tree). Copy counts are kept in each reference as before. Then any reference can be discarded by sending its generation and copy count to the owner, even if the copy count is non-zero. For each o, the owner keeps an array called a *ledger* whose entries (initially zero apart from that for the zeroth generation) record the number of outstanding o-references of each generation. On receiving a discard message for a reference with generation i and copy count n, $o.ledger[i]$ is decremented (possibly becoming negative in the process) and $o.ledger[i+1]$ is incremented by n. When all its ledger entries are zero, o may be collected. In our terminology, the dirty calls for generation $i+1$ references have been deferred and piggy-backed on the clean calls from generation i.

The generational scheme also requires reliable messaging, needs no non-local synchronisation and has an overhead of one extra message per discarded reference. It allows a process to terminate after sending discard messages for all its references, but cannot reclaim an object o after a process has terminated while holding a o-reference.

Weighted reference counting [Bevan 87, Watson 87] has similar communication and synchronisation requirements to the indirect and generational schemes, and has the same intolerance to process failure as the latter. In this algorithm, a strictly positive *weight* is stored with each o-reference and on o itself such that the sum of the weights of the o-references is invariantly equal to the count on o, initially a large number. When a new reference s is created from an old one r, the previous weight of the old reference is split (in practice equally) between r and s. When a reference is deleted, a *discard* message is sent to the owner, which decrements o's reference count by the weight of the deleted reference. When o's reference count reaches zero, no references remain. In practice, weights are always powers of two. They are therefore stored as logarithms, and thus effectively record the generation of the reference (subtracted from the initial log weight). The relationship between the generational and weighted schemes is that the latter effectively destroys a generation i reference and creates *two* of generation $i+1$ when copying, while the former stores (negative) weights in exponent and mantissa (generation and copy count) form. Both algorithms deal with underflow of weights (overflow of generations, copy counts or ledgers) when a reference is copied repeatedly by creating fresh indirection objects, but we will not discuss this further.

7.2.2 Robust Reference Counting

Among existing collectors, ours is most closely related to the fault-tolerant reference counting schemes of Mancini and Shrivastava (MS) [Mancini 91] and Shapiro *et al.* (SGP) [Shapiro 90, Shapiro 92]. All three can tolerate lost, duplicated and delayed messages. The SGP scheme is more expensive than ours, particularly in the face of process failure, while the MS scheme, though it handles process failure simply, imposes more restrictions on reference transmission than we do.

The SGP scheme, like ours, distinguishes local from remote references and maintains various data structures to allow independent local collectors to cooperate in reclaiming public objects for which no references from remote spaces exist. It tolerates lost, duplicated and delayed messages, but because o's owner is not informed when transmitting an o-reference between other spaces, it can only handle process failure by a very expensive mechanism, and potentially imposes a high overhead both for reclamation and when making an invocation on an object.

A process P holds a reference to a remote object o in the form of a *stub* for o. A stub in P refers to a structure called a *scion* for (o, P) in a remote space; because scions record the identity of the client process, they can be used to make various changes to the set of

scions idempotent in much the same way as the entries of the owner's *cap.exports* records for capsule P in our implementation. Scions at the owner protect concrete objects from the local collector in the familiar way. Stubs are analogous to our surrogates, except that P may contain more than one stub for each concrete object o. However, a stub for o need not point directly to a scion at o's owner, as we shall now see.

To transmit an o-reference to Q, P first creates a scion for (o, Q) in its own address space which refers to the local stub in P (thus protecting the stub from P's local collector), then sends a reference to this indirection scion to Q, which creates a new stub. This technique, like indirect reference counting, avoids making a dirty call to o's owner and builds a propagation tree for o's stubs. Unlike Piquer's scheme, Q need take no special action if it already holds an o-stub: it simply creates another one. An invocation on the new stub proceeds indirectly via P; this involves a potentially unbounded number of nested RPCs. When Q's local collector detects that the new stub has become unreachable, a *removal* message is sent to its parent (P) in the propagation tree. Because indirection scions keep stubs alive, this will only happen when Q is on the fringe of the tree, as before.

To maintain safety in the face of lost, duplicated and delayed messages, the SGP scheme, like ours, uses strictly increasing counters maintained at each process. Each transmission of a reference from P is stamped with the current value of P's counter. The scion at P for (o, Q) contains P's timestamp value for its last transmission to Q; this contrasts with the corresponding *cap.exports* record of our algorithm, which would contain a timestamp from Q. Like all processes, Q maintains a *timestamp vector* which records the highest timestamp of any message received by Q from each other process. When Q sends a removal message for o to P, it includes both its own current counter value *and* the value recorded for P in its timestamp vector. The former is used to update P's vector. If the latter is less than that stored in P's (o, Q) scion, there is an o-reference in transit from P to Q or lost, and the removal message from Q is ignored. Similarly, if Q receives a transmission out of order (that is, with an earlier timestamp than the highest seen by Q from P), the transmission is ignored as if it had been lost. Our scheme manages timestamps rather differently: reference transmissions need not be timestamped because they will be associated uniquely with any dirty call necessary; clean and dirty calls themselves need contain only the caller's current timestamp.

Maintaining liveness after lost or misordered transmissions or removal messages is handled by a separate mechanism. Periodically, Q sends to P a *live* message, timestamped like a removal message with Q's time value and the highest timestamp it has seen from P. The live message contains a list of Q's live stubs for scions at P. Q-scions at P which do not appear in Q's live message can be removed if their timestamp indicates that no

reference is in transit. For this to ensure liveness in the face of lost transmissions from P, Q's view of P's timestamp must make progress. This is achieved by assuming that the periodic live messages in the opposite direction (from P to Q) will eventually be received. The analogous mechanism in our scheme is the repetition of clean calls from Q until they succeed or communication is deemed permanently lost; because of our pinning mechanism and the fact that a surrogate cannot be created at Q until its dirty call has returned successfully, we do not need to send a complete record of the live surrogates at Q. Notice that the live messages suffice to ensure the liveness of the SGP algorithm: removal messages are a performance optimisation.

Thus far, the SGP scheme can be viewed as a more robust version of indirect reference counting. However, it goes further by introducing a background task which eliminates indirection scions by *path compression*. When an indirect invocation via P from Q on o returns, o's true owner becomes known to Q. Q can now make what is effectively a dirty call to cause the owner to transmit an o-reference directly to Q, thus creating an (o, Q) scion at the owner. The new direct stub at Q can now replace the old one, which is dropped once any outstanding calls on it have returned. Eventually, the cleanup for the old stub will cause P to reclaim its (o, Q) scion, possibly unprotecting the stub in P. Where we rely on pin lists to keep surrogates alive in a transmitter until the receiver's dirty call is known to have been processed by the owner, the SGP scheme creates indirection scions which will eventually be eliminated by path compression and normal cleanup processing. Our algorithm requires fewer messages when P transmits o to Q as a call argument: because Q's dirty call is nested under this call, the fact that it returns is sufficient acknowledgement that it is safe to release pins.

Suppose a process P that is an internal node of the propagation tree of o-references crashes before it is bypassed by path compression. An invocation on a downstream o-stub will discover this fact, and will be delayed until a global search has been made for o's owner. When such a crashed process is detected, every other process must bypass P by using the exhaustive search. Once it is known that this has been achieved (which implicitly requires a distributed termination detection algorithm), an empty live message can be faked from P; since it is not known locally for which spaces P held references when it crashed, this final clean message must be sent to and acknowledged by every process. It is not clear how the death of processes on the fringe of the propagation tree is detected. By contrast, because our scheme informs the owner before every surrogate creation, the owner always has enough information to maintain liveness in the face of client crashes.

The MS scheme addresses the problem of process failure explicitly, by making use of an existing orphan detection facility in the underlying RPC system. At the owner,

the scheme maintains an *export* list (analogous to the *Concrete* entries of our object table) whose elements record the reference count for each public (concrete) object and protect such objects from the local collector. For each client process, the owner keeps a structure recording the set of object *references* (one per transmission) which have ever been transmitted to the client. By contrast, the *cap.exports* table of our implementation contains one entry per *object* (section 5.3.3). These transmission lists are related to our pin lists. Before transmitting an o-reference to process P, o's owner increments o's reference count in the export table (possibly creating an entry) and adds the reference to P's transmission list. There may be many distinct o-references at P.

After P's local collector has deleted an o-reference, it is included in a *del* (clean) message to the owner; because the references are distinguishable, the death of *each* results in its removal from the owner's transmission list and the decrementing of o's reference count. When this reaches zero, the protective export list entry is removed in the usual way. When the orphan detection mechanism at the owner discovers that P has crashed, either because of the arrival of a call from P with a later *crash count* (incarnation) than that recorded for P, or because P has failed to respond to a periodic request for its crash count, all the references in the transmission list for P are deleted as before, and the transmission list itself is discarded to ensure this happens only once. This works because o-references are only transmitted to P in results of calls from P to the owner: thus, the death of one incarnation of P will be discovered before any references are transmitted to a later incarnation.

When an o-reference is transmitted to Q from a process P which is not its owner, the MS scheme needs an additional mechanism. Rather than make a dirty call from the *receiver* (Q) as in our algorithm, it maintains safety by making the call from the *transmitter* (P). The owner puts the new reference on its transmission list for Q, and increments the reference count as before. Only once P's dirty call has returned successfully can it send the reference to Q. However, the owner must check that the new reference is eventually received at Q: otherwise liveness would be compromised should P or Q crash after the dirty call but before transmission.

To this end, each reference in the transmission list for Q at the owner contains an initially true *unused* flag, which is cleared on receipt of an invocation on that reference. When the orphan detection mechanism comes to check that Q has not crashed, it includes Q's currently unused references in the query. There is a possible race if the query arrives at Q before the transmission from P; this is resolved by having Q regard a reference possibly in transit as already in its possession. For this to work, Q must already know the identity of any object for which it is expecting to receive a reference. In the MS scheme, a reference

is only transmitted to Q as a result of an explicit request from Q. (This implies that Q must not be able to forge its own o-reference on the basis of o's identity alone.)

Neither the MS scheme nor ours can distinguish process failure from extended communication failure, so both may violate safety in this case (section 5.2.6). Where the MS scheme relies on orphan detection, we rely on death detection and the semantics of our RPC system to cope with failures. In particular, the fact that a dirty call which has returned successfully can never execute in the future at the owner is what allows us to forget timestamps after a clean call if no previous dirty has failed (section 5.2.7). In the case of transmission from the owner, where the dirty call is local, the MS scheme uses one less RPC than ours. In the general case, it imposes the same overhead. However, we support a more general programming model in which client processes need not know in advance the identities of the objects for which they will hold references, allowing server objects to be created and collected dynamically.

None of the reference counting schemes we have described can detect garbage lying on inaccessible multi-process cycles. Various modifications have been proposed to address this problem without resorting to full tracing. **Bishop's technique** [Bishop 77] consists of migrating a locally inaccessible object to one of the remote processes holding a reference to it. By ordering processes appropriately, it is possible to arrange that garbage cycles will eventually end up in a single process, where they will be detected by the local collector. We cannot use this trick because we do not (yet) support object migration, whose meaning in our heterogeneous environment would need clarification. **Vestal's technique** [Vestal 87] consists of making a trial deletion of an object suspected of lying on a garbage cycle and propagating its effect on reference counts. If after this restricted trace the minimal set of traced objects which would have a zero reference count includes the seed object, the set is garbage. This technique suffers from the problems of identifying potential seed objects, and requiring cooperation from remote processes' local collectors to maintain and propagate hypothetical reference counts.

7.2.3 Tracing Collectors

Reference counting schemes cannot detect cyclic chains of distributed garbage essentially because they only consider changes to a small part of the reference graph, while reachability is a *global* property. An object o is reachable in a global state (a snapshot at an instant of external global time) if and only if there is some path of references starting at a *root* object or a message in transit and ending at o. The difficulty of distributed garbage collection arises in part because such global states (and the unique total ordering

of events they imply) cannot be observed within the system [Lamport 78] (though the partial order induced by the relation of causality of events is observable). Simply deciding which processes are to be regarded as containing roots is not completely straightforward, either.

The distributed garbage collection problem is well defined in these circumstances because being garbage is a *stable property*. Once an object has become unreachable (in some global state), no further references to it can be created and it stays unreachable forever. Stable properties can be detected by distributed termination detection algorithms [Chandy 85], but these are expensive in messages and do not scale well with the number of processes among which consensus must be reached, especially when these processes are subject to failure. (Conversely, termination detection can be formulated as a garbage collection problem [Tel 93].)

Nearly all of the tracing collectors are based on mark-and-sweep rather than copying ([Rudalics 86] is an exception). A typical tracing scheme which is claimed to be robust to process failure is the Emerald collector described in [Juul 92]. It is related, as are many others [Hudak 82, Augusteijn 87, Vestal 87, Schelvis 89], to the classic on-the-fly algorithm of [Dijkstra 78].

The Emerald collector proceeds in three phases. In the first, a collection cycle is initiated by one process and labelled with a cycle number. In the second, which starts when all processes agree on the current collection cycle number, local tracing collectors gather reachability information and propagate it amongst themselves in parallel with mutator activity, which can transmit references between processes. The end of this phase is detected by a distributed termination detection algorithm. In the final phase, between collections, unmarked objects can be discarded as they are encountered by local sweeping interleaved with allocation at each process.

To allow tracing to take place in parallel with mutator activity, the Emerald collector uses the traditional three-colour marking scheme. Objects which have not yet been considered by the tracing algorithm and are potentially garbage are *white*, those which are alive and whose references are currently being traced are *gray*, and those which are alive and have had all their references traced are *black*. Initially, all objects are white except the roots, which are gray, and all mutators are halted. The collector makes progress by making objects darker (converting gray objects to black once all their descendents are at least gray) under the invariant condition that no black object holds a reference to a white object. New objects are created black. Mutators can never access a white object. Gray objects are protected and made black (by shading their descendents, possibly by calls to

remote processes) on an attempted access by a mutator; since such an access must be by an invocation, this protection is relatively cheap in Emerald. Thus mutators only see black objects, and any references they transmit must therefore have at least gray referents. When no gray objects remain and there are no outstanding non-local shading requests, all remaining white objects are garbage. The termination of the tracing phase is detected by a robust two phase commit protocol [Gray 79], which involves communication with every process in the system.

Black, gray and white markings only have a meaning relative to a particular global collection cycle. The Emerald collector in fact marks objects with collection cycle numbers. During the tracing phase, the colours black and white are represented by the current and previous cycle numbers and gray objects are identified by membership in *gray sets* which are kept on the side. Objects marked with cycle numbers strictly less than that of a cycle whose tracing phase is known to have terminated are garbage and can be reclaimed.

Unlike the Emerald collector, which requires global agreement to start a collection cycle, **Hughes' collector** [Hughes 85] allows many collections to run in parallel. A collection is identified by a timestamp which is propagated through the reference graph by tracing. Objects are thus marked with the highest timestamp at which they are known to be accessible. As long as collections continue, the timestamps of live objects will increase. By use of a distributed termination detection algorithm, a global lower bound on the timestamps of live objects can be be determined. As this lower bound makes progress, garbage objects with earlier timestamps can be reclaimed. The algorithm is expensive in a distributed environment because of the communication required to establish the lower bound.

Liskov and Ladin's collector [Liskov 86b, Ladin 92] allows each process to collect its own heap independently, like ours. Rather than requiring tracing activity to propagate between processes, like most tracing algorithms, it relies on a logically central highly-available service to store information about inter-process references.

A process P maintains a structure called the *Inlist* whose entries point to objects owned by P for which it has transmitted references in messages, and which may therefore be accessible from other processes. Like a *Concrete* entry in our object table, the *Inlist* entry for object o protects o from P's local collector; in addition, the entry records $o.id$ and the local time of the last transmission of o from P. When P transmits an o-reference to Q, an entry recording $o.id$, Q and the local time is added to *Trans*, a list of references transmitted since the last local collection.

At the end of each local collection, P sends to the central service three pieces of information about its portion of the global reference graph, accompanied by the time at which the collection took place. The set of remote objects for which P holds references is split into two parts: *Acc* and *Paths*. The *Acc* list contains the identities of those remote objects which are accessible from the roots at P *excluding* the public objects recorded in the *Inlist*. The *Paths* list contains the identities of those which are reachable only from the public objects, and are inaccessible from the other roots. For each such remote object, the entry in *Paths* records some public object at P from which it is (directly or indirectly) accessible. If an object in *Paths* is globally accessible, it is only by virtue of a root outside P. The third item sent to the central service is the *Trans* list. In return P receives a list of the objects it owns that are known to be globally inaccessible.

The central service maintains data structures recording the complete inter-process reference graph. Since the processes' local collectors operate asynchronously with each other, the information in this graph is never completely accurate. However, with care and some cooperation from the local collectors in the selection of which public objects to use in making *Paths* entries, the central service can use Hughes' algorithm to propagate timestamps and derive a global lower bound on the timestamps of reachable objects. A universal bound on message delay, implemented by loose synchronisation of strictly increasing clocks at all processes, allows the central service to maintain liveness even though its information about references in transit is out of date.

The central service is made highly available in the face of process and communication failures by replication. Each process need only contact one replica. The replicas exchange background "gossip" messages to ensure that their views of the global reference graph and the current global lower bound for live objects' timestamps make progress.

Though Liskov and Ladin's collector does not suffer from the common problem of global synchronisation between ordinary processes, it requires more cooperation from local collection algorithms than ours; in particular, each collector must be modified to calculate the *Paths* set. Also, the storage and processing required to maintain global (though out of date) state in each replica of the central service places a bound on the number of the processes in a practical system. This can be circumvented by introducing a hierarchy of reference services at the cost of increasing the complexity of non-local communication between processes.

Lang, Queinnec and Piquer [Lang 92] propose a collector which explicitly addresses the problem of defining the roots for a tracing collection without requiring global control. Their scheme is designed to be used as an adjunct to distributed reference counting in

order to collect inaccessible cycles lying entirely within *process groups*. Processes in a given group perform a conservative distributed tracing collection relative to that group by regarding objects referenced from outside the group as roots in addition to the local roots of the processes themselves. Once a group has been formed, and the effective root set for the collection cycle is established, reachability information relative to the group is propagated by a tracing algorithm. As usual, this requires cooperation from local tracing collectors to propagate colours from public concrete objects to surrogates in the same process, and as usual, a distributed termination detection protocol between group members discovers when a tracing cycle has stabilised. Unmarked public objects are then deleted, thus breaking any garbage cycles, and the group (notionally) disbands.

A process may be a member of more than one group simultaneously: marks for the tracing algorithm must be maintained separately for each group on concrete and surrogate objects and propagated between them accordingly. For any given group, the scheme is as expensive as a standard distributed tracing collector, and requires the same cooperation from local collectors. It is useful because groups need not include every process in the system, but may be defined (whether dynamically or statically) according to a local policy based on knowledge about where cycles are likely to occur in an application.

Stencil listened attentively. The tale proper and the questioning after took no more than thirty minutes. Yet the next Wednesday afternoon at Eigenvalue's office, when Stencil retold it, the yarn had undergone considerable change: had become, as Eigenvalue put it, Stencilized.

— THOMAS PYNCHON, *V* (1963)

Chapter 8

Conclusion

8.1 Summary

In this dissertation, we have presented the design, implementation and evaluation of a practical network objects system for Modula-3 and the ANSA testbench. Its purpose is to support programmers of *distributed* applications with language-level tools previously available only in centralised or homogeneous systems. To the language-level support for distributed communication provided by traditional RPC, it adds language-level support for the dynamic creation, binding, typechecking and garbage collection of remotely accessible services.

We have had two broad goals. The first was to deal with the heterogeneity and failure properties of the distributed systems encountered in practice. The second was to minimise the burden of distribution on the application programmer by supporting powerful and familiar language-level tools. The main problem to be overcome was the conflict between these goals. The key feature of our system which addresses this conflict is that abstract network object type definitions for a particular target language are generated automatically from interface specifications in a common interface definition language.

With this bridge in place, we were able to come close to achieving our more detailed design goals. In our system, service instances are represented by language-level objects, and are therefore simple to create. They are made remotely accessible by the same mechanism used in a single program: by being passed by reference as arguments or results of operations. The resulting client-side bindings are themselves ordinary language-level objects. We

explicitly map failures of distribution transparency into language-level exceptions.

Services are typed, and their types are propagated with references. Again, the global type system is presented in the familiar terms of the application programming language. New service types can be introduced without stopping old programs, and references can pass through old programs without loss of type information. Our type mechanisms use fingerprints to reduce the overhead of managing the space of service types.

Concurrency is an unavoidable companion of distribution. We regard concurrency as independent of object structure. Within a single program, we support well understood techniques for language-level concurrency control, but our system takes no special measures in this respect. In particular, we have so far made no attempt to provide distributed atomic transactions or support for transparent replication.

The network objects style encourages the use of dynamically created objects as handles for transient shared state. We have succeeded in supporting this usage with fault tolerant distributed garbage collection. This solves a common core of the distributed resource management problem in an application-independent way. However, in order to allow an efficient implementation, our collector must rely on some cooperation from the programmer if garbage cycles are to be reclaimed.

We do not free the programmer from the burden of deciding in advance which objects are to be remotely accessible. Furthermore, their specifications must be written in IDL. We regard this lack of transparency as a positive feature of our system for several reasons:

- The restrictions of IDL are precisely those which allow interface specifications to be implemented via efficient heterogeneous RPC mechanisms.
- The performance and failure properties of network object invocation in a distributed system are sufficiently different from the local case that they should be taken into account early in the design of a distributed application.
- We believe that "choosing the proper boundaries between functions is perhaps the primary activity of the computer system designer" [Saltzer 84]. Therefore, we wish to encourage programmers to design their interfaces carefully—both as abstractions and in the light of the properties of likely implementations.

Perhaps our most significant contribution is that our system is completely implemented. The tradeoffs we have made are those we have found useful in practice. We have demon-

strated by example that programmers of applications in heterogeneous distributed computing systems can be supported with many of the tools that are currently common only in centralised systems.

8.2 Further Work

A number of common problems in distributed computing are not directly addressed in our system. Persistence of service state, secure communication between authenticated principals, mobility of service objects, and reliability through replication, consensus protocols, or automatic fail-over are all left to the programmer. These problems have received considerable attention in distributed systems research. The extent to which solutions can be provided in an application-independent manner is still a topic for debate, as is the best division of responsibility between languages, runtime libraries and operating systems. Nevertheless, solutions to these problems *are* available. It would be interesting to investigate how best these solutions can be presented to the application programmer in the framework of our system. M. D. Schroeder's *software clerks* [Schroeder 92], M. Shapiro's *proxies* [Shapiro 86] and the analogous notion in the ANSA architecture of *selective transparencies* seem good starting points. Language-level network objects and their associated (possibly annotated) IDL definitions seem good candidates to act as the interface between application-level code and the implementations of the various transparency mechanisms.

One of the strengths of the network objects style is that the application programmer sees binding to remote entities as the familiar process of binding to local objects. The targets of bindings in our system are essentially RPC-like service interfaces, but perhaps extensions are possible. For instance, some RPC systems allow stream connections of one form or another to be established as part of an RPC binding. These vary from thinly-disguised TCP byte streams or UNIX pipes to reliable typed message streams with sophisticated facilities which control the sequencing of messages and conventional RPC calls sent across the same interface [Gifford 88].

Our system contains mechanisms which trigger secondary activity (such as dirty calls for our garbage collector) during the binding process for network object references of a particular type (`<: DGC.T`). It seems quite possible that a more general version of this mechanism could be made available to higher-level software. For instance, binding to a network object which represents the source of a video stream could trigger connection setup. The network object itself would provide control operations for this stream. Notice that it might be possible to relax the requirement that the surrogate object created at the

receiver has the same abstract type as the concrete object to which it is bound. Arbitrary translation mechanisms could then be interposed. Operations on the surrogate need not even make calls on the concrete object. Such a generic mechanism would probably be based on *marshalling specials*. These are procedures (or closure objects), registered with the network object runtime in an extended stub type registry (section 4.8). When a network object reference is marshalled or unmarshalled, the marshalling special for its type is given the opportunity to produce or interpret an additional value of some arbitrary IDL type which is transmitted together with the standard interface reference.

A similar mechanism could also be provided at the level of the stub generator. In our current system, `stubm3` makes certain fixed translations from IDL types (like `STRING`s) to idiomatic Modula-3 types (like `TEXT`s). Given a way of registering their names and signatures, custom type translations and marshalling procedures could be supplied for application-specific types. This would solve problems like the one mentioned in section 3.3.1 (in which we would like to translate certain fixed-length IDL character arrays to and from Modula-3 `TEXT`s during marshalling).

Other extensions to our system would be more straightforward. User-defined exceptions for IDL would be both very useful and extremely simple to implement. A lightweight RPC mechanism with single-shot semantics, in the style of CCLU "maybe" calls, would be a useful addition for systems in which application-level retry mechanisms are intrinsically necessary. IDL interfaces should document the "maybe" semantics of calling an operation. This is no great burden since the operation signatures are themselves likely to have been designed for application-level retry, for instance by being idempotent.

Finally, we would like to gain more experience with our distributed garbage collector. In particular, further investigation of the use of garbage collection to trigger associated application-level resource management mechanisms would be useful. Such investigations would allow us to evaluate the extent to which our collector's safety and liveness guarantees reflect the requirements of practical higher-level applications. They would also give us more insight into the requirements of a supplementary collector for distributed cyclic garbage.

Everybody gets told to write about what they know.
The trouble with many of us is that at the earlier stages
of life, we think we know everything—or to put it more usefully,
we are often unaware of the scope and structure of our ignorance.

— THOMAS PYNCHON, *Slow Learner* (1984)

Bibliography

[Abdullahi 92] Saleh E. Abdullahi, Eliot E. Miranda, and Graem A. Ringwood. *Collection Schemes for Distributed Garbage.* In Bekkers and Cohen [Bekkers 92], pages 43–81. (p 127)

[Anon 91] *How the language got its spots.* In Nelson [Nelson 91b], chapter 8, pages 218–252. (p 27)

[ANSA 89] Architecture Projects Management Ltd., Cambridge, UK. *ANSA Reference Manual release 01.01*, March 1989. (pp 6, 19)

[ANSA 92a] ANSA. *Interface References.* In ANSA [ANSA 92c], chapter 2. Document RM.101.01. (p 101)

[ANSA 92b] ANSA. *Notification service implementation notes.* In ANSA [ANSA 92c], chapter 12. Document RM.101.01. (p 88)

[ANSA 92c] Architecture Projects Management Ltd., Cambridge, UK. *ANSAware 4.1: System Programming in ANSAware*, November 1992. Document RM.101.01. (pp 19, 143)

[ANSA 93] Architecture Projects Management Ltd., Cambridge, UK. *DPL Programmers' Manual*, February 1993. Document TR.031.00. (p 123)

[Augusteijn 87] Lex Augusteijn. *Garbage Collection in a Distributed Environment.* In PARLE '87: Parallel Architectures and Languages Europe, volume 259 of *LNCS*, pages 75–93. Springer-Verlag, June 1987. (p 134)

[Bacon 93] Jean Bacon. *Concurrent Systems: an integrated approach to operating systems, database, and distributed systems.* Addison Wesley, 1993. (p 12)

[Bartlett 88] Joel F. Bartlett. *Compacting Garbage Collection with Ambiguous Roots.* Research Report 88/2, DEC Western Research Laboratory, 100 Hamilton Ave., Palo Alto, CA 94301, February 1988. (p 112)

[Bartlett 91] Joel F. Bartlett. *Don't Fidget with Widgets, Draw!* Research Report 91/6, DEC Western Research Laboratory, 100 Hamilton Ave., Palo Alto, CA 94301, May 1991. (p 6)

[Bayer 79] R. Bayer, R. M. Graham, and G. Seegmuller, editors. *Operating Systems: an Advanced Course*, volume 60 of *LNCS*. Springer-Verlag, 1979. (pp 147, 152)

[Bekkers 92] Y. Bekkers and J. Cohen, editors. *Memory Management: International Workshop IWMM '92*, volume 637 of *LNCS*. Springer-Verlag, September 1992. (pp 143, 148, 153)

[Bennett 87] John K. Bennett. *The Design and Implementation of Distributed Smalltalk.* Proc. of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, SIGPLAN Notices, 22(12):318–330, October 1987. (p 15)

[Bershad 87] Brian N. Bershad, Dennis T. Ching, Edward D. Lazowska, Jan Sanislo, and Michael Schwartz. *A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems.* IEEE Trans. on Software Engineering, SE-13(8):880–894, August 1987. (p 14)

[Bevan 87] D. I. Bevan. *Distributed Garbage Collection using Reference Counting.* In PARLE '87: Parallel Architectures and Languages Europe, volume 259 of *LNCS*, pages 176–187. Springer-Verlag, June 1987. (p 129)

[Birman 87a] Kenneth P. Birman and Thomas A. Joseph. *Exploiting Virtual Synchrony in Distributed Systems.* Proceedings of the 11th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review, 21(5):123–138, November 1987. (p 125)

[Birman 87b] Kenneth P. Birman and Thomas A. Joseph. *Reliable Communication in the Presence of Failures.* ACM Transactions on Computer Systems, 5(1):47–76, February 1987. (pp 88, 125)

[Birrell 84] Andrew D. Birrell and Bruce Jay Nelson. *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, 2(1):39–59, February 1984. (pp 12, 52)

[Birrell 88] Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber. *A simple and efficient implementation for small databases.* Research Report 24, DEC Systems Research Center, January 1988. (p 51)

[Birrell 91a] Andrew D. Birrell. *An introduction to programming with threads.* In Nelson [Nelson 91b], chapter 4, pages 88–118. (p 24)

[Birrell 91b] Andrew D. Birrell, John V. Guttag, Jim J. Horning, and Roy Levin. *Thread synchronisation: a formal specification.* In Nelson [Nelson 91b], chapter 5, pages 119–129. (p 26)

[Birrell 93a] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Ted Wobber. *Distributed Garbage Collection for Network Objects.* Submitted for the ACM Symposium on Operating Systems Principles, 1993. (pp 2, 91, 120)

[Birrell 93b] Andrew Birrell, Greg Nelson, Susan Owicki, and Ted Wobber. *Network Objects.* Submitted for the ACM Symposium on Operating Systems Principles, 1993. (pp 93, 119)

[Bishop 77] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection.* Technical Report MIT/LCS/TR-178, M.I.T. Laboratory for Computer Science, May 1977. (p 133)

[Black 87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. *Distribution and Abstract Types in Emerald.* IEEE Trans. on Software Engineering, SE-13(1):65–76, January 1987. (pp 15, 52, 58)

[Boehm 93] Hans-Juergen Boehm. *Space Efficient Conservative Garbage Collection.* In ACM SIGPLAN'93 Conference on Programming Language Design and Implementation, pages 197–206, June 1993. (p 112)

[Brown 92] Marc H. Brown and James R. Meehan. *The FormsVBT Reference Manual.* Draft Report, DEC Systems Research Center, July 1992. (p 107)

[Campin 91] Jack Campin, Richard Cooper, and Francis Wai. *Type Management in a Heterogeneous System.* In D. J. Harper and M. C. Norrie, editors, *The Glasgow Collection of Comandos Papers*, pages 1–27. Department of Computing Science, University of Glasgow, September 1991. Departmental Research Report CSC/91/R16. (p 122)

[Cardelli 85] Luca Cardelli and Peter Wegner. *On Understanding Types, Data Abstraction and Polymorphism.* ACM Computing Surveys, 17(4):471–522, December 1985. (p 58)

[Cardelli 89] Luca Cardelli. *Typeful Programming.* Research Report 45, DEC Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, May 1989. (p 58)

[Cardelli 91] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. *Modula-3 Language Definition.* In Nelson [Nelson 91b], chapter 2, pages 11–61. (p 21)

[Carriero 86] Nicholas Carriero and David Gelernter. *The S/Net's Linda Kernel.* ACM Transactions on Computer Systems, 4(2), May 1986. (p 11)

[Carriero 89] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A Guide to the Perplexed.* ACM Computing Surveys, 21(3):323–357, September 1989. (p 11)

[Chandy 85] K. Mani Chandy and Leslie Lamport. *Distributed Snapshots: Determining Global States of Distributed Systems.* ACM Transactions on Computer Systems, 3(1):63–75, February 1985. (p 134)

[Chin 91] Roger S. Chin and Samuel T. Chanson. *Distributed Object-Based Programming Systems.* ACM Computing Surveys, 23(1):91–124, March 1991. (p 18)

[Craft 85] Dan H. Craft. *Resource Management in a Distributed Computing System.* PhD thesis, University of Cambridge Computer Laboratory, 1985. Available as Technical Report No. 73. (p 88)

[Crosby 93] Simon Crosby. *MSRPC—A Lightweight RPC system for MSNL.* In ATM Document Collection. University of Cambridge Systems Research Group, February 1993. (p 110)

[Dahl 70] O.-J. Dahl, B. Myrhaug, and K. Nygaard. *(Simula 67) Common Base Language.* Publication N. S-22, Norsk Regnesentral (Norwegian Computing Centre), Oslo, October 1970. (Revised version, Feb. 1984). (p 22)

[Danforth 88] Scott Danforth and Chris Tomlinson. *Type Theories and Object-Oriented Programming.* ACM Computing Surveys, 20(1):29–72, March 1988. (p 58)

[Demers 90] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Schenker. *Combining Generational and Conservative Garbage Collection: Framework and Implementations.* In Proceedings of the

17th annual ACM Symposium on Principles of Programming Languages (POPL'17), pages 261–269, January 1990. (p 112)

[Dijkstra 78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. *On-the-fly Garbage Collection: An Exercise in Cooperation.* Communications of the ACM, 21(11):966–975, November 1978. (p 134)

[Dixon 92] M. Joe Dixon. *System Support for Multi-Service Traffic.* PhD thesis, University of Cambridge Computer Laboratory, January 1992. Available as Technical Report No. 245. (p 105)

[Evers 89] David Evers. *Gawp: a Tool for Graphical Inter-Terminal Conversations.* Diploma Dissertation, University of Cambridge Computer Laboratory, August 1989. (p 34)

[Evers 92] David Evers and Peter Robinson. *Modula-3 Network Objects over ANSA.* In 5th ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring, September 1992. Paper No. 19. (p 30)

[Garnett 80] N. H. Garnett and R. M. Needham. *An Asynchronous Garbage Collector for the Cambridge File Server.* ACM Operating Systems Review, 14(4):36–40, 1980. (p 75)

[Gifford 88] David K. Gifford and Nathan Glasser. *Remote Pipes and Procedures for Efficient Distributed Communication.* ACM Transactions on Computer Systems, 6(3):258–283, August 1988. (p 140)

[Goldberg 89] Benjamin Goldberg. *Generational Reference Counting: a Reduced-Communication Distributed Storage Reclamation Scheme.* In ACM SIGPLAN Conference on Programming Languages Design and Implementation, pages 313–321, 1989. (p 128)

[Gosling 86] J. Gosling. *Sundew: A Distributed and Extensible Window System.* In Proceedings of the 1986 Winter Usenix Technical Conference, pages 98–103. Usenix Association, January 1986. (p 6)

[Gray 79] J. N. Gray. *Notes on Database Operating Systems.* In Bayer et al. [Bayer 79], pages 459–481. (pp 124, 135)

[Hamilton 84] K. Graham Hamilton. *A Remote Procedure Call System.* PhD thesis, University of Cambridge Computer Laboratory, December 1984. Available as Technical Report No. 70. (pp 13, 43, 64)

[Hayes 92] Barry Hayes. *Finalization in the Collector Interface.* In Bekkers and Cohen [Bekkers 92], pages 277–298. (p 78)

[Herlihy 82] Maurice Herlihy and Barbara Liskov. *A Value Transmission Method for Abstract Data Types.* ACM Transactions on Programming Languages and Systems, 4(4):527–551, October 1982. (p 16)

[Hoare 74] C. A. R. Hoare. *Monitors: An operating system structuring concept.* Communications of the ACM, 17(10), October 1974. (p 24)

[Hudak 82] Paul Hudak and Robert Keller. *Garbage Collection and Task Deletion in Distributed Applicative Processing Systems.* In ACM Symposium on Lisp and Functional Programmng, pages 168–178, August 1982. (p 134)

[Hughes 85] John Hughes. *A Distributed Garbage Collection Algorithm.* In Functional Programming Languages and Computer Architecture, volume 201 of *LNCS*, pages 256–272. Springer-Verlag, September 1985. (p 135)

[Jones 86] Michael B. Jones and Richard F. Rashid. *Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems.* Technical Report CMU-CS-87-150, Carnegie Mellon University Dept. of Computer Science, Pittsburgh, PA 15213, September 1986. Also appeared in Proc. ACM OOPSLA '86. (p 9)

[Jordan 90] Mick Jordan. *An Extensible Programming Environment for Modula-3.* Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments, ACM SIGSOFT Software Engineering Notes, 15(6):66–76, December 1990. (p 47)

[Jul 88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. *Fine-Grained Mobility in the Emerald System.* ACM Transactions on Computer Systems, 6(1):109–133, February 1988. (p 15)

[Juul 92] Niels Christian Juul and Eric Jul. *Comprehensive and Robust Garbage Collection in a Distributed System.* In Bekkers and Cohen [Bekkers 92], pages 103–115. (p 134)

[Ladin 92] Rivka Ladin and Barbara Liskov. *Garbage Collection of a Distributed Heap.* In Proc. 12th International Conference on Distributed Computing Systems, pages 708–715, June 1992. (p 135)

[Lamport 78] Leslie Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM, 21(7):558–565, July 1978. (p 134)

[Lampson 80] Butler W. Lampson and David D. Redell. *Experience with processes and monitors in Mesa.* Communications of the ACM, 23(2), February 1980. (p 24)

[Lampson 92] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. *Authentication in Distributed Systems: Theory and Practice.* ACM Transactions on Computer Systems, 10(4):265–310, November 1992. (p 63)

[Lang 92] Bernard Lang, Christian Queinnec, and José Piquer. *Garbage-Collecting the World.* In Proceedings of the 19th annual ACM Symposium on Principles of Programming Languages (POPL'19), pages 39–49, January 1992. (p 136)

[Lauer 79] Hugh C. Lauer and Roger M. Needham. *On the Duality of Operating System Structures.* Reprinted in ACM Operating Systems Review, 13(2):3–19, 1979. (p 12)

[Lea 92] Rodger Lea and Christian Jacquemot. *The COOL architecture and abstractions for object-oriented distributed operating systems.* In 5th ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring, September 1992. Paper No. 28. (pp 120, 121)

[Leino 93] K. Rustan M. Leino. *Multicomputer Programming with Modula-3D.* Technical Report CS-TR-93-15, California Institute of Technology, Pasadena, CA 91125, June 1993.
`ftp.cs.caltech.edu:/tr/cs-tr-93-15.ps.Z`. (p 120)

[Lermen 86] Claus-Werner Lermen and Dieter Maurer. *A Protocol for Distributed Reference Counting.* In ACM Symposium on Lisp and Functional Programmng, pages 343–350, August 1986. (p 89)

[Levy 91] Henry M. Levy and Ewan D. Tempero. *Modules, Objects and Distributed Programming: Issues in RPC and Remote Object Invocation.* Software—Practice and Experience, 21(1):77–90, January 1991. (p 16)

[Liskov 81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual,* volume 114 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1981. (pp 13, 22)

[Liskov 86a] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1986. (p 61)

[Liskov 86b] Barbara Liskov and Rivka Ladin. *Highly-Available Distributed Services and Fault-Tolerant Distributed Garbage Collection.* In Proceedings of the 5th annual ACM Symposium on Principles of Distributed Computing (PODC'5), pages 29–39, August 1986. (p 135)

[Liskov 88a] Barbara Liskov. *Data Abstraction and Hierarchy.* OOPSLA '87 Addendum to the Proceedings, SIGPLAN Notices, 23(5):17–34, May 1988. (p 61)

[Liskov 88b] Barbara Liskov. *Distributed Programming in Argus.* Communications of the ACM, 31(3):300–312, March 1988. (p 124)

[Manasse 91] Mark Manasse and Greg Nelson. *Trestle Reference Manual.* Report 68, DEC Systems Research Centre, 130 Lytton Ave., Palo Alto, CA 94301, December 1991. (p 105)

[Mancini 91] L. Mancini and S. K. Shrivastava. *Fault-tolerant Reference Counting for Garbage Collection in Distributed Systems.* The Computer Journal, 34(6):503–513, December 1991. (p 129)

[McAuley 90] Derek McAuley. *Protocol Design for High Speed Networks.* PhD thesis, University of Cambridge Computer Laboratory, January 1990. Available as Technical Report No. 186. (pp 53, 105)

[Mitchell 79] James G. Mitchell, William Maybury, and Richard Sweet. *Mesa Language Manual Version 5.0.* Technical Report CSL-79-3, Xerox PARC, April 1979. (p 23)

[Mullender 86] Sape J. Mullender and Andrew S. Tanenbaum. *The Design of a Capability-Based Distributed Operating System.* The Computer Journal, 29(4):289–299, 1986. (pp 9, 88)

[Mullender 93] Sape Mullender, editor. *Distributed Systems—Second Edition.* ACM Press/Addison Wesley, 1993. (p 151)

[Needham 82] R. M. Needham and A. J. Herbert. *The Cambridge Distributed Computing System.* International computer science series. Addison-Wesley, 1982. (p 5)

[Needham 93] Roger M. Needham. *Names.* In Mullender [Mullender 93], pages 315–327. (pp 75, 93)

[Nelson 91a] Greg Nelson. *Supplementary Material for Greg Nelson's Lecture on Network Objects.* Unpublished notes, July 1991. (pp 30, 119)

[Nelson 91b] Greg Nelson, editor. *Systems Programming With Modula-3.* Prentice-Hall, Englewood Cliffs, NJ 07632, 1991. (pp 21, 43, 57, 143, 145, 146)

[Nelson 92] Greg Nelson. Private communication, March 1992. (p 68)

[OMG 91] OMG. *The Common Object Request Broker: Architecture and Specification.* Object Management Group, December 1991. Document no. 91.12.1, Revision 1.1. (p 122)

[OSF 91] OSF. *OSF/DCE 1.0 DCE Application Development Guide.* Open Systems Foundation, December 1991. (p 121)

[Parnas 72] David Parnas. *On the Criteria to be Used in Decomposing Systems Into Modules.* Comm. ACM, 15(12), December 1972. (p 58)

[Piquer 91] José M. Piquer. *Indirect Reference Counting: A Distributed Garbage Collection Algorithm.* In PARLE '91: Parallel Architectures and Languages Europe, volume 505 of *LNCS*, pages 150–165. Springer-Verlag, June 1991. (p 127)

[Rabin 81] Michael O. Rabin. *Fingerprinting by Random Polynomials.* Technical report TR-15-81, Center for Research in Computing Technology, Harvard University, 33 Oxford Street, Cambridge, MA 02138, 1981. (p 65)

[Raj 91] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. *Emerald: A General-Purpose Programming Language.* Software—Practice and Experience, 21(1):91–118, January 1991. (pp 15, 58, 62)

[Rees 93] R. T. O. Rees. *The ANSA Computational Model.* Architecture Projects Management Ltd., Cambridge, UK., February 1993. Document AR.001.01. (p 123)

[Roscoe 93] Timothy Roscoe. *It's All About Beers at the End of the Day.* Private communication, August 1993.

[Rovner 85] Paul Rovner, Roy Levin, and John Wick. *On extending Modula-2 for building large, integrated systems.* Research Report 3, DEC Systems Research Center, January 1985. (p 24)

[Rozier 88] Marc Rozier, Vadim Abrossimov, et al. *CHORUS Distributed Operating Systems.* Computing Systems Journal, 1(4), December 1988. (p 9)

[Rudalics 86] Martin Rudalics. *Distributed Copying Garbage Collection.* In ACM Symposium on Lisp and Functional Programmng, pages 364–372, August 1986. (p 134)

[Saltzer 79] J. H. Saltzer. *Naming and Binding of Objects.* In Bayer et al. [Bayer 79], chapter 3.A, pages 100–208. (p 73)

[Saltzer 84] J. H. Saltzer, D. P. Reed, and D. D. Clark. *End-to-End Arguments in System Design.* ACM Transactions on Computer Systems, 2(4):277–288, November 1984. (pp 13, 75, 139)

[Schaffert 86] C. Schaffert, T. Cooper, B. Bullis, M. Killian, and C. Wilpolt. *An Introduction to Trellis/Owl.* Proc. of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, SIGPLAN Notices, 21(11):9–16, November 1986. (pp 26, 58)

[Schelvis 89] Marcel Schelvis. *Incremental Distribution of Timestamp Packets: A New Approach to Distributed Garbage Collection.* Proc. of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, pages 37–48, October 1989. (p 134)

[Schroeder 92] Michael D. Schroeder. *Software Clerks.* In 5th ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring, September 1992. Paper No. 43. (p 140)

[Shapiro 86] Marc Shapiro. *Structure and Encapsulation in Distributed Systems: the Proxy Principle.* In Proc. 6th International Conference on Distributed Computing Systems, pages 198–204, June 1986. (p 140)

[Shapiro 90] Marc Shapiro, David Plainfossé, and Olivier Gruber. *A Garbage Detection Protocol for a Realistic Distributed Object-Support System.* Rapport de Recherche 1320, INRIA, November 1990.
`ftp.inria.fr:/INRIA/Projects/SOR`. (p 129)

[Shapiro 92] Marc Shapiro, Peter Dickman, and David Plainfossé. *Robust Distributed References and Acyclic Garbage Collection.* In Proceedings of the 11th annual ACM Symposium on Principles of Distributed Computing (PODC'11), pages 135–146, August 1992. (p 129)

[Shrivastava 89] S. K. Shrivastava, G. N. Dixon, G. D. Parrington, F. Hedayati, S. M. Wheater, and M. C. Little. *The Design And Implementation of Arjuna.* Technical Report 280, Computing Laboratory, University of Newcastle upon Tyne, March 1989. (p 125)

[Snyder 86] Alan Snyder. *Encapsulation and Inheritance in Object-Oriented Programming Languages.* Proc. of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, SIGPLAN Notices, 21(11):38–45, November 1986. (p 62)

[Stamos 90] James W. Stamos and David K. Gifford. *Remote Evaluation.* ACM Transactions on Programming Languages and Systems, 12(4):537–565, October 1990. (p 6)

[Tel 93] Gerard Tel and Friedmann Mattern. *The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes.* ACM Transactions on Programming Languages and Systems, 15(1):1–35, January 1993. (p 134)

[Tennenhouse 89] David L. Tennenhouse. *Layered Multiplexing Considered Harmful.* In Protocols for High Speed Networks, IBM Zurich Research Lab., May 1989. IFIP WG6.1/6.4 Workshop. (p 55)

[Thekkath 93] Chandramohan A. Thekkath and Henry M. Levy. *Limits to Low-Latency Communication on High-Speed Networks.* ACM Transactions on Computer Systems, 11(2):179–203, May 1993. (p 116)

[Vestal 87] Stephen C. Vestal. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming.* PhD thesis, University of Washington, Seattle 98195, January 1987. Technical Report 87-01-03. (pp 78, 133, 134)

[Want 90] Roy Want. *The Active Badge location system.* Technical Report, Olivetti Research Laboratories, Old Addenbrooke's Site, 24a Trumpington St., Cambridge, UK CB2 1QN, 1990. (pp 39, 107)

[Watson 87] Paul Watson and Ian Watson. *An Efficient Garbage Collection Scheme for Parallel Computer Architectures.* In PARLE '87: Parallel Architectures and Languages Europe, volume 259 of *LNCS*, pages 433–443. Springer-Verlag, June 1987. (p 129)

[Wilson 92] Paul R. Wilson. *Uniprocessor Garbage Collection Techniques.* In Bekkers and Cohen [Bekkers 92], pages 1–42. (p 74)

[Xerox 81] Xerox Corporation. *Courier: the Remote Procedure Call Protocol*, 1981. Xerox System Integration Standard 038112. (p 19)