Technical Report

UCAM-CL-TR-70
ISSN 1476-2986

Number 70

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

A remote procedure call system

Kenneth Graham Hamilton

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

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

This technical report is based on a dissertation submitted December 1984 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Wolfson 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

The provision of a suitable means for communication between software modules on different machines is a recognized problem in distributed computing research. Recently the use of language-level Remote Procedure Call (RPC) has been advocated as a solution to this problem.

This thesis discusses the rationale, design, implementation and supporting environment of a flexible RPC system for an extended version of the CLU programming language. It is argued that earlier RPC systems have adopted an undesirably rigid stance by attempting to make remote procedure calls look as similar as possible to local procedure calls. It is suggested instead that the inevitable differences in performance and failure properties between local and remote calls are such that remote calls should be regarded as being essentially different from local calls. Following from this, it is proposed that RPC systems should offer at least two complementary call mechanisms. One of these should attempt to recover from network errors and should only report unrecoverable failures. The other should never attempt automatic recovery from network errors, thereby giving implementors the convenience of a language-level mechanism without losing sight of the underlying network.

Other specific areas that are discussed include binding issues, protocols, transmission mechanisms for standard data types, and the particular problems posed by abstract data types. A new transfer mechanism for abstract types is proposed which would permit software using new representations to communicate with software using earlier representations. The provision of special operating system support for the CLU RPC mechanism is also discussed.

Preface

I wish to thank my supervisor, Dr Andrew Herbert, for his encouragement during my research. I would also like to thank the members of the Mayflower group, and the rest of the Titan Room community, for many useful discussions and for providing a pleasant working environment.

The work described here was carried out during a UK Science and Engineering Research Council studentship, for which I am grateful.

Except where otherwise stated in the text, this dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration. Furthermore, this dissertation is not substantially the same as any I have submitted for a degree, diploma or any other qualification at any other university. No part of this dissertation has already been or is being concurrently submitted for any such degree, diploma or other qualification.

Contents

Chapter 1. Introduction.

The provision of a suitable means for communication between program components on different machines is a recognized problem in distributed computing research. A number of solutions have been proposed to this problem and there have recently been attempts to provide explicit linguistic support for distributed processing via either typed message passing or Remote Procedure Call (RPC).

Language-level RPC attempts to make remote interactions appear similar to local procedure calls. Thus when an RPC is executed, it causes a set of type-checked arguments to be passed to a remote machine, where a body of code is executed and a set of type-checked results returned to the calling machine. This provides the programmer with both a simple model for remote interactions and an automatic mechanism for the transfer of language-level objects.

This thesis discusses the rationale, design, implementation and supporting environment of a flexible RPC system for an extended version of the CLU programming language. It is argued that earlier RPC systems have adopted an undesirably rigid stance by attempting to make remote procedure calls look as similar as possible to local procedure calls. It is suggested instead that the inevitable differences in performance and failure properties between local and remote calls are such that remote calls should be regarded as being essentially different from local calls. Following from this, it is proposed that RPC systems should offer at least two complementary call mechanisms. One of these should attempt to recover from network errors and should only report unrecoverable failures. The other should never attempt automatic recovery from network errors, thereby giving implementors the convenience of a language-level mechanism without losing sight of the underlying network.

1.1 Dissertation structure.

The body of the dissertation starts with a discussion of existing and proposed techniques for communicating between software on different machines. This is followed by a discussion of the design issues involved in providing software support for distributed pro-

cessing (including some comments on the limitations of existing solutions), leading into an exposition of the design of a new remote procedure call system for the CLU programming language.

There then follow chapters describing specific issues in the design and implementation of a CLU RPC system, including discussions of binding issues, protocols, transfer mechanisms for standard types and the particular problems provided by abstract types. A new transfer mechanism for abstract types is proposed which would permit software using new representations to communicate with software using earlier representations.

The integration of the RPC system into an operating system is then described. This includes descriptions of additional linguistic extensions made to CLU, and particular points arising from the implementation and tuning of a specialized operating system kernel.

Finally the performance of the CLU RPC system is discussed and the main conclusions of the dissertation are summarized.

Chapter 2. Existing tools for distributed processing.

2.1 An example of the problem.

In late 1981, I implemented a simple mail system within the Cambridge Distributed System. The Cambridge Distributed System [Needham 82] was by this time a relatively large and stable distributed system, encompassing a wide variety of servers and acting as the main computing resource for a group of about 20 students and staff. Despite this, it proved to be a surprisingly inhospitable environment for the development of even a fairly simple distributed application.

The mail system was based around a simple mail database kept on the Cambridge fileserver. The bulk of the mail processing was performed either in users' personal machines, which were unprotected and thus untrusted, or by a trusted mail daemon which was run, when required, in a machine automatically allocated from a pool. A simple Z80 mail server, which was accessed by an authenticated protocol, granted users access to their own mail directories and also stored requests for action by the daemon.

Thus, in their normal operation the components of the mail system needed to communicate both with each other and with a variety of servers, including the fileserver, an authentication server, a date and time server, and a resource management server. As a result the mail system (despite its small size) appeared likely to typify a large class of distributed applications.

It came as no surprise that the code for intermachine communication represented a large proportion of the total code for the mail system. However, this code proved both more time consuming to implement and more error prone than had been expected. All the server interfaces used took the form of the client transmitting a request packet consisting of a function code together with some arguments, to which the server would send a reply packet, normally consisting of a return code plus some results. The formats of all the request and reply blocks were defined in terms of byte patterns at specified offsets in network packets.

On the client machines, a library routine was normally written for each server function invoked. These library routines provided language-level interfaces that were functionally equivalent to the servers' byte-level interfaces. Each library routine stored its arguments in the appropriate places in a transmission buffer; called a protocol handling routine to attempt a packet exchange with some specified degree of persistence; and then decoded the results from the reply buffer. For those procedures supported by the mail server, a reception handling routine provided the converse function. After identifying the nature of the operation required from the function code, this reception routine decoded the anticipated arguments and use them in a call on an appropriate internal routine. The results of this call were then encoded and returned to the caller. The encode and decode implementation was the source of a significant number of bugs in the mail system's initial development, as well as being tedious to implement.

Communication between components of the mail system also had to follow this route and suffered the same problems as calls across public interfaces.

The protocol handling routine that tried to carry out the calls had a very high success rate during testing. The principal cause of failure was congestion, whereby a server was unable to accept new request packets because all its request buffers were already in use (a not uncommon problem at the fileserver).

These problems with remote interface implementation appeared to be general in the Cambridge Distributed system and represented an obstacle to the implementation and extension of servers. Furthermore, they discouraged efforts to build distributed applications, whose internal inter-machine interfaces could be expected to become significantly more complex than the relatively simple interfaces that had been adequate for servers.

2.2 Distributed operating systems.

One approach to simplifying distributed processing is to build an extended operating system that encompasses all the machines in the network. Intercommunication between processes on different machines can then be given the same format as intercommunication between processes on a single machine.

There have been a number of designs and some implementations of such distributed operating systems, e.g. Roscoe [Solomon 79], Trix [Ward 80], the Stanford V Kernel [Cheriton 83] and Accent [Rashid 81]. Accent is a good example of such a system, since it offers particularly rich facilities which are based on experience with earlier distributed systems, such as RIG [Lantz 82] and Medusa [Ousterhout 80].

Accent aims to support an indeterminate number of processes running in separate address spaces and communicating by message passing. These processes may be scattered across a number of hosts on a network. The Accent kernel on each host is expected to ensure that messages are delivered safely across the network.

Accent's message passing facilities are considerably more complex than those normally found on single processor systems. The message system is based around "ports", which are message queues that can contain only a pre-defined number of messages. There are several options to the message transmission primitives which determine what happens when an attempt is made to transmit to a full port. The transmitting process may be suspended until the message can be sent; it may also request to be notified of an error condition if this suspension exceeds some specified period of time. A special option permits a process to deposit with its local kernel one message per port that it wishes to be delivered when that port can accept it.

Both ports and processes can be moved between hosts. It is a feature of the implementation, however, that a port belonging to a crashed host cannot be reassigned.

Unfortunately, Accent's facilities solve only a small part of the problems of building distributed systems. By expanding the message passing system of some single machine operating systems it has provided extremely pleasant facilities for transmitting and receiving reliable datagrams. However, in the case of our earlier example the distributed mail system, these facilities would only serve to simplify the implementation of the already elementary protocol handling routine and leave the handling of both congestion and server failures with application level code. The error prone decoding and encoding of transmitted values into buffers is still necessary when using the Accent kernel.

2.3 Loosely typed interfaces.

One way of simplifying the encoding and decoding of transmitted arguments is to define conventional transmission representations for common language types and require all network interfaces to be defined in terms of these types. Thus an interface that had been previously defined as taking "a size byte at offset 0, followed by that number of characters of file name" might be defined as taking "a string representing a file name". Language-level routines can then be defined for encoding and decoding these standard representations. This approach has been quite widely adopted, for example a conventional type format has been adopted for use with Accent, supported by a set of Pascal and Lisp library routines.

Clearly this approach greatly simplifies the implementation of remote interfaces, but there are still several problems. It is still in general necessary to implement encode and decode routines for each remote interface, even although these routines only consist of calls on the encode and decode routines for the appropriate data types. This would not be essential in cases where the implementation language supports polymorphic operations and run-time type identification. More importantly, library functions will normally only be able to cope with simple types such as integers, arrays, strings, etc. Structured types, such as a record containing a variety of basic types plus other records, must still be composed by hand, albeit largely by calls on the standard type operations. Thus transferring complex data structures is still difficult. Finally, since the types required for any given interface are only known by convention, the implementor must still ensure that the correct types are deposited in the correct order in each network packet. Interfaces cannot be type-checked at compile time and only limited checking can be performed at run time, without transmitting an identifying tag with each type. A client may erroneously encode a 3 byte string instead of a 4 byte integer and the server's decode function may be unable to detect this substitution.

2.4 Linguistic solutions.

Loosely typed systems constitute a step forward from completely untyped solutions, despite their disadvantages. One way to advance beyond these disadvantages is to move

to strong typing, by providing language-level mechanisms to support type-checked remote interfaces. (Ideally these mechanisms would reinforce compile-time type-checking with a run-time consistency check to ensure that all parties to a remote interface have been type-checked against the same interface definition.) This implies that either the remote interfaces must be coerced to look like the language's normal interfaces or the language must be extended.

2.4.1 Guardians.

At an early stage in her investigation of primitives for distributed computing [Liskov 79a], Liskov suggested that inter-machine communication should be based on passing typed messages between program modules termed "guardians". Guardians are intended to be the units of distribution within a multi-machine system, so that each guardian is seen as an autonomous server, possibly containing a number of processes, all residing on a single node.

Guardians create "ports" to act as reception message queues. Each port specification includes a list of named message types which that port may receive. A guardian process may utilize a special syntax to wait for a message on a port and then enter a message dependent handler where the messages' component values are automatically assigned to correctly typed local variables. Each message specification may include a description for a corresponding reply message which will be returned by the guardian's message handler.

The message transfer semantics proposed for use with guardians was "no-wait send", wherein the sending process is only delayed while the message transmission is initiated and is not suspended until the message is delivered or a reply received. These transmission semantics are very similar to those of conventional datagrams. When a process sends a message it must specify a message name and a set of values, and it may also specify a port to which any reply message should be sent.

This proposal for guardians represents one of the most detailed designs for a message passing language. (Other designs include Pascal-M [Cook 82] and Conic [Kramer 82].) However this form of guardians was never implemented and Liskov's later designs for

guardians used essentially procedural interfaces for remote communication (see section 2.4.3).

2.4.2 Remote Procedure Call.

The guardian message passing scheme is based around detailed linguistic support for a fundamental networking concept (the packet transfer). An alternative approach is to take a fundamental linguistic concept (the procedure call) and attempt to provide a mechanism that will permit it to operate across machine boundaries.

The term Remote Procedure Call (or RPC) will be used here to mean a type-checked mechanism that permits a language-level call on one machine to be automatically turned into a corresponding language-level call on another machine. Such a mechanism will require a network protocol to support the transfer of its arguments and results. The term Remote Procedure Call is sometimes used to describe just this low level transfer protocol. For instance, the Newcastle Connection uses a remote procedure call protocol [Shrivastava 82], but this is not integrated into a language and merely supports a few standard hand coded routines, which must explicitly encode and decode their arguments and results from network buffers.

The most thorough investigation to date of implementing a language-level RPC system has occurred at Xerox PARC with the Mesa [Mitchell 79] and Cedar languages. (Cedar is a modified version of Mesa, developed for the Cedar programming environment project [Teitelman 84].) Nelson [Nelson 81] investigated several potential mechanisms and produced the Emissary design for a Mesa RPC system that aimed to make remote calls totally equivalent to local calls in all but performance. Later, Birrell and Nelson [Birrell 84a] developed an efficient mechanism for Cedar which, while not realizing many of Emissary's goals, became a standard tool in the Xerox development system.

The Cedar RPC system is based around RPC stub modules that are automatically generated by a special program (called "Lupine") from standard Cedar procedure interface specifications. The stub that is to run in the client supports the given procedure interface and transfers its arguments into a network buffer (a process Nelson called "marshaling"). It then uses RPC specific protocols to call the server stub and finally decodes the results

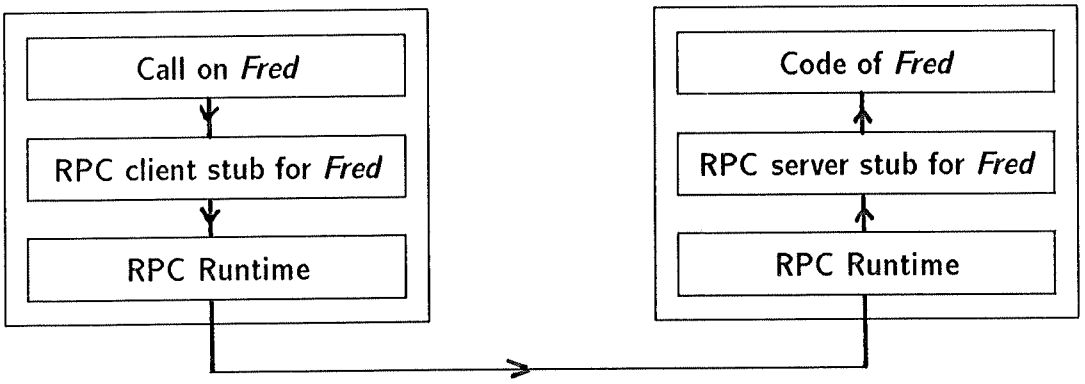

Figure 1. An RPC in Cedar

from the network buffer and returns them to the original caller. The server stub waits for network calls, decodes their arguments and calls the destination Cedar procedure. Thus a client's RPC on a procedure *fred* takes the form of a standard Cedar procedure call on the client stub module *fred* which arranges for the server stub modules to call the real *fred* procedure on its machine (see Figure 1).

The stubs approach had the significant implementation advantage that it did not require any changes to the Cedar compiler. It also had the side effect that remote calls are indistinguishable from local calls in the Cedar source code despite the fact that they have different call semantics. Because of this lack of special syntax for RPCs, the only way that call-specific control information can be passed into the RPC mechanism is by quoting it as an argument to the call. Thus when the encrypted call facility [Birrell 84b] of the RPC mechanism is required, an argument of the encryption related type *conversation* must be included in the procedure's specification. When Lupine detects such an argument it will generate stubs that use the encrypted form of RPC and the server stub will use the *conversation* argument in performing the call. This argument represents control information on how the call should be performed and is not logically part of the call at all, even though it must be delivered to the called procedure.

2.4.3 Argus.

Both the original Guardians proposal and the Cedar RPC system aimed to ease the production of distributed software, but only through the provision of robust, type-checked inter-machine communication. The Argus language design [Liskov 82a], [Liskov 83a] aims

to go much further. By embedding support for atomic actions within the language, it aims to ensure that the system is automatically resilient to node crashes.

Argus is an extension of the Guardians system. Each guardian is now viewed as being principally represented by a set of "stable objects", whose state is maintained in some form of stable storage which can survive crashes. A guardian will also have some "volatile objects" which are only used for temporary working and are not retained in stable storage. When a guardian crashes its volatile objects will be lost, but its stable objects will be preserved and the guardian will execute special crash recovery code to restore its state.

Argus's atomic operations are referred to as "actions". Each action contains a series of "subactions" which are seen as a single indivisible operation by processes that are not involved in them. Each action should always leave Argus's data structures in a consistent state.

Argus cannot guarantee that user level operations are genuinely atomic, but it does provide many built-in types whose representations are stable objects and whose operations are atomic, e.g. atomic records and atomic arrays. Each action or subaction that accesses an atomic built-in type takes a lock on that object on behalf of its top-level action. When a top-level action commits, then all the component atomic objects which it has modified are written incrementally to stable storage and their locks freed. When a subaction commits, its modified objects remain locked until its top level action commits, in which case the objects are written out, or until the top-level action aborts, in which case the objects are wound back to their previous values. (Within an action, subactions may not access data locked by other active subactions, but they may claim locks that are being held by the top level action on behalf of committed subactions.)

Actions are performed at other Guardians by calls on "handlers", which are essentially remote procedures. Each handler call constitutes a subaction which may terminate by returning results, signalling an exception, returning abort results, or raising an abort exception. Remote actions may be aborted by the Argus system if they cannot contact the destination Guardian.

The implementation of Argus has only begun, so its performance can only be guessed at and the suitability of its primitives is unproven. It is clear that even with its linguistic

support great care must be taken in programming supposedly atomic clusters if genuine atomicity is to be achieved. Liskov gives an example of how the incremental nature of the updates to stable storage can lead to problems if there is a crash part way through the process [Liskov 84]. Moreover, in discussing a mail system program written to demonstrate Argus's syntax, Liskov explains how the use of atomic types has led to an unexpected loss of concurrency and has introduced considerable scope for deadlock. This is because all atomic objects that are accessed during a operation must be locked for the duration of the top-level containing transaction. It appears that programs written in Argus must still be written with great care if both atomicity and a high degree of concurrency are to be achieved.

Chapter 3. Design Issues.

The motivation of this thesis was to provide better support mechanisms for the implementation of programs in distributed systems. Now that several existing solutions to this problem have been described, the underlying design issues will be discussed and the design decisions that led to the Mayflower RPC mechanism will be explained.

3.1 Types.

The provision of a consistent view of data structures passed between machines is a useful bottom layer for an implementor to build on. If this is provided merely by convention then it is an unreliable tool and implementors must take considerable care to avoid type errors. Thus it was decided to provide language-level interfaces which could be type-checked by a compiler and whose types could be automatically encoded and decoded from buffers by the language's run-time system.

The provision of compile time type-checking does not guarantee that type mismatches will not occur at run-time. Two modules which depend on each other may be compiled together and one module may then be modified and recompiled with a different interface. Thus, though the two modules have been type-checked they have inconsistent interfaces. In single machine programs the linker may check that all the modules it deals with have been compiled with consistent images, but this is not possible for multi-machine programs, whose elements may be linked separately, as well as being compiled separately. Since interface changes may occur frequently during program development, it was decided that consistency checks should be made at run-time to ensure that remote interfaces conformed to an expected type, thus ensuring that the mechanism was type-safe, as well as being type-checked.

3.2 Message passing compared to remote procedure call.

Both type-safe intermachine message passing and type-safe remote procedure call appear potentially useful tools for building distributed systems. These two alternatives are clearly duals in the same way that message passing operating systems and monitor

based operating systems are duals [Lauer 79]. That is, a procedure call and return can be represented by an exchange of messages and passing a message can be represented by forking a process to call a remote procedure returning no results. Given their duality, the decision to go for one primitive rather than the other is somewhat arbitrary. It would be possible to support both primitives in a single language, but offering a choice of two ultimately equivalent mechanisms in the same language might only serve to complicate the implementor's task.

Message passing initially appears the more flexible of the two alternatives, since a single message can result in zero, one, or many responses, and the responses need not come directly from the original message destination. For example, a distributed filing system might arrange that all its fileservers automatically forwarded file transaction requests to the fileserver where the file was located. Thus a request message would be sent to any fileserver and the reply message might come back from some other fileserver. However in practice this flexibility rarely seems to be used.

The Cambridge ring system offers simple message passing as the main low level primitive, using ring packets. However, most simple transactions take the form of an exchange of request and reply packets (the SSP protocol [Ody 79]). The option of sending a packet to one machine and getting a reply from another has rarely been used. Some servers send multiple replies in response to a single request. For example, in response to a read request, the Cambridge fileservers send a multi-packet message holding the data and also a separate packet holding control responses. However these two separate messages can be regarded as part of the same logical reply. A rare example of a request causing logically distinct multiple replies is the Clock service, which in response to a simple SSP transaction will send clients "tick" messages at regular intervals.

The Tripos operating system [Richards 79] at Cambridge is based on inter-process message passing. In theory no structure is imposed on message transfers, but in practice an exchange of call and reply messages is almost invariably used for inter-process communication.

If a message passing protocol were to avoid acknowledging incoming messages (leaving error detection and recovery to higher level software) message passing would be more

efficient than RPC at handling cases where requests do not require individual acknowledgement.

In favour of RPC, it can be observed that most languages are fundamentally procedural, so that using procedural interfaces for remote communication avoids an unnecessary conceptual change. The procedural emphasis of existing languages means that low level library routines involved in remote interactions will tend to offer procedural interfaces to higher level software. The tendency in both Tripos and the Cambridge Ring for message passing to degenerate to request/reply exchanges tends to confirm that the procedural model is the more satisfactory of the two.

After originally proposing to use a typed message passing system, the Argus designers moved to a procedural system, arguing that this provided a more convenient base for the construction of reliable software [Liskov 82b].

3.3 Semantics.

The largest single problem in integrating remote procedure call into a language is the need to reconcile the semantics of remote calls with those of local calls. This section discusses the extent to which remote calls should be made to look like local calls. Before doing so it is necessary to consider the environment in which RPC will be used, in particular the forms of errors that may occur.

3.3.1 The target environment.

This thesis is principally concerned with language-level RPC mechanisms designed for high performance local area networks, rather than for slower wide area networks. So it may be useful to consider experience with the Single Shot Protocol (SSP), which is widely used in communications with public servers in the Cambridge Distributed Computing System. This protocol involves an RPC like exchange of request and reply packets, even though it does not support the linguistic features of a true RPC system.

An SSP transaction simply involves the client transmitting a request packet to the server and waiting for some user-specified time for a reply packet to come back from the

server. SSP itself involves no retries.

The Cambridge Ring has a sufficiently low error rate that checksum errors due to packet corruption are virtually unknown. The two main forms of SSP failures are persistent transmission errors and timeouts.

The low level protocol handlers on Cambridge Ring systems normally try quite vigorously to get transmitted packets to their destinations by repeatedly trying the transmission over a period of several hundred milliseconds. The normal cause of persistent transmission failures is that the server's ring interface is congested, has crashed or is switched off.

The inter-ring bridges will occasionally drop packets, but only because of persistent failures in transmitting to the destination node. This is one cause of well chosen timeouts expiring. Timeouts also expire, as intended, when the target server fails to carry out the transaction within the required time, e.g. due to a crash or to congestion at other servers.

Thus, both persistent congestion and failure to meet a timeout reflect applications layer problems rather than being based on communications failures.

3.3.2 The end-to-end argument.

It might appear that the provision of a completely reliable inter-machine communication mechanism is essential to the construction of reliable distributed applications. However, it has long been argued that since communication failures are only one of the classes of failures that a robust application must protect itself against, the implementors of a robust application will tend to perform "end-to-end" checking to ensure each transaction's success. This implies that communication errors will be detected by the end-to-end check and can be recovered from by the same mechanisms as exist to handle other errors. Thus the failure rate need only be reduced to some acceptably low level rather than needing to be reduced to zero.

In their discussion of this issue, Saltzer, Reed and Clark [Saltzer 81] considered the case of a careful file transfer mechanism. Errors in the file transfer might occur in the network, in the network protocol handlers, in the disc input/output systems or in the file transfer software itself. They note that one way of increasing overall reliability would be

to try to increase the reliability of each component in the transaction. However, they argue that the best way of safeguarding the transaction would be to checksum the entire file before and after its transfer. The transaction would then be repeated if this were incorrect, i.e. if there had been an error in any part of the transfer. The probability of a transfer failing due to a communications error will rise with the size of the file, so the error rate must be kept below some threshold if there is to be an acceptable success rate in transferring large files. They argue that any lowering of the error rate beyond this is merely a potential optimization, serving to reduce the number of file retransmissions. If the error rate of the network is naturally low, the costs of checksumming each network packet and using error recovery protocols may be higher than that of occasionally retransmitting a file.

An interesting example of a file transfer mechanism that does not use end-to-end checking is that from VAXs to Motorola MC68000s on the Cambridge ring, which I had occasion to scrutinize when a minor corruption occurred in a large file transfer. The Cambridge ring has an acceptably low error rate (around one bit in 10^{11} [Spratt 80]) and the packets are also individually checksummed. Unfortunately the target 68000s' memories include no parity checking. The file transfer software was considered normally reliable, but was not guaranteed to be error free. Thus the network transfer appears to have been made by far the most reliable component in the overall transaction – without however guaranteeing the transaction's success.

The crux of the end-to-end argument is that robust applications must include error recovery code even if network transfers are totally reliable. This application level code may also be used to recover from occasional network errors. If the network is inherently highly reliable then the use of error detecting and correcting protocols will only serve to slow the application down.

3.3.3 Transparency and RPC.

In his thesis, Nelson included the following definition:

> *Transparency.* Two programming languages mechanisms are *transparent* if they have identical syntax and semantics. In particular, a transparent language-level RPC mechanism is one in which local procedures and remote procedures are (effectively) indistinguishable to the programmer.

Nelson argued that RPC transparency is both desirable and achievable. However, his Emissary design for a Mesa RPC mechanism does not fully meet his own definition. For example, Emissary does not permit pointer types to be passed as arguments to remote procedures. More importantly, Emissary does not properly handle the passing of objects, such as semaphores, for which concurrent access is important. Arguments passed by reference via Emissary are copied to the target machine at the start of the call and copied back afterwards. If other processes on the transmitting machine update the transmitted data structure during the call, this change will be lost when the call completes.

Emissary guarantees to execute the call once and only once in the absence of crashes (Nelson refers to this as "exactly-once" semantics). In the presence of server crashes the call may be executed several times, but only the results of the last call will be seen by the client ("last-one" semantics). This is similar to what happens when a local call fails. There is some difference however, in that the client will keep running with its pre-crash state without ever being made aware that a crash had occurred. For example, if the called module is maintaining some internal state in the forms of sessions, the client may find that its session identifiers have suddenly become invalid in a way that would never happen with a local call.

Nelson proposes no completely satisfactory solution to these problems. To provide uniform treatment of arguments passed by reference, Emissary could be modified to perform remote memory access operations to access such arguments, but Nelson rejected this as being unacceptably expensive. To help achieve totally transparent call semantics, clients could be crashed whenever their servers crash. Obviously this destroys any attempts to gain resilience from distributing an application.

It is interesting that the Birrell-Nelson RPC system for Cedar, which postdates the Emissary design, places much less emphasis on transparent semantics, though a transparent syntax is still used. Binding is under explicit program control. Persistent errors in communicating with servers will cause the client RPC mechanism to raise exceptions.

Neither the Emissary nor the Cedar RPC systems can conceal one other important distinction between local and remote calls – performance. A local call may be up to 100 times faster than an equivalent remote call using the Cedar mechanism.

Thus, it appears that a totally transparent RPC mechanism with acceptable performance is unattainable, so that an implementor must be aware which calls are local and which are remote. As a consequence of this it seems undesirable to hide the nontransparent semantics of remote calls behind a totally transparent syntax or to insist that implementors be denied the opportunity to exploit more flexible call semantics.

One aim of the Cedar RPC system is to enable a program to be written and tested, and then arbitrarily fragmented, without source code changes, so as to be distributed over several processors. However, the performance costs and the semantics of remote calls are such that the units of distribution and the nature of their inter-communication ought to be regarded as fundamental design decisions that should be taken early in the design of a program, rather than being made after its implementation.

3.3.4 Error handling.

When a client RPC mechanism encounters an error in communicating with a server, it can either attempt recovery itself or pass the error up to the application layer as an exception. The Emissary and Cedar RPC systems both aim to take recovery action within the RPC mechanism wherever possible. The Cedar mechanism will, however, eventually deem certain error conditions to be fatal and report them to the applications level, but normally only after attempting recovery for several minutes. However, in many cases the best response to an error may be application specific. For instance, when calling a congested server it is important not to call too persistently, as this may only overload the server further, by bombarding its network interface with calls it will instantly discard. At the same time is also important not to back off too far, otherwise there may be an excessive delay in getting a call through.

Passing error exceptions up to the application level also permits application specific error action to be taken immediately. For instance, a file handler might issue a warning message in response to persistent congestion at a fileserver as well as adopting some selective retry strategy. Other client programs might try to contact alternative servers when faced with persistent errors at one particular server.

3.3.5 Resilience and atomicity.

In any system, many applications which wish to provide a reliable service will need to ensure that parts of their data structures remain intact and internally consistent across machine crashes. Lampson [Lampson 81] describes how this can be achieved by preserving such data structures in stable storage which is only updated by atomic actions. These atomic actions must ensure that the required consistency is maintained within the data structures and that client processes' transactions are either totally executed or have no effect. Gray discusses how such transactions can be efficiently performed in a distributed system through the use of a two phase commit protocol [Gray 78].

The atomicity model appears a useful one for constructing resilient servers, although it requires considerable care to implement. Increasing the complexity of the software may only compound the scope for software errors. Even careful implementors will have difficulty in protecting their data structures from software errors, even if they were to use hardware checks when updating stable storage [Needham 83]. For these reasons, it appears desirable to provide some degree of software support for atomicity rather than requiring that each implementor re-invent the wheel. It is as yet unclear at what level this support should be provided.

The Argus designers decided to embed atomic actions within their language. This has advantages over a system based on library procedures, principally that implementors cannot accidentally forget to lock or unlock objects. However, the Argus system has two major disadvantages.

Argus aims to provide atomic operations on its built-in atomic types, but to provide the same operations syntax as for non-atomic types, despite the significantly different semantics. This means that using Argus's standard array element accessing syntax, "a[x]" on an atomic array causes that array to be locked for the duration of the top-level containing transaction. The locking of objects in complex transactions requires care if deadlocks are to be avoided, and it seems undesirable to hide this problem behind transparent syntax. Similarly, the automatic nature of the commit process conceals the fact that care must still be taken to ensure that the overall transaction is not left in an inconsistent state in the event of a crash during Argus's incremental storage of updated objects. In both

these cases there appears a real danger that Argus's syntax has only served to conceal the pitfalls of transaction design, without entirely removing them.

The other major problem with Argus is that it forces its users into a single, possibly over simplistic, locking strategy. Argus acquires either a read or a write a lock on every built-in atomic object which is touched during a transaction and only releases these locks when the transaction is committed or aborted. Liskov states that Argus does not attempt to detect or break deadlocks and relies on application level timeouts and aborts on top-level actions, saying "These timeouts will generally be very long, or will be controlled by someone sitting at a terminal" [Liskov 84]. This approach is not altogether satisfactory. Processes beyond those which created the original deadlock may be stalled if they need to access deadlocked data, thereby creating an expanding circle of waiting processes. Relying on very long time-outs or human intervention may cause unacceptable delays.

There have been several suggested schemes for avoiding deadlocks. Gray [Gray 83] advocated aborting the entire transaction and retrying after a few seconds, if a locked record were encountered at any point. He argued that contention for locks was very rare, so that this simple deadlock avoidance mechanism will involve only very occasional inconvenience and will avoid the overheads of more complex schemes. A similar scheme is to adopt an optimistic strategy and avoid locking entirely [Kung 81]. Kung's approach involves keeping track of all transactions' read and write requests, and deferring all updates until commit-time. This permits interfering transactions to be identified and aborted when they come to commit. Both Gray's and Kung's strategies would cause problems if some records were very frequently updated, but the designers of a large distributed system may be expected to avoid designing such bottlenecks, which can be expected to cause delays whatever locking strategy is used.

Schwarz and Spector [Schwarz 84] have proposed a formal model for transaction locking that is very different from Argus's. They propose that each abstract type should be able to define its own transaction locking discipline so as to achieve the maximum degree of concurrency. For instance, a directory type might permit several independent transactions to update a directory concurrently, provided they did not conflict over individual entries. Thus type-specific implementation knowledge is utilized to permit transactions which update the same object to proceed concurrently, provided there is no genuine conflict.

Given these different locking strategies, it appears that individual applications may wish to adopt particular locking disciplines that are appropriate to their needs. Thus, it appears inappropriate to enforce a single locking discipline on all transactions in the way that Argus aims to do.

The provision of appropriate inter-machine communication primitives and the provision of support for transactions are two distinct problems, with type-safe inter-machine communication merely providing a base on which application level transactions can be built. It was decided to concentrate on remote communication primitives in this thesis while bearing in mind that they only represent one component of an overall strategy for providing reliable distributed processing.

3.3.6 The case for a nontransparent mechanism.

In the preceding sections it has been argued that total transparency cannot be achieved between local and remote calls and that permitting the application layer to handle transient failures may improve performance and increase flexibility. One approach to RPC is to try to minimize the inevitable differences between local and remote calls, with the aim of simplifying the applications programmer's task in return for some loss of flexibility and performance. This is the approach followed in the Emissary and Cedar RPC systems.

An alternative approach is to recognize the inevitable nontransparency and to provide the applications programmer with the opportunity of exploiting the increased flexibility and higher performance of lightweight protocols. The assumption here is that remote interfaces will be designed to withstand crashes and other major failures, so that handling transient failures will involve the designer in little or no extra effort. This is the approach investigated in this thesis.

It is tempting to see the RPC interface provided by a server as its public interface, which must be negotiated by each implementor who wishes to access the underlying service. However, while it may be desirable to permit the RPC interface to be called directly only by those implementors requiring unusual retry strategies or unusual flexibility, it may also be useful to provide a standard library package for accessing each remote server. This library package could be regarded as an integral component of the overall service being

provided, with the RPC interface being regarded primarily as an interface between separate components of an application that has been produced by one team of implementors. This removes the decisions of how to handle failures from the undemanding implementor, who may treat this library package like any other.

If remote interfaces are regarded primarily as internal interfaces (which may also be called directly by some demanding applications) then this increases the utility of a flexible call system. The implementors of a server can use their knowledge of its characteristics to adopt retry strategies and to deal flexibly with transient failures in more effective ways than less knowledgeable implementors could.

This strategy for implementing remote servers does not diminish the role of language-level RPC (since the provision of type-safe interfaces is extremely useful within an application), instead it should permit the greatest benefits to be obtained from a flexible RPC system.

3.3.7 Two alternative RPC call semantics.

Once the decision has been taken to offer different call semantics for local and remote calls, the next issue is to decide what the RPC call semantics should be. Rather than constraining implementors to one set of semantics for remote calls, it was decided to offer two forms of remote call.

From the foregoing discussion, the utility of a lightweight form of RPC, where the call mechanism takes only minimal error recovery, should be clear. Following Spector's taxonomy [Spector 82], this will be referred to as the "Maybe" semantics. "Maybe" requires that the call is attempted once and only once, and that all errors are reported to client software. For this to be most useful there must be some way for a timeout to be specified for each call, so that the RPC mechanism abandons the call if it cannot be carried out within the user's required time.

The Maybe call semantics anticipate a very low network error rate, such as may be found on most local area networks. Thus most errors reported to client software are application specific errors caused by server congestion or server unavailability, and there is little expense in having the very occasional network error also handled at user level. If,

however, the underlying network is extremely unreliable, it might be appropriate for the RPC system to maintain some form of error-recovery protocol so as to reduce the network failure rate to some acceptably low value. Such a protocol must be regarded as a mere optimization, which must not significantly delay calls or mask failures which occur at the destination node.

While some clients may be interested in transient errors, it is likely that many will be unable to take action other than merely retrying the call until it succeeds or until further retries seem pointless. Such a layer can easily be implemented on top of the Maybe system, but this would tend to offer "last-of-many" semantics whereby the call was performed one or more times and the final results retained, rather than the more desirable "exactly-once" semantics. This is because a call that is in fact executing successfully may be incorrectly timed out and repeated. When the reply to the earlier call arrives, the Maybe mechanism will reject it, since it is not the reply to a currently executing call. (Alternatively, the Maybe mechanism may simply lose the reply message.) For those calls which are naturally idempotent (such as reading from an absolute position in a file) this will be unimportant. For other calls (such as obtaining an interlock on a file), it may be highly undesirable to have the call executed twice.

For user level code to avoid repeated execution of Maybe calls, it would be necessary to transmit a tag in each remote call and for the server to retain a list of recently executed tags and their results. Then if the server saw a repeated call it could return the original result once more. It is reasonable for the RPC system to provide a special mechanism to satisfy this function, rather than requiring each server to implement its own system. This system will be referred to as the "Exactly-once" system, since it should ensure that the call is executed once and once only in the absence of prolonged server or network failures. The requirement here is for the RPC mechanism to repeat the call a large number of times until it succeeds or is deemed impossible to carry out. The RPC system must ensure that repeated calls are discarded at the server, so that the call is only executed once. As with Maybe, a user specifiable timeout is useful for Exactly-once, but in this case representing a retry time, after which the call should be retransmitted.

Maybe and Exactly-once represent different ends of the spectrum of call semantics. Intermediate semantics such as "at-least-once" (where the number of times a call is executed

is unimportant) can easily be constructed from Maybe.

Superficially, Exactly-once may appear to be a more suitable base than Maybe for constructing reliable distributed applications, but this is not necessarily true. Exactly-once makes no guarantees in the face of server crashes or prolonged network failure. If an application wishes to be genuinely resilient then it must incorporate its own safeguards against server crashes. This may involve implementing its transactions with servers as multiple component atomic actions, which is likely to mean that a higher level call discipline will be constructed from the type-safe primitives provided by the RPC mechanism. Given such a higher level discipline, designers may prefer the probability of obtaining higher performance and a faster notification of problems from Maybe than the slow and only half-sure semantics of Exactly-once. Exactly-once is likely to be most useful in constructing simple minded packages that do not wish to attempt recovery from server crashes. For example, an inter-machine character pipe mechanism built on top of an RPC system may be content to abort a pipe when faced with a crash at the pipe's destination machine.

3.4 Orphans.

So far the discussion of RPC semantics has only considered network failures and server crashes. It is now necessary to discuss the implications of client crashes.

Consider the case of a service running on some machine A which as part of its normal business wishes to carry out some lengthy transaction on some server B. It might be desirable to model this transaction as a single remote procedure call. However, if A issues the RPC and then crashes, the call will of course continue to execute at B. A call that continues to run in this way after its caller has crashed is known as an "orphan". When A is restarted it may be unaware of the existence of this orphaned call and attempt to carry out other transactions on B. The orphaned call may interfere with these new calls in undesirable ways, e.g. by overwriting data structures they have updated or by denying them access to files it has locked.

3.4.1 Nelson's orphan extermination schemes.

The potential presence of such orphaned calls represents a significant difference between local and remote calls. As part of his quest for a transparent RPC mechanism, Nelson proposed four separate schemes for automatically coping with this problem.

"Extermination" requires each restarting node to locate and terminate any orphaned calls that originated with it. This scheme is fairly complex due to the need to cope with multiple crashes. Nelson noted that recovery might have to be postponed until all crashed (or removed) servers had also been restarted. In such a case he suggests one of the other three schemes should also be used.

"Expiration" requires that each remote call have a time limit associated with it by the caller. If this time limit were exceeded, the call would be assumed orphaned and killed. When a server restarts it will wait for its maximum call time before resuming work, so as to allow all orphaned calls to die. If the time limit is too short calls may be killed unnecessarily, if it is too long crash recovery may be unacceptably delayed.

Both "regular reincarnation" and "gentle reincarnation" only guarantee to kill orphaned calls when they interfere with new calls, either by executing on the same machines or by calling the same servers. These schemes require an epoch identifier to be maintained across the entire network. When a server restarts and finds "exterminate" inadequate to kill its orphans it will cause a new epoch to be started. Each node in the network is required to perform "reincarnation" when it sees a new epoch has started. "Regular reincarnation" requires each node to abort all remote calls at each epoch change. This amounts to a network wide reset of all remote processing. "Gentle reincarnation" merely requires that each node verify that it is not executing any orphaned calls. This requires a check to be made back to the root process in every chain of remote calls. The fact that both these "reincarnation" solutions require action on all network nodes is disturbing. On a small network the cost may be tolerable, but on very large networks the overhead may be unacceptable.

Nelson assumes that orphaned calls can be aborted as soon as they are detected. The Argus designers [Liskov 84] observe that orphan deletion must be postponed while the orphan process is executing in a critical subsection (equivalent to holding a monitor lock).

They suggest an orphaned call that stays in a critical region for too long can always be deleted by crashing its Guardian.

Nelson did not actually implement any of his orphan elimination schemes, nor did the later Birrell-Nelson RPC system for Cedar attack this issue. Since each of Nelson's proposals has its own problems it may be useful to consider what advantages even successful orphan extermination would offer.

3.4.2 Other aspects of client crashes.

Nelson's motivation appears to be a belief that terminating orphaned calls will, by itself, terminate any remote actions a crashed node may have started. This is not necessarily the case. During its execution a node may establish a variety of different forms of long term state on remote servers. This state will not always take the simple form of orphaned calls and may be highly application dependent, so that recovery from a client crash may require application level action.

Consider the simple example of a printer server which accepts requests to print named files. Such a server might maintain state in stable storage so that it can acknowledge print requests as soon as they are received, rather than when they are actually carried out. Thus an RPC request to this server will merely update the print queue before returning. A client that submits a request and then crashes will not have this request cancelled by an orphan extermination scheme. The restarted client may, however, find itself unexpectedly denied write access to one of its files when the printer server actually comes to print it.

The Tripos Filing Machine at Cambridge [Richardson 83] gives a further example of the importance of application level recovery action after client crashes. The Filing Machine is accessed via an RPC like protocol, but maintains filing system authentication information for each of its active clients. When it detects that one of its clients has crashed, it closes any open files and deletes the authentication information. The filing machine makes no attempt to suppress orphaned calls, since calls are sufficiently short that any active calls will have terminated by the time a crash is detected.

3.4.3 Support for application level recovery.

It must be possible for servers to detect client crashes if they are to undertake application level recovery. This might be done by requiring restarted machines to notify their previous servers of their crash. However, this may result in long delays. Alternatively each server could require that all clients maintaining state on it had to call it a regular intervals. This scheme has been used for a number of servers in the Cambridge Distributed System. This can result in a large number of calls being executed as each client issues reassuring calls to all its servers.

As part of his design for a new Resource Management service [Craft 83] Craft has designed an Event Notification server. One use of this is to centralize the detection of client crashes. Each resource is required to register itself with the event notification server and to operate a "dead man's handle" periodically to demonstrate it is still alive. Interested parties can then request to be notified if the dead man's handle expires. This service is a useful higher level tool for use with the current RPC system.

3.5 Configuring a system.

It is necessary for each component of a distributed system to be able to locate those other components whose services it wishes to use. The simplest solution to this configuration problem is to bind into the program at link time the addresses of all the services it requires. However, this strategy means that the program is unable to adapt at run time to changes in the locations of servers or to the addition of new instances of servers. Moreover, a program in a distributed system may at various times require different servers to be available to it, but it may not require all these servers to be available at all times. It was thus considered desirable that the RPC system should permit server binding to be under program control, so that a program might bind and unbind as necessary during its execution. This permits easier switching to alternative servers and also permits the program to use dynamically instantiated servers, which have only been temporarily allocated, to perform some functions. These issues are more fully discussed in the chapter on binding (chapter 5).

3.6 Summary.

In this chapter several significant differences between local calls and remote calls have been discussed which can only be eliminated at considerable cost:

- remote calls do not share memory with the calling process, restricting argument passing semantics (3.3.3)
- RPC clients can survive server crashes and detect their consequences (3.3.3)
- remote calls may continue to execute after client crashes (3.4)
- remote calls may acquire state on remote machines that will survive client crashes (3.4.2)

Furthermore, it has been observed of network transfers that:

- communication difficulties often reflect application level problems (3.3.1)
- making network transfers totally reliable does not guarantee a call's success (3.3.2)

For these reasons, it has been argued that total transparency between local and remote calls is both excessively expensive and undesirable. Instead it has been proposed that the differences between local and remote calls should be made explicit and that implementors should be permitted to respond directly to communication failures. Thus, it is proposed that two alternative call mechanisms, Maybe and Exactly-once, should be provided to cope with differing application requirements. The Maybe mechanism passes all errors directly to user software, permitting immediate, application specific, error recovery. The Exactly-once mechanism attempts recovery from communication failures and only reports unrecoverable failures to user software.

Chapter 4. Extending CLU for RPC.

4.1 Choosing a language.

It would be highly desirable for an RPC mechanism to permit calls to be made from client modules written in any language to server programs written in any other language. Unfortunately the problems of cross-language type-checking and data conversion appeared too large to be tackled as part of this thesis. Superficially similar constructs in different languages often have subtly different properties, which may be hard to reconcile. For instance, arrays in CLU are highly dynamic objects, whose bounds may grow and shrink, whereas arrays in Modula-2 are relatively static, having their sizes fixed at creation time.

For this thesis only a single language RPC mechanism was investigated. A fairly small number of languages were readily obtainable for the available hardware (Motorola MC68000s) and ultimately a decision had to be taken between using the relatively straightforward conventional language Modula-2, or the richer, but more experimental language CLU. CLU was eventually chosen, partly just because it permitted investigation of a wider range of problems, but partly because its structure appeared well suited to distributed processing.

CLU [Liskov 81] was developed at MIT from 1974 to 1979, mainly as a base for experimenting with abstract types. CLU is oriented around **clusters** which provide sets of operations on abstract types whose internal representations are visible only within their defining clusters. (See Figure 2 for an example.) All objects in CLU are regarded as having independent existence on the CLU heap and are only visible to other objects as pointers. These object-oriented features in CLU provide a relatively clean model for transferring objects between machines.

```
stats = cluster is create, add, average
% Cluster to maintain sets of simple statistics.

% The representation describes a simple set of statistics.
rep = record [total: real, entries: int]

create = proc ( ) returns (cvt)
% Create a new representation object and return it as an abstract type.
% (Cvt indicates we wish an argument or result object to be seen externally
% as our abstract type, but we want to view it internally in terms of our
% representation.)
    return (rep${total: 0.0, entries: 0})
end create

add = proc (c: cvt, value: real)
% Update the representation of the object "c".
    c.entries := c.entries + 1
    c.total := c.total + value
end add

average = proc (c: cvt) returns (real)
% Calculate an average from the information in the object "c".
    return (c.total / (real$i2r(c.entries))
end average

end stats
```

Figure 2. A very simple CLU cluster.

CLU has a powerful exception mechanism [Liskov 79b] which can be thought of as a way of expressing alternative return paths for a procedure, rather than a mere error handling feature. This meant that any additional exceptional events that might be encountered by the RPC mechanism during a remote procedure call could be cleanly integrated into the existing language-level exception handling mechanism. Both when signalling an exception and when returning normally, a CLU procedure may return an arbitrary number of results of arbitrary types. This largely avoids the need to use Modula-2 style **var** (i.e. call-by-reference) arguments as a way of returning extra results, thus clarifying the nature of the data being returned.

4.2 Integrating remote procedure call into CLU.

An early decision was taken to support only simple remote procedures and not to support either generic remote procedures (referred to in CLU as "parameterized" procedures) or remote iterators. (Iterators are loop control modules that are re-entered whenever the caller goes round its loop.) Iterators were seen as inappropriate for remote use, due to the high degree of interaction between the calling and called modules. Both iterators and parameterized procedures were seen as raising too many CLU specific issues of detail that would not generalize to other languages. (Parameterized **remoteprocs** in particular pose several problems. At present parameterized procedures are instantiated at link-time, but run-time instantiation would be necessary for parameterized **remoteprocs**, since the range of their instantiations depends upon their separately linked RPC clients. It is not clear under what circumstances such instantiations should be discarded.)

Birrell and Nelson chose to add remote procedures to Cedar by using a preprocessor to produce stubs rather than by modifying either the language or the compiler. This approach may have advantages where there is a desire to avoid compiler modifications but it led to the non-transparent semantics of remote calls being masked by a transparent syntax. The CLU RPC system aimed to emphasize the special nature of remote calls, so it was decided to modify the CLU compiler so as to provide a special syntax for remote procedures and remote calls. This also permitted extra syntax to be added for RPC control information (thereby avoiding the Birrell-Nelson problem of having to provide such information in the form of extra arguments to the call) and allowed the compiler to generate optimized code for entering the main body of the RPC mechanism.

Remote procedure definitions were given the same format as local procedure definitions, but with the keyword **remoteproc** replacing **proc** in the header. Argument, result and exception definitions were unchanged, as was the syntax for returning results and signalling exceptions. For instance:

```
tester = remoteproc (n: int) returns (bool)
% Trivial remote procedure.
    if n = 0
    then return (true)
    else return (false)
    end
end tester

reserve = remoteproc (f: flight, s: seat_type, count: int)
            returns (array[seat_info], flight_status)
            signals (full (array[seat_type]), trouble(string))
    % Uses various abstract types concerning flight reservations.
    % Returns two result objects, may signal some exceptions.
    seats: array[seat_info], status: flight_status :=
                local_reserve (f,s,count) resignal full,trouble
    return (seats, status)
end reserve
```

A new syntax, the **call** expression, was introduced for performing remote procedure calls. The **call** expression can be thought of as a veneer around a normal CLU procedure invocation. The keyword **call** proceeds the invoked procedure's name and other control keywords may follow the invocation's arguments. The results of the call expression are of course simply the results of the call, e.g. :

```
b: bool := call tester (1)

seats, status := call reserve (f,s,1) resignal full,trouble
```

The results of RPCs may be used in the same way as the results of local calls, e.g. :

```
if ~ call tester (0)
then signal Failure ("tester failed")
end
```

In CLU all objects are represented via pointers so that local variables merely contain pointers into the heap. When a local CLU procedure is called it is given copies of pointers to the argument objects. This is referred to as "call-by-sharing" because any changes the called procedure makes to the argument objects will be visible to owners of the other

pointers. Since the use of remote memory operations to access the argument objects of a remote call appeared prohibitively expensive, it was necessary to use "call-by-copy" to transfer the argument objects to the destination machine. It would still be possible to obtain some of the effects of call-by-sharing by copying the argument objects back to their original locations after the call. However, in a concurrent system call-by-sharing cannot be totally emulated by this means, since other processes may access the objects while the call is proceeding. Because CLU permits several distinct objects to be returned by a procedure, it appeared reasonable to require that implementors explicitly return those objects whose changes they wished to be visible on the original machine. This gave remote calls pure call-by-copy argument passing semantics. This can be expected to boost performance on calls which involve the transfer to a server of large data structures which do not need to be returned.

The calls so far described have used the default call semantics, Maybe. If the alternate Exactly-once call semantics are desired then the keyword **zealously** is appended to the call. This keyword was chosen to try and emphasize the persistent nature of this call mechanism, without implying that its use guaranteed success.

call logger ("Kernel running low on heap!!") **zealously**

The keyword **timeout** followed by an integer expression representing a time in milliseconds may be appended to the call. For Maybe calls this represents the time after which the call should be abandoned. For Exactly-once calls it represents the recommended interval between retries of the call. If no timeout value is specified then the RPC mechanism will use a default value of a few seconds. E.g.

call logger ("Proceeding OK") **timeout** 1000

One further optional keyword **at** can be used for specifying a call address and is described in the chapter on binding (Chapter 5).

If an error happens during the execution of a remote call the RPC will signal either a *hard_error* or a *soft_error* exception together with an error code. *Soft_error* is only signalled by the Maybe call mechanism and indicates that an error has occurred on the call, but

that a retry of the call may succeed. *Soft_errors* include such things as timeouts expiring or apparent congestion at the server machine. *Hard_error* is signalled by both the Maybe and the Exactly-once and indicates that an apparently unrecoverable error has occurred. *Hard_errors* include such things as persistent inability to contact the server when using the Exactly-once mechanism, or the receipt of a denial from the server that it supports the called **remoteproc**. Thus, remote calls will normally have exception handlers attached to cope with these exceptions, e.g.

```
% Report we're all right, catch and discard any exceptions:
call logger ("Still OK") except
when hard_error, soft_error (reason: int):
    % Null exception handler body.
end
```

4.3 Implementing the extended syntax.

Because the **call** expression introduces new syntax to the language, the Mayflower CLU RPC system could not be implemented merely through an RPC stub generation mechanism. It would have been possible to use a combination of a stubs generator and a preprocessor where the preprocessor replaced each RPC **call** by calls on a corresponding stub generated procedure which had additional arguments for each possible control option. Such an approach appeared unnecessarily contorted. It proved straightforward to modify the CLU compiler to accept the new syntax and to convert **call** expressions into appropriate entries to the RPC mechanism. The control options (or their defaults) were stacked as arguments to the RPC system, together with a vector holding pointers to the call's argument objects.

Chapter 5. Binding.

This chapter discusses how remote procedures are identified at run-time and how remote calls are directed to an appropriate server.

5.1 The CLU library mechanism.

CLU aims to support the separate compilation of program modules consisting of clusters, procedures and iterators. The CLU library system is the mainstay of the inter-module type-checking system in current CLU implementations (all of which originate from the Programming Methodology Group at MIT).

Each interactive session of the CLU compiler maintains a "library" holding descriptions of the external interfaces of various CLU modules. Information in this library can be dumped into library files and later merged into the library of future compilation sessions. Libraries are initialized with the definitions of the standard CLU modules. Whenever a module is successfully compiled its definition is inserted in the library, replacing any previous definition of a module with the same name. When modules are type-checked any use of modules described in the library must be consistent with their stored interface definitions. (However, the compiler merely warns of references to modules not in the library.) The compiler also provides a method for checking and recording a module's interface without checking the module's internals. This permits the definition of sets of modules with circular dependencies.

The linker assumes that modules have been checked for consistency at compile time and blithely tries to link whatever it is given. It will only complain if certain very gross errors occur, such as missing cluster operations or procedures being called with the wrong number of arguments.

The fact that references to unknown modules are permitted by the compiler is inexplicable. Even when type inconsistencies are avoided, the VAX and 68000 compilers sometimes generate faulty code in compiling a call on an unknown procedure, due to ignorance of the number of results the procedure is returning. It would appear straightforward to treat the use of an unknown interface as a fatal error.

The other major problem is that interface compatibility is not checked at link time. This means that when the external interface of a module changes, the module can still be linked with other modules that were compiled in terms of the old interface.

One solution would be for the compiler to include in the object file sent to the linker full definitions of both exported and imported interfaces. The linker could then perform full type-checking of interfaces. This would be quite flexible but would probably slow down the linker significantly.

Another solution would be for the compiler to generate unique interface identifiers for each each new or changed module interface. It would be useful to associate separate interface identifiers with each operation of a cluster, so that adding an extra operation did not require existing code to be recompiled. It would then be possible (as in Mesa and Modula-2 implementations) for lists of imported and exported interface identifiers to be included in the object files so that the linker could check interfaces were compatible. This approach would require users to become more careful in their use of library files, since calls on an interface would have to have access to the library information produced when the interface was first compiled.

Even though there are defects in the current implementations, the standard CLU language is potentially type-safe. Extensions to the language should reflect this and the implementations of these extensions should avoid introducing new type-checking loopholes.

5.2 Remoteproc identifiers.

To permit the independent development of client and server programs, RPC client programs will normally be linked independently from the RPC server programs that they use. This means that interface compatibility cannot be guaranteed at link time and must be checked at run-time if type-safety is to be guaranteed. A system of compiler generated unique identifiers (UIDs) are used for this purpose. These UIDs are 62 bit values generated from a simple formula by the compiler. They are guaranteed to be unique, but they are not intended to be unguessable.

Whenever a **remoteproc** is compiled for the first time a new UID is generated for it by the compiler and this is saved with the **remoteproc's** descriptor in the library. Whenever the **remoteproc** is recompiled its new type is checked against its previous type. If these match then the old unique identifier is retained, otherwise a new one is generated. This UID is stored as part of the linkage control information in the CLU code file. The linker uses this information to produce an ordered table containing UID-to-program-address mappings for all the **remoteprocs** supported by the program.

The compiler will only regard a reference to a named **remoteproc** as valid if the library contains a descriptor for it, in which case the UID from the library is noted by the compiler. The compiler and linker act together to produce a list of the **remoteproc** UIDs used by a program together with a table which has a slot for each such use. This table is used at run-time to hold the current binding of each **remoteproc**.

5.3 Remoteproc bindings.

A programmer in a distributed system will frequently want to arrange for all calls on a particular **remoteproc** to be directed to a particular network address. For example, a programmer might desire that all calls on the **remoteproc** *Authenticate* were directed to a particular authentication server. It is thus desirable to provide a run-time mechanism for establishing long term bindings between **remoteprocs** and network addresses.

The RPC mechanism operates in terms of "binding objects". Each binding consists of a **remoteproc** UID together with a network address. The RPC run-time procedure for performing remote calls expects to be given a binding object as one of its arguments and will normally direct the call to that binding's address.

The RPC run-time mechanism ensures that for each **remoteproc** referenced in a program a default binding object is kept in the bindings table prepared by the linker. When the compiler generates a remote call on a named **remoteproc** it has to ensure that the current binding object corresponding to the **remoteproc** is picked up from the bindings table and passed to the RPC mechanism.

5.4 Updating the bindings table.

Programs in a distributed system often wish to select some particular server to provide a set of services. What this implies for the bottom level of the CLU RPC mechanism is that the program wants to update its bindings tables entries for all the **remoteprocs** supported by the server, so that their new default bindings point to that server.

Each RPC mechanism supports a low-level interface to permit RPC mechanisms on other machines to obtain copies of its list of supported **remoteprocs**. On demand an RPC mechanism will obtain such a list from another RPC mechanism at a specified network address. It will then compare this list against its own bindings table. For each match it will create a new binding object using the new network address and store this in the bindings table.

5.5 Remoteproctype objects.

Standard CLU provides procedure variables, of type **proctype**, and it seemed desirable to provide analogous remote procedure variables, of type **remoteproctype**. It then follows that **remoteproctype** variables should have network addresses associated with them, describing the location to which any call should be directed. This network address should describe the call destination of the **remoteproc** at the time it was assigned to the **remoteproctype** variable, so that a **remoteproctype** variable is seen as describing a particular binding of a **remoteproc**. Thus if a client wishes a server to call it periodically on some **remoteproc** *fred*, it can pass the server a **remoteproctype** variable describing *fred* without any risk of the server erroneously calling an equivalent *fred* at some other client.

Remoteproctype objects are represented directly by binding objects. (However a **remoteproctype's** compile-time type reflects the types of its possible arguments, results and signals, whereas a binding does not.) This means that an assignment of a **remoteproc** to a **remoteproctype** simply involves picking up an entry from the bindings table. Calling a **remoteproctype** is little different from calling a named **remoteproc**, except that the binding is that of the **remoteproctype's** representation, rather than an entry from the bindings table.

5.6 Sessions.

A binding can either be regarded as merely describing a particular **remoteproc** on a particular machine or as describing a particular incarnation of a particular **remoteproc** on a particular machine. In the latter case it would be necessary for the binding to include some kind of session identifier for the destination server, so that if a program has bound to a server and that server crashes and is restarted, subsequent calls on the old bindings will be rejected.

Birrell and Nelson argue that bindings should be broken when a server is rebooted, on the grounds that client software may wish to know about the potential loss of state at the server. However, it seems possible that session management of this sort is best handled in the client software itself and that little is gained from replicating it at a lower level.

Some simple servers, e.g. nameservers, may not maintain any state on their clients at all. In this case forcing clients to be aware of server reboots confers no obvious advantages.

If a server is maintaining some state for a client across a series of calls, it is likely that the server will require its clients to quote some kind of session identifier in all their calls so that this state can be identified. (For example on the Cambridge fileserver [Dion 80] an open file is identified by a 64 bit "temporary unique identifier".) A cautious server can arrange that these session numbers are unique across system reboots, (e.g. by including timestamps in its session identifiers) and signal exceptions on any clients quoting obsolete identifiers.

Important servers may chose to retain their state in stable storage. (For example, the authentication server in the Cambridge distributed system [Girling 82] retains a copy of its "Active Object Table" at the fileserver.) After being rebooted, such servers may be able to resume their previous activities without disturbing their clients.

For these reasons no concept of sessions was included within the CLU RPC mechanism. The contention is not that server crashes should be concealed from user software, but that the RPC mechanism need not go out of its way to try and distinguish different incarnations of a server, since application level software will easily be able to detect undesired restarts.

5.7 Call time binding.

There are times when the programmer may not wish to establish a long term association between a named **remoteproc** and a particular network address, e.g. when dealing simultaneously with a large number of identical servers. For example, an RPC based system of inter-machine pipes was implemented in CLU which involved each pipe manager in frequent remote calls to the pipe managers on other machines. Several such calls might be in progress to different machines simultaneously.

It would be possible in such cases to bind to each server in turn and assign the resulting bindings to **remoteproctype** objects associated with the server. Calls on a particular server would then take the form of calls on a particular **remoteproctype** object. However, this solution seems to mask what is actually occurring. The aim is not to call different **remoteprocs** instances at indeterminate addresses, but to call a particular **remoteproc** at a particular network address.

For this reason, the RPC syntax was extended to permit the destination network address of a call to be specified directly. The keyword **at** followed by an object of the *network_address* type may be appended to a remote call. This will cause the RPC mechanism to direct the call to the given network address instead of to the default binding of the named **remoteproc**.

```
orange: network_address := network_address$lookup("rpc-orange")
x: int := call fred (1,2) zealously at orange
```

Allowing remote calls to be sent to any network address without a preliminary binding phase permits great flexibility, but implies that the binding must be validated at call time.

It is necessary that every remote call incorporates some kind of identifier so that the destination RPC mechanism can direct the call to the correct piece of code. The obvious choice for this purpose is the **remoteproc's** compiler generated UID, which the RPC run-time mechanism knows both for calls to bound **remoteprocs** and for calls to explicit addresses. When decoding an incoming call request, the RPC run-time mechanism searches for the call's UID in its UID-to-program-address mapping table. If the UID is not found then an internal return code is sent to the calling RPC mechanism, which will cause it to signal a *hard_error* exception to the caller with the argument "rpc_destination_denies_binding".

5.8 Locating servers.

In order for the RPC mechanism to either bind a set of **remoteprocs** to a server, or to execute a call with call-time binding, the network address of the server must be known. This represents a higher-level binding problem, beyond what has been discussed so far.

The crudest approach would be to say that all servers resided at well known network addresses. This approach means client code must be changed whenever a service is moved to a different machine. This makes it extremely difficult to reconfigure the overall system, which may be necessary to cope with either short term hardware problems, or with long term changes in available resources.

The use of nameservers to map service names onto network addresses represents a more flexible approach. For example, the Cambridge nameserver [Gibbons 81] can, by human intervention, be updated to reflect a service being moved onto a different host. However, all that nameservers provide is a way of locating existing services. This is clearly useful, but represents a rather static view of the world.

The "processor bank" at Cambridge consists of a public pool of around 30 mini-computers which are allocated as personal computers to individual users via requests to a simple resource management service. It might be desirable in such a system for some servers to be loaded into processor bank machines on demand rather than running on permanently allocated hosts. For example a mail server might be loaded as a consequence of one client's needs and then be accessed by other clients. A new resource management server capable of handling such dynamic services is currently under construction [Craft 83].

The CLU RPC binding mechanism has been designed to cope with both of the last two possibilities, i.e. the use of either a nameserver to locate a service, or a resource manager to fabricate one on demand.

5.9 Related work elsewhere.

The Argus designers are still exploring the issues of binding and server location. They propose to use a form of nameserver called a "catalog" to locate Guardians, but its interface

is still undefined [Liskov 83a].

Birrell and Nelson base their locating of services around sophisticated nameservers, the Grapevine distributed database servers [Birrell 82]. The Xerox environment does not support any system for dynamically instantiating services and their RPC system does not consider such dynamic services.

Birrell and Nelson describe their low-level binding as occurring in terms of interface names. Individual procedures are identified by entry point numbers within interfaces. It appears that interface names are user created. In the current system this compromises type safety, in that a client may be compiled in terms of a server interface which is then modified while retaining the same name. Since no type-checking is performed at run-time, incorrectly typed calls may then occur. Normally, the fact that different versions of an interface definition had been used would be detected from interface timestamps that are checked at link or load time, but this does not seem to happen with remote binding. A solution to this problem is for the Lupine stubs generator to generate a unique interface name for each pair of stubs it produced and for it to bind this name into the code of both the client and the server stub.

Chapter 6. Restrictions on transmittable objects.

If an RPC mechanism is to be a useful programming tool, it should permit the widest possible range of language-level objects to be transferred between machines. However there are some objects whose transfer is either not meaningful or is unacceptably complex. These cases in 68000 CLU are discussed here.

6.1 Procedure variables.

Procedure variables of CLU's **proctype** type can be represented on a single machine by their code addresses. Clearly these cannot simply be transferred to other machines. It would be possible to modify **proctype**'s representation to include a network address so that when a proctype variable was invoked the call was transmitted to the correct machine. Such an approach goes against the philosophy of the current mechanism which tries not to obscure the differences between local and remote calls.

Remote procedure variables of the **remoteproctype** type can be passed between machines. It may sometimes be appropriate to write a **remoteproc** stub that called a particular local procedure and then pass that **remoteproc** to other machines. It would not be desirable to do this automatically, because of the different semantics of local and remote calls.

6.2 Non-CLU references.

Some CLU objects include references to things that are outside the normal CLU world. For example, the *buffer* abstract type exists to provide access to delimited areas of store outside the CLU heap. (This abstraction is used by clusters such as *stream* in interfacing to the non-CLU world.)

It is not clear what it means to transfer such objects to other machines. Simply ignoring references outside the CLU heap is clearly inadequate, but, for example in the case of *buffer*, merely copying the delimited area of store into the destination CLU heap

is also inappropriate. It was thus decided to prohibit the transfer of non-CLU objects in CLU RPC calls.

6.3 The any type.

Most type-checking in CLU occurs at compile time, but there is one exception to this rule: the *any* type. *Anys* act as unions of all types, to which an object of any type can be assigned. (CLU also supports two forms of limited tagged discriminated unions, *oneofs* and *variants*, so that *anys* are only rarely required.) To avoid creating a type-checking loophole, CLU requires a run-time test, the **force** operation, to be performed on an *any* before its value may be re-assigned to a normal type (see Figure 3 for an example). Thus a procedure may take an *any* argument and perform one of a set of operations on it depending on its actual type. An *any* can be thought of as a having two components: a CLU object of arbitrary type and a unique type-identification tag. It is CLU's definition of what constitutes a unique type that causes difficulties in transferring *anys* between machines.

```
x: any := "this isn't an integer"
n: int := force [int] (x) except
when wrong_type: Message ("x isn't an integer!")
end
```

Figure 3. Use of *anys*.

Different languages have subtly different definitions of type equivalence. In many languages (e.g Modula-2 and Ada) two variables are of the same type only if they are declared to be of the same named type. Thus in the following Modula-2 declarations, *fred* and *bert* have different types, but *jack* and *harry* are of the same type.

```
VAR fred: ARRAY [1..7] OF CHAR;
VAR bert: ARRAY [1..7] OF CHAR;
TYPE a = ARRAY [1..7] OF CHAR;
VAR jack: a;
VAR harry: a;
```

It is clear that the same primitive array operations can be applied to *fred*, *bert*, *jack* and *harry* and that they are all structurally equivalent.

In CLU, two types are equivalent if they have the same structure, i.e. if the same range of operations can be applied to both types with the same effects. Thus in the following CLU variable declarations, *leonid* and *mikhail* are of the same type, even though their elements are defined in different orders. Both *leonid* and *mikhail* are however different from *sergei*.

```
leonid: record [a,b: int]
mikhail: record [b,a: int]
sergei: record [x,y: int]
```

In CLU, unlike Modula-2 or Mesa, type-equivalence does not depend on the compilation environments where the types were defined. Thus two types A and B can be equivalent even though they were declared and compiled on separate machines without any shared environment. Abstract types will never be equivalent to other types, since their internal structure is hidden.

In current implementations of CLU, the linker is informed of the structure of all the types used in a program and it generates a unique identifier for each structurally distinct type. It is these identifiers that the CLU run-time system uses in implementing *any* and **force**. These identifiers are only meaningful inside a program that has been linked as a single unit. They cannot usefully be passed between the component programs of a multi-machine CLU application, since the separately linked components may have been allocated different identifiers at link time for structurally equivalent types.

I did not attempt to implement a solution to this problem. One solution would be for the linker to produce type description trees describing the actual structure of types used in *anys* or **forces**. The run-time test for type equivalence would then involve a detailed comparison of two type descriptors. These descriptors could then be transferred between machines. This solution involves a significant run-time cost, both in comparing the trees and in shipping them between machines. It appears likely that these type descriptors would be several times the size of their type's normal representation. This may be acceptable for

the specialized case of *anys*, it appears unacceptable for more general type identification of abstract types as discussed in chapter 8.

It is not clear that structural type equivalence offers any great advantages over Modula-2's form of type equivalence by name. If type equivalence by name were used, it would be possible to employ the CLU library mechanism to generate and retain a unique type identifier whenever a new type definition was encountered, in a similar way to that used for **remoteproc** UIDs in the CLU RPC system. Then types could be identified at run time by these library UIDs. Changing the nature of type equivalence in CLU was avoided because it would have unacceptable repercussions on many existing programs. However it appears that this use of type equivalence by name is the most efficient way for any new language to provide run-time type identification across several machines.

6.4 Enforcing transmission restrictions.

It might seem desirable for the compiler to reject any **remoteproc** definitions which involved types containing non-transmittable sub-types. However the compiler cannot assume knowledge of the internal representations of abstract types, so it cannot flag as erroneous the use of abstract types containing non-transmittable sub-types. Also the widespread use of *oneofs* in CLU means that a complex type might have nested sub-options that ultimately include a non-transmittable type, but these sub-types may never, or only exceptionally, be used in actual instances of the object.

For these reasons the CLU RPC mechanism checks for non-transmittable types at RPC call time rather than at compile time. An exception is generated if an attempt is made to transfer a non-CLU object, a procedure variable, or an *any*.

6.5 Tokens.

There are times when CLU implementors may wish to send a handle for an object to another machine rather than sending the object itself. This may be because the object is non-transmittable or contains sub-objects which are of only local significance (e.g. a monitor lock). Alternatively, it may be because the object is very large and is only intended

to be returned, unmodified, to its originating machine at some future time.

A new abstract type generator *token* has been implemented to satisfy this need. For each object of some type *t*, a corresponding object of type *token*[*t*] can be created. Later the original object can be recovered from its token. Tokens are implemented so as to be unique between machines and an attempt to decode a token on the wrong machine will cause an exception. A side-effect of creating a token for an object is that the object will be retained by the token cluster so that it will not be garbage collected even when all other pointers to it have been deleted. This means that the object will still exist if its token is later presented by some other machine, but means that tokens must be explicitly deleted if their associated objects are to be garbage collected.

Tokens are a way of representing objects resident on one machine at other machines. An example might be the case of an event notification server, which is requested to call a given **remoteproc** variable with a given argument when some event occurs. A useful form for the argument might be that of *token*[*any*], so that the client can map the argument onto some appropriate object of its own when the event occurs.

Chapter 7. Marshalling.

In implementing remote procedure calls it is necessary to transfer language-level data structures between the two machines involved in the call. This typically involves packing the data into a network buffer on one machine and then unpacking it at the other machine. Nelson [Nelson 81] refers to the two operations as "marshaling" and "unmarshaling". This chapter discusses the marshalling system used by the CLU RPC mechanism. Here, marshalling user defined abstract types is regarded as equivalent to marshalling their representations. The special requirements of user defined abstract types are discussed more fully in chapter 8.

7.1 Particular requirements of CLU.

A marshalling system must reflect the data structures of the target language, and CLU's goal of being an object-oriented language has affected the organization of its data structures. For instance, all objects in CLU are regarded as having an independent existence on the heap. Structured objects, such as arrays and records, consist of pointers to their component objects, rather than including them directly. Similarly the variables on the CLU stack are merely pointers into the heap. (The CLU run-time system does not actually store certain simple values, such as integers or booleans, in the heap, but substitutes the value for the pointer. This important optimization does not affect the language's semantics.)

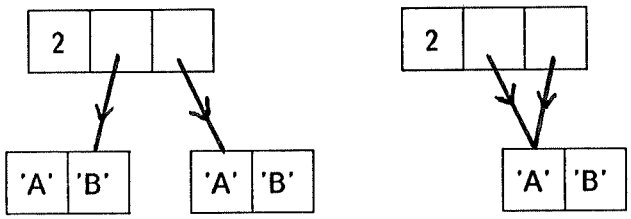

Figure 4. Two instances of type "sequence[record[x,y: char]]"

All objects in CLU are potentially sharable and the existence of such sharing will change the effect of certain operations. For example, Figure 4 shows two objects of the same type "sequence[record[x,y: char]]". These two objects will appear identical for read-only operations. However, because of the sharing present in the second object, updating the

component records will have different effects for the two objects and may cause subsequent read operations to return different results.

In section 4.2, CLU RPC was defined as implementing "call-by-copy" rather than CLU's normal "call-by-sharing". Thus it is not necessary for the marshalling and unmarshalling mechanisms to be concerned with sharing between marshalled and non-marshalled objects. It would also be possible for the marshalling mechanism to ignore sharing within transmitted objects, so that the right-hand object of Figure 4 was transformed into the format of the left-hand object as a result of its transmission. Such a transformation of transmitted objects would appear to introduce an unnecessary and avoidable inconsistency between local and remote calls. It is unclear whether it is necessary to preserve sub-object sharing between distinct argument or result objects, but there is no obvious reason for disrupting it.

CLU does not permit directly recursive type definitions, i.e it is not possible to define a type T to include an object containing a pointer of type T. However it is possible for an abstract type's representation to contain a pointer to the abstract type. This makes it possible for cyclic data structures to be constructed and accessed. However a programmer may be unaware that abstractions he is using contain such cyclic structures. Similarly the programmer may unwittingly participate in the construction of cyclic structures running through several abstraction boundaries. For these reasons any prohibition on the use of cyclic data structures in RPC arguments or results would conflict with CLU's model of abstraction.

Hence an RPC marshalling system for CLU should permit cyclic data structures and preserve sharing within transmitted objects.

7.2 Courier.

Xerox have proposed the Courier standard for remote procedure call [Xerox 81a]. Courier is an integrated part of their Network Systems standards and is intended to use their Sequenced Packet Protocol [Xerox 81b], a virtual-circuit protocol. The Courier specification includes definitions of standard formats for the transmission of data objects.

These formats appear to have been influenced by the needs of Xerox's Mesa language, but they are intended for general use and merit consideration.

Courier explicitly aims to minimize the amount of data transferred rather than the processing involved, on the grounds that it may be used over slow network connections.

Courier's formats for its simple data types (integers, long integers, cardinals, long cardinals, booleans and strings) are straightforward and cause no major problems with CLU, although 68000 CLU's use of 31 bit integers cause a minor embarrassment. The lack of an explicit format for characters, other than in strings, is a surprising omission.

The Courier formats for constructed data types (arrays, sequences, records and choices) are less satisfactory. For example, the closest format to CLU arrays is the sequence format. This contains size information but does not specify the current lower bound of the object, which makes it inadequate for describing CLU arrays which have mutable bounds.

More importantly, Courier requires constructed types to textually contain the representations of their sub-objects. For example the representation of an array of records will contain the representations of the component records (and of their components, etc.) within itself. This makes it impossible to transfer cyclic data structures or to describe sharing of sub-objects. This means Courier's data formats are unsuitable for CLU.

7.3 ISO presentation layer.

The International Standards Organization has issued a number of discussion documents on how data should be presented to application layer programs. One of the more relevant of these is the "Draft Transfer Syntax for OSI Datatypes" [ISO 83]. There are a number of public formats which shall be discussed here. There is also an allowance for private formats.

This proposal aims to include full type checking information with the transmitted data and each item of data has headers describing its type and size. For instance a CLU integer would be transmitted as three fields: the 16 bit ID code "2" (signifying an integer), a 16 bit length field holding the value "4", and only then the 4 data bytes holding the

integer's value. This assumption that detailed type information is necessary at run-time is in contrast to Courier, which expects types to have been agreed at a higher level, e.g. through the use of unique procedure identifiers with known argument and result types. The CLU RPC system aimed to follow Courier in using a language-level library mechanism together with unique identifiers to guarantee that only objects of the correct types are given to a procedure, so extra typing information in the data will normally be redundant.

The ISO proposal's formats for simple types are suitable for CLU, but like Courier its constructed types are inadequate to represent CLU arrays. Also the public transmitted types do not permit a distinction between mutable and immutable versions of structurally equivalent objects. This would make the standard type-checking data insufficient for a CLU run-time type-checking mechanism to distinguish between, say, records and structs.

This proposal also seems to share Courier's requirement that all constructed types directly include their subtypes, which is an unacceptable restriction for CLU datatypes.

7.4 A transmission format for CLU objects.

Since these proposed standards are inadequate for CLU, it was necessary to devise a transmission format specially for CLU objects. Since the RPC mechanism was intended to be for a single language and in view of the difficulties the multi-language proposals had encountered in being sufficiently general, it was decided to make the format CLU specific.

An arbitrary limit of 32 Kbytes was imposed on the size of a transfer. This particular limit allowed the marshalling mechanism to use 16 bit operations for greater efficiency at various points.

To avoid unnecessary translation it was decided to transfer all things other than pointers in their standard 68000 CLU heap formats. These formats are designed to ease the garbage collector's task. Their details are fairly straightforward and they are not worth discussing here, except to mention that since complex objects are represented by pointers, all record fields and array element fields are 32 bits in size.

The ability to describe pointers to other objects in the buffer is the most important feature of this format, as this makes possible the description of shared sub-objects and

cyclic structures. A pointer to an object in the buffer is represented by a 15 bit value giving the object's halfword offset within the buffer. These 15 bit values are stored in 32 bit fields whose other bits are such that the 32 bit field does not represent a valid simple object in 68000 CLU. These bits have been chosen so that the recipient RPC mechanism can easily detect and relocate the pointers.

The first item in the buffer is a record whose fields represent the call's arguments, some of which may be pointers to other objects in the buffer.

7.5 Partial transfers.

When a remote call has only very simple arguments it is clearly advantageous to transfer these immediately to the destination machine for the use of the called procedure. However, when a remote call takes a very large or complex argument object it might be useful to transmit only part of that argument object immediately and to transmit other parts only as they are needed. The PS-Algol persistent storage system [Cockshott 84] uses invalid machine addresses to identify data structures held on backing store. A hardware trap is caused by any attempt to access such an address, which causes an exception handler to retrieve the data structure and to resume the faulting code with the structure's new address. This system could clearly be adapted to cope with references to remote data structures.

The principal advantage of such a system is that it avoids the transfer of large sub-objects which are never accessed by the called procedure. Thus calls which are unable to perform their main function for some reason, such as a required resource being unavailable, can be executed quickly by avoiding the transfer of most of their argument objects. More importantly, some calls may simply never require large parts of their argument objects, even when executing normally.

CLU's abstraction mechanisms lead to significant difficulties in implementing an automatic partial transmission system. The called remote procedure may call various abstract type operations on the transferred objects. These operations are entitled to retain parts of their arguments objects within own variables or to pass them to other processes via shared data structures. Thus when a remote call terminates, various components of its

argument objects may still persist in assorted nooks and crannies of the server machine, to be accessed at some future time. It does not appear feasible to keep track of the movement of pointers to partially transmitted objects without performing a check on all assignment operations which might conceivably be carried out during the execution of the remote call. This would be unacceptably expensive. Clearly while the call is in progress the calling machine will keep the argument objects available to satisfy read requests, but at some point it must free these objects for garbage collection. Each server could keep a list of partially transmitted objects it has received. Then after its normal garbage collections it could arrange to read the remaining parts of any such objects still in its heap and to free any other partially transmitted objects being held by its clients. Clients could be permitted to request server garbage collections so as to free their pending objects, although this starts to introduce a worrisome degree of interdependency. (Garbage collections are relatively expensive operations which ought not to be performed more frequently than necessary. Garbage collecting after every call is certainly unacceptable.)

Furthermore if a partial transmission system is not to modify RPC call semantics it must arrange that, after an RPC call has begun, any changes made to the argument objects on the calling machine are not visible at the called machine. This means that a copy of the arguments must be made to satisfy any read requests from the server machine.

All in all, an automatic partial transfer system seems to introduce more difficulties than it solves, particularly as it only confers advantages when implementors try to transfer redundant information.

Theoretically, in an abstract, modular language such as CLU, the internal workings and structure of abstract objects are invisible to implementors, so they have little control over the sizes of the objects they are dealing with and may inadvertently include large amounts of little-used data in transmitted objects. However, it could be argued that this problem is rather less important in practice than in theory. In order to write even moderately efficient code, implementors must be able to make rough estimates of the relative costs of various operations, e.g. updating an array element can be expected to be much cheaper than copying a large array. These estimates must be based on semantic knowledge of an operation's effects beyond that provided by its formal interface specification. It would be possible to regard the need to estimate the size of an object as a similar process – an

object that has an unexpectedly large size will degrade performance in the same way as an operation that takes unexpectedly long to execute. Unfortunately this is not an entirely valid approach since some objects that are apparently independent may, nevertheless, wish to retain some form of pointers to each other, for internal implementation reasons.

Deciding which data structures to transfer between machines can be viewed as an integral part of the design of a distributed application. The data structures that are to be used should be designed with remote transfers in mind, so that they only contain direct pointers to sub-objects which are logically integral parts of the main object. Other related objects can be described by the machine independent *tokens*, described in the previous chapter, so that the indicated objects will only be transferred between machines by special programmer action. This system requires considerably more thought by implementors of large data structures than an automatic partial transfer system. Unfortunately in distributed processing ignorance rarely brings bliss – an implementor who arbitrarily transfers complex data structures between machines will encounter abysmal performance despite the RPC mechanism's best efforts, in the same way as someone who uses inappropriate algorithms in a large numerical application.

7.6 The CLU marshalling mechanism.

Nelson argued that the compiler should always generate in-line marshalling code for every remote call. This should permit more efficient marshalling than interpretive schemes, but can lead to unacceptably large amounts of code. For example CLU programs tend to make extensive use of *oneofs* (which are a form of tagged, discriminated unions). Code would have to be dumped to handle all the cases and their potentially large sub-structures. The possible sub-type tree for a CLU type may be quite large compared to the size of typical objects. The need to cater for cyclic structures and shared sub-structures would also make CLU marshalling code significantly more complex than that for Mesa.

For these reasons, it was decided to use an interpretive scheme for marshalling CLU objects. For a remote call the compiler dumps code to stack various pieces of control information together with an argument count for the call and that number of argument pointers. There is then a call into the RPC run-time system which allocates a buffer, calls

the interpretive marshaller on the arguments and then implements the RPC packet transfer protocols described in chapter 9.

To allow garbage collection, CLU objects' formats must include a large amount of structural information so that the size of each object and the whereabouts of its sub-objects can be deduced. This information proved adequate for a simple interpretive marshalling system. Since it is not possible for CLU programs to access this structural information, this marshalling code was written in the same CLU-compatible assembler code used for the garbage collector.

The low level marshalling code also maintains a hash-chained table describing objects that have been marshalled. If a pointer about to be marshalled is found to be already in this table then its existing buffer offset is used. This handles both simple object sharing and cyclic structures.

The corresponding unmarshalling code merely has to copy the data objects into its heap and relocate any buffer relative pointers it finds.

7.7 A template marshalling system.

The simple interpretive scheme described above is not entirely satisfactory. The need to determine an object's characteristics from its garbage collection tags and to distinguish between pointers and simple types, such as integers and booleans, does impose a performance overhead. However, this appears to represent quite a small proportion of the total marshalling time. The more important difficulty is that, since only structural information is available to the system, it lacks the ability to distinguish user-defined abstract types and thus is unable to call user defined action routines to transform transmitted and received objects, as proposed in section 8.1.

An alternative system might involve descriptive templates for each transmitted object. These templates would take the form of a directed graph, with each object's template including pointers to templates for its sub-objects. Each component of a structured type would be represented by two elements: an action routine and a pointer to a template for the sub-object. For CLU's built-in types these action routines would be fairly straightforward

recursive functions. For instance, the action routine for *array* would check if the array had already been marshalled and, if not, initialize an area for it in the buffer, call the sub-object action routine with the sub-object template on each array element and then store the resulting simple objects or buffer offsets in the buffer. This will probably be marginally faster than the current system.

For user defined abstract types it would then be possible for the compiler to generate action routines for each cluster. These might just consist of calls on the action routine of the cluster's current representation. They could, however, include calls on user defined routines to reformat the transmitted object.

A similar template driven unmarshalling system would be necessary to permit user defined action routines to reformat incoming abstract types.

Chapter 8. Abstract data types.

The existence of abstract data types in CLU presents particular problems for an RPC mechanism, but also offers potential advantages for the implementors of distributed applications. These issues have not been fully addressed in the current CLU RPC implementation, but are discussed here.

An abstract type in CLU is seen in terms of a set of operations that may be performed on abstract objects of that type. Abstract types are defined by **clusters** which specify a "representation", defining the actual data structure of each object of the type, and an "implementation", which consists of the code that carries out the type's operations. The representation and implementation of an abstract type are invisible outside its defining cluster, so that the type of an abstract data object does not change when its representation changes.

An abstract object can only be successfully manipulated by an implementation if the object's representation conforms to that expected by the implementation. Within a single machine program this presents no difficulties, since the current linker will only permit a single implementation (and thus a single representation) for each named abstract type. The equivalent restriction for distributed processing would be that each abstract type has a single implementation residing on one machine, with all operations on the type's objects being performed via RPC. A special case might be made of the built-in types, so that all primitive operations such as integer addition did not have to be turned into remote calls. Even so, building such a requirement for centralization into a language intended for building distributed applications is undesirable, particularly as many such applications may consist of replicated components.

The current CLU RPC system permits each abstract type to have implementations on several machines in a multi-machine configuration. (The linker will arrange that any component program that invokes a type's operations will have an implementation linked into its machine). This is done by requiring that all extant implementations of an abstract type share the same representation. Unfortunately, this cannot be enforced within the current CLU type-checking system. Consider the case of a **remoteproc** *fred* taking an argument of the abstract type *boris*. When *boris's* representation changes its type will of

course remain unchanged. This means that the type of *fred* is also unchanged. Normally this is exactly the effect desired, since there is no reason why calls of *fred* should be affected by an internal change in *boris*. Unfortunately, a linked CLU program containing an old implementation of *boris* can continue to invoke instances of *fred* that have been linked with the new version of *boris*. The RPC run-time test for **remoteproc** type compatibility will not object, since the type of *fred* has not changed. This constitutes a serious type-safety loophole, with no simple solution. A **remoteproc's** type could be defined to change whenever the representation of any of its abstract arguments changed. What this would mean is that an abstract type's type changed whenever its representation changed. This is a contradiction in terms and would mean that all code that used such an abstraction would have to be recompiled whenever its representation changed, even if all implementations were updated simultaneously.

Even if it were possible to enforce the requirement that a distributed configuration should only include a single representation for each abstract type, this would not be a completely satisfactory solution. In a configuration consisting of a large number of cooperating servers (e.g. a national air traffic control system) it may be either very difficult or simply very inconvenient to have to update the code in all machines simultaneously. If would be very desirable to allow different representations of abstract types to co-exist on different machines. Satisfactory solutions to this problem appear to require significant extensions to the way abstract types are specified in CLU.

8.1 Herlihy's encode and decode operations.

Herlihy and Liskov [Herlihy 82] have proposed a system for the transmission of CLU abstract objects between machines on which the objects' abstract types have different representations. Their scheme requires each abstract type to define a single "external representation" type. This is the representation that will be used for transmitting the type's objects over the network. Despite its name, this "external" representation is only used within the type's implementations and does not form part of the cluster's type. Each implementation of the type is required to provide an *encode* operation to convert objects from its internal representation to this external representation. Similarly, it must provide a *decode* operation to convert from the external to the internal representation. When

abstract objects are transmitted across the network the appropriate *encode* operation must be called at the transmitter to obtain the external representation. When an abstract object is received from the network, the appropriate *decode* operation must be called to convert it into the local internal representation.

This system is a step in the right direction. It allows several different implementations and representations of an abstract type to coexist simultaneously. The use of implementor specified *encode* and *decode* operations allows the implementor to perform useful transformations on transmitted objects even when all internal representations are the same. For example, an abstract type might include a monitor lock, which cannot meaningfully be transmitted between machines. Since remote communication is based around call-by-copy, the implementor might chose not to transmit any representation for the monitor lock, but simply to create a new monitor lock at the destination machine as part of the *decode* operation.

Unfortunately, this solution only succeeds in pushing the type-checking problem back one level. Different implementations of an abstract type need no longer have the same representation to communicate, but they must have the same external representation. Herlihy does not discuss how conformity of external representations can be type-checked, either at compile time or at run-time. The problem is essentially the same as that of checking for consistency between internal representations. It is undesirable for all modules using an abstraction to have to be recompiled whenever the abstraction's external representation changes, but the validity of these modules' RPC calls may depend on the abstract type's actual external representation.

Herlihy envisages different implementations of an abstract type choosing to have permanently different internal representations, e.g. the DEC10 representation of an integer might be 36 bits long and the VAX representation 32 bits long. His system is aimed at permitting the coexistence of essentially equivalent representations, rather than allowing the progressive introduction of more sophisticated representations.

In my observations of the use of CLU in Cambridge it has been relatively common for a user defined abstract type's representation to be extended so that the abstract type can support additional operations. One of the principal reasons for using abstract types is to

facilitate such changes in an object's representation. (The addition of new operations to a cluster merely extends its type, without invalidating any earlier uses, so there should be no need for other modules to be recompiled.) In these cases, the use of a separate external representation for inter-machine communication does not reduce the problem of introducing a new internal representation, since the external representation will probably also have to be changed to reflect any extension to the abstract type's semantics. To allow a new representation for an abstract type to be introduced gradually, a cluster's *decode* operation could be made to cope with several different transmission formats.

8.2 Tagged representations.

If each distinct transmission representation of an abstract type were given some unique tag, then this tag could be transmitted with each transferred object and used in a run-time compatibility test by the *decode* operation on the recipient machine. It would then be possible for each *decode* operation to accept some particular subset of tagged representations from the undelimited set used for its abstract type and to reject any unexpected or unconvertible representations. It is now necessary to consider the generation and representation of the tags themselves.

The compiler might simply assign a unique identifier for each new or changed representation, much as is done with **remoteprocs**. These identifiers would be suitable for internal use in checking that only a single representation were used, but would not help the implementor to cope with multiple representations.

Standard CLU provides two *selector* type generators, *variant* and *oneof*, that support tagged unions of types. Superficially, either of these might appear suitable for describing a set of possible representations. Unfortunately, they both operate in terms of delimited sets whose members are all known at compile time. Also the type of a *oneof* or *variant* reflects the types of its components. Two abstract types could not exchange *oneofs* with incompatible sets of subtypes in a type-safe way.

CLU also supports a union of all possible types, the *any* type. Unfortunately, as has already been discussed in section 6.3, there are particular problems in transferring *anys* between machines, because of the nature of type equivalence in CLU. The only proposed

solution involved transferring a type description tree with each *any* object. This would represent an unacceptable overhead on the transfer of abstract objects.

Since tagged representations cannot be implemented efficiently within standard CLU, explicit language support may be useful.

8.3 Flotsam.

In this section a new primitive CLU type *flotsam* is proposed for use in transferring abstract types between machines. *Flotsam* will require a combination of language extensions, modifications to the CLU library system and some run-time support.

Flotsam is intended to permit abstract type implementations to cope with a range of different representations from a set of unknown size. It thus takes the form of a tagged discriminated union whose overall scope need not be known by every implementation. To permit each abstract type's transmission type to be distinct, *flotsam* will take the form of a parameterized type, so that for each abstract type *t* there exists a type *flotsam*[*t*].

As part of each cluster definition it will be possible to declare one or more **flotsam_tags**. A **flotsam_tag** will consist of an identifier and a type specification (see Figure 5).

In the same way that the CLU selector operation "oneof$make_jack(x)" can be used to create a *oneof* with the tag *jack* and the value *x*, it will be possible to provide a selector operation "flotsam[t]$tag_bert(y)" to create a *flotsam* object for the abstract type *t* with the tag *bert* and the value *y*.

Similarly, it will be possible to provide an overloading for *flotsam* of the **tagcase** syntax used for *oneofs* and *variants*. This will allow a *flotsam* object to be compared with a variety of **flotsam_tags** and its element retrieved if its tag matches.

This raises the problem of how *flotsam* objects are matched against **flotsam_tags**. A comparison based solely on identifier names is clearly not type-safe, but for the reasons discussed in section 6.3 we do not wish to pass type descriptors between machines at run-time. The solution is for the compiler to generate a unique identifier whenever it

```
fred = cluster is encode,decode
% This cluster doesn't actually support any "user" operations, so as to simplify
% the example. The cluster's rep used to be a sequence of integers, but was
% upgraded to be an array of integers instead. There are two transmission
% formats reflecting the different representations. We are just beginning to
% upgrade to the new format, so we will accept either format, but only transmit
% the old format.

rep = array [int]      % The current internal representation.

flotsam_tag aformat: array [int]          % The current transmission format.
flotsam_tag sformat: sequence [int]       % The new transmission format.

encode = proc (w: fred) returns (flotsam[fred])
% This is called whenever a "fred" object is about to be transmitted.
% We have to de-abstract w and convert it from an array to a sequence.
    r: rep := down (w)
    s: sequence[int] := sequence[int]$a2s (r)
    return (flotsam[fred]$tag_sformat(s))
end encode

decode = proc (fw: flotsam[fred]) returns (fred)
% This is called whenever a "flotsam[fred]" object is received off the network.
% We try to convert "fw" into our current internal representation.
    tagcase fw
    tag aformat (a: array[int]):
        w: fred := up (a)
        return (w)
    tag sformat (s: sequence[int]):
        w: fred := up (sequence[int]$s2a(s))
        return (w)
    others:
        signal failure ("Unexpected format in fred$decode")
    end
end decode

end fred
```

Figure 5. An example of the use of *flotsam*

sees a new or changed **flotsam_tag**. These unique identifiers can then be stored in the compiler's library for use whenever the **flotsam_tag** is referenced again. This system is similar to that described in section 5.2 for identifying **remoteprocs** and shares a similar

restriction, that two tags will only be equivalent if they are compiled using related library environments.

Given these tagging and untagging operations it will then be possible for a cluster's implementor to provide *encode* and *decode* operations that map between the cluster's current representation and a tagged *flotsam* object. A very simple example is given in Figure 5, showing a cluster whose internal representation has changed, but which is still supporting an earlier external representation.

8.4 Discussion.

Herlihy requires a cluster to define a single external representation. **Flotsam_tags** can be viewed as a way of defining multiple external representations which can coexist. This will permit new representations of important abstract types to be introduced incrementally into a distributed system. This updating will probably occur in several distinct phases. Initially all nodes will use the old representation with the old **flotsam_tag**. Then over a period of time new implementations will be introduced that can receive and interpret a new representation but will only transmit the old version. When this phase is believed to be complete, final versions of the new implementation can be installed that both receive and transmit the new representation. The ability of all nodes to accept a new representation may be an important aid to the development and testing of a new implementation.

Flotsam is significantly different from other CLU types. The meaning of each call on *flotsam* is dependent on the library environment in which that call is compiled. *Flotsam's* external appearance is a combination of a parameterized type, for normal type-checking, and a selector type, to permit the selector style *tag* operations and the **tagcase** overloading. This combination of properties is a necessary reflection of the fact that *flotsam* does not operate on the clearly delimited sets used by normal selector types.

Chapter 9. Protocols.

The call semantics of the CLU RPC system proposed in chapter 4 are not tied to any particular protocol implementation. These semantics can be supported in a variety of ways on different local area network architectures. This chapter describes the protocols that were used to support CLU RPC on a Cambridge ring system.

9.1 The Cambridge ring.

The Cambridge ring is a slotted ring local area network [Wilkes 79] [Needham 79]. Hosts are attached to the ring via "stations". Stations are identified by 8 bit addresses and up to 255 stations may be attached to a ring.

Each slot in the ring consists of a minipacket which includes a 16 bit data field, an 8 bit source address, an 8 bit destination address and several response bits. A station can choose to receive from a particular station, from all stations or from no station. When a station wishes to transmit a minipacket it waits for a free slot in the ring and then overwrites that slot with its minipacket. When the destination station sees the minipacket it can read the data bits and source address and may modify the response bits. When the minipacket returns to the source station it will be marked as free and the response bits will be passed back to the host. The response bits may indicate one of four things. The "accepted" response means that the destination station has read the data bits and will pass them to its host. The "unselected" response means that the destination station was receiving from some particular station other than the transmitter. The "busy" response indicates that the destination host had not yet read a previous minipacket from its station, so the station was temporarily unable to receive. The "ignored" response occurs when the response bits have not been modified by the destination station. This normally indicates that there is no active station at that address.

The ring has a bit speed of approximately ten megabits/sec. Since each station is only permitted to have a single minipacket in circulation at a time, the point-to-point bandwidth depends on the number of slots in the ring. For example, in a 3 slot ring the maximum point-to-point data bandwidth is around eight hundred kilobits/sec.

9.2 The Basic Block Protocol.

The 16 bits of a minipacket are clearly inadequate for most transfers. The Basic Block packet protocol is normally used for ring transfers. Each Basic Block packet may contain from 2 to 2048 data bytes. These are preceded by a header, size field and port number and are followed by a checksum. The port number is a 12 bit field designating a logical route into the destination host.

Between packet receptions a host will request its station to receive minipackets from all stations. When it receives a packet header minipacket it will instruct the station to receive from only that station until the packet is finished or a timeout expires. Normally the host will use the port number to select an appropriate reception buffer and copy the data bytes directly from the station to that buffer.

Some hosts have adopted a policy for refusing to accept unwanted packets, i.e. packets which are directed at ports for which the host has no outstanding reception requests. After receiving an unwelcome port number these hosts will instruct their stations to reject all minipackets for a period of about 1 millisecond. The transmitting host is expected to understand the meaning of this unselected response in mid-packet and to abort the transfer.

Hosts will generally try vigorously to get their transmitted packets through to their destinations. For example, the Tripos operating system on the processor bank 68000s will ask its ring interface processor to make ten major attempts to transmit each packet. In the course of each such attempt, the interface processor will retransmit the header minipacket up to 100 times in the face of unselected and busy responses. When a host persistently attempts to transmit to a busy or unselected station, the station will insert an increasing delay in its transmission attempts until it is only attempting to transmit once per 16 ring rotations. This means that each major transmission attempt can take up to 30 milliseconds. Thus an attempted transmission will only fail due to congestion after the header minipacket has been transmitted unsuccessfully 1000 times over a period of around 300 milliseconds.

9.3 Virtual circuits versus specialized protocols.

It would be possible to implement CLU RPC on top of a standard virtual circuit protocol. However, such protocols are normally designed to handle streams of data, which will tend to make them inefficient at handling the call/response pairs typical of RPC.

Typical virtual circuit protocols (e.g. X25 [CCITT 81], BSP [Johnson 80]) offer flow control, acknowledgements and error recovery. In an RPC system, stream oriented flow control is largely unnecessary, since each client process can only issue a new RPC after the completion (or timeout) of its previous call. Virtual circuits only offer point-to-point flow control and do not solve the problem of global congestion control. If 20 clients call into an important server simultaneously, it is likely to become congested (in the sense of providing very poor response) with or without point-to-point flow control.

Acknowledgements and error recovery both introduce significant performance overheads. The Maybe call semantics do not require these features and the presence of a lower level recovery mechanism would destroy the very flexibility which the Maybe semantics were designed to exploit. The Exactly-once semantics would more obviously benefit from low level recovery action, but even here a specialized protocol which understands exactly when and where acknowledgements and retries are necessary, is likely to offer higher performance.

9.4 RPC protocols.

Distinct protocols were developed for the Maybe and the Exactly-once semantics, but they have the following points in common.

Some consideration was given to developing a minipacket level protocol for RPC, but no scheme was found that had decisive advantages over Basic Block. The use of Basic Block also held out the possibility that the RPC protocols could eventually be used through interring bridges (see section 9.8). The same transmission strategy as Tripos was used, so that a transmission would only be abandoned if the destination were persistently congested.

Each call is allocated a 32 bit unique call identifier by its transmitter. These identifiers are allocated in a monotonically increasing sequence, which is initialized to include a date

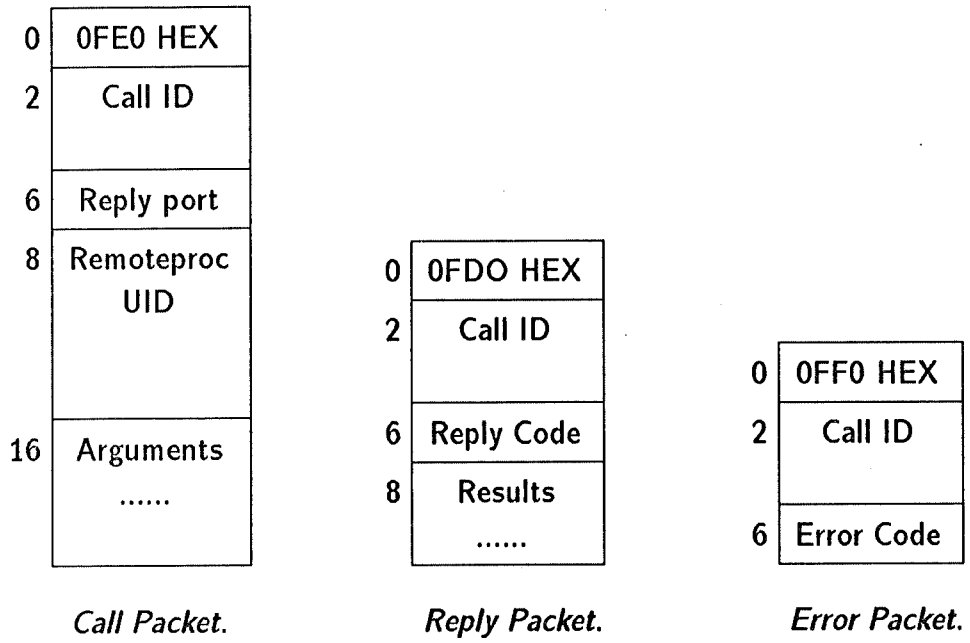

Figure 6. The Maybe protocol.

and time value in its top 24 bits. This ensures that call identifiers remain unique across caller crashes, so that rebooted clients will not confuse replies to pre-crash calls with replies for post-crash calls. (This is similar to the call identifier system used in the Newcastle RPC mechanism [Shrivastava 82])

Calls use the 62 bit **remoteproc** UIDs described in section 5.2 to describe the invoked **remoteproc**. Each processor might contain several different programs using CLU RPC. It is not possible to distinguish which program an incoming RPC is destined for merely by the **remoteproc** UID, since more than one program might support that **remoteproc**. Each program is regarded as having a distinct RPC mechanism which is allocated its own port number for incoming RPCs. Each packet in the RPC protocols includes a descriptive header so that a single port may be used for both Maybe and Exactly-once calls and also for responses.

9.5 The Maybe protocol.

The Maybe protocol is based on a simple exchange of packets between a client and a server. There are potentially three types of packets involved: Call, Reply and Error. Their formats are shown in Figure 6.

A client transmits a request packet and then awaits a response packet from the server. If the transmission is ignored by the destination station the call is abandoned and a *hard_error* exception is signalled to the calling process. If the transmission fails for any other reason, or a response packet does not arrive within the call's timeout period then the call will be abandoned with a *soft_error* exception.

When an RPC mechanism receives a request packet it will normally attempt to execute the call and send the results back in a reply packet, to the designated port at the transmitting station. A return code in the packet either indicates the call terminated normally or identifies a user level CLU exception that was signalled. The server mechanism may return an error packet if it is unable to carry out the RPC call. This will occur if the server program does not support the designated **remoteproc** or if it is short of buffering, due to congestion.

The client RPC mechanism will convert congestion error packets into *soft_error* exceptions and unsupported **remoteproc** error packets into *hard_error* exceptions. For normal responses it will return the results or signal an exception as appropriate.

9.6 The Exactly-once protocol.

The Exactly-once protocol is required to ensure that calls are executed once and only once in the absence of crashes or prolonged communication failure. It is also required to behave sensibly across server crashes, so that calls are either aborted or re-issued to a new incarnation of the server. Similarly, servers should not be discomfited by client crashes.

The Exactly-once protocol was based around four types of packets (Calls, Replies, Errors and Terminators), whose formats are shown in Figure 7.

In its simplest form an Exactly-once call will only involve three packets. The client will transmit a call packet, the server will respond with a reply packet and the client will acknowledge the reply with a terminator packet, permitting the server to discard the call's results. Just as the call packet specifies a port to which the reply should be sent, the reply packet specifies a port to which the terminator should be sent. This simple three part exchange is in fact what happens for calls of less than two seconds duration.

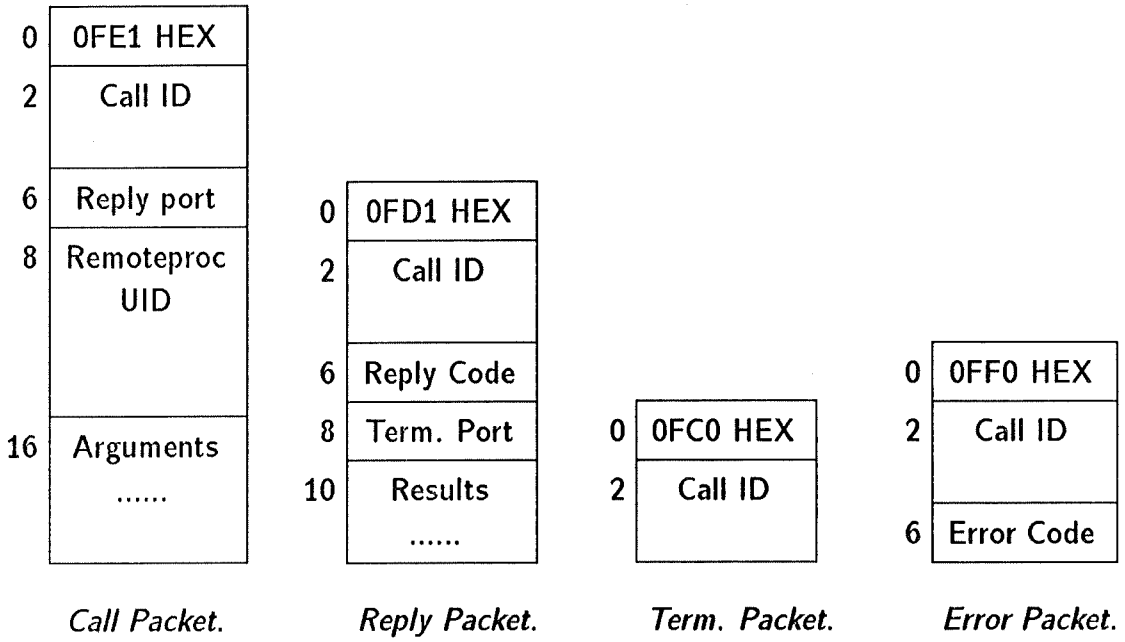

Figure 7. The Exactly-once protocol.

Until it receives a reply from the server or there is a fatal error, the client will every two seconds retransmit the call packet to the server. If the original call packet did not reach the server or the server crashed then this packet will be treated as a fresh call and execution commenced. If the server is in fact executing the request, it will discard the call packet and send an error packet to the client informing it of the situation. This error packet does not abort the call and in fact serves to reassure the client RPC mechanism that all is well.

When the server finishes executing a call it will transmit a reply packet to the client every two seconds until a terminator arrives for the call or there is a fatal error. This permits recovery from lost reply packets. When a node receives a reply packet which it is expecting it will cease transmitting call packets and send an appropriate terminator packet. However if a node receives an unexpected reply packet it will also send a normal terminator to the transmitter. This may occur if an earlier terminator had been lost or if a client had crashed after issuing a call. In neither case is there a need to inform the server of the error, but it is desirable to free it from its obligation to retransmit reply packets.

As with the Maybe protocol a server will send error packets in response to a call if it is congested or does not recognize the **remoteproc** UID. A "congestion" response causes

no immediate action at the client, but it notes the server is active and will retransmit the call when the normal two second timeout expires.

If call or reply transmissions fail because the destination is persistently busy or unselected, this is ignored. The normal two second retransmission timeout will recover from this. If a call fails due to being ignored at the receiver this is regarded as a fatal error and the call is aborted with a *hard_error* exception. If a reply fail due to being ignored then the client is assumed to be inaccessible and the reply information is discarded. All transmission errors are ignored on terminators.

If a client sends 50 call packets in a row without receiving any response from the server it will abort the call, with a *hard_error* exception. However, even error packets count as valid responses, so that calls of very long duration or calls to persistently congested servers will not be aborted. Similarly a server will cease retransmitting a reply packet after 50 repeats. It is believed that these retransmission counts are sufficiently large that they will only be reached if the destination node has crashed or otherwise become inaccessible.

9.7 Large transfers.

So far only RPC transfers that can be fitted into a single Basic Block packet (i.e. up to 2048 bytes), have been described. Many RPC transfers may involve larger amounts of data. To permit this, it was decided to extend the RPC protocols to permit up to 32 Kbytes to be transferred either as the arguments of the results of any RPC call.

Large transfers are handled as a series of maximal sized packets. If all the packets were to be sent to the target RPC mechanism's standard reception port, then they would be read into its standard buffers. This would cause several distinct problems. Firstly, the large transfer may swamp the available buffer space. Secondly, the packets would have to include headers identifying the call (and possibly including a sequence number) so that each packet can be correctly demultiplexed. Finally, the arriving packets are likely to be scattered throughout store, complicating the task of the unmarshalling mechanism.

The allocation of reception buffers for large requests is a problem area. If the RPC mechanism automatically allocates memory for incoming large transfers, a server may find

itself with little free memory should all its server processes receive large calls simultaneously. As well as requiring temporary buffer space, incoming RPCs will also require roughly equivalent heap space for their arguments. Currently the size of 68000 CLU heaps is fixed at link time. Thus, unless a server has some control over the size of incoming RPCs it risks running out of heap. The solution adopted was that the RPC mechanism should maintain a pool of packet sized buffers for normal RPC requests, but require that higher level software advise it as to how many larger buffers and of which sizes it should keep available. Thus as part of its initialization a server might call into the RPC mechanism to make, say, one 30 Kbyte and two 10 Kbyte buffers available for handling incoming transfers.

When one RPC mechanism wishes to transfer more than 2 Kbytes of data to another machine (either as arguments or results) it first calls ahead to reserve a buffer. This call itself takes the form of a Maybe RPC to the target RPC system. If this system has a suitable buffer available, it will reserve a port for the transfer and stand by to receive a series of packets on that port into the buffer. The reserved port number is passed back to the transmitting RPC mechanism, which sends off the packets. Thus the packets arrive in a contiguous area of store, without being intermingled with other transfers.

If the transmission does not arrive safely at the receiver within a few seconds, the transfer will be assumed to have failed and the port and buffer will be freed. If either the reservation RPC, or the multi-packet transmission itself cannot be transmitted for any reason, this is treated exactly as if a transmission failure had occurred in the top-level RPC. Similarly, if the reservation RPC is timed out, this is treated as a simple transmission failure. As before, the Exactly-once mechanism attempts to recover from such errors, but the Maybe protocol raises an appropriate exception. Thus the multi-packet protocol involves little extra work in the main RPC protocol handling routines.

If the target RPC mechanism is temporarily without a buffer large enough for the transfer it will signal a "congestion" error to its caller, which will be treated in the same way as for a congestion response to a single packet transfer. If the target RPC mechanism has never been allocated a large enough buffer by its higher level software, it will signal a different error, which will be turned into a special *hard_error* exception for both Maybe and Exactly-once calls.

9.8 Bridges.

So far the RPC protocols have only been implemented for a single ring system. In fact at Cambridge there are now three rings connected by two "bridges" [Leslie 83]. Each bridge receives packets from a station on one ring and transmits them via a station on another ring.

The minipacket response bits lose their significance when transmitting through a bridge. Since packets may be accepted by the bridge and then dropped due to a transmission failure in the target ring, it is no longer possible to ascertain from the ring hardware that a packet has been accepted by its ultimate destination. After accepting a packet the bridge will try for around 100 milliseconds to transmit it to its final destination. It does not report retransmission failures back to the original transmitter.

The bridge is only represented by a single station address on each ring. This means it is necessary for it to somehow map incoming packets from one ring onto station addresses in the destination ring. This has been done by creating a mapping in the bridge between reception ports and destination stations and ports. It is clearly not possible for the bridge to map its set of port numbers onto all the station and port numbers available in the destination ring, so hosts are required to set up "routes" through the bridge to named destinations. The nameservers provide a mechanism whereby a host can create a route to a specified port number at a named station. The bridge will delete routes which appear to have gone out of use. The bridge recognizes certain formats of protocol headers as designating request packets containing reply port numbers. In such cases it will construct return routes for the requests and then modify the reply port field in the request to use this return route. Thus when the destination host sends its reply to the bridge it will be retransmitted to the original reply port of the station that transmitted the request.

9.9 Names, bridges and the RPC system.

It is interesting to consider how the CLU RPC system could be modified to work through bridges, since while some properties of the ring bridge are implementation specific (e.g. the exact protocol mappings performed) others are typical of linked local area networks (e.g. the loss of response bits).

The lack of global addressing in a multi-ring network causes the main problems for the RPC system, since it must always ensure that routes are created whenever addresses are used. This implies that to work through bridges, RPC addresses must consist of more information than just a station number and port. Since RPC services may be created dynamically, is is not feasible to have an entry in the nameserver for each possible RPC mechanism. Thus RPC addresses would have to be represented by a combination of a node name and a port number.

The RPC mechanism could call the nameserver before each remote call to establish a route to the destination node name and port. In fact, it is possible to optimize this. One of the flags a nameserver returns on creating a route indicates whether the destination is actually on the same ring. Thus the first time the RPC mechanism uses an address it must check with the nameserver, but it can cache the resulting station and port. If the address is on the same ring, then the cached station and port can be used in all subsequent calls, without further communication with the nameserver. If the address is on a different ring and only a short interval has elapsed since the last call, then the cached station and port can also be reused. Otherwise, the nameserver must be called to establish a new route to the destination.

The Maybe protocol could be easily modified to use the same packet formats as those of the SSP protocol, which is understood by the bridge. Unfortunately, the Exactly-once protocol cannot be similarly mapped onto any protocol known by the bridge. However, the difficulties are principally over details of route reallocation, and it appears that the bridge could be easily modified to add the Exactly-once RPC protocol to its repertoire.

The response bits on a single ring act only as an optimization in the Exactly-once RPC protocol, since the retry mechanism will recover from packets being dropped by the bridge. (This should only occur rarely unless the destination is genuinely not interested.) Transmissions to stations persistently refusing packets will be aborted when the retry limit is reached. The Maybe semantics only commit the RPC mechanism to making a good effort at carrying out calls, so limited interference from the bridge is acceptable.

9.10 Reflections.

The original goal for the CLU RPC protocols was that they should exploit the low level architecture of the Cambridge ring so as to achieve high performance. In fact, in their final forms the CLU RPC protocols are relatively network independent, only requiring that their transmission layer can deliver packets with a high probability of success. The ring's response bits are essential to the success of the supporting Basic Block packet protocol, but are not essential to the RPC protocols themselves. (As described above both the Maybe and the Exactly-once protocols only use the ring's response bits on packets as an optimization and would perform successfully without them.) The reasons for not further utilizing the ring's response bits reflect the lightweight nature of the protocols and the need to cope with node failures.

The ring's response bits suffice to inform a host that a destination station has accepted a packet. Since the error rate on the ring is very low, accepted packets will almost never be subsequently dropped due to checksum errors. However, this implicit acknowledgement is of limited use to the RPC system. The Maybe protocol must enforce a time-out on its calls anyway, to fulfill its required semantics. The Exactly-once protocol must still retransmit its call packet periodically, in case the target server has crashed. When returning results, the Maybe protocol is uninterested in failures, provided they only happen infrequently. The low level acknowledgement of delivery could simplify the returning of Exactly-once results, but the server RPC mechanism would still have to retry periodically in the face of transmission failures.

Chapter 10. The supporting environment.

An RPC system does not exist in isolation. This chapter describes the support system that was implemented for the CLU RPC system. This included the addition of concurrency features to CLU and the provision of a specialized operating system kernel. The next chapter (Chapter 11) will discuss how the RPC system was integrated into this specialized kernel.

10.1 Adding concurrency to CLU.

Standard CLU programs are intended to be executed entirely on one machine by a single system process. As a result, the standard language does not include any features for the support of concurrent programming.

In a distributed system, concurrency is both more obvious and more important than on a single processor. The use of multiple processors means that genuinely parallel execution is occurring in place of the illusory concurrency that can be achieved on a single processor. To exploit this parallelism, it is useful to permit related tasks that might be executed sequentially if they occurred on a single processor, to be executed concurrently when they occur on distinct processors.

Since distributed processing involves concurrent execution between nodes it seems useful to permit concurrent execution within each node. It is implicit that such concurrent execution will require type-safe communication and synchronization between the processes involved and will benefit from linguistic support.

The choice between a procedural form of synchronization, such as monitors [Hoare 74], and a message passing system is related to the choice between remote message passing and remote procedure call discussed in section 3.2, where it was decided to support RPC principally because it reflected the built-in procedural orientation of standard languages. Following from this, it was decided to extend CLU to include monitors for concurrency control.

10.2 Monitors and CLU.

Monitors have been successfully included in many languages, notably Concurrent Pascal [Brinch Hansen 73] and Mesa [Lampson 80a]. CLU is an object oriented language where actions are seen as primarily occurring on independent objects. This implies that locking should not occur merely on a piece of code (as happens in Concurrent Pascal) but should be data oriented (as with Mesa's monitored records). However, even in CLU it is sometimes useful for a cluster to maintain global state in an own variable data structure. For example, a port manager cluster might maintain a list of allocated port numbers, to avoid accidentally reusing them. For this reason, it was regarded as desirable that a uniform monitor locking system system should be provided which would permit locking either on a group of own variables or on individual objects.

The solution adopted was to introduce a new optional clause in procedure headers specifying how locking occurs. This clause consists of the keyword **needs** accompanied by an expression that should evaluate to an object of a new built-in type *monitor_lock* (see Figure 9). Procedures using this clause are only permitted inside clusters whose header uses a new keyword **monitor** in place of the normal **cluster** keyword. The **needs** clause's expression will be evaluated on each entry to the procedure and the resulting *monitor_lock* will be used for enforcing exclusion. The **needs** expression is not re-evaluated when the *monitor_lock* is freed or reclaimed (so Mesa's potential embarrassment of freeing the wrong lock is avoided).

The **needs** clause permits considerable flexibility in implementing a monitor cluster since each procedure that locks can specify a different method for obtaining its *monitor_lock*. This means that a cluster might use a single lock when creating or invalidating objects, in order to permit an own variable data structure to be updated, but use a separate lock for each object when performing normal operations upon them. This increased flexibility inevitably increases the scope for creating deadlocks. For example a careless implementor might try to lock on two objects in turn when doing a comparison, without performing any global locking first. If undue complications start appearing when object based locking is used, there remains the option of obtaining the effect of Concurrent Pascal style locking by using a single, own variable, lock.

```
Buffer = Monitor [t: type] is Create, Add, Remove
% Simple minded parameterized cluster that provides buffers for objects
% of any type t.

rep = record [ lock: monitor_lock, pending: semaphore, items: array[t] ]

Create = proc () returns (cvt)
    % Create a new buffer object. No locking is necessary.
    b: rep := rep${  lock: monitor_lock$create(),
                     pending: semaphore$create(),
                     items: array[t]$new()}
    return (b)
end Create

Add = proc (b: cvt, item: array[t]) needs (b.lock)
    % Add an item to the given buffer.
    array[t]$addh (b.items, item)
    semaphore$notify (b.pending)
end Add

Remove = proc (b: cvt, time_out: int) returns (t)
                                        signals (timed_out) needs (b.lock)
    % Wait till there is at least one item in the given buffer and
    % then remove the oldest item.
    semaphore$wait (b.pending, time_out) resignal timed_out
    item: t := array[t]$reml (b.items)
    return (item)
end Remove

end Buffer
```

Figure 8.

10.3 Synchronization within a monitor.

Monitors need to provide some method of synchronization between their client processes, so that a process executing in the monitor can wait for a change in its state. Concurrent Pascal and Mesa use condition variables, which are regarded as having no state, but merely an attached queue of processes. A notify on a condition variable merely awakes the first queued process. Thus notifying on a condition variable on which no-one is waiting has no effect whatsoever. This is adequate, but slightly misrepresents the common situation where there is a one-to-one relationship between events and actions, e.g. where one process is adding buffers to a queue and another is removing them. For this

reason, Dijkstra's general semaphores [Dijkstra 65] were adopted for use with CLU, with the obvious modification that a wait on a semaphore caused the most recently acquired monitor lock to be freed. (A corollary of this was that semaphores should only be used within locked monitor procedures.) An example of their use for the buffering problem is shown in Figure 8. Here the state of the semaphore will either represent the number of buffers in the queue (if positive) or the number of processes awaiting buffers (if negative).

As with Mesa a timeout time can be specified for each wait and there will be a "timed_out" exception if the semaphore cannot be claimed within that time.

10.4 Creating new processes.

Mesa has a special keyword **fork** which allows a procedure call to be "forked" to run as a separate process. This special syntax permits type-checked arguments to be passed to the newly created procedure in a way that a simple library routine, e.g. Modula-2's *newprocess* cannot provide. A similar system was added to CLU, but with one extension. The use of the keyword **stacksize** followed by an integer expression, permits the implementor to advise the run-time system on the expected maximum stack requirements of the new process. (A default size of 2 kilobytes is used which has normally proved adequate.) This feature was unnecessary in Mesa which allocates stack frames dynamically, but necessary for CLU which uses a simple linear stack. Some simple examples of **forks** are shown in Figure 9.

```
fork foo (1,2,3)
p1: process := fork foo (x,y,z)
p2: process := fork foo (z,y,x) stacksize x+y+2000
```

Figure 9. Some example uses of **fork**

10.5 Concurrency and abstraction mechanisms.

One of the guiding principles of CLU is that a module is only viewed by other modules in terms of its external interface, i.e. the results and signals it returns, plus any changes

it makes to argument objects. It has not been necessary for modules to be aware of how target modules implement their interfaces. This changes significantly when modules may block as a result of waiting on semaphores during their normal execution.

The nested monitor problem is fairly well known [Lister 77]. It arises when a process P which holds a monitor lock A calls into a monitor B and requires to wait there. The wait will cause the current monitor lock B to be freed, but not the earlier lock A. If the process which is intended to notify P then waits on the monitor lock A, deadlock results.

There is no entirely satisfactory automatic solution to this problem. The strategy that was originally adapted for programming in monitors with CLU was to organize monitors hierarchically, so that monitor locks were always claimed in the same order, thus avoiding simple deadlocks, and to avoid holding monitor locks for other than very short periods. This last requirement implied that a procedure holding a monitor lock should avoid calling any other procedure that blocks. Herein lay the problem. Knowledge about a module's internal behaviour (its blocking characteristics) had become essential to writing correct code.

Consideration was given to extending a procedure's type to include information about its blocking characteristics. Thus all procedures that waited on semaphores would be required to have the keyword **blocking** in their headers. Similarly any procedure that called a **blocking** procedure would be required to be **blocking** itself. Conformity to declared **blocking** characteristics could be enforced by the compiler. Unfortunately a quick scan of the Mayflower kernel suggested that the vast majority of kernel procedures would be **blocking** by this definition. The problem is that a large number of modules which do not normally block include code that is only executed under exceptional conditions which does block. For instance, many modules invoke blocking error handlers if internal consistency checks fail, i.e. if a bug is detected.

What is desired is some easy way for an implementor to detect whether a procedure that he calls normally blocks, without requiring him to read the procedure's source together with the sources of all the procedure it calls. Requiring that all blocking procedures contain an appropriate comment is an inadequate solution. Comments are all too easily omitted or overlooked.

As described above, waits on CLU semaphores are required to provide a timeout value, which may be zero, meaning forever. This timeout is intended to reflect how long a delay higher level software is prepared to tolerate in acquiring the resource. For instance the Mayflower stream manager permits other kernel modules to specify timeouts when performing potentially blocking operations. This proved useful when implementing a remote pipe module in the kernel, since incoming remote reads could specify a timeout of a few seconds so as to avoid outliving the interest of their client machines. It also served the beneficial purpose of making the blocking characteristics of those procedures explicit. It was decided to make this a programming convention for use with concurrent CLU, so that all procedures that blocked should have an argument that controlled the blocking characteristics. Standard CLU stream input/output procedures were exempted from this convention, since their blocking characteristics were fairly self evident.

The provision of a blocking control argument (probably, but not necessarily a semaphore timeout value) to a procedure serves two purposes. Firstly, it acts as a explicit reminder to implementors of the fact that a procedure blocks. The exact nature and significance of blocking arguments may change at abstraction boundaries, but the fundamental blocking characteristics should still be obvious. Secondly, the argument also permits implementors some degree of control over the nature of any blocking. The potential deficiency of this scheme is that the procedure must be capable of honouring its blocking argument. If it only discovers the need to block after doing some complex processing, it may have to perform significant backtracking if the semaphore wait (and thus the entire operation) has to be aborted.

10.6 Overview of the Mayflower kernel.

The CLU RPC system could be implemented on any reasonable operating system. However, the RPC system's performance will be heavily dependent on the structure and performance of the supporting system. In order to investigate a high performance RPC system, it was decided to implement a specialized kernel which would support concurrent CLU efficiently and could be tailored to include special support for CLU RPC. This kernel, called the Mayflower kernel, was implemented on Motorola MC68000 processors without memory mapping.

The Mayflower kernel was designed to support CLU efficiently, but also to permit the use of other languages. Kernel functions are accessed procedurally (via trap instructions) rather than by message passing, reflecting the procedural bias of concurrent CLU and of the CLU RPC system.

The kernel supports groups of processes running in "domains", which are similar to Thoths's "teams" [Cheriton 79]. System resources, such as memory, input/output streams, etc., are allocated to domains rather than to individual processes, so that all the processes within a domain have equal access to its resources. The kernel will try to prevent a domain accessing another domain's resources, but this is done to facilitate program development rather than to provide absolute protection (which is anyway infeasible on a machine without hardware memory protection). Each concurrent CLU program, containing an arbitrary number of processes, executes as a single domain. The kernel itself constitutes a domain, which is entered whenever a supervisor call is executing.

The kernel supports two simple synchronization primitives, which can be used for communication between processes within a domain. Processes can run at one of eight priority levels and timeslicing occurs between processes at the same priority level.

10.7 Interrupt handling.

In their discussion of concurrency in Mesa [Lampson 80a], Lampson and Redell describe how the Pilot operating system [Redell 80] integrates physical devices into Mesa's concurrency system. Each device is controlled by a monitor. Hardware interrupts from devices are turned into notifys on condition variables. Thus a device interrupt will cause an appropriate control process to be scheduled to run via Mesa's normal mechanisms.

This system appeared an elegant way of controlling devices and an attempt was made to use a similar scheme, with CLU monitors and semaphores, for the Mayflower kernel. However it proved unsatisfactory in two distinct ways and had to be abandoned.

Pilot assumes that device interrupts will be dropped when they are transformed into notifys, so that normal execution can be resumed. This assumes a high degree of uniformity in the device controllers, so that they will all either drop their interrupts automatically

on being serviced, or provide a uniform method for clearing interrupts. Unfortunately, not all hardware conforms to this model. For example, the M6850 input/output controller will drop its "character available" interrupt only in response to very device specific action. This means that a device specific handler must be entered to clear the interrupt before any processing at normal interrupt level can resume. This is difficult to guarantee in a semaphore based system, since the device control process may be temporarily suspended e.g. due to contention over a monitor lock.

A different problem arose with the module controlling access to the "Mace" network interface processor attached to each 68000. This module was structured so that when a process issued a request to the Mace it would obtain a control object containing a semaphore that would be notified on when the request completed. Thus, when an interrupt arrived, the Mace semaphore was notified and the Mace control process activated, but this process would merely store a return code from the Mace in the control block and then locate and notify whichever process was really interested in the reply. Two process switches were thus involved in informing a process that its network request had completed, with the intermediate process only fulfilling a trivial demultiplexing function. Given the importance attached to building a high performance RPC mechanism, this was an undesirable overhead. It was also suspected that the Mace controller typified a fairly general class of intelligent device controllers where the time spent in the control process would be significantly less than the cost of a process switch. The performance penalty may have been more acceptable in Pilot, where process switches have special microcode support.

The solution adopted to these problems was to revert to a simpler form of interrupt handling, whereby kernel modules could register simple CLU procedures which were to be called directly off hardware interrupts. This approach required that interrupt inhibition be used for mutual exclusion within device control modules, since the use of monitors for exclusion would lead to deadlock when a process holding the monitor lock was interrupted by a procedure which itself required that lock.

This solution proved adequate for handling the M6850 and boosted performance on ring transactions, where the interrupt procedure simply performed the demultiplexing previously carried out by the Mace control process.

10.8 Garbage collection.

Both kernel and user level CLU programs required garbage collection of their distinct heaps. An existing compacting garbage collector for sequential CLU was adapted for use with concurrent CLU. This garbage collector was given special kernel support to enable it to locate the active areas of concurrent CLU processes' stacks.

Within user level concurrent CLU, the fact that the garbage collector compacted retained objects meant that all buffers used in communication with the kernel had to live outside the heap and thus be invisible to the garbage collector. Otherwise, while one process was in the middle of a system call using a buffer in the heap, another process might cause a garbage collection which would cause that buffer to be moved in the heap, unbeknown to the kernel. This constituted a considerable nuisance, since abstract types could only include input/output buffers in their representations at the cost of requiring that an explicit delete operation be performed on their objects, which conflicted with the normal style of programming in CLU. For these reasons it would appear desirable to use a non-compacting garbage collector for a concurrent language, despite the slightly slower heap allocation (due to the need to maintain one or more free chains) and some risk that heap fragmentation might unnecessarily prevent the creation of large objects.

10.9 Synchronization.

Concurrent CLU uses both monitor locks and semaphores for inter-process synchronization. The first version of the Mayflower kernel provided explicit support for both kinds of objects, including performing such tasks as freeing the most recently acquired monitor lock when a semaphore wait occurred. This meant there was a relatively expensive language independent kernel call on each semaphore or monitor action. More significantly, it led to a kernel object being associated with each user level semaphore or monitor lock. This raised the problem that the corresponding user level objects had to be explicitly deleted (rather than dropped and garbage collected) so that their kernel level resources could be freed. In the case of monitor locks and semaphores this proved even more of a nuisance than with buffers, since it is in the nature of monitor locks and semaphores that they will be shared between processes, which makes their deletion more complex.

To avoiding requiring state to be kept in the kernel for each semaphore and monitor lock, these objects' state variables were moved into user space where they could be incremented and decremented by indivisible instructions, and the kernel synchronization mechanism simplified to merely provide *wait* and *kick* primitives on user supplied 23 bit identifiers. Conceptually, when a process *waited*, the kernel would suspend it and record in its process descriptor that it was waiting on the specified identifier. When another process *kicked* on that identifier the waiting process would be located and resumed. (The kernel actually maintained slightly more complex data structures, for performance reasons, but there was still no separate state for each monitor lock or semaphore). This meant that monitor locks and semaphores could be created and garbage collected like standard concurrent CLU objects.

Additionally, since the state variables now resided at user level, it was possible for the concurrent CLU run-time system to detect when an indivisible state variable change implied that a genuine kernel level *kick* or *wait* was required. Thus actions involving no immediate synchronization, such as notifying on a semaphore on which no-one was waiting or claiming a monitor lock that was free, could be resolved entirely at user level without requiring a kernel call. This considerably improved performance on monitor calls, where genuine contention is rare.

10.10 Reflections.

After extensive use of the above concurrency system for CLU, some observations can be made.

10.10.1 Internal routines.

The failure to provide an explicitly flagged form of "internal routine" which could only be called after a monitor lock had been acquired was an error. In modules which included non-locking external procedures and which had large numbers of subroutines, the inability to easily distinguish subroutines which expected to already own a particular monitor lock was unnecessarily confusing.

10.10.2 Procedural exclusion.

Following the standard monitor paradigm, procedures were used as the unit of exclusion. As discussed above, it was also decided to adopt a policy of only holding monitor locks for minimal periods in order to avoid deadlock. This combination resulted in complex monitor calls being implemented as unlocked procedures which mixed calls outside the monitor that involved blocking with calls on local **needs** procedures to carry out actions that required mutual exclusion. Unfortunately, the resulting **needs** procedures were frequently little more than scattered code fragments which happened to require mutual exclusion. Procedural locking is useful when the procedures form logical units of the module, otherwise it tends to lead to awkward structures.

There is no obvious reason why the **needs** syntax for monitor exclusion could not be extended from procedures to arbitrary **begin...end** blocks. This would result in a form of exclusion more akin to Hoare's critical sections [Hoare 72] than to the original monitor system. This system would permit sections of code which required sole access to a data structure to be intermingled with code which required to call blocking procedures in other monitors, in a way that should improve program clarity.

Chapter 11. An integrated RPC mechanism.

Under the Mayflower kernel, simple ring transmissions and receptions go through three levels of software processing. The user level library modules merely transform requests from applications programs into kernel calls. The kernel level software mediates access to the Mace and directs replies from the Mace to appropriate requesting processes. The software in the Mace performs the actual work of transmitting and receiving ring packets and maps incoming packets onto outstanding requests from the kernel.

The CLU RPC mechanism could have been implemented entirely by library code within the user level, using the standard kernel ring primitives for all I/O, and a preliminary version of the RPC system was in fact implemented at this level. However, in order to improve the efficiency and resilience of the RPC system an investigation was made of how the processing could be better split between the three levels. This section discusses the approaches considered and describes the integrated mechanism that was eventually implemented.

11.1 The Mace ring interface.

The final implementation of the CLU RPC system was heavily influenced by the interface hardware and software used, so this will now be described.

The host systems used consisted of Motorola MC68000 processors with either 512 or 1024 kilobytes of memory. The MC68000 has a full range of 32 bit instructions, but uses a 16 bit external data bus. The 68000s that were used executed with an 8 MegaHertz clock, which permitted a 16 bit memory access every 500 nanoseconds, since four internal cycles are necessary for each memory access.

Instead of having direct access to the ring station (which would result in the 68000 processing all minipacket transfers itself), the 68000s access the ring via an intermediate network interface processor, the Mace. The Mace hardware consists of a Motorola MC6809 processor with 64 kilobytes of memory. This 6809 executes at 2 MegaHertz, permitting an 8 bit memory access every 500 nanoseconds. The 6809 is basically an 8 bit processor, although it does support a few 16 bit instructions. The Mace has three way DMA hardware

connecting the 68000's bus, the 6809's bus and the station interface. The DMA hardware is capable of block transfer into 68000 memory, but must be reset when a transfer crosses a 64 Kbyte boundary. The design of the DMA logic makes software control rather cumbersome.

Since the 6809 can only access an 8 bit value in memory in the time the 68000 can access a 16 bit value, it is clear the Mace will be significantly slower than the host 68000 at manipulating 16 or 32 bit values.

The Mace was intended to execute at least the Basic Block protocol and possibly higher level protocols. Two Mace programs were produced, Spectrum and Supermace [Garnet 83]. Spectrum merely performed Basic Block functions, but Supermace also implemented the Byte Stream Protocol. Spectrum was significantly faster than Supermace at Basic Block packet functions, so it was selected as the starting point for the CLU RPC Mace program. The code that accepted requests from the 68000 was extensively modified to achieve higher performance. For small transfers this control code originally took considerably more time to execute than the actual ring transmission or reception. Speeding up this code cut 40% off the critical path time for the reception and transmission of small packets.

11.2 Marshalling.

One option considered for the RPC system was to move most of the RPC processing into the Mace, with the kernel merely providing a route for a minimal piece of user level library code to pass RPC control information into the Mace. This offered the prospect of merging the marshalling process with the Mace's low level transmission mechanism, so that instead of marshalling CLU objects into a buffer for later transmission, the Mace would transmit objects across the ring as soon as they were marshalled. This initially appeared to hold out prospects for a significant performance gain, but on closer examination certain problems emerged.

When a CLU structure is being marshalled, forward references to other objects will inevitably be encountered. When marshalling into a buffer this is easily resolved by recursively marshalling the sub-objects into the buffer and then storing their offsets in the marshalled copy of the original object. When marshalling directly onto the ring, this would not be possible. It would however still be necessary to assign offsets for the sub-objects

immediately, so that these could be transmitted as part of the parent object. (Sending a list of corrections at the end of the transmission would be extremely expensive, given the number of inter-object references typical in CLU.) Assigning offsets would necessitate peeking into the sub-objects to ascertain the size of their immediate representations and maintaining a queue (not a stack) of objects to be marshalled at specific positions. This would represent a significant additional implementation cost. The gain from not marshalling into a buffer was essentially the saving of a write, a read and some simple boundary checking for each longword transferred. Since the Mace software was already incapable of transmitting at the maximum rate the ring hardware permitted, there appeared little scope for gaining performance by interleaving marshalling and transmission actions. Thus it appeared that marshalling directly onto the ring would be unlikely to lead to a performance gain.

Even although marshalling directly onto the ring appeared infeasible, the option of having the Mace perform the marshalling remained. This would offload some processing from the 68000 and would make the eventual ring transmission go faster, since it could take place directly from Mace memory. However, simple hardware performance considerations ruled this out. Marshalling involves extensive manipulation and comparison of 32 bit values together with scattered accesses within the 68000's memory. It thus appeared likely that the 68000 would be significantly faster at marshalling than the Mace. Even if the 68000 had a ring interface processor as fast as itself, unloading the marshalling into the ring interface processor would not speed up an individual RPC. Since the ring interface processor can be the bottleneck determining the system's overall performance, it appeared undesirable to move extra processing into it unless there were some clear performance gain.

It was thus decided to perform marshalling within the 68000. There appeared no advantage in carrying it out within the kernel, so it was implemented by user library code, which benefited from having direct access to user level heap control structures.

11.3 Unmarshalling.

Unmarshalling is a much simpler process than marshalling, merely involving uncomplicated parsing of the transferred objects, leading to identification and relocation of internal

pointers. This process could either be carried out in the Mace, or in the 68000 itself.

In the Mace the unmarshalling process might be combined with data reception so that data was transferred directly from the ring station into the destination CLU heap. In fact, in such a system the Mace would probably unmarshal incoming objects into an area above the normal CLU heap, for which the Mace itself controlled space allocation. Whenever the CLU system performed a garbage collection it could merge this area into the main heap, after alerting the Mace.

Unfortunately, similar performance considerations to marshalling applied. The 68000 is sufficiently faster than the Mace at transferring 16 bit values and manipulating 32 bit values that no performance advantage appeared likely from performing unmarshalling in the Mace.

If the host system had a network interface processor of comparable speed to itself, unmarshalling in the network interface processor might lead to a small performance gain. Such a system would only save three of four instructions for each 32 bit value transferred, but it would also simplify the RPC system's buffer management.

11.4 The role of the Mace.

Since the Mace's inferior performance relative to the 68000 made is unsuitable for complex data manipulation, such as marshalling or unmarshalling, the emphasis of the investigation changed from trying to put as much as possible of the RPC system into the 6809, to an attempt to discover if any part of the RPC processing could be better performed in the Mace rather than in the 68000.

One advantage of the Supermace program for Byte Stream Protocol handling was that it reduced the amount of interaction necessary between the Mace and the host 68000. For example, if client software tried to transmit 5 packets worth of data down a byte stream, this could be presented to Supermace as a single request and Supermace would send off the 5 packets itself and handle the protocol acknowledgements from the destination without disturbing the 68000. Similarly, the Mace can handle for itself the occasional protocol interactions that are necessary to keep the byte stream active. This leads to a cost saving

both in the Mace, which need process fewer host commands, and in the 68000 which need neither issue the extra commands nor process their response interrupts and associated process switches.

The RPC protocols involve very little protocol handling of the sort that could be done in the Mace. Multi-packet transfers of the form required for the RPC system were already accommodated by the standard Spectrum program. The periodic retries potentially required by the Exactly-once protocol were of such low frequency as to be of little importance.

The final implementation of the CLU RPC mechanism merely pushed congestion control into the Mace. Spectrum was modified to permit certain ports to be designated as RPC ports. If an RPC format packet arrived on such a port when there was no reception request outstanding the Mace would accept the packet and send back a "congestion" error packet. This meant that an overloaded host would not be troubled by incoming requests it did not wish to process, while still offering some chance that calling systems would be alerted to the nature of the problem rather than merely having their transmissions fail. However, it is likely that if the host is overloaded the Mace will be also, so callers may be unable to get their packets through to a persistently busy Mace.

11.5 Reflections on the Mace.

The main motivation behind providing the Mace was to offload network processing from its host, freeing it for other work [Garnet 83]. This presupposes that the host will commonly have other work to perform when awaiting the completion of a ring transfer. This will be true in many cases, e.g. when a system process is performing buffered i/o across the network on behalf of a client program. However, there are also many cases where it will not be true, e.g. when an object module is being loaded on an otherwise idle machine. Furthermore, because a CLU RPC call is an essentially synchronous action, it appears likely that (despite the provision of language-level concurrency) there will be many cases where an application will suspend, leaving the machine idle, awaiting the results of some remote call.

Thus, while there are some cases where the parallelism between the host and the Mace is useful, there are other cases where it is not. Hence it is desirable that the Mace

should not significantly impede those transfers which do not benefit from its parallelism. This is clearly not the case. The Mace is a significantly slower machine than its host and there is also a considerable cost overhead in transferring control information between the software of the two machines. (Garnet observed that it took over a millisecond to pass a null command back and forth between the Tripos operating system and the Supermace program.) It thus appears likely that the 68000 would be significantly faster than the Mace at performing ring transfers.

If a network interface is to boost system performance by providing parallel execution of network actions without also significantly slowing down critical path network transfers, it is necessary that the network interface should be at least as fast as its host and that there should be some convenient method for them to exchange complex control information (e.g. by tables in shared memory). It would then also be possible, as described above, for the RPC system to offload some extra processing, such as marshalling and unmarshalling, into the network interface. This would benefit those applications where there was concurrent execution occurring on a host, without significantly delaying individual RPCs.

11.6 Tailoring the kernel.

Since both kernel level and user level code execute on the same processor, there is no scope for increasing performance by merely moving processing from user level to kernel level. However, performance gains can be achieved by providing special functions inside the kernel so as to minimize the number of process switches and kernel calls (both of which are relatively expensive operations) involved in performing an RPC.

The management of reception buffers was a source of significant complications in the original RPC system. When an RPC call was issued it was necessary that a reception buffer should be available at the Mace to hold the reply. This request could not be issued after the call, since, due to timeslicing, there was a significant chance of the reply to a short call arriving before the reception request was issued. To avoid having to issue a reception request before each call, the RPC system used a single port for all receptions and endeavoured always to have several buffers waiting on that port. This meant that when a reply arrived it had to be analyzed by an intermediate process in the RPC mechanism so

that the call could be identified and the correct calling process notified. After issuing the call, the calling process would submit any free reception buffer to the Mace before waiting for its reply to arrive. (For extremely short calls this led to the interesting effect that the reply to the call would frequently have arrived by the time the calling process got round to waiting for it.) To increase the efficiency of this manoeuvre, buffer management was partly moved into the kernel and a new kernel call was introduced for performing a CLU RPC. The new kernel call combined the request transmission with the addition of a new buffer and also waited for the call's completion. For each RPC port, the kernel maintained a list of call control objects describing the active calls. When an interrupt from the Mace indicated that a request had arrived on an RPC port, the CLU interrupt handling procedure would analyze the incoming packet and if it were a reply, the calling process would be notified immediately. This eliminated a process switch from the critical path time for an RPC and also reduced the system overheads involved in the processing done while the call was in progress.

Some care had been taken in the original kernel design to ensure that ring actions happened efficiently and with a minimum of process switching and other system overheads. The net result was that a CLU Maybe RPC involved a total of only two process switches on its critical path, one at the server to switch to an appropriate RPC server process and one at the client to switch back to the calling process when its reply arrived. The number of kernel calls and monitor actions was also minimized.

Chapter 12. Performance.

The degree of interaction that is possible between software on different machines is dependent on the performance of the inter-machine communication mechanism. If communication is slow then software must be designed so as to minimize the number of remote interactions. As communication becomes faster, a higher rate of interaction becomes feasible, simplifying the design of distributed applications. Since the CLU RPC system aims to encourage distributed applications by making remote interactions easier to implement, it appears logical that it should also try to minimize the cost of remote communication, so that implementors may use RPCs as freely as possible.

Thus, the CLU RPC system on the Mayflower kernel was intended to provide a high performance inter-machine communication mechanism. In assessing how well it achieved this goal, an appropriate yardstick appears to be the Single Shot Protocol (SSP) under the Tripos operating system. Tripos was intended to be a high performance system and SSP is essentially equivalent to the network protocol used for the Maybe RPC system.

Table 1 shows the minimal times for SSPs executed by the standard Tripos SSPLIB library and the times for equivalent application level send and receive exchanges down a bi-directional stream using the Byte Stream Protocol (BSP). The comparatively small difference in performance between these two protocols reflects the fact that the Tripos SuperMace implementation of BSP was tuned to support interactive terminal traffic efficiently (for example, it pauses slightly before acknowledging any received block, so that this protocol level acknowledgement can be merged with any application level response). This tuning also rendered it an excellent medium for generalized send and reply exchanges. Even so, it remains slower than even a comparatively untuned implementation of SSP.

Tripos-Tripos message exchange down BSP (SuperMace)	15.1 msec
Tripos-Tripos SSP (Spectrum)	12.5 msec
Mayflower-Mayflower SSP (Spectrum)	8.3 msec

Table 1. System & protocol comparisons.

Table 1 also shows the comparative times for SSPs between Tripos systems and between Mayflower systems. In both cases the measurements were based on several

thousand repeats (without even transient errors) of a minimal sized SSP using the systems' standard SSP libraries. Both the Mayflower SSP cluster and the Tripos SSPLIB library module provide equivalent functionality. For example they both allocate a new reply port for each SSP execution. They are also based on very similar operating system primitives, i.e. a port allocation and de-allocation function and various ring basic block functions. These times do not represent a lower bound on SSP execution times for either system – better times could be obtained for both systems by a variety of tricks. What they do represent is the performance the average programmer can expect to obtain from a general purpose mechanism.

The Mayflower kernel's higher performance on SSPs reflects the emphasis in its design on supporting network interactions efficiently. This performance gain comes despite the evidence, shown in table 2, that Mayflower's process switches are significantly slower than Tripos's. This reflects Mayflower's use of a language independent system interface and a more complex scheduler. However, more important than the cost of Mayflower process switches is the relative cheapness of Mayflower monitor calls. Since the Mayflower kernel is organized procedurally, a single monitor call often takes the place of a pair of process switches in a message based system such as Tripos. Thus allocating and then deallocating a port for each SSP requires 4 process switches in Tripos but only two, cheaper, monitor calls in Mayflower.

Tripos process switch (packet transfer)	210 μsec
Mayflower process switch (semaphore notify)	314 μsec
Mayflower process switch between monitors (semaphore notify)	378 μsec
68000 CLU null procedure call	18 μsec
68000 CLU monitored procedure call	75 μsec

Table 2. Operating system comparisons.

Table 3 shows the performance of the different network interface programs. The first time represents the interval between one host issuing a transmission request to its interface and the destination host being interrupted by its interface. This represents the network delay for a minimal transfer.

	SuperMace	Standard Spectrum	Mayflower Spectrum
Transmission time from host interface to host interface for a minimal transmission	2510	1400	830
Average cost per additional byte transferred	22.4	19.1	19.1

Table 3. Network interface performance. (All times in μsec.)

Table 4 shows the performance of the CLU RPC mechanism, using versions of the Spectrum ring interface program, at various stages of its development. The first system had no special kernel support and ran on an early version of the Mayflower kernel, which represented interrupts as semaphore events for device control processes. Replacing this interrupt scheme with a system which permitted CLU device handler procedures to be called directly on interrupts brought an immediate 15% saving. A fairly small further saving of 12% was obtained by providing explicit support for RPC within the kernel. (It must be remembered that the kernel's ring processing had always been designed with RPC in mind.) Finally another 15% was saved by modifying the Spectrum program to process host commands and replies more efficiently.

	Maybe	Exactly-once
First system, with interrupts handled as semaphore actions	10.6	—
After introducing CLU interrupt handling procedures	9.0	—
After integrating RPC into Mayflower kernel	7.9	11.3
After replacing standard Spectrum with Mayflower Spectrum	6.7	9.0

Table 4. Times in msec for minimal CLU RPCs as the system evolved.

The final CLU RPC times initially appear disappointing compared with the 1.1 msec RPC call times of the Birrell-Nelson Cedar RPC system, but these figures were for calls between Dorados [Lampson 80b], which have considerably faster processors than the Motorola MC68000s used for the CLU RPC system. (On a simple operating systems benchmark, Cedar on a Dorado is approximately nine times faster than its kindred language Modula-2 on a 68000 [Mitchell 83].) Furthermore, the Dorados ran the single language operating system Pilot [Redell 80] easing the interaction of the RPC run time system with the operating system. Dorados use microcoded device drivers running on the main processor to control input/output. This loses the parallelism available with a network

interface processor such as the Mace, but reduces the critical path time for each RPC by simplifying controller interactions.

The CLU RPC performance measurements were based on the time for 10000 calls one after another. No detailed information is available on the breakdown of processing time within either the client or server host. However, it appears likely that a significant proportion of the time goes in internal synchronization and buffer management. It is necessary for the RPC system to lock certain structures (such as buffer pools) at certain points of the call, but it must unlock these structures whenever it engages in a ring transaction to avoid unnecessarily hindering other calls. There is only a minimal amount of process switching on the critical path of a call, once to a server process on the called machine and once to get back to the client process on the calling machine. The client RPC mechanism manages to carry out part of its duties (e.g. issuing an additional ring reception request, if the outstanding pool is getting small) while its call is in progress at the server.

The bulk of the Mayflower kernel's ring processing and also of the RPC run-time system (other than the marshalling and unmarshalling procedures) is written in CLU. Undoubtedly higher performance could be achieved by rewriting this code in assembler and by better integrating it with the existing assembler components. Such an effort appeared unlikely to result in a large enough performance gain to justify the additional development and maintenance cost involved.

	Extra time (μsec)	Extra data (bytes)	Network delay (μsec)	Other delays (μsec)
One integer argument	216	4	76	140
5 integer arguments	644	20	382	362
A 5 integer record	1114	28	535	579
A 5 integer array	1854	48	916	938
An integer oneof	904	12	229	675

Table 5. Extra cost of various arguments to a Maybe call.

Table 5 shows the additional cost of various arguments to CLU Maybe RPCs. The costs for returning results and for arguments and results to Exactly-once calls will be the same. These times include the additional network delay in transferring the extra data and

"other delays" which represents the marshalling and unmarshalling times. Marshalling essentially involves copying a directed graph from the CLU heap into a buffer and unmarshalling involves copying objects from a buffer into the CLU heap and relocating them. For marshalling there is a significant overhead on each reference in determining firstly whether it is a valid pointer and secondly whether the designated object has already been transmitted. Because this code involves looking inside CLU objects and performing direct heap manipulation, it proved most convenient to write it in assembler. A certain amount of data compression could be carried out to reduce the amount of data transferred, e.g. an array dope vector might be compressed from 20 to 12 bytes, but the reduced network delay would have to be balanced against the increased complexity in the unmarshalling mechanism, which currently has a very simple role.

A Tripos SSP with a 25 Kbyte buffer	508 msec
A Maybe RPC with a 25 Kbyte string	578 msec

Table 6. Bulk transfer times (using Mayflower Spectrum).

When transferring complex application level data structures it can be argued that the overheads of the RPC marshalling mechanism must be compared with the cost of the application encoding and decoding the data structures itself. However some servers, such as fileservers, are likely to require the transfer of large amounts of uninterpreted data. Table 6 shows the comparative times for transferring 25 Kbytes of uninterpreted data by CLU RPC (as a CLU string) and by Tripos SSP. CLU RPC is around 15% slower, which represents the time spent copying the string from the heap into a buffer at one end and copying it back from a buffer into the heap at the destination machine. This relatively small performance overhead may be acceptable for some applications, such as inter-machine pipes, but might be deemed unacceptable for some critical applications, such as file i/o. (A specialized bulk transfer protocol might be more appropriate for file i/o anyway, disregarding performance. CLU RPC is essentially an application level service which transfers data between CLU heaps. This is inappropriate for, say, loading a program.)

Chapter 13. Conclusion.

This thesis has described a Remote Procedure Call system, its underlying philosophy and the framework within which it was implemented. This chapter summarizes the main issues explored, the conclusions reached and the scope for further work.

The main goal of this thesis was to explore ways of easing the production of software for distributed systems. It was decided that the provision of an automatic, type-safe transfer mechanism for arbitrarily complex user data structures was an important tool and that this tool could be most usefully integrated into a procedural language by providing it as part of a Remote Procedure Call mechanism.

Most work on linguistic extensions for distributed processing has aimed to provide remote operations with semantics as similar as possible to local operations. This thesis has argued that this is undesirable, that remote interactions should be clearly distinguished from local operations, and that it should be possible for client software to handle all network failures directly. This permits client code to be tuned to adopt optimal retry strategies for busy servers and also permits client software to take application specific adaptive action immediately when faced with a congested or inaccessible server. Building high reliability into the bottom level of a distributed system does not obviate the need for high-level recovery action, nor does it necessarily ease the task of providing end-to-end reliability.

An RPC system was implemented for an extended version of CLU and two alternative call mechanisms were made available to implementors, Maybe and Exactly-once. These two mechanisms permit implementors to either handle all errors themselves or to let a standard mechanism cope with all network failures. By removing irksome problems of implementation detail they should allow implementors to concentrate on the more important design issues involved in producing distributed systems. The CLU RPC system tries to avoid interfering with these design decisions, as might happen if it insisted on performing error handling itself. It would be interesting to investigate providing support for atomic transactions with a similar philosophy, with the atomicity layer merely trying to provide detailed support (type-safe stable storage, commit chains, etc.) for application level decisions.

The CLU RPC system makes binding between machines a run-time decision, under full program control. This permits the construction and adaptation of complex configurations to reflect run-time information.

A set of object-oriented concurrency features was added to CLU, based on the monitor paradigm. This meant that the entire system was procedurally oriented, with concurrency, remote interactions and even abstract types all being viewed procedurally. (CLU's abstract data types are accessed by procedural interfaces.)

A general purpose operating system kernel was implemented in CLU to support the RPC system and the concurrency extensions. This demonstrated the benefits to be gained from integrating a single simple protocol into an operating system and its device drivers.

In developing the CLU RPC system and its supporting kernel, CLU's abstraction mechanisms proved a useful tool. Individual components of the system could be modified without affecting other parts. The *flotsam* system proposed in section 8.3 shows how the abstraction mechanism can be further exploited to permit new representations and implementations of abstract objects to be incrementally introduced into a distributed system in a manner that appears impossible without data abstraction.

13.1 Applications of the CLU RPC system.

The Mayflower kernel, concurrent CLU and the CLU RPC system are currently being used for the development of a Resource Manager [Craft 83], a distributed compilation system [Wei 84] and a distributed debugging system [Cooper 84].

Craft's new Resource Manager service, and its associated Event Notification service, provide both CLU RPC and SSP interfaces to clients, although Craft uses CLU RPC for internal communication. The Resource Manager's clients expect it to either provide a resource or report failure within a period of only a few seconds. Thus Craft uses the Maybe call mechanism throughout, so that he can rapidly detect if particular servers are inaccessible and attempt to use alternative resources.

Wei intends to construct a distributed compilation system for CLU, based around a "library server" which will maintain a CLU type-checking library for a set of "compila-

tion servers". The library server will provide type-checking information to the compilation servers on demand and will also update the library on demand (and possibly notify interested compilation servers if an interface definition changes). The CLU data structures that are maintained in the compiler library are highly complex and could not easily be moved between machines without some form of automatic marshalling system.

The distributed debugging system being developed by Cooper will use a remote debugger on one machine to debug a number of client machines. The remote debugger currently uses Exactly-once CLU RPC to communicate with debugging stubs in each client machine. Cooper intends moving to the Maybe call mechanism so that client crashes which affect the debugging stub can be more rapidly detected and reported to users.

13.2 Multi-language RPC.

Only the problems of single language RPC have been discussed in this thesis. Further investigation is required into the provision of a multi-language RPC system which could permit, say, CLU clients to call servers written in Mesa.

Such a system would have to be able to provide cross language type-checking and data conversion. This is difficult, partly because not all features are common to all languages (e.g. CLU lacks the "enumerated types" of Mesa), and partly because different languages may have different representations for superficially similar types. For example, most languages on the VAX support 32 bit integers, but at least one language (CLU) only supports 31 bit integers. Other types such as arrays or pointer types are even less uniformly represented. Pointer types are important in Mesa, but irrelevant in CLU, since all CLU objects are represented by pointers.

It seems desirable to provide a system which permits interfaces to be defined either for a particular language, using all the richness of that language, or to be defined in terms of some well known set of types which can be mapped onto a wide variety of languages. Having a single RPC system accommodate, say, Mesa-Mesa, CLU-CLU and CLU-Mesa calls would represent a major boon to system implementors.

Chapter 14. References.

[Birrell 82] A D Birrell, R Levin, R M Needham & M D Schroeder.
"Grapevine: An Exercise in Distributed Computing",
CACM Vol 25 No 4, April 1982.

[Birrell 84a] A D Birrell & B J Nelson.
"Implementing Remote Procedure Calls",
ACM Trans. on Computer Systems Vol 2 No 1, Feb 1984.

[Birrell 84b] A D Birrell.
"Secure Communication Using Remote Procedure Calls",
Private communication, 13 Feb 84.

[Brinch Hansen 73] P Brinch Hansen.
"Operating Systems Principles",
Prentice-Hall 1973.

[CCITT 81] CCITT.
"CCITT Recommendation X25",
CCITT Yellow Book, Volume VIII, Fascicle VIII.2, Geneva 1981.

[Cheriton 79] D R Cheriton, M A Malcolm, L S Melen & G R Sager.
"Thoth, a Portable Real-Time Operating System",
CACM Vol 22 No 2, Feb 1979.

[Cheriton 83] D R Cheriton & W Zwaenepoel.
"The Distributed V Kernel and its Performance for Diskless Workstations",
Proc. 9th ACM Symposium on Operating Systems Principles, 1983.

[Cockshott 84] W P Cockshott, M P Atkinson, K J Chisholm, P J Bailey, R Morrison.
"Persistent Object Management System",
Software Practice and Experience Vol 14 No 1, Jan 84.

[Cook 82] S Cook & S Abramsky.
"Pascal-M in Office Information Systems",
Proc. 2nd International Workshop on Office Information Systems,
(Ed) Naffah, North Holland 1982.

[Cooper 84] R C B Cooper.
"Towards a Distributed Programming Environment for CLU",
Cambridge University PhD Proposal, 27 Sept 1984.

[Craft 83] D H Craft.
"Resource Management in a Decentralized System",
Proc. 9th ACM Symposium on Operating Systems Principles, 1983.

[Dijkstra 65] E W Dijkstra.
"Cooperating Sequential Processes",
Technological University, Eindhoven, 1965.

[Dion 80] J Dion.
"The Cambridge Fileserver",
ACM Operating Systems Review Vol 14 No 4, Oct 1980.

[Garnet 83] N H Garnet.
"Intelligent Network Interfaces",
PhD Thesis, Cambridge 1983.

[Gibbons 81] J J Gibbons.
"The Cambridge Nameserver",
Systems Research Group Documentation, Cambridge, Jan 1981.

[Girling 82] G Girling.
"Object Representation on a Heterogeneous Network",
ACM Operating Systems Review Vol 16 No 4, Oct 1982.

[Gray 78] J N Gray.
"Notes on Database Operating Systems",
Operating Systems - an Advanced Course,
(Eds) Goos & Hartmanis, Springer-Verlag 1978.

[Gray 83] J N Gray.
Private communication, Nov 83.

[Herlihy 82] M Herlihy & B Liskov.
"A Value Transmission Method for Abstract Data Types",
ACM Trans. on Programming Languages and Systems Vol 4 No 4, Oct 82.

[Hoare 72] C A R Hoare.
"Towards a Theory of Parallel Programming",
Operating Systems Techniques,
(Eds) Hoare & Perrott, Academic Press, New York, 1972.

[Hoare 74] C A R Hoare.
"Monitors: An Operating System Structuring Concept",
CACM Vol 17 No 10, Oct 1974.

[ISO 83] International Standards Organization.
"Draft Transfer Syntax for OSI Datatypes",
ISO Working Paper ISO/TC97/SC16/N1664, October 1981.

[Johnson 80] M A Johnson.
"Byte Stream Protocol",
Systems Research Group Documentation, Cambridge, April 1980.

[Kramer 82] J Kramer, J Magee, M Sloman & A Lister.
"Conic: An Integrated Approach to Distributed Computer Control Systems",
Imperial College Research Report DOC 82/6, 1982.

[Kung 81] H T Kung & J T Robinson.
"On Optimistic Methods for Concurrency Control",
ACM Transactions on Database Systems Vol 6 No 2, June 1981.

[Lampson 80a] B W Lampson & D D Redell.
"Experience With Processes and Monitors in Mesa.",
CACM Vol 23 No 2, Feb 1980.

[Lampson 80b] B W Lampson & K A Pier.
"A Processor for a High-Performance Personal Computer.",
Proc. 7th ACM Symposium on Computer Architecture, 1980.

[Lampson 81] B W Lampson.
"Atomic Transactions",
Distributed Systems - Architecture and Implementation,
(Eds) Lampson, Paul & Siegert, Springer-Verlag 1981.

[Lantz 82] K A Lantz, K D Gradischnig, J A Feldman & R F Rashid.
"Rochester's Intelligent Gateway",
IEEE Computer Vol 15 No 10, Oct 1982.

[Lauer 79] H C Lauer & R M Needham.
"On the Duality of Operating System Structures",
ACM Operating Systems Review Vol 13 No 2, April 1979.

[Leslie 83] I M Leslie.
"Extending the Local Area Network",
PhD Thesis, Cambridge 1983.

[Liskov 79a] B Liskov.
"Primitives for Distributed Computing",
Proc. 7th ACM Symposium on Operating Systems Principles, 1979.

[Liskov 79b] B Liskov & A Snyder.
"Exception Handling in CLU",
IEEE Trans. on Software Engineering Vol 5 No 6, Nov 1979.

[Liskov 81] B Liskov, R Atkinson, T Bloom, E Moss, J Schaffert, R Scheifler & A Snyder.
"CLU Reference Manual",
Springer-Verlag 1981.

[Liskov 82a] B Liskov & R Scheifler.
"Guardians and Actions: Linguistic Support for Robust, Distributed Programs",
Proc. 9th ACM Symposium on Principles of Programming Languages, 1982.

[Liskov 82b] B Liskov.
"On Linguistic Support for Distributed Programs",
IEEE Trans. on Software Engineering Vol 8 No 3, May 1982.

[Liskov 83a] B Liskov, M Herlihy, P Johnson, G Leavens, R Scheifler & W Weihl.
"Preliminary Argus Reference Manual",
MIT Programming Methodology Group Memo 39, Oct 1983.

[Liskov 84] B Liskov.
"Overview of the Argus Language and System",
MIT Programming Methodology Group Memo 40, Feb 1984.

[Lister 77] A Lister.
"The Problem of Nested Monitor Calls",
ACM Operating Systems Review Vol 11 No 3, July 1977.

[Mitchell 79] J G Mitchell, W Maybury & R Sweet.
"Mesa Language Manual, Version 5.0",
Xerox PARC Report CSL-79-3, April 1979.

[Mitchell 83] J G Mitchell.
"Martin Richard's Benchmark Program",
Private communication, 16 Sept 1983.

[Needham 79] R M Needham.
"Systems Aspects of the Cambridge Ring",
Proc. 7th ACM symposium on Operating Systems Principles, 1979.

[Needham 82] R M Needham & A J Herbert.
"The Cambridge Distributed Computing System",
Addison-Wesley, London, 1982.

[Needham 83] R M Needham, A J Herbert, J B Mitchell.
"How to Connect Stable Memory to a Computer",
ACM Operating Systems Review Vol 17 No 1, Jan 1983.

[Nelson 81] B J Nelson.
"Remote Procedure Call",
PhD. Thesis, CMU Report CMU-CS-81-119, 1981.

[Ody 79] N J Ody.
"A Protocol for "Single Shot" Ring Transactions",
Systems Research Group Documentation, Cambridge, April 1979.

[Ousterhout 80] J K Ousterhout, D A Scelza & P S Sindhu.
"Medusa: An Experiment in Distributed Operating System Structure",
CACM Vol 23 No 2, Feb 1980.

[Rashid 81] R F Rashid & G G Robertson.
"Accent: a Communication Oriented Network Operating System Kernel",
Proc. 8th ACM Symposium on Operating Systems Principles, 1981.

[Redell 80] D D Redell, Y K Dalal, T R Horsley, H C Lauer, W C Lynch,
P R McJones, H G Murray & S C Purcell.
"Pilot: An Operating System for a Personal Computer",
CACM Vol 23 No 2, Feb 1980.

[Richards 79] M Richards, A R Aylward, P Bond, R D Evans & B J Knight.
"Tripos - A Portable Operating System for Mini-computers",
Software Practice & Experience Vol 9 No 7, July 1979.

[Richardson 83] M F Richardson & R M Needham.
"The Tripos Filing Machine, a Front End to a File Server",
Proc. 9th ACM Symposium on Operating Systems Principles, 1983.

[Saltzer 81] J H Saltzer, D P Reed & D D Clark.
"End-to-end Arguments in System Design",
2nd International Conf. on Distributed Computing Systems, 1981.

[Schwarz 84] P M Schwarz & A Z Spector.
"Synchronizing Shared Abstract Types.",
ACM Trans. on Computer Systems Vol 2 No 3, Aug 84.

[Shrivastava 82] S K Shrivastava & F Panzieri.
"The Design of a Reliable Remote Procedure Call Mechanism",
IEEE Trans. on Computers Vol 31 No 7, July 1982.

[Solomon 79] M H Solomon & R A Finkel.
"The Roscoe Distributed Operating System",
Proc. 7th ACM Symposium on Operating Systems Principles, 1979.

[Spector 82] A Z Spector.
"Performing Remote Operations Efficiently on a Local Computer Network",
CACM Vol 25 No 4, April 1982.

[Spratt 80] E B Spratt.
"Operational Experiences with a Cambridge Ring Local Area Network in a University Environment",
Local Networks for Computer Communications,
(Eds) West & Janson, North-Holland 1980.

[Teitelman 84] W Teitelman.
"A Tour Through Cedar",
Proc 7th International Conference on Software Engineering, 1984.

[Ward 80] S A Ward.
"Trix: a Network-Oriented Operating System",
IEEE Compcon 1980.

[Wei 84] M Wei.
"A Distributed Compiling System",
Cambridge University PhD Proposal, 15 Sept 1984.

[Wilkes 79] M V Wilkes & D J Wheeler.
"The Cambridge Digital Communication Ring",
Proc. Local Area Communications Networks Symposium, Mitre & NBS,
Boston, May 1979.

[Xerox 81a] Xerox Corporation.
"Courier: The Remote Procedure Call Protocol",
Xerox System Intergration Standard XSIS 038112, 1981.

[Xerox 81b] Xerox Corporation.
"Internet Transport Protocols",
Xerox System Integration Standard XSIS 028112, 1981.