Technical Report

UCAM-CL-TR-117
ISSN 1476-2986

Number 117

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Distributed computing with RPC: the Cambridge approach

J.M. Bacon, K.G. Hamilton

October 1987

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

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

© 1987 J.M. Bacon, K.G. Hamilton

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

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Distributed Computing with RPC: the Cambridge Approach

J M Bacon

University of Cambridge Computer Laboratory, Corn Exchange Street, Cambridge CB2 3QG, UK

K G Hamilton

Digital Equipment Corporation, 100 Hamilton Avenue UCO-2, Palo Alto, CA 94301 - 1616, USA

The Cambridge Distributed Computing System (CDCS) is described and its evolution outlined. The Mayflower project allowed CDCS infrastructure, services and applications to be programmed in a high level, object oriented language, Concurrent CLU. The Concurrent CLU RPC facility is described in detail. It is a non-transparent, type checked, type safe system which employs dynamic binding and passes objects of arbitrary graph structure. Recent extensions accommodate a number of languages and transport protocols. A comparison with other RPC schemes is given.

1. Background: The Cambridge Distributed Computing System

CDCS [Needham82] is a heavily used heterogeneous research environment based on the Cambridge Ring local area network and employing the "pool of processors" approach to distributed computing. Infrastructure is provided by a number of small service nodes and supports **processor bank management** and **invocation of common services** by heterogeneous processor bank systems. The major research issues of its development were the distribution of system functions and their underlying communications support. Typical usage of CDCS is for a user, via a terminal server, to ask the Resource Manager for a processor from the processor bank. The user specifies a software system to be loaded into the acquired processor and then runs applications on this single machine. Although the user is free to acquire more than one machine, CDCS originally provided virtually no support for users to spread a task across a number of machines.

1.1. Evolution of CDCS

CDCS is now implemented on three bridged Cambridge Rings. The program development environment provided through the processor bank and file server, which for practical reasons were small research systems, was augmented by two Vax UNIX™ systems, with local discs, and a number of Ethernet-based MicroVax2s. Sun and Xerox distributed systems also use the Ethernet [fig.1]. The advent of the Cambridge Fast Ring gives potential for a unified approach.

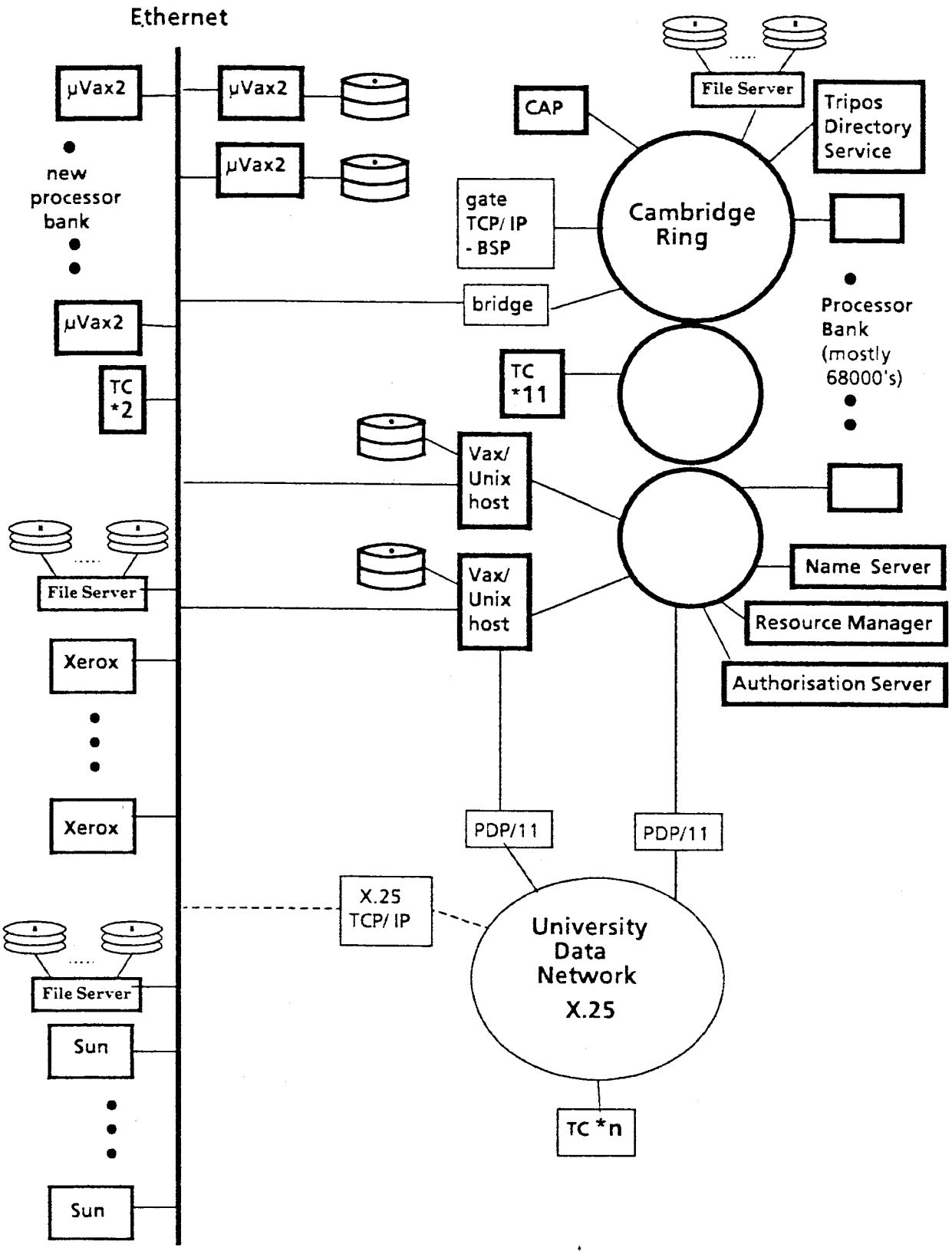

FIGURE 1. The Cambridge Distributed Computing Research Environment

1.2. CDCS design overview

A processor bank may contain heterogeneous hardware and software systems and each system may have its own implementation of naming, protection, reliability, etc. An aim of CDCS was to allow such systems to share common services such as printing and disc storage.

System and dialogue structure

The system software is partitioned and distributed. A request-response protocol was provided for remote service invocation and a simple byte stream protocol supports the connection of terminals to processor bank systems.

Naming of services

Each service has a flat text string name which is mapped to a network address together with information needed for service invocation. This mapping is carried out by a name server at a fixed network address. This is the only address which may be embedded in software.

Naming and protection of objects

The file storage service (CFS) is the only CDCS service that functions as an object type manager. This "universal file server" is designed to be used by any number of different file directory servers, each with its own text naming conventions and access control policies.

Authorisation for Service Use

Each processor bank system carries out its own user authentication but must register its current user with a CDCS authorisation server, the Active Name Table manager (ANT). ANT issues a session key, with a random component, which, together with information on the category of the user, functions as a capability for service use, in that a server may check with ANT that a request comes from an authenticated user.

Reliability

Each service may take its own independent approach to reliability, for example, CFS provides atomic transactions on special files. The infrastructure was designed for rapid rebooting through the boot server. The network interfaces and node software provide facilities for remote control and debugging. A dead man's handle technique is used to monitor allocated processor bank systems.

Summary

A major advantage of the processor bank approach is that new systems may be made available to users as technology evolves without any change in the underlying system. Also, the model of independently managed subsystems sharing common services is widely applicable.

2. The Mayflower Project

The Mayflower group was set up in 1982 to provide an environment for developing and running distributed applications and services. It comprises a language, Concurrent CLU, a communications protocol, RPC, integrated into the language system, and an operating system, the Mayflower supervisor.

CLU [Liskov81] was selected and extended by the Mayflower group. CLU is object oriented in style and provides procedures for procedural abstraction, iterators for control abstraction and clusters for data abstraction. It is strongly typed and supports user defined abstract types very well. It has separate compilation facilities and the compiler generates and checks interface specifications. Parameterised clusters go some way towards providing the facilities usually associated with a polymorphic typing system.

CLU was originally extended with monitors, semaphores and a lightweight fork primitive. Experience with Concurrent CLU showed that classical monitors unduly restrict concurrency in large systems [Cooper 85] and a critical region construct, with programmer specified locking, was added. Also, a conflict was shown between the need for abstract interface specifications to handle the complexity of large systems and the need for systems implementors to have knowledge of the concurrency behaviour of modules. A methodology was developed to integrate these requirements.

Mayflower RPC is a type-checked, type-safe, language-level construct incorporating dynamic binding under program control. Arbitrarily complex objects of practically any type in the CLU language, including user-defined abstract types, can be passed in arguments to RPCs. Mayflower philosophy is that RPC syntax should not be transparent and the programmer has a choice of semantics; MAYBE or EXACTLY ONCE. A detailed explanation of RPC semantics is given as an appendix. Mayflower RPC also allows new representations of a data type to be incrementally introduced into a distributed system without simultaneously halting and reloading every node. During the upgrade period objects using both the old and new representations can co-exist. Mayflower RPC operates through bridges and across a ring-ethernet gateway. Details are given in section 3.

The Mayflower supervisor was designed for implementing high performance services with internal concurrency. It therefore supports lightweight processes running in a shared address space (or domain) and a fork primitive is provided for dynamic process creation within a domain. Resources are allocated to a domain and are shared by all processes therein. Multiple domains per node may be used but inter-domain communication is by (expensive) RPC. Language independence was seen to be desirable but was not a major design focus.

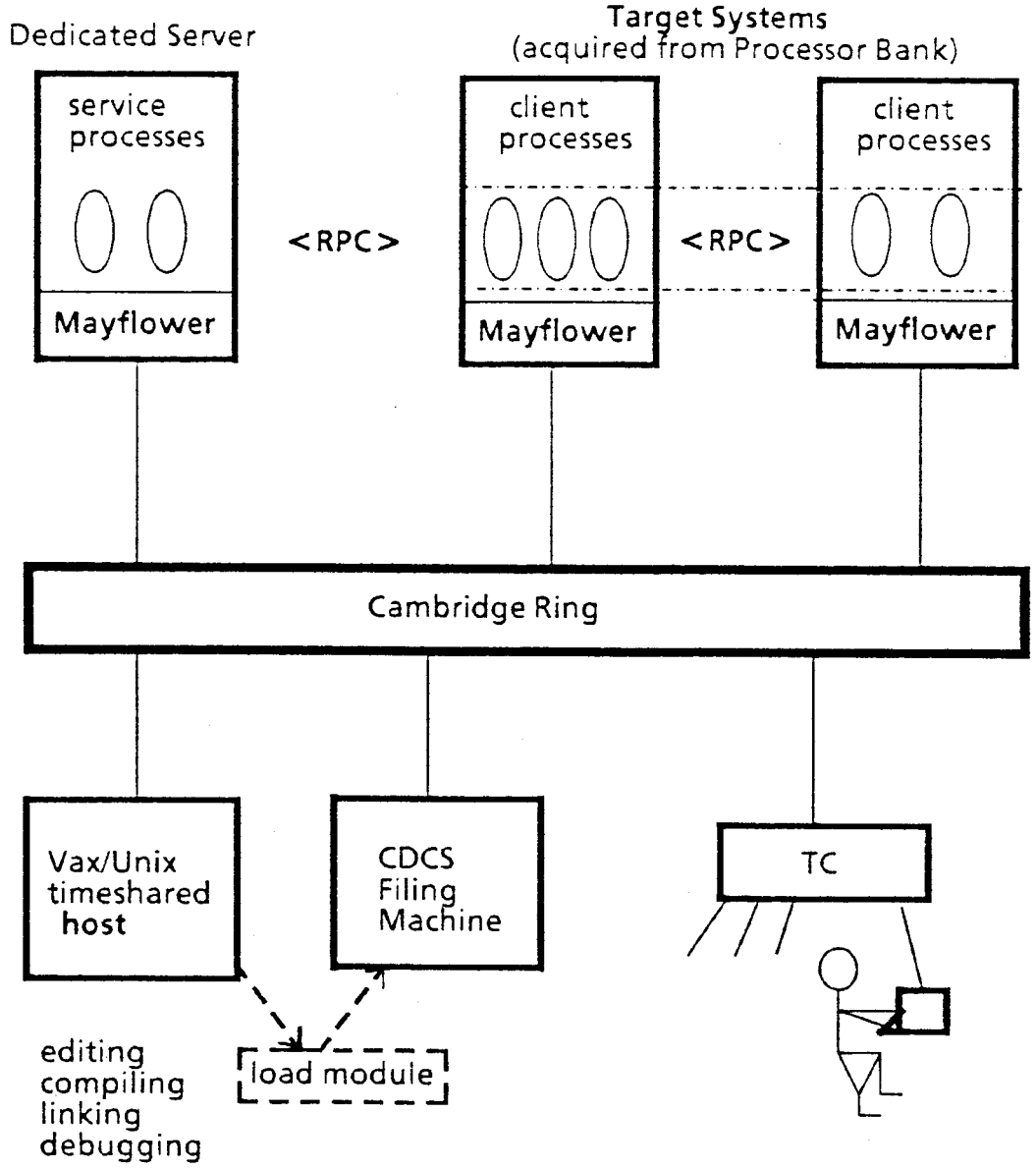

FIGURE 2. Mayflower on CDCS

System development is carried out in Concurrent CLU on (Micro)Vaxes under Unix and target code runs on processor bank MC68000s under the Mayflower supervisor and, for preliminary testing, on Vaxes under Unix. CDCS infrastructure, services and applications can be written in Concurrent CLU to run under Mayflower and have RPC interfaces [fig.2]. The CDCS authorisation server ANT was reimplemented and a new processor bank manager is being developed. Several research projects have been carried out under the Mayflower environment including a distributed compilation system [Wei87], a debugger for distributed concurrent programs [Cooper87] and distributed directory services [Seaborne 87]. Applications in the areas of graphics and multi media services have also used Mayflower.

3. Mayflower Remote Procedure Call (RPC)

The philosophy of Mayflower RPC is not to hide from the programmer that certain processing is remote. The peculiar semantics and overhead of remote operations are made explicit. The language was therefore modified to add new syntax rather than using the method of stub generation often associated with transparent RPC [Birrell 84].

3.1. Definition and Call Syntax

The definition of a procedure which may be called from a remote node contains the keyword **remoteproc** replacing **proc** in the header. Other aspects remain the same.

```
a_remote_proc = remoteproc ( <args> )
                returns ( a_type )
                signals ( problem )
                ...
end a_remote_proc
```

A new syntax, the call expression, is used for performing RPC's. The keyword **call** precedes the invoked procedure's name and a number of control keywords (**resignal, zealously, timeout, at**) may follow the invocation's arguments, eg.

```
v: a_type := call a_remote_proc (<args>) resignal problem
```

where problem is an exception signalled by the remote procedure, see also below. (A CLU procedure, both when returning normally and signalling an exception, may return an arbitrary number of results of arbitrary type).

3.2. Argument Passing

All objects in CLU are represented by pointers so that local variables merely contain pointers into the heap. When a local procedure is called, it is given copies of pointers to the argument objects. This is known as **call by sharing** since any changes the called procedure makes to the argument objects will be visible to owners of other pointers to these objects. The argument passing semantics used for RPC are **call by copy.** Arguments are copied into the remoteproc machine's heap and results are copied back. Modified argument objects are not shipped back, for efficiency and semantic reasons. Copying an argument to a remote machine and back (as a result) may cause it to cease to be equal with its former self. The partial transfer of large objects, with fetch on

demand of additional object components, was considered but not implemented because of its complexity.

3.3. Argument Marshalling and Unmarshalling

Primitive types, structured types and user defined types must be considered. Almost all CLU types can be marshalled, with the exceptions of semaphore, monitor-lock, any and proctype. The new type, remoteproctype may be used to transfer procedure names between nodes. Arbitrary structures including pointers may be transferred and sharing of subobjects and cycles within the object are preserved.
The standard CLU heap format for the representation of system types is used to transfer all types except pointers. A special, detectable, 32 bit format is used for pointers which includes a 15 bit offset within the network buffer (a 32Kbyte limit on the size of data transferred makes this possible). Objects are stored as contiguous vectors of pointers which are buffer offsets. This is optimised for speed of conversion from heap format rather than compactness.
User defined types must include marshal (and unmarshal) conversion operations which are called to obtain representations, eventually in terms of system types, suitable for transmission, for example:

```
thing = cluster ...
        rep =
            record [ v: thing_value,
                     m: monitor_lock,
                     g: global_info
                     ]
        ...

marshal = proc ( t: thing )
                        returns ( thing_value )
        return ( t.v)
end marshal

unmarshal =
                    proc ( tv: thing_value )
                    returns ( thing )
        ...
end unmarshal
```

3.4. Run Time Binding

Compile time type checking is inadequate without runtime consistency checking. The compiler generates a UID for every procedure interface including remoteprocs. This is stored with the interface definition and the programmer must ensure that this information is made available, as a library module, to the compiler when any call to this remoteproc is compiled. Recompilation of a procedure may cause its interface to

change, in which case a new UID is generated. At run time, the interface UID is transmitted with the call, thus allowing the RPC facilities of the caller and the called procedure to check that the exported and imported interfaces are still identical. Currently, the programmer must find the network address of the called procedure, by a name server lookup for example, and pass it to the RPC service.

```
carver: network_address := ...
...
t: a_type := call a_remote_proc ( <args> )
                              at carver
...
a_proc: remoteproctype := a_remote_proc

...
f:= call a_proc ( <args> )
```

Work is in progress to improve naming, binding and configuration services.

3.5. Call Semantics

The CLU RPC programmer has a choice of call semantics. The default, used in the above example, is the lightweight MAYBE. If the alternative "reliable" EXACTLY ONCE (in the absence of node crashes, see Appendix for details) is required, the keyword **zealously** is appended to the call.

```
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 any remote 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.
If an error occurs during 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 such as a timeout or apparent congestion at the remote node, but that a retry may succeed. Hard-error is signalled by both the MAYBE and EXACTLY ONCE options and indicates that an apparently unrecoverable error has occurred, for example, failure to contact the server node or denial by the server node that the called remote proc is to be found there.

An exception handler for the MAYBE protocol would have the form:

```
begin
  ...
  t: a_type:= call a_remote_proc ( <args> )
                                  timeout 2000
  ...
end except
  when problem ( p: problem ):
    ...
  when hard_error ( why: int ):
    ...      % not worth retrying
  when soft_error ( why: int ):
    ...      % worth retrying a few times
  end
```

An exception handler for the exactly once protocol would have the form:

```
begin
  ...
  t: a_type:= call a_remote_proc ( <args> )
                                  zealously
  ...
end except
  when problem ( p: problem ):
    ...
  when hard_error ( why: int ):
    ...
  end
```

3.6. Multiple Transport Protocols

CDCS had started out with heterogeneity as a central design aim. The initial phase of Mayflower produced a single language subsystem within CDCS; Concurrent CLU over the Mayflower supervisor on 68000's. Recent work has extended the RPC system to allow interworking between Concurrent CLU programs running on ring based 68000's over Mayflower and on Ethernet based Microvaxes over UNIX. RPC runs over UDP (User Datagram Protocol) and IP (Internet Protocol) on the Ethernet and over the Basic Block protocol on the Cambridge Ring.
The transport protocol required is selected at RPC bind time and the network address now includes network type as well as a network specific address. RPC gateways can be written in CLU and other networks are easy to add.

3.7. Multiple Data Formats

Some interworking was also seen to be desirable between programs written in Concurrent CLU running over Mayflower and programs running on Xerox or Sun workstations written to use Xerox Courier RPC and Sun RPC respectively. It was found that a subset of the types supported by Concurrent CLU RPC, the built in immutable types, are used in Sun XDR (eXternal Data Representation Standard) and

the Courier data standard. CLU RPC was therefore extended to allow selection of any one of these data representations.
CLU clients can access existing Xerox and Sun services, UNIX and Xerox XDE (Xerox Development Environment) and Interlisp clients can access CLU servers and limited support is provided for new multi language distributed applications.

4. Comparable Services

4.1. Loosely Typed Interfaces

A transmission representation may be defined for common language types and network interfaces may be defined in terms of these types. A string, for example, may be represented by a one-byte count followed by that number of character bytes. The types required for any interface are only known by convention, however, and interfaces cannot be type checked at compile time. Examples of systems with this style of communication are Amoeba [Tanenbaum 85] and the Newcastle Connection [Shrivastava 82] both supported by a set of C library routines. Type checking of system types can be carried out at run time if an identifying tag is transmitted with each type, but this increases transmission and processing overhead. The ISO presentation layer standard, ASN.1 (Abstract Syntax Notation 1) defines such a representation [ISO 85].

4.2. Language Level Solutions Based on Message Passing

Strong typing and typed checked remote interfaces may be incorporated into a language which uses message passing for inter-process communication. In the Conic system [Sloman 82] for example, typed messages are associated with typed input and output ports for modules. The Conic module programming language is strongly typed and the compiler can check that a message sent to an output port or received from an input port corresponds to the type of that port. The configuration language checks for type compatibility when linking an output port to an input port. There is no explicit type identification in the transferred data.

4.3. Transparent Remote Procedure Call

Pioneering work in language level RPC was carried out at Xerox PARC for the Mesa language [Nelson 81] and the Cedar language and environment [Birrell 84]. In his Emissary design, Nelson aimed to make remote calls equivalent to local calls in all but performance. In fact, the problem of achieving argument passing transparency for pointers and call by reference was not solved. Later, Birrell and Nelson developed an

efficient mechanism for Cedar in which a transparent syntax is used but which places less emphasis on transparent semantics. The Cedar RPC system uses a preprocessor to generate stubs; the compiler has no knowledge that certain calls are to remote procedures. This method implies that any call-specific control information, such as a request for an encrypted call, must be passed into the RPC mechanism as a call argument.

4.4. Atomic Remote Procedure Call

The Cambridge and Xerox RPC systems provide a robust, type checked inter-machine communications facility. They do not claim to recover from node crashes. The Argus language [Liskov 83,84] embeds support for atomic actions within the language, ensuring that the system is automatically resilient to node crashes. It is reported, however [Liskov 84] that Argus programs must be written with great care if both atomicity and a high degree of concurrency are to be achieved and if deadlock is to be avoided.

4.5. Interface Specification Languages and Heterogeneous RPC

Interface specification and configuration languages are often provided within programming support environments even when they are directed towards producing single node programs within a single language system. They contribute towards the management of large, multi module software systems by allowing inter module dependencies to be expressed, and thus ensuring that strong typing is maintained when modules are recompiled.
The techniques developed may be extended for distributed programs with heterogeneous components. Accent and later Mach at CMU employ such a language, Matchmaker [Jones 86] as does the HCS (Heterogeneous Computer Systems) project at the University of Washington [Black 87].

5. Current Work, Summary and Conclusions

Some ten years of experience with distributed computing systems have shown that the two functions provided by CDCS, processor bank management and support for service invocation by heterogeneous, independently managed subsystems, form a good basis for distributed system design. Although single user, diskless systems were originally envisaged, a range of configurations may be accommodated. Special purpose hardware may be included, several systems may be acquired to run a parallel application and a range of operating systems may be made available to users.

CDCS had started out with heterogeneity as a central design aim. The initial phase of Mayflower produced a single language subsystem within CDCS; Concurrent CLU over the Mayflower supervisor on 68000's. Subsequent work has allowed a number of transport protocols and data formats to be selected and work on naming, binding and configuration is still in progress. The next phase is to port Mayflower to other hardware.

After some years of experience with Concurrent CLU RPC we feel that non-transparent syntax best reflects the realities of an environment comprising distributed programs. Finer control over timeout and retry strategies than those provided may be desirable. A fully type checked high level language facility is clearly required. Research will continue on achieving heterogeneity with performance.

Acknowledgements

This paper presents the work of the Systems Group, and in particular the Mayflower project, at the Computer Laboratory over a number of years. Many acknowledgements are due: Graham Hamilton wrote the Concurrent CLU RPC system. Concurrent CLU and the Mayflower supervisor were written and maintained by Graham Hamilton, Dan Craft, Robert Cooper, Andy Seaborne and Mian Wei. Derek McAuley extended CLU RPC to operate through ring bridges and to employ multiple transport protocols. Joe Dixon has assisted with protocol implementation and Willy MacDonald with multiple data formats. Andrew Herbert led the Mayflower project until 1985.

References

[Birrell 84] Birrell A D and Nelson B J, "Remote Procedure Call" ACM TOCS, 2(1), Feb 84

[Cooper 85] Cooper R C B and Hamilton K G "Preserving Abstraction in Concurrent Programming" Cambridge University Computer Laboratory TR 76, August 1985 (accepted for publication in IEEE Trans. on Soft. Eng.)

[Cooper 87] Cooper R C B, "Pilgrim: A Debugger for Distributed Systems" Proc IEEE 7th ICDCS, Berlin 1987

[Craft 83] Craft D H, "Resource Management in a Decentralised System" ACM SOSP9, Operating Systems Review 17(5), 11-19, Oct 1983 and PhD thesis, Cambridge 1885, TR 73

[Hamilton 84] Hamilton K G , "A Remote Procedure Call System" PhD thesis, Cambridge University, TR 70, 1984

[ISO 85] Specification of Basic Encoding Rules for Abstract Syntax Notation One, ASN.1, ISO/DIS 8825 June 85

[Liskov 81] Liskov B et al , "CLU Reference Manual" Springer Verlag, LNCS 114, 1981.

[Liskov 83] Liskov B et al, Preliminary Argus Reference Manual
MIT Programming Methodology Group Memo 39, Oct 83

[Liskov 83] Liskov B, Overview of the Argus Language and System
MIT Programming Methodology Group Memo 40, Feb 84

[Needham 82] Needham R M and Herbert A H, "The Cambridge Distributed Computing System" Addison Wesley 1982.

[Seaborne 87] Seaborne A F, "Filing in a Heterogeneous Network"
Cambridge University Computer Laboratory, thesis in preparation, 1987

[Shrivastava 82] Shrivastava S and Panzieri F, "The Design of a Remote Procedure Call Mechanism", IEEE Trans Computers, 31(7), July 1982

[Sloman 84] Sloman M et al "Building Flexible Distributed Systems in Conic" in Duce D A (ed) Distributed Computing Systems Programme, Peter Peregrinus, Sept 84

[Swinehart 86] Swinehart D, "A Structural View of the Cedar Programming Environment" ACM TOPLAS, 8(4), 419-523, Oct 86

[Tanenbaum 85] Tanenbaum A S and van-Renesse R, "Distributed Operating Systems" ACM Computing Surveys 17(4), 419-470, Dec 85

[Wei 87] Wei M, "Distributed Compilation"
Cambridge University Computer Laboratory, thesis in preparation, 1987

Appendix: RPC Semantics

Fig. 3 shows a simplified view of system components that are invoked when a remote procedure call is made. Binding is not included and marshalling is shown as a single activity. A request, reply, acknowledge (RRA) protocol is assumed. An alternative is request, acknowledge, reply, acknowledge (RARA). In a local procedure call, caller and called procedures crash together. In a remote procedure call the following possibilities must be considered (the node containing the call is referred to as the client, that containing the called procedure, the server):

- **Congestion**: the network or server may be congested, causing the timeout at **A** to expire.
  In CLU RPC the MAYBE protocol does not retry, the EXACTLY ONCE protocol retries a number of times.
- **Client Failure**: the client may fail after sending the request. The remote call will go ahead (termed an orphan) as will any further related calls that it may make (more orphans) but the timer at **E** will expire and no acknowledgement will be received on prompting. The server may have made permanent state changes as a result of the call. Some server operations may be made repeatable (idempotent), for example, a file server operation that reads a file from byte n rather than from the

current pointer, but this is not possible in general. The client, on being restarted, may repeat the same call but the repeat will not be detected by the RPC service and a new id will be generated.

The CLU RPC system aims to provide an efficient communications facility and makes no attempt to exterminate orphans. Software at higher levels may provide atomic transactions with checkpointing and rollback facilities. The performance penalties associated with such a service should not be made mandatory for all users.

- Server Failure: the server may fail before the call is received or at several points during the call (in all cases the client timeout at A will expire).
  B - after the RPC service receives the call but before the call to the remote procedure is made,
  C - during the remote procedure invocation,
  D - after the remote procedure invocation but before the result is sent.
  In all cases the client might repeat the call when the server restarts. In cases C and D this could cause problems since the server could have made permanent state changes as a result of the call. It is feasible that the RPC service could save received RPC id's on stable storage or that a timestamp could be included in the id so that the server could distinguish pre and post crash calls.
  These cases are indistinguishable to the CLU RPC service. The MAYBE protocol tries once then signals an exception to the caller, the EXACTLY ONCE protocol makes repeated tries then signals hard error. Higher level software is assumed to handle any required recovery in cases C and D.

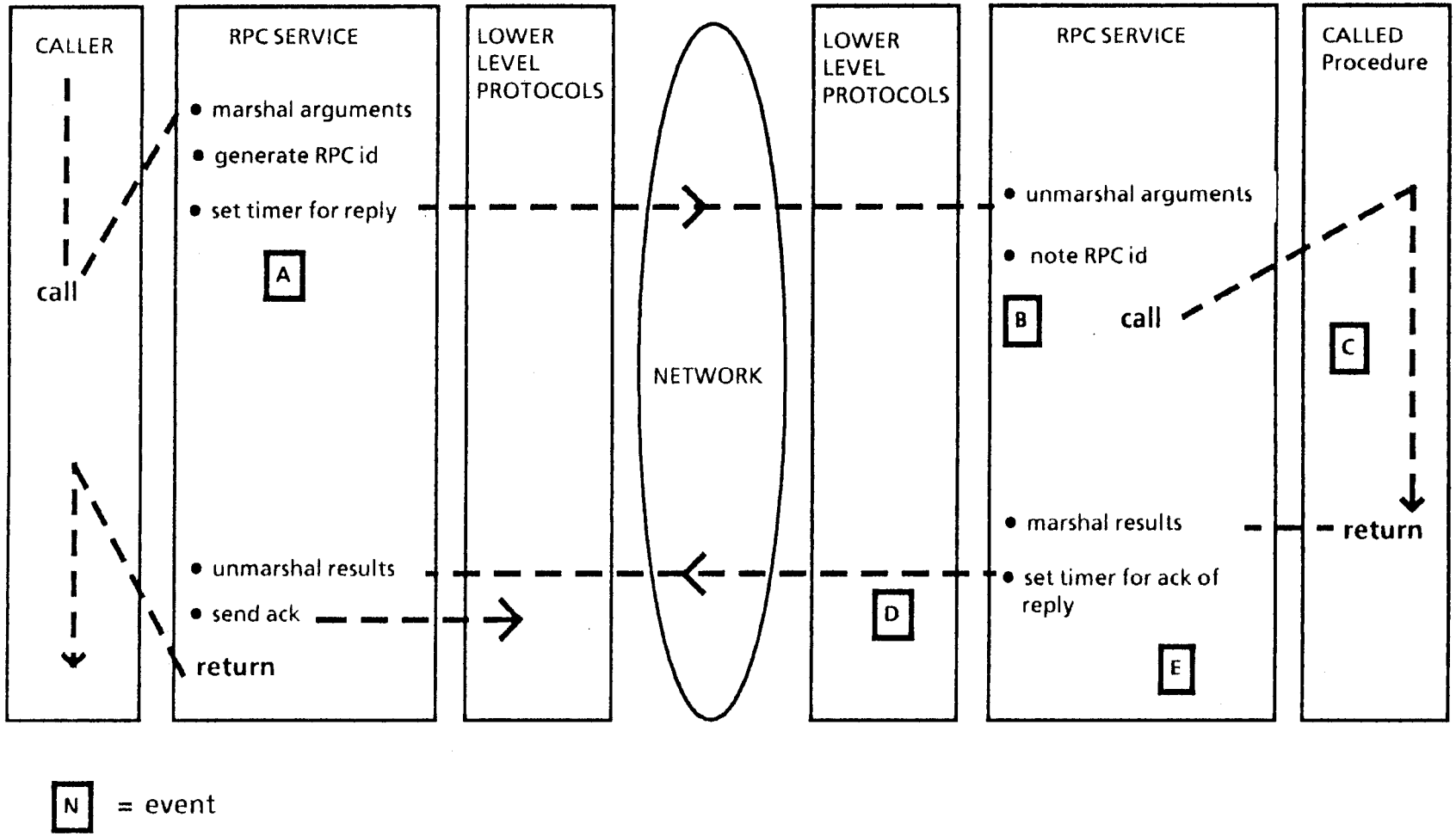

FIGURE 3.
RPC Semantics