National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

Multicasting in a High-Level Language

by

James Russell Gunson

B.A., Oxon., 1965
Ph.D., Dunelm, 1968
M.Ed., British Columbia, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

© James Russell Gunson 1989

SIMON FRASER UNIVERSITY

April 1989

All rights reserved. This thesis may not be
reproduced in whole or in part, by photocopy
or other means, without permission of the author.

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-59346-6

Canada

Approval

Name: James Russell Gunson

Degree: Master of Science

Title of Thesis : Multicasting in a High-level Language

Examining Committee:

Chairman: Dr. Joseph G. Peters.

Dr. M. Stella Atkins, Assistant Professor,
School of Computing Science,
Senior Supervisor.

Dr. F. Warren Burton, Professor,
School of Computing Science,
Supervisor.

Dr. Robert D. Cameron, Assistant Professor,
School of Computing Science,
Examiner.

Date of Approval: June 27 89

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Multicasting in a High-Level Language

Author: _____
(signature)

James Russell Gunson
(name)

27 June 89
(date)

Abstract

Multicasting allows a sender to send the same message simultaneously to a group of receivers, which may be required to reply. In comparison to a semantically equivalent series of one-to-one messages, multicasting facilitates greater parallelism among receivers, reduces network traffic, and reduces the work performed by the sender. At the lower levels, multicasting occurs between processes, one of which may send to others in a given process group. Members of the group may be distributed over a local area network.

The high-level distributed programming language SR was chosen as the testbed for our multicasting experiments. In an operation-oriented language such as SR, the receivers of a multicast must be a group of like operations. The set of receivers is termed a multicast network, a distributed entity. Multicast network access may be controlled by means of capabilities.

This thesis discusses several issues concerned with multicasting in SR, including semantic, linguistic and implementation issues. The syntax and semantics of multicasting are discussed from the perspective of message passing and remote procedure call paradigms. The use of an explicit structured reply queue is discussed. The thesis also proposes a way of implementing multicasting within the current SR implementation.

Dedication

In memory of my father,

George Charles Gunson,

who died during the preparation of this work.

Acknowledgements

My thanks Dr. Stella Atkins for suggesting the subject of this work, and for her help and guidance. My thanks also to the other members of the examining committee for their helpful criticisms.

This work was completed while the author was employed by Kwantlen College. Thanks go to John Levin, for his support of my studies.

Last, but not least, my thanks to my wife, for her tolerance and support.

Table of Contents

1. INTRODUCTION

The trend to replace single large computers by networks of smaller machines has increased interest in the design of parallel and distributed algorithms, in distributed operating systems, and in new languages which facilitate the programming of such algorithms and operating systems.

A crucial aspect of distributed systems is the communication scheme which permits processors or processes to pass information and synchronise their activities. For efficient communication there must be a suitable physical link between machines, and software to provide the reliability not provided by the hardware. In addition, the operating system and language must provide the user with a convenient interface, permitting reliable process-to-process communication. If the user employs a relatively low-level language, such as C, this service will be provided by system calls.

Communications may be one-way, from sender to receiver, or two-way, if a reply is required. The most common form of communication is from one entity to another, and possibly back, the *unicast*. Recently interest has been aroused in one-to-many communications, *broadcasts* and *multicasts*, which are the focus of this work. Various ways of synchronising the work of sender and receiver have been well addressed with regard to one-to-one communications[Andrews83], but only recently with regard to one-to-many [Cheriton85], [Navaratnam88], [Ahamad85], [Atkins89], [Birman87], [Martin87].

Issues related to one-to-many communications include dynamic process groups (the 'many' may change), how to incorporate suitable communications primitives into languages, reliability (atomicity and same order delivery), and methods of implementation.

1.1 Unicasts, Multicasts and Broadcasts

Ahamad and Bernstein describe three basic schemes of inter-process communication: the **unicast, multicast,** and **broadcast** [Ahamad85]. The *unicast* is the usual one-to-one scheme. The *multicast* consists of a message being sent to **a group of processes** running on any subset of the hosts in a network. A *broadcast* is sent to **all hosts**. It is noted that one-to-many communication may be applied to such problems as distributed commit protocols, elections, reliable storage and others. Navaratnam et al. describe multicasting as a communication with a process group using the group's **logical name** [Navaratnam88].

Communication on the Sun-Network [Leffler83] illustrates the relationship between broadcasting and multicasting: broadcasting takes place between host machines, multicasting between processes. Machine-to-machine communication is implemented using machine addresses and ports, which function as mailboxes on each machine. One-to-one communications use specific ports and machine addresses; broadcasts employ a wildcard value for the destination address, with a specific port number, and are delivered to that port on all machines in the network. A daemon process may be created on each processor to listen to a port and react to messages received. Each broadcast message is thus delivered to a group of daemon processes that typically offer a specific service to the user. As the message is received by a subset of all processes, the machine level broadcast effects a multicast at the process level.

Broadcasts and multicasts have several potential advantages over unicasts. First, as in *V* [Cheriton85], they may permit a user to request a service without knowing the identity of the server. Second, network traffic may be reduced as one message replaces several. Third, the average time for a member of the process group to receive the message will be reduced. Fourth, the user code may be reduced in size and simplified as one statement replaces several. The extent of these advantages depends on the particular nature of algorithm being executed. A multicast replacing a series of unicasts, must

carry the information contained in all the unicasts. The practicality of this depends on the degree of duplication in the unicasts' data. Overall, the relative costs depend on the ratio of processing time to communication time, the speed of communications, the nature of the algorithm and the degree of reliability required.

1.2 Synchronisation

Synchronisation refers to restrictions imposed on the order in which code of the sending process and that of the receiving processes may be executed. These constraints may be necessary for the sender to receive replies from the receivers, or because the sender must be assured that the receivers have completed their task before it (the sender) continues.

Andrews and Schneider[Andrews83] discuss at length the relationship between the communication of data and synchronisation. They observe that two options exist for synchronisation: shared variables and message passing. The authors also identify three main types of language: **procedure oriented, message oriented** and **operation oriented**. *Procedure oriented languages*, such as *Modula* [Wirth77] or *Concurrent Pascal* [Brich Hansen77], use shared variables to effect process interaction. Most such languages are monitor based. *Message oriented languages*, such as *CSP* [Hoare78], *Gypsy* [Good79] or *PLITS* [Feldman79] employ *send* and *receive* statements to pass messages between processes. *Operation oriented languages*, such as *SR* [Andrews87] and *Ada* [U.S.D.D.81], employ a form of remote procedure call as their main communication scheme.

[Andrews83] outlines a number of issues related to message passing: how the source and destination of a message are to be specified, and what synchronisation should apply. *Direct naming* is when both the sender and receiver name each other, creating a one-to-one *channel*. This paradigm does not permit a server to serve multiple clients; in that case a many-to-one communications scheme is required.

In message passing, various synchronisation schemes are used, based on the sender executing a *send* statement and the receiver executing a *receive* statement. These statements may be blocking or non-blocking: the statement may effect some action and continue immediately (non-blocking), or may await the completion of the action, perhaps on a remote site (blocking). Whether a *send* is blocking or non-blocking depends on the buffering provided. If there exists an effectively unlimited buffering capacity, the *send* may return immediately, assured that, short of failure, the message will eventually be passed to the receiver. This is termed **asynchronous message passing**. In this scheme the sender can get arbitrarily ahead of the receiver, as in Figure 1. (In Figures 1, 2 and 3, the time axis points downward, a solid line indicates an executing process, and a hatched region indicates that a process must block.)

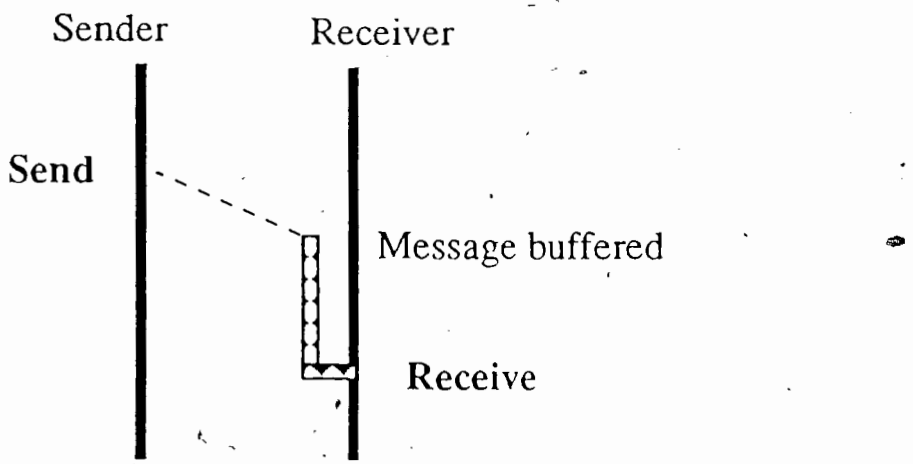

Figure 1 Asynchronous Message Passing

If, at the other extreme, no buffering exists, the *send* must block until the message has been received by the sender, as in Figure 2: this is termed **synchronous message passing**. In this scheme the exchange of a message represents a synchronisation point for the two processes.

Receive statements are mostly blocking, as the process is often unable to proceed until the message is received. However, operating systems may require a non-blocking *receive*.

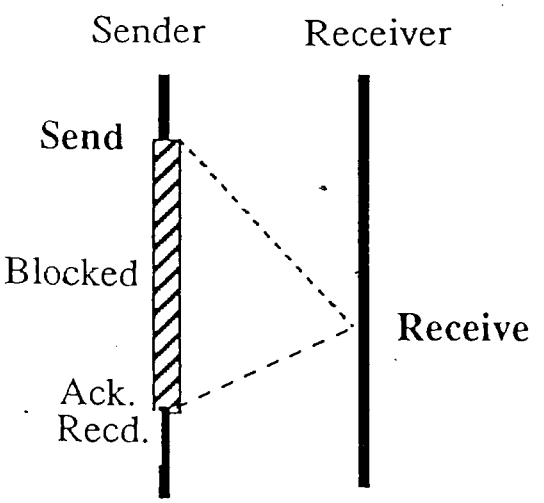

Figure 2 Synchronous Message Passing

Send and *receive*, taken in combination, allow the user to program a number of communication and synchronisation schemes. In client-server interactions, the client may execute a *send* and the server a *receive*, for the request for service to be made. The client may then execute a *receive*, and wait for a reply for the server, which executes a *send*, as illustrated in Figure 3(a).

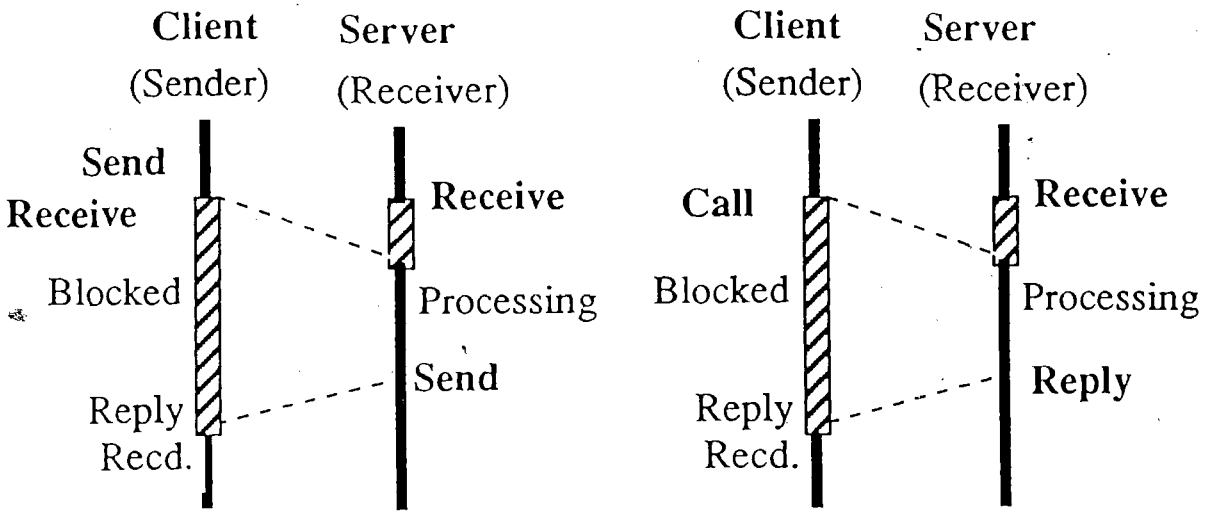

Figure 3 A Remote Procedure Call using (a) Send and Receive, or (b) Call

This use of *send* and *receive* is sufficiently common for many languages to support it directly as a **remote procedure call** (Figure 3(b)). The client executes a *call* which blocks until the reply is received. The server may serve a *call* in two ways. A *call* may be serviced by a process, which will execute when the *call* arrives, or to a *receive-statement*, which may be placed at some point in the code of a process. The first method resembles the conventional procedure call in that each *call* results in the execution of a body of code from beginning to end. In the second method, the receipt of a message by a *receive statement* is a synchronisation point for the client and server processes, termed a **rendezvous**. The *rendezvous* provides the server with greater flexibility in choosing when and how to serve the client. This is particularly true if *selective communications* are implemented: these are *receive-statements* which permit the server to choose to receive one of a number of competing messages, possibly based on the contents of the messages and the server's state.

1.3 Applications of Multicasting

[Cheriton85] states that **group communication** (multicasting) has two generic uses: *query* and *notification*. *Query* refers to a common situation in operating systems, where a number of servers offer a desired service, and a client, wishing to make use of the service, multicasts to the server group: the server(s) appointed to provide the service to that client, or those that are currently available, reply. This application illustrates one advantage of group communications: the client may need to know only the group identity, but not that of the individual servers. This scheme is particularly valuable at boot-time as a new host may use a multicast to appeal for service. The alternative is, either for each client to retain a list of its servers, or obtain the same from a much more extensive name server. *Notification* refers to the situation in distributed programming when one process wishes to inform others of new information, or to control their operation.

Cheriton in [Cheriton85] and Martin in [Martin87] present several specific examples of multicast use. These may be categorised as either inter-process communication, within the operating system, or as parallel programs

1.3.1 Inter-process Communication

Within *V* several server groups exist: kernel servers, file servers, pipe server, time servers and team servers. The team server group uses multicasts to locate under-loaded processors: the multicast specifies a load-level, and only processes with a lesser load need reply [Cheriton85]. In this case the team server makes use of a built-in reply queue, that permits the team server to access replies subsequent to the first, which is returned with the multicast statement.

A second server group application in *V*, is a decentralised name server. The name and request are multicast to the appropriate server group. Those servers who recognise the name respond. It is noted that this type of application requires process groups with unrestricted access, or access based on user privileges. Depending on the circumstances, such applications may require the user to use the first reply only, or any number of replies. *Lighthouse algorithms* [Martin87], in which hosts communicate their view of the network, may also be implemented by multicast sends, with no replies.

1.3.2 Parallel Programs

Cheriton and Zwaenepoel [Cheriton85] describe the programming of a distributed game. Multicasts are used to update local game managers as to the global state. A second application noted is that parallel programs, such as a one for playing checkers, may employ a group of processes performing a parallel search. Multicasting may be used to pass information, *reducing and focusing the search effort*. A similar application is a concurrently executing rule-based system, with the resolution of subgoals being exchanged.

Cheriton and Stumm [Cheriton 87] promote a model of parallel computing using a **multi-satellite star**, a central controlling processor with a number of satellite workers. They argue that certain types of distributed algorithm allow the code to be pre-loaded on the satellite processors, and execution to be controlled by means of relatively short messages. They also state that in a distributed branch-and-bound algorithm, multicasting could be used to update the satellites on the best result so far. They say :

> *Group communication has proven to be useful in terms of efficiency and program simplicity. It is used for control purposes and for data transfer.*

A further application is the implemention of distributed two-phase commit protocols for atomic transactions. The initiator sends a *prepare-to-commit* message, to which all members of the process group reply with *yes* or *no*. As the initiator considers the replies, and at some point sends a *commit* or *abort* message to all group members. The initiator may use multicasts for each phase: the first must be reliable, and have a reply queue, as all group members must commence to execute the protocol; the second need not be reliable, as a receiver failing to get a final message may time-out and request a retransmission [Cheriton85]. Specific examples of a data-base update algorithm that may be performed using a multicast with replies are the *Gemini Voting Algorithm* [Burkhard87], which requires a quorum to vote in favour of committing (see section 2.3), and the *available copies* scheme [Bernstein83].

1.4 Process Groups : Static or Dynamic

As Ahamad and Bernstein [Ahamad85] note, a (multicast) process group may be **static**, fixed before execution begins, or **dynamic**, with processes joining or leaving the process group while the program executes. They also observe that multicasts may be one-to-many, with one unique sender and many receivers, or many-to-many, with several senders. In this latter case, processes may both send and receive multicasts.

The one-to-many multicast group is simpler to implement than the many-to-many, since the requirement of some degree of reliability necessitates each host knowing at

least the size of the process group, and possibly the membership, in order to collect acknowledgements and, if necessary, time-out and retransmit. Navaratnam employs a single group manager to maintain the membership list, and secondary managers on all member sites serve as backup [Navaratnam88]. With one-to-many multicasting, the group membership need be stored only on the unique sender's processor: in the case of dynamic groups, changes need be made only on that processor.

1.5 Multicasting Semantics.

In section 1.2 we discussed synchronisation issues as they relate to unicasts. It should be noted that a unicast returning a value is, of necessity, synchronous, since the statement may not return until the value is received. Since multicasts may return multiple values, the issue here is not so clear. It is clear, however, that provision must be made for a reply queue, which the user may access by some structured means.

Since reliability is a concern with most multicasts, we must consider how this may be provided. It should be noted that any scheme that allows the multicast statement to terminate without assurance that all implied actions have completed, is implicitly prepared to ignore remote and communication errors: if these are significant, error detection and handling may be implemented at a higher level. We concur with [Atkins89], in proposing that the sender assume responsibility for determining the level of reliability required.

In the case of unicasts, the operating system implements protocols to ensure that a transmission is received and acknowledged, or that the user is informed of the failure to do so. There are two approaches to this: providing a communication statement with provisions for exception handling, or having the communication statement return a boolean, indicating success or failure. *SR* takes the first approach; *V* takes the latter.

Reliability is commonly implemented by requiring the receiver to acknowledge the receipt of a message. If the sender fails to receive an *ACK* within some time interval, it

must resend the message. Sequence numbers on messages prevent duplicates being mistaken for new messages. Navaratnam extends this technique to provide for the reliable delivery of multicasts [Navaratnam88]. A sender multicasts and then collects *ACKs* from the process group members. Since each sender knows the composition of the process group, both the size and individual members, it knows the number of *ACKs* it should receive, and their source. If the *ACKs* fail to arrive within a fixed time interval, the sender transmits individual unicast duplicates of the message to each member that failed to acknowledge. It should be noted that a re-multicast would have served the same purpose: it requires all receivers to process the duplicate message, but would avoid each sender having to know the identity of the individual group members, as opposed to merely their number. Reliable multicasting is *hard to support, unless the number and identity of group members is known* [Navaratnam88].

An issue closely related to synchronisation and reliability is the **early termination** of a multicast: the greater the desired reliability or synchronisation constraint, the later a multicast must terminate. This might mislead one to consider that early termination is subsumed by the other issues. However, early termination is, per se, a means to achieving greater efficiency: a multicast should terminate as soon as sufficient replies have arrived. This sufficiency may be determined by the *number* of replies, in which case early termination and reliability may become synonymous, or by the *content* of the replies, in which case the two issues diverge. This latter situation occurs when a client wishes to access servers with sufficient capacity to perform a given task: when respondents have reported sufficient capacity, the client has no use for additional replies, and the multicast may terminate. A particular case of this is when the client requires one server and will accept the first offer of service.

1.6 Multicasting in SR

This thesis concerns the use of multicasting within a high level language, *SR*, **Syn-**

chronised Resources [Andrews87], [Andrews 88], [Olsson86]. Our choice of *SR* was prompted by a number of considerations. Previous discussions and implementations of multicasting have been in the context of low-level languages, employing system calls; our desire was to explore multicasting as an integral feature of a high level language. The specific choice of *SR* was made because of the elegant way in which simple, yet powerful, communication primitives are integrated into the language. As *SR* is *operation oriented*, and encapsulates data and code using *resources*, the integration of multicasting primitives is particularly challenging. Issues include:

- what semantics are required
- how the required semantics may be incorporated cleanly into the language
- how to implement the scheme
- how efficient is the proposed scheme.

1.7 Remainder of Thesis

Chapter 2 discusses in detail the related work. Chapter 3 gives an introduction to the *SR* language, with particular emphasis on the semantics of the communication primitives. Chapter 4 presents our proposal for introducing multicasting within *SR*, using a new pseudo-resource, the Multicast Network (**MCN**), which represents the group of receivers. Multicasting is discussed with reference to message passing and remote procedure call paradigms, which suggest different syntax and semantics. We also discuss the use of the **collector**, a structured reply queue. Chapter 5 deals with the design issues of the proposed scheme, within the context of the current implementation of *SR*, using *UNIX*. We also describe the implementation of a prototype, and the gains in efficiency that a multicast or a pseudo-parallel *co-statement* may give. Chapter 6 gives the conclusions of the work, and points to future research.

2. RELATED WORK

Most of the work on multicasting has concentrated on providing low-level primitives and system calls, either in *UNIX* or *V*. Ahamad and Bernstein[Ahamad85], implemented a new multicasting scheme in *UNIX*, creating a new type of socket: the effect is to allow a number of receivers to bind to one socket, and each receive a multicast. The service is unreliable, being based on the unreliable datagram service provided in *Sun's* 4.2BSd *UNIX* operating system [Leffler83].

[Cheriton85], [Atkins89], and [Navaratnam88] deal with communications in the *V* operating system, a distributed message-based operating system, running on Sun workstations connected by an Ethernet. As part of the messaging scheme they introduce what they term *group communications* (*multicasting*). Dynamic process groups are provided, as it is necessary to maintain groups of like processes, when the message passing scheme makes no lexical distinction between process types. Operations are provided to permit a process to create, join or leave a process group. A sender can send a message to the group, and can receive multiple replies. Cheriton's group built *V* with a semi-reliable group communication primitive (one guaranteeing that one reply or acknowledgement will return), arguing that implementing greater reliability would be too costly.

Navaratnam, Chanson and Neufeld [Navaratnam88] implemented a reliable multicasting scheme on top of the *V*. The protocol provides two levels of reliability, using a centralised control scheme. The system is somewhat tolerant of machine failure and the partitioning of the network.

Atkins, in [Atkins89], provided reliable multicasting within the *V* kernel itself, showing that the cost of reliability is small, contrary to [Cheriton85]. Different degrees of reliability may be chosen by the sender, requiring all or a specific number of replies or acknowledgements. Note that *V* multicasts must be invoked as *C* library calls

invoking *V* system calls. It should be also noted that the fundamental entity in *V* is the *process*, and that the communications are *process-oriented*, as both senders and receivers of a messages are processes.

2.1 Multicasting Operations

Cheriton provides a number of function calls to provide multicasting.

AllocateGroupId() allocates and returns a new group identifier. The process executing this function is automatically a member of the process group.

JoinGroup(*groupId, pid*) makes the process specified by *pid* a member of the group given by *groupId*. The operation **LeaveGroup**(*groupId, pid*) removes the process.

Send(*message, groupId*) sends the contents of *message* to all members of the process group specified by *groupId*. The first reply is inserted into *message*, and the *Send* returns. Subsequent replies may be read using *GetReply* (see later). The same function (*Send*) is used for multicasts and unicasts. Note that the messages employed here are of a fixed, small, size: larger messages may be copied between process address spaces, using different primitives. *Send* blocks until a reply is available, in which case the id. of the responding process is returned, or until the kernel times-out, and zero is returned, indicating failure.

Receive(*message*) blocks the invoking process to receive a message, in *message*. The function returns the process id. of the sender.

Reply(*message, pid*) sends *message* to the process specified by *pid*.

GetReply(*replyMessage*) copies the next reply to a group send into *replyMessage*, returning the process id. of the replying process (zero if the reply queue is

empty). Note that unread and subsequent replies are discarded when the next *Send* is performed by that sender.

Navaratnam employs two new primitives to facilitate reliable multicasting [Navaratnam88]:

ugsend(*msg, group_id, msg_type*)

ogsend(*msg, group_id, msg_type*)

Unlike Cheriton's *Send*, **ugsend** provides reliable delivery (to all group members). In addition, **ogsend** provides *same order delivery*, in cases of many simultaneous senders. Multicasts may be made non blocking by a suitable choice of *msg_type*.

Atkins, Haftevani and Luk modified *V* functions to provide reliable multicasting, within the *V*-kernel in [Atkins89]:

Send(*message, id*), sends *message* to the process group specified by *id*. *Message* contains two fields that specify whether replies or only *ACKs* are required, and whether all, or some number of them, must return for the *Send* to succeed.

GetReply(*message, time_limit*), is intended to take a reply from the sender's reply queue and copy it to the variable *message*, within the time-limit, and return success.

2.2 Reliability

There are a number of potential problems that can make communication unreliable: the communications service provided by the network may be unreliable, work-stations or their network interfaces may fail, and the processes on the work-station may fail.

The modes of reliability described in the literature include *atomicity (reliable delivery)* and *same order delivery. Atomic delivery* implies that either all group members receive a multicast or none. Naturally the first event is preferred, and thus every attempt is made to deliver the message. *Same order delivery* implies that a sequence of multicasts are received by each receiver in the same order.

Navaratnam implement these delivery modes using a two layered system, with an underlying group of managers, one per active host, as shown in Figure 4. Each manager maintains a list of local receivers for each group, and a list of other managers. The managers themselves comprise the process group at the lower level. Inter-host multicasts and unicasts are then used to provide process-to-process communication. The authors are concerned that the multicasting method be *general and not dependent on specific characteristics of the underlying network* [Navaratnam88]. If the network provides broadcasting, each manager will receive multicasts which must then be demultiplexed to the receivers on that host. If the network does not support broadcasting, hosts may communicate via a sequence of unicasts.

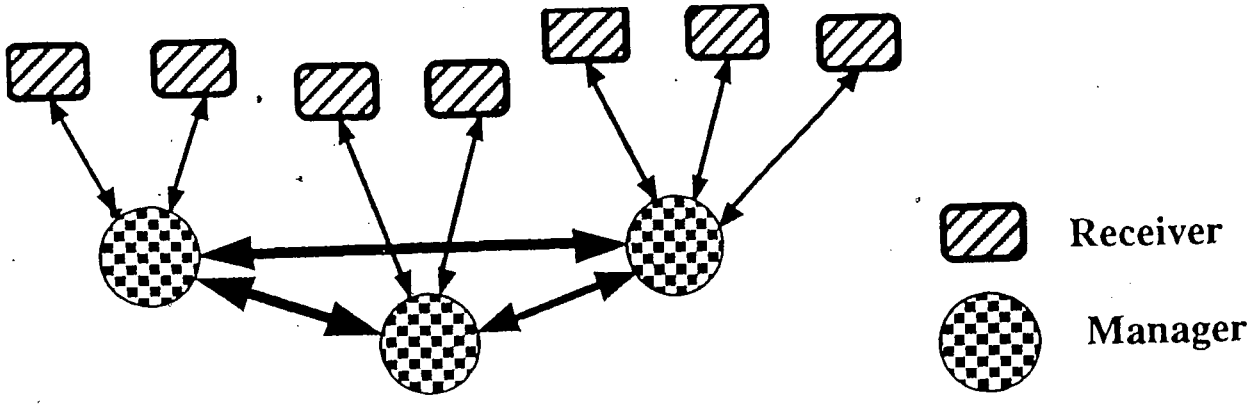

Figure 4 Two-level Multicasting.

In implementing efficient reliable multicast communication in the V-system, Atkins, Haftevani and Luk [Atkins89] provide two sets of semantics for terminating a

multicast: ALL_DELIVER (ALL_REPLY) and K_DELIVER (K_REPLY). Deliver-type multicasts return after the message has been delivered to the receiver, reply-type when the receivers' replies have returned to the sender's queue. The user may specify how many *ACKs* or replies must arrive before the multicast succeeds and terminates. The ALL option specifies reliable delivery (reply) to all the group members extant at the time of the initial send. The K option permits the user to specify the number of *ACKs* (replies) needed, as in Byzantine agreements. Setting the required number to one in the K option provides for the multicast to return after the first *ACK* (reply), the level of reliability that *V* itself provides. It should be noted that in this scheme early termination and reliability are controlled by the same parameters, and are thus synonymous. The values of the parameters may be set to achieve a certain reliability, or to achieve early termination, depending on the application.

Same order delivery requires that each receiver receive a sequence of multicasts in the same order, and is a requirement of replicated data-base systems, when it is necessary to maintain consistency at all sites. This may be implemented by executing a multi-phased protocol [Birman87] , or by using a a central controller, through which multicasts are *funnelled* [Navaratnam88]. This second scheme unfortunately requires an additional unicast, from the sender to the group manager, increasing the time taken for each complete transmission. Navaratnam provides two modes of multicasting, one providing reliable delivery only, and the other providing complete same order delivery. We do not attempt to provide atomicity, due to its cost, and as it may be provided at the user level.

2.3 Parallel Procedure Calls

The *PARPC* scheme [Martin87] gives semantics and syntax for *parallel procedure calls* in a distributed *UNIX* environment, modelled on the remote procedure calls already provided in such environments as the Sun network [Leffler83]. *PARPCs* are

system calls within *C* or *C++*, simultaneously invoking a number of procedure calls. By default, the calling process blocks until a result from one of the calls arrives: it then unblocks and may service the result. The syntax for a *PARPC* is as follows:

```
<PARPC-invocation> ::=
        <PARPC_name> ( <distributed_address_spaces>, <parameter_list> )
        <result-statement>
```

The *distributed_address_spaces* is *the set of distributed address spaces in which the procedure is to be executed.* The *parameter_list* comprises the parameters to be passed to each remote invocation of the basic procedure. The *result-statement* is an optional statement, normally a block of code, permitting the caller to process results as they arrive. Within the result-statement various semantics are possible: the calling process may ignore a reply by executing a *continue* (blocking until another result arrives) or cause early termination of the result-statement by executing a *break.* Data is passed to the *PARPC* and returned by *in* or *out* parameters (equivalent to *SR's* VAL and RES parameters). If the procedures invoked by a *PARPC* have no *out* parameters, the call is asynchronous and non-blocking: any *result-statement* does not execute. A user requiring a synchronous *PARPC* with no replies, must employ a dummy *out* parameter. Replies to a *PARPC* may only be received only by the result-statement, within the scope of the PARPC-invocation.

The syntax and use of the *PARPC* is well illustrated by Martin's example, in which a replicated data-base is updated, after a quorum of hosts have replied [Martin87]. *Ropen* is a *PARPC*, which invokes a set of database servers corresponding to the address space *hl*. *Filename* and *ballot* are parameters which define the transaction to be attempted at each site. The result statement counts replies, and, if a quorum is reached, executes a *break*, to exit the *PARPC*. If a quorum has been reached, a commit order is sent to all the databases, otherwise an abort order. As Martin notes, *commit* and *abort* may be implemented using *PARPCs*. The result statement also processes errors, and

will exit when all members of the remote procedure group have replied, directly or through an error message. Early termination is flexible, as the result-statement may use the replies' contents in judging when to perform a *break*.

```
votes = 0;
ropen(hl,filename,&ballot) {                    /* PARPC          */
        if (host_error(hl)) continue;           /* remote error */
        votes++;                                /* positive vote */
        if (votes > size(hl)/2) break;
        }

if (votes > size(hl)/2) commit(hl);             /* commit or abort in parallel */
else abort(hl);
```

PARPC programs require the programmer to write a header file which describes the *PARPC* interface using type and procedure declarations, and procedure argument specifications. This header file permits the compilation and linkage of user code and special *PARPC* code. *PARPC* programs permit the procedures called to be on different, potentially heterogenous, host machines. The procedures forming the process group are, however, statically determined at compile time.

3. THE SR LANGUAGE

3.1 Overview

Synchronizing Resources, SR, [Andrews88], [Andrews87], [Olsson86], is a high level distributed programming language, intended for both distributed operating systems and distributed applications. *SR* is, in Andrews' taxonomy, an *operation based language* [Andrews83]. Using two communications primitives (*call* and *send*), *SR* provides a variety of synchronisation schemes in a way that is both simple and linguistically consistent with other language constructs.

The basic building block of an *SR* program is the *resource*, an entity that has associated data structures and code, which may be accessed via structured *operations*. Note that *resource* refers to both the lexically defined *resource-pattern* and the dynamically created instance of it, the *resource-instance*. Resource-instances may be created by other resource-instances: the ability to access a given resource-instance, or one of its operations, may be passed from its creator to other resource-instances as a *capability*. A resource may access another resource by invoking an operation on that resource, providing that it holds the capability to either the operation or the entire resource.

3.2 Resources

Since *SR* embodies the philosophy that a resource's external appearance and internal workings should be separated, a resource declaration has a *specification* and a *body*. This facilitates modularity, permitting the design of the user interface of a resource and its compilation as part of other resources, prior to, and separate from, its implementation. The specification defines the interface that the resource has with other resources, declaring its operations and also the resources (patterns) that it *imports*. A resource instance may create or destroy another, or invoke an operation on it, only if it imports the pattern for that resource.

We present the *specs* of two resources, which will serve as examples throughout the text. The first resource, *dbase*, serves a site of a distributed data-base and has operations *vote, commit, and abort*, to provide the semantics required by the Gemini Protocol [Martin87]. These operations have a parameter *transaction*, of a globally defined type, *trans_type*, which specifies the particular data-base transaction required. In addition each site has an operation *report_status*, which is a request for the site to report certain statistics. This is to be done by the site invoking some operation on a central controlling resource, asynchronously, in order to avoid the multicaster blocking. The operation report_status_now is a synchronous form of this operation, returning the status.

```
resource dbase
        op vote(transaction: trans_type)
        op commit(transaction: trans_type)
        op abort(transaction: trans_type)
        op report_status()      {send}  # must be invoked via a send
        op report_status_now(status: status_type)
         .....
end
```

The second resource, *server*, provides a service to clients. The operations *query_load* and *query_capacity* return the current load and capacity of the server. The RES parameter *server_id* permits the multicaster to identify each respondent.

```
resource server
        op query_load(RES: server_id : integer) returns load : real
        op query_capacity( RES server_id : integer) returns capacity : real
         .....
end
```

Resource variables corresponding to these resource-patterns may be declared as follows:

```
var my_server : cap server
var my_dbase : cap dbase
```

The *body* of a resource describes its implementation, which may involve four types of code: *initial, final, procs and processes*, all executing in the same address space. *Initial* and *final* code are executed immediately after the resource is created and immediately before its destruction. *Procs* and *processes* service operation invocations, and perform other tasks.

When an operation serviced by a proc is invoked, a new instance of the proc of that name is created and commences execution. A *process*, on the other hand, begins execution after the resource is created and initialised, and normally continues execution until the resource is terminated. Typically a process has a loop containing *input-statements*, to service one or more operations.

An *SR* program begins with the creation of the main resource instance, which may then create other resources instances, possibly on other machines: these may, in turn create other resources instances. Capabilities to resources, or individual operations, may be passed as a parameters, permitting the holder to invoke the operations of the resource, or the individual operations.

3.3 Operations

SR treats operations as being of the same type if, either the operations are explicitly declared as being of the same *op_type* (as defined in an operation-type declaration), or the operations are implicitly of the same type, having the same number and types of parameters, and the same type of return value. This approach permits one variable to be assigned the capabilities of different, but equivalent, operations with similar semantics (eg. a set of sorting routines). This continues the *SR* philosophy of separating semantics from implementation: operations with the same user interface are considered equivalent from a user perspective.

An operation on a resource may be serviced by the creation of new procs, or by *input-statements* in one or more processes. When an invocation is received by a

resource, the *SR* run-time support (**RTS**) queues the invocation, and checks how the invocation is to be serviced. When an operation serviced by a proc is invoked, a proc instance is created and executes until termination, returning, if required, values to the invoking resource. If the operation is serviced by a process, the situation is more complex: operations waiting to be serviced are queued, as are processes waiting to service them. Operations may be waiting on a *receive-statement*, which services only one specific operation, or be waiting on an *input-statement*, which may service a number of different operations, one-by-one. Processes blocked on input compete to service eligible invocations. Invocations competing to be serviced will be dealt with FIFO, unless *synchronisation* or *scheduling expressions* require otherwise.

3.4 Implementation of SR

The current implementation of *SR* makes use of the *UNIX* operating system. Each physical machine in the distributed system may act as host for one or more virtual machines. A virtual machine (**VM**) is an address space in *SR*, allowing resources on the same *VM* to pass data by pointers. *VMs* are implemented as *UNIX* processes, and execute two sets of code, that of the *RTS* and that of user resources, linked by the *SR* compiler. The *RTS* provides for the creation and destruction of resources and operation invocations, and links the *VMs* via the network.

3.5 Communication Primitives in SR

SR provides two communications primitives, the *call* and the *send*, along with the *co-statement* that allows calls to be carried out in parallel*.

A send is asynchronous: the sender is blocked only until its *VM* receives acknowledgement that the invoking message has been buffered on the operation's *VM*. No

* The use of *sends* within a *co-statement* was not implemented in the version of *SR* used for this research. It has been subsequently.

reply is returned to the sender. A send statement is of the form:

send *my_dbase.report_status()*.

A call is synchronous: the caller is blocked until the invoked operation completes, and replies (returned value and parameters) are delivered to the caller. A call may take place within a call-statement:

call *my_dbase.commit(transn)*

or by using the denotation for the invocation as an expression, whose value will be the value returned by the invocation:

write('Load is : ', *my_server.query_load(id)*).

The parameters of an operation may be of type VAL (value or in), RES (result or out), or VAR (in and out). *SR* does not, per se, restrict the type of parameters that may appear with a call or send. A returning call assigns values to arguments of type RES and modifies the values of those of type VAR. In addition, when the invocation denotation occurs as an expression, the expression is assigned the value returned by the operation. In the case of sends, no value will be assigned to RES parameters, and VAR parameters will not change.

It should be noted that the two types of invocation, call and send, and the two ways in which an operation may be serviced, by proc or by process, permit the user to achieve several of the common forms of synchronisation [Andrews88].

Invocation	Serviced by	Effect
call	proc	procedure call
call	process	rendezvous
send	proc	dynamic process creation
send	process	message passing

The co-statement permits a number of calls to take place in quasi-parallel. The invocations are passed one-by-one to the *RTS*, which transmits them. Replies may be received in any order: when one is received, the *RTS* may call back the *SR* code to

execute the optional post-processing block corresponding to the returning invocation. Early termination may be programmed by executing an *exit*, within the post-processing block.

Invoking two servers within a co-statement is achieved as follows:

```
co
        my_load := my_server.query_load(id)
        your_load := your_server.query_load(id)
oc
```

The purpose of the co-statement is to overlap the times taken to execute the operations contained within it. The actions performed in response to a co-statement, on the invoker's physical host, on the network and on the physical hosts of the *VMs* of the invoked operations, may occur in parallel. The invoker's *RTS* processes the co-statement, calls the network, processes replies and calls back to the user's post-processing block. The network transmits the invocation and return values. When instances of the same operation are to be invoked on separate machines, further savings might be made by employing a single multicast. This reduces network traffic, and reduces the average time for a message to be buffered on each *VM*.

4 MULTICASTING IN SR

4.1 Overview

The thrust of the research described in this thesis is the design and implementation of a suitable multicasting scheme for the *SR* language. *SR* already provides a certain degree of parallelism, via the *co-statement*. This, however, requires one message per operation invoked, as opposed to only a single multicast message. In addition, the programming of multiple calls within a *co-statement* is not a particularly elegant way of describing parallel invocations.

We refer to the entity that permits multicasting, equivalent to the process group, as a *multicast network* (**MCN**). We argue that an *MCN* be considered, like the *VM*, a *pseudo-resource*, which is, however, distributed over the entire set of *VM*s. Multicasts are to be performed by invoking the *MCN-capability*. We discuss the syntax and semantics of an *MCN*: in particular, we examine and compare the effect of choosing a message passing or remote procedure call paradigm. We also discuss the use of another *pseudo-resource*, the **collector**, which acts as a separate reply queue.

The proposed implementation is similar to that described in [Navaratnam88], in that each *SR RTS* will have multicasting code. In order to implement protocols for changing group composition, we employ a central controller, modifying the central name server, *srx*, for this additional purpose.

A number of criteria were employed in making design decisions:

Power

The system should allow the user to program a range of useful semantics with regard to synchronisation, early termination, reply handling, reliability, and error handling.

Simplicity

The scheme should be linguistically and semantically simple for the user. The number

of new primitives should be minimal. The potential for programmer error should be minimised.

Compatibility

The scheme should be as consistent as possible with existing *SR* semantics, and any extension to the semantics should be minor. Extensions to the language should be consistent with standard *SR*. The scheme should be in the spirit of *SR*.

Efficiency

Changes to the compiler and *RTS* should be minimised and have little effect on performance. The implementation should be as efficient as possible.

4.2 Multicasting within standard SR

It is possible to implement multicasting in standard *SR* by means of a true resource, which holds the capabilities of receivers and invokes them all, when itself invoked by a sender. The multicaster resource has operations *join* and *leave*, which allow a receiver's capability to be given to, or removed from, the resource. Multicasting operations cause the multicaster to invoke its receivers via a series of separate unicast invocations. Further *get_reply* operations are needed in order for all replies to be made available to the user. A multicaster resource for multicasting to receiver resources of type *dbase* (see section 3.2) would have specification:

```
resource mcn
        import dbase
        op join(new_op:cap dbase)
        op leave(old_op: cap dbase)
        op multicast_vote(transaction: trans_type)
        op multicast_commit(transaction: trans_type)
        op multicast_abort(transaction: trans_type)
        op multicast_report_status()
        op get_reply_vote(transaction: trans_type)
        op get_reply_commit(transaction: trans_type)
        op get_reply_abort(transaction: trans_type)
        op get_reply_report_status_now(status: status_type)
end
```

The true-resource multicaster has merit in demonstrating that multicasting can be provided using only standard *SR*, and as a model for the interface of a built-in multicaster, but has major limitations. First, though a single user statement may invoke multiple receivers, the multicast is implemented as a sequence of unicasts: this is marginally less efficient than a series of unicasts performed directly. Second, there is a proliferation of operations, two per operation of the basic operation, if replies are required: a multicast operation and a *get_reply* operation. Lastly, as noted, *SR's* strong typing requires each multicasting resource to be customised to correspond to the type of the receiver, and the type and number of its parameters. This sort of modification should be performed mechanically by a pre-processor or compiler. These limitations provide the justification for a built-in multicasting facility.

The work on multicasting described in chapter 2 referred to process-to-process multicasting, as the languages and systems used were process oriented. *SR* is an operation-oriented language [Andrews83]: its unit of encapsulation is the *resource*, which sends messages by *invocations* and receives messages via *operations*: thus, multicasts will be invocations by the sending resource of operations on the receiving resources, as with the true-resource multicaster. While, in the work cited, process groups were groups of peers, there is a basic asymmetry with *SR* multicasts: the receivers for a multicast are a set of operations, the senders a set of invocations. There is no requirement that the parent resources of the senders correspond to the parent resources of the receivers.

The multicast, being a single operation with one set of parameters, must invoke operations whose formal parameters correspond to the multicast's parameters in both number and type. This does not imply that the operations invoked need be instances of the same operation, merely operations of the same type.

An *MCN* may comprise a set of operations, or a set of resources. With a *resource-based MCN* the operation required in a given multicast must be specified in

the multicast-statement. The choice of operations as the multicast group members is the most general one; however, in the case of file access, a separate *MCN* would be required for each file operation (*read, write, open*, etc.), causing an excess of user-code to create, and maintain the *MCN*s, and potentially a greater burden on the *RTS*: here the choice of an entire resource as the multicast group member is more convenient. In addition, as will be discussed later (section 4.4.4), the resource-based *MCN* is necessary to implement *same order delivery* for a set of operations, such as read and write. Unfortunately the *resource-based MCN* precludes the invocation of operations of the same type, on resources of different types. Thus both *resource-based* and *operation-based MCN*s should be provided. Note that this parallels the existence in *SR* of both resource and operation capabilities.

4.3 Group Composition : Dynamic or Static

V provides for dynamic group creation, and for dynamic changes to group membership; with *PARPC*, however, group composition is statically determined at compile-time, raising the question as to whether dynamic groups are necessary. With most distributed user programs, the required processing power will be known at compile-time, and statically defined groups will suffice. However, at the OS level, or with ongoing systems such as distributed data-bases, the processing power may be required to change dynamically. A server group, such as the servers of the sites of a distributed data-base, may change as new sites are added, and old ones taken out of service: sites may also *go-down*, and require servicing and *bringing-up*. One option is to stop all transactions, make the change, and then restart the system.: this may be satisfactory, if the changes are infrequent, and system down-time can be tolerated. In other circumstances, the preferred approach would be to create and initialise the site, and then add the server to the group dynamically, with minimal interruption to the service.

Distributed algorithms may also require dynamic groups: as the job progresses, it may prove advantageous to redistribute the work-load, creating new processes to perform certain tasks and reducing the number performing others. In this context, dynamic changes to the group composition are essential, if time savings are to be achieved.

4.4 Required Semantics of Multicast Statement

The literature [Atkins89] [Martin87] suggests a number of semantic features that should be incorporated into a multicast statement: synchronisation, early termination, reply-handling, reliability, error-handling, and same order multicasting.

4.4.1 Synchronisation and Early Termination

Multicasts may be asynchronous or synchronous. Synchronous multicasts are required for applications in which results are required, or where the user needs to be assured that a set of operations have completed. A synchronous multicast will thus provide reliability similar to that provided by the synchronous unicast (call), guaranteeing that all, or some number of operations have completed.

In order for the asynchronous multicast to provide the same reliability as the asynchronous unicast (send), its success must guarantee that the message has been buffered on all *VMs*, or some number of them.

As noted in chapter 1, there are algorithms that permit the multicast to terminate early, and thus be more efficient. The examples given in section 1.5 are cases in which the user is satisfied when the replies returned so far satisfy some property: this may be a function of the number of replies, or depend on the values returned. Early termination for synchronous multicasts should permit these semantics, and those required when no data is returned and the user needs to be assured that some number of tasks have been completed, as in some data-base applications. Note that if no values are required, the termination semantics can only depend on the **number** of replies.

Early termination of asynchronous multicasts appears to be of a lesser importance. A potential application is when a user requires assurance that data has been buffered on at least one site, before continuing and erasing the local copy. However, such a scheme might prove inadequate if the buffering is in volatile storage. It should be noted that setting the number of required *ACKs* to zero would specify totally unreliable delivery, which may be permissible in some cases, with error checking at a higher level. We discuss the provision of early termination for asynchronous multicasts, but do not regard them as a crucial feature.

4.4.2 Reply Handling

Multicasts require replies in some applications, and not in others. Atkins permits this choice, as does Martin (via the mechanism of in and out parameters) in the *PARPC*: in both cases, the absence of replies is taken to imply asynchrony. In *PARPC*, the absence of *out* parameters makes the *PARPC* asynchronous, and a user requiring no replies, but wishing to be assured that a process has terminated, must employ a dummy *out* parameter. This mechanism is inelegant and should be avoided, if possible, in a high-level language.

4.4.3 Reliability and Error Handling

Reliability and error handling are opposite sides of the same coin. The *kind* and *degree* of reliability required depend on the application. There are three possible *kinds*: the scheme could be totally unreliable, with the message being sent, but nothing more, *deliver-reliable*, with the sender being assured that the message has been buffered on the remote machine(s), or *reply-reliable*, with the sender being assured of a reply, or replies. The *degree* of reliability applies to multicasting, when the number of *ACKs* or replies required may vary.

It is important to distinguish between the different types of failure. Atkins provides for a multicast to return success or failure, what may be termed *semantic* success or failure: the multicast-statement itself does not fail, in the sense of generating an exception, it merely returns a process id. of zero. A *PARPC* does not fail either: exceptions may be handled within the result-statement. *SR's* unicast primitives do not return success or failure: errors provoke exceptions, causing run-time errors, unless the user has provided a *handler*, a block of error-handling code associated with the multicast statement. It should be noted that returning success or failure, is consistent with standard C programming practice; in a high level language, this practice is less common.

The range of reliability is specified by Atkins, by means of a parameter which specifies the *kind* (ALL_REPLY, K_REPLY, ALL_DELIVER, K_DELIVER) and, in the case of the K-option, an integer expression giving the required number of *ACKs* or replies, the *degree*. It should be noted that this reliability could have been provided using Cheriton's earlier unreliable scheme, using higher level code. Atkins's scheme has the advantage of being more efficient, as it performs the checks within the *V*-kernel. It appears that the justification for specifying reliability via parameters is based on two grounds, efficiency and simpler code.

Martin employs a more flexible scheme, permitting the *PARPC* user to have explicit access to both genuine replies and exceptions, which have separate entry points within the result-statement. This is, however, at the potential cost of extra processing time, as this functionality is provided by the user code, rather than at a lower level.

SR intends *send* and *call* to be deliver-reliable and reply-reliable, respectively: by default, an exception provokes a run-time error. Rendering errors benign requires the use of a *handler*. If we meld *SR* multicasts with the parametric approach to specifying reliability, we are faced with a greater potential for exceptions, due to generally less reliable lower level communications, and to the greater number of hosts and receivers that may fail. It should be noted that deliver-reliability guards against communication

failure, including the possibility of a receiver being unable to buffer a message. Reply-reliability guards against both communication failure and remote exceptions. However, it should be noted that k-reply-reliability does not imply (k+1)-deliver-reliability, and a scheme providing k-reply-reliability must be prepared to deal with communication errors.

In distributed systems, errors are often passive, discovered by the absence of action, rather than by action. An error checking scheme inevitably requires a local timer: if no response occurs within a time-limit, an error is declared. This time-limit must be set within the operating system, after a process of fine tuning. This is not applicable to calls, as the remote action may take an indefinite time to complete.

The present implementation of *SR* is not designed to withstand processor, communications, or process failure. The failure of the name server process, *srx*, would eventually be fatal, as its information is not replicated. However, the cost of building in the necessary redundancy is substantial [Birman87], and beyond the scope of this experimental language. In addition, such reliability comes at the cost of slower processing.

4.4.4 Same Order Multicasting

Same order multicasting with operation-based *MCNs*, does not provide the necessary functionality for replicated data-base applications: *reads* and *writes* would have separate *MCNs*, and ordering on each *MCN* would not imply that each site saw the same ordering of *reads* and *writes* taken together. This problem is solved by resource-based *MCN*s, on which all operations are ordered, and is a potent argument for their implementation. However, a distributed data-base may permit other operations, of a diagnostic nature, which need not be part of the ordering. This example suggests the need for resource-based MCNs, in which all operations, subsets of operations, or individual operations would be ordered. However, specifying all possible ordering semantics would be confusing.

Same order multicasting could be a static attribute of a given receiver group, a boolean to be dynamically toggled, or a property of individual multicast statements, requiring the programmer to correctly specify the nature of each multicast. Specifying same order multicasting in an *MCN* declaration has several advantages. Firstly, as noted, this provides the functionality required to update replicated data-bases. Secondly, by defining this attribute statically, the programmer is relieved of responsibility for correctly specifying the nature of each multicast, with the side effect of slightly simplifying the user code: as will be described shortly, the semantics of same order multicasts, are, in some cases, far from simple. The other options for specifying same order multicasting appear less useful, unless the algorithm being implemented has different phases, requiring ordering and non-ordering multicasting at different times.

The effect of same order multicasts depends on the type of multicast, and on how the operations are implemented. Same order synchronous multicasts guarantee that all the operations invoked will terminate before the next multicast, of whatever type, is started. Same order asynchronous multicasts guarantee only that the invocations will have been buffered before the next ordered multicast is allowed to take place. When an operation is implemented by a process, and in the absence of *scheduling* or *synchronisation expressions*, FIFO will prevail and same order asynchronous multicasting will imply that each receiving process deals with the stream of multicasts in the same order. If an operation is implemented by a proc, same order asynchronous multicasting only implies that the procs are created in the order in which the multicasts are sent. Even though time-slicing is not presently implemented in *SR*, the first proc to start may not finish first, as it may block, permitting the second multicast's invocation, with different parameters, to execute and possibly terminate first.

As our scheme requires a central controller to facilitate *VM-to-VM* multicasting, we propose that *same order delivery*, be implemented using this same controller, as with the scheme of Navaratnam et al. [Navaratnam88].

4.5 Multicast Group Semantics

Access to *SR* resources is controlled by means of capabilities: an operation may be invoked only by a resource holding its capability or that of its parent resource. To explicitly maintain these semantics with dynamic process groups would be impractical, as each potential sending resource would be required to hold the capability to each receiving operation. What is required is a way of controlling process group membership, for both sending resources and receiving operations, that maintains the spirit of *SR*.

The true-resource multicaster of section 4.2 provides a model for the semantics of a built-in multicaster. The right to multicast is restricted to those holding the multicaster capability, consistent with *SR* practice. We therefore propose that the right to multicast to a given process group be controlled in this same manner. The resource creating the group would obtain the capability, and could pass the same to other resources. The **multicast network (MCN)** is the entity which has this capability.

Controlling the receivers of a *MCN* requires a different approach. Again the true-resource multicaster provides a model, in that a resource holding the capability of an *MCN* and that of a potential receiver may pass the receiver capability to the multicaster resource, thus adding it to the group of receivers. Though this approach is suitable for a built-in *MCN*, some clarification of the semantics is required.

This raises an interesting point: resource *A* creates a multicast group, passes the multicast capability to resource *B*, and joins an operation *O* to the group. Resource *B*, which does not possess the capability of *O*, may now multicast to the group and invoke *O*. This appears to violate *SR* semantics in that a resource can invoke an operation whose capability it does not possess. This violation is, in fact, apparent only: a resource may not directly invoke an operation whose capability it does not possess, but may do so via an intermediary: the ability to invoke is transitive. In our case the multicast group acts as intermediary.

We suggest that the violation of *SR* semantics depicted above, is apparent only, as the semantics proposed do not differ from those that the true-resource multicaster provides within standard *SR*. The true-resource multicaster provides this functionality because the ability to invoke is transitive. (*A* may *indirectly* invoke *C*, if *A* has the capability of *B*, and *B* that of *C*.) If we consider the *MCN* a *pseudo-resource*, the semantics of standard *SR* will be maintained. We thus propose a minor clarification of *SR* semantics:

that an MCN be a pseudo-resource, and that the right to multicast to the receivers of an MCN be regulated by the sender being required to hold the capability to the MCN, initially owned by the group creator,

and

that a new receiver may be added to the MCN by any resource that holds both the MCN and the receiver capabilities.

4.6 Maintaining a Built-in Multicaster

We now describe the features of a built-in *pseudo-resource MCN*, its semantics and the changes required to the *SR* language for its implementation. In describing the ways in which a *MCN* may be accessed, each language addition will be referenced to the appropriate section of the Revised Report on the *SR* Programming Language[Andrews87], which we refer to as **RR**.

As a pseudo-resource, an *MCN-instance* may be declared, created, or destroyed, like a resource. To distinguish between a declaration of resource or operation, and that of an *MCN*, we use a new keyword **net** in place of **cap** in **RR** (3.1):

```
<capability_definition> ::=
        <captype> <resource_or_operation_or_optype_identifier> |
        <captype> <component_identifier.operation_or_optype_identifier> |
        <captype> <operation_specification>
```

<captype> ::= **cap** | **net**

This permits operation-based and resource-based *MCN*-declarations of the type:

```
var loads : net servers.query_load    # an operation-based MCN **
var commits : net dbase.commit    # an operation based MCN
var dbases :  net dbase          # a resource-based MCN
var servers :  net server        # a resource-based MCN
```

When creating or destroying an *MCN*, we use standard syntax, with the word *net* being the name of the object created or destroyed. This *generic* name avoids the complexities of specifying the type of each *MCN*, which is apparent from that of the variable. There may be no initialisation values nor a host *VM*. Thus:

```
dbases := create net()
destroy dbases
```

Changes to a multicast group may be made by means of *join* and *leave* statements, which must contain the *capabilities* of both the receiver *operation* or *resource*, and of the *MCN*. We use the following code to join/remove *my_dbase* to/from the *MCN dbases*:

```
join dbases(my_dbase)
leave dbases(my_dbase)
```

A new statement, **mc_update_statement**, must be added to the list of resource_control_statements in RR (6.3):

```
<resource_control_statement> ::=
        <create_statement> | <destroy_statement> | <mcn_update_statement>

<mcn_update_statement> ::= <join_statement> | <leave_statement>

<join_statement> ::=
        join <mcn_identifier> ( <resource_or_operation_identifier> )

<leave_statement> ::=
        leave <mcn_identifier> ( <resource_or_operation_identifier> )
```

An operation or resource may be joined to a *MCN* as long as their types agree.

** In *SR* # indicates a comment.

Atkins and Martin provide two paradigms for the semantics and syntax of multicasting: the **message passing (MP)** and **remote procedure call (RPC)** paradigms [Atkins89][Martin87]. It is worth describing the essential differences between them. In the *MP* paradigm, separate primitives are used to perform multicasts and to access replies. This provides great flexibility, but implies that the scope of the entire multicast activity is not lexically defined. In the *RPC* paradigm, one primitive is used for the entire multicast activity, including result processing. The scope of the multicast is well defined, resulting in an integrated approach. We examine how these may be used to describe multicasting in *SR*, and discuss their relative merits.

4.7 SR Multicasting : Message Passing Paradigm

In this paradigm, based on [Atkins89], reliability and early termination are specified by an optional integer expression. The ALL_DELIVER option provides the same semantics as the *SR* send, in that all receivers must acknowledge the receipt of the message before the statement terminates: ALL_REPLY is similar to *SR's* call, in that all receivers must reply. It should be noted that the semantics required for multicasts correspond to those for unicasts, suggesting that, as in *V*, the same two primitives (in this case, send and call) be used to specify asynchronous and synchronous multicasts. A send or call statement will be a unicast or multicast depending on whether the associated denotation contains an operation or a multicast_operation.

We have made two changes to the syntax, to accommodate the optional termination semantics and to provide the *get_reply* functionality. We have added the optional termination semantics to the multicast denotation, rather than requiring new call and send statements. This change also takes into account the use of denotations as expressions. The following invocation terminates after the first reply has been received:

call *dbases.report_status_now(status)* **return_after** 1

providing an optimisation when the first reply will suffice.

The inclusion of the *get_reply* primitive presents some problems. In the absence of return-values, we could provide a *get_reply_statement* as follows:

get_reply *servers.query_load(load,id)*

where the operation query_load would have been redefined so that the load is returned as a parameter, not as a return-value:

op *query_load(RES load: integer, RES server_id : integer)*

If, however, return-values are required, the problem is how to specify the *get_reply* functionality in a denotation, along with the multicast operation, which identifies the reply-queue being accessed. *Get_reply* is effectively an operation on the reply-queue, which is implicitly specified by naming the multicast operation. One approach is to explicitly define a reply-queue for each multicast: this is discussed in section 4.9. If return-values are required, and there is no explicit reply-queue, there appears to be no satisfactory way to incorporate *get_reply* into the language. The option we have chosen is to allow *get_reply* to be added as a prefix to a denotation, which would otherwise represent a multicast. The weak justification for this is that *get_reply* appears where *call* would appear, were it not omitted when a call appears as an expression. Thus:

load := **get_reply** *servers.query_load(id)*

Modifying **RR** 7.3 to incorporate *get_reply* and termination semantics:

<denotation> ::=
[**get_reply**] <object_identifier> ([<argument_list>])
[<termination_semantics>]

<termination_semantics> ::= **return_after** <integer_expression>

It is noted that only one of the termination semantics and *get_reply* may be used, and only if the denotation otherwise represents a multicast. The use of *get_reply* is so awkward that it is perhaps better to specify that multicasts should have no return-values, and that RES parameters be used in their place.

4.7.1 Multicast Send

Asynchronous multicasting is indicated by *send*:

send *commits(trans)*
send *dbases.report_status()*

The denotation for a *multicast_operation* contains either the capability of a operation-based *MCN*, or that of a resource-based *MCN* qualified by a choice of operation.

By default, a multicast-send returns when all *VMs* have acknowledged receipt of the message, providing asynchronous message passing. The optional termination semantics provide for early termination, after a specified number of *ACKs* have been received. Following the approach taken with unicasts, we permit any type of parameter (RES, VAR or VAL), but of necessity, the VAR parameters will be unchanged and RES parameters will not have been assigned values. However, compiler warnings are warranted.

4.7.2 Multicast Call

Synchronous multicasts may appear in two types of statement, as is the case with unicasts. Apart from the call-statement, a synchronous multicast may be performed when the appropriate denotation is employed as an expression. The two cases are illustrated by the following examples:

call *dbases.vote(transaction)*

load := *servers.query_load(id)*

The synchronous multicast emulates the synchronous unicast in guaranteeing that the invoked operations have terminated, before it terminates. In some cases early termination may be desired, such as when the user requires a guarantee of a minimum number of successful invocations (as when working with a replicated data-base, or with Byzantine agreements): in these cases the optional termination semantics may be used.

Parameters of all types (VAL, VAR and RES) are permitted, and have the same semantics as for the unicast call, with the stipulation that the new values of RES and VAR parameters will be those derived from the first reply.

It is necessary to consider the pathological case when the user specifies in the optional semantics a value exceeding the size of the process group, the multicaster being unaware of the size of the group. The options appear to be either to have the multicast fail, or set the number of replies required to *ALL*.

4.7.3 Get_reply

In *V*, *get_reply* only blocks for a specified period: it then times-out and returns failure. This approach may be satisfactory in a message passing environment, where replies can be expected within a reasonable, known, time. In *SR*, however, the time for an operation to complete may be large and is not likely to be known. Thus *get_reply* must be blocking, which may cause a deadlock, unless the user can be assured that a reply is on the queue, or will arrive. To make proper use of *get_reply*, the user needs to know of the size of the group, which may be volatile. Without this knowledge, maintained at a higher level, there appears no simple way to permit the user to consume all replies. A primitive allowing the user to determine the size of the receiver group would return a potentially erroneous value. (This problem does not exist if we model our syntax on *PARPC*, as we show in section 4.8.) The reply_queue is automatically emptied when a new multicast-statement in the same process invokes the same *MCN*.

We propose that the replies to a call be consumed only by the process which made the call. This is the functionality provided by others, and that required by distributed algorithms. This implies that there must be a separate reply-queue for each *MCN* operation and process, rather than one per resource or *VM*. Such queues need not be created until a multicast occurs, but may not be removed until it is certain that no further use can be made of them, when all replies have been consumed or the next mul-

ticast occurs. The simplest approach is to create a queue when required and never remove it, unless the *MCN* is destroyed. The only maintenance required is to empty the queue when a new synchronous multicast occurs for the corresponding *MCN* operation. If it is required that multiple processes or resources consume the replies, the functionality may be provided using a *collector* (section 4.10).

As the first and subsequent calls are returned differently, the code for processing replies must be duplicated, leading to more complex code. To determine the total load on the group *servers*, the following code is required:

```
total_load := 0
total_load := total_load + servers.query_load(id)  # load from first reply
fa i:= 1 to reply_total - 1 ->
        total_load := total_load + get_reply servers.query_load(id)  # load from
        subsequent reply
af
write('Total load :', total_load)
```

A solution to this asymmetry is to make *call* non-value-returning, and to have all replies returned to the reply queue.

```
total_load := 0
call servers.query_load(id)
fa i:= 1 to reply_total  ->
        total_load := total_load + get_reply servers.query_load(id)
af
write('Total load :', total_load)
```

This simplifies code when multiple replies are to be used, but leads to more complicated code when a single reply is required, as in a request for service in which the first server responding is chosen.

4.7.4 Error-handling

Both synchronous and asynchronous multicasts are provided, and each may terminate early, after a specified number of replies or *ACKs* have returned. It should be noted that early termination of the multicast statement may not depend on the contents of the replies; the user may, however, chose how many replies to process. This means

that the user many not commence reply processing until the replies specified by the termination semantics have returned.

Error handling is problematical in this paradigm, as the high level syntax provides no simple way of returning success or failure for a multicast. If the required semantics fail, then the statement must fail, and either a run-time error occur, or a handler be invoked. With unicasts the handler can be attached to the multicast statement, and will deal with any exception detected for the single operation invoked. With multicasts requiring all *ACKs* or replies, the multicast statement will be current when an exception returns, and a handler attached to the statement may be invoked.

If early termination is used, the handling of late arriving exceptions is difficult, as the executing statement will be without the scope of the multicast. The problem is that the message passing paradigm does not permit the lexical determination of the scope of a multicast plus reply-handling: with this paradigm there is no semantically and linguistically simple way to specify error handling in all cases. It thus appears that the *MP* paradigm does not admit the handling of remote exceptions arising from a multicast.

It is our view that the early termination semantics have the intention of permitting exceptions to occur, as long as the required number of replies/*ACKs* return. Thus we suggest that the handler should be concerned with the failure of the multicast, but not explicitly with those of its receivers.

4.8 SR Multicasting: Remote Procedure Call Paradigm

PARPC [Martin87] suggests a syntax in which a reply-processing-block is associated with a multicast, to process replies and deal with exceptions. Modelled on other *SR* statements with associated blocks of code, such as the do- or co-statements, we use a keyword (**mc**) to begin a multicast statement and its reverse (**cm**) to terminate it. To accommodate this a new multicast statement must be added to the grammar in **RR**,

section 6.1:

<invocation_statement> ::=
<call_statement> | <send_statement> | <multicast_statement>

<multicast_statement> ::= **mc** <denotation> [-> <reply_processing_block>] **cm**

<reply_processing_block> ::= <block>

The multicast statement implicitly contains a loop, in that the optional reply_processing_block is executed, by default, once for each reply. With this syntax a whole range of semantics may be specified: early termination, synchronous or asynchronous multicasts, reply-handling and error-handling.

It is necessary to specify how the parameters of the multicast should behave, and in particular their scope. VAL parameters are expressions which may include variables accessible within the scope of the block which contains the multicast: VAR parameters must be such variables. RES parameters and the return-value (if any) may be variables local to the scope of reply_processing_block, or variables with wider scope. The values returned by the multicast, those of the VAR and RES parameters, must be accessible within the reply_processing_block. While RES parameters could be specified to be local to the reply-processing-block, this is not feasible with VAR parameters. We thus specify that all variables used as actual parameters for a multicast must be declared lexically prior to the multicast statement. It should be noted that this syntax precludes the use of return-values, a minor inconvenience.

Synchronous Multicasts with replies

In order to employ *RPC* multicasts, the specification of the resource *server* must be changed to use arguments rather than return-values. We also add the server id. as a RES argument, to allow the multicaster to identify the source of each reply. Thus:

op *query_load*(RES *load* :real; RES *id* : integer)
op *query_capacity*(RES *cpty* : real; RES *id* : integer)

The load-querying example of section 4.7.4, may be written as follows:

```
total_load := 0
mc servers.query_load(load,id) ->
        total_load := total_load + load
cm
write('Total load is :', total_load)
```

Synchronous Multicasts with no replies:

Our example is the second phase of a Gemini Protocol, using synchronous multicasts, in which all sites are to commit, before the statement terminates.

```
mc voters.commit(transaction) -> skip cm
```

The skip_statement is a null statement, which will be executed until all operations have acknowledged.

Synchronous Multicasts with Early Termination:

Our example is that of a client wishing to identify servers whose total capacity is sufficient to handle the client's task.

```
# initialise required_capacity to capacity needed for task.
mc servers.query_capacity(capacity,id) ->
        required_capacity := required_capacity - capacity
           # instruct server (identified by id) to undertake part of task
        if not (required_capacity > 0) -> exit fi      # all work apportioned
cm
```

The *SR exit* forces termination of the smallest unclosing iterative statement, which is, in this case, the implicitly iterative reply_processing_block.

Asynchronous Multicasts:

Our example is the second phase of the Gemini protocol, using an asynchronous multicast: failure to receive a commit or abort will be detected by the receiver at a higher level, via time-out, and a retransmission requested.

```
mc voters.commit(transaction) cm
```

The absence of a post-processing block is taken to indicate asynchrony: the statement terminates after all *VMs* acknowledge.

4.8.1 Error-handling

The *RPC* paradigm provides asynchronous and synchronous multicasts, with flexible early termination semantics, which may depend on the nature of the replies. This flexibility permits the user to process both genuine replies and exceptions: the handler may deal with individual exceptions, and not merely the failure of the entire statement, as in the *MP* paradigm. Error handling may be achieved by providing two separate blocks of code: a reply-processing-block and an exception handler.

Our scheme does not provide early termination for asynchronous multicasts. As noted earlier (4.4.1) this functionality does not appear essential; if it does prove necessary, a simple addition to the syntax could be made.

As the authors of *SR* have not specified how to represent *handlers* we feel free to choose the most appealing option. The same syntax will apply to both synchronous and asynchronous multicasts. Our choice is to use *mc_handler* followed by a block of code, thus:

<multicast_statement> ::= **mc** <denotation> [-> <reply_processing_statement>]
[mc_handler -> <exception-handling-statement>]
cm

The exception-handling-statement is a block, but with special semantics that will permit it to determine which group member(s) provoked the exception, and its nature.

To illustrate error-handling, we use Martin's example of the Gemini protocol (section 2.3):

```
votesfor:=0
mc dbases.vote(transaction) ->
        votesfor++
        if (votesfor > quorum) -> exit fi

mc_handler -> # error-processing-block

cm          # end of multicast statement

if (votesfor > quorum) -> mc dbases.commit(transaction) cm
[] else mc dbases.abort(transaction) cm
fi
```

4.9 Conclusions with regard to the two paradigms

A comparison of the two paradigms reveals that the remote procedure call paradigm and syntax is the more appropriate. This may be seen from examining how the two models compare with regard to the functionality specified in section 4.4 and the criteria given in section 4.1. It should be noted that generally either scheme can be made to provide the same range of functionality. The reasons for preferring one paradigm over the other relate to the relative simplicity and *naturalness* of the paradigm, both semantically and syntactically, and its ease of use.

Synchronisation

Both paradigms permit synchronous and asynchronous multicasts.

Early Termination

Both paradigms permit early termination, but the *RPC* paradigm is more flexible as the termination semantics are programmed by the user and may make use of the values contained in the replies. This is at the expense of using higher level code.

Reply Handling

Both schemes permit reply handling, but again the *RPC* is cleaner, semantically and syntactically. The syntax for *get_reply* in the *MP* paradigm is very awkward; however, it should be noted that if return values are prohibited, as in the *PARPC* paradigm, this awkwardness is reduced. Even so, there remains the problem of the blocking nature of

get_reply and the volatile nature of the receiver group.

Reliability and Error handling

Here the *RPC* paradigm is clearly more powerful, as the linguistic encapsulation of reply-processing facilitates the handling of remote exceptions. The *MP* paradigm's error-handling capability is restricted to handling the multicast-statement's failure.

Same Order Multicasting

The two paradigms offer equal advantages in specifying same order multicasting. If this is done lexically, there there is clearly no difference. If it is specified dynamically, a keyword (**ordered**) may be attached to a multicast statement of either type.

In section 4.1, we outlined certain criteria for judging multicasting schemes: we now compare the *MP* and *RPC* paradigms with respect to these criteria. From the above discussion we can conclude that neither scheme is intrinsically more powerful, in an absolute sense, in that either type of syntax can be enlarged to provide semantic equivalence to the bare form of the other. However, if we consider these bare forms, the *RPC* model provides the greater power.

The *RPC* paradigm is also superior with respect to simplicity and ease of use, requiring only one new primitive. In language design there is a tension between flexibility and structure: the more flexible a language, the less structured it is, and hence it has a greater potential for programmer error. We believe that the previous examples show the *RPC* paradigm to be flexible enough to provide all necessary semantics, and yet sufficiently structured to minimise the chance of programmer error. The *MP* paradigm, on the other hand, is highly unstructured and would increase the potential for error. The *RPC* is also more *SR-like* and more compatible with standard *SR*.

As far as efficiency is concerned, there appears to be little to choose between the two (see section 5.4) However, the more flexible error handling in the *RPC* paradigm, is

bought at the price of being potentially less efficient, as it is performed at a higher level.

Overall we conclude that the *RPC* paradigm is superior. This is perhaps not surprising, as its single logical thread of control is more in tune with the semantics of a high-level operation-based language, than is message passing.

4.10 Remote Reply-queues

Both multicasting paradigms permit replies to be returned to the multicaster; the possibility exists, however, that some algorithms may exploit a scheme whereby replies are delivered to other resources or processes, which we term *consumers*. The case for having a remote reply-queue is circumstantial. A model with a central controlling resource, and a number of lesser controllers might employ this functionality by having the central controller multicast, instructing all its servers to send data to a subsidiary controller. Note that the functionality we proposed for call did not allow the replies to a multicast to be consumed by different processes, even within the same resource or *VM*: this ability might be desirable in an multi-processor architecture. A second paradigm that might support this functionality is a version of Cheriton and Stumm's multi-star satellite, in which all the satellites might be instructed by the star to send partial results to a particular satellite.

An argument against providing this functionality via a remote reply-queue is that it may also be provided using the orthodox scheme, with one additional message, from the resource initiating the action to that which is to receive the data, instructing the latter to perform a conventional multicast. The limitations of this approach are the extra code required to support this extra message, and the extra time and network traffic incurred. None of these appear to be major problems. However, we consider it worthwhile to discuss the incorporation of a separate reply queue, which we term a **collector**.

A collector has most aspects of a resource: it holds data, performs tasks, and must be invoked by a *consumer* process to get replies, and by the *MCN* to queue replies. We therefore choose to implement the reply queue as a *pseudo-resource*, the *collector*, located on a specific *VM*.

There appears little value in having resource-based collectors, as replies to different operations would have to be handled differently by the user-code. We thus restrict a collector to be associated with either a operation-based *MCN*, or with a specific operation on a resource-based one.

The use of a separate reply-queue appears more in tune with the *MP* rather than the *RPC* paradigm. We therefore examine the collector principally in the *MP* paradigm, with a brief addendum discussing it in the *RPC* paradigm.

The use of a collector introduces a new control thread: the question arises what the multicast-statement's synchronisation should be, and what should happen to exceptions: should they be returned to the multicaster or the collector?

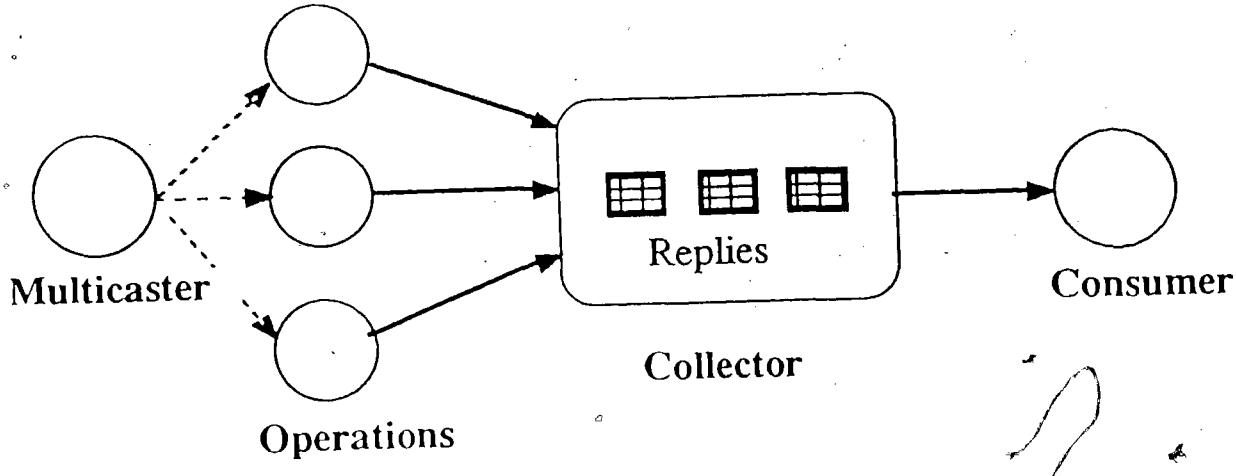

Figure 5 Multicasts with Replies delivered to a Collector.

4.10.1 Synchronisation and Reply_queues

In general, the user might desire a number of different synchronisation schemes, corresponding to the following situations where the multicast would not complete until

(1) all invocations had completed, or

(2) the collector received all (or all necessary) replies, or

(3) all (or all necessary) replies, had been removed from the collector.

Note that synchronisation points must occur in this order. Another option is to make collector multicasts asynchronous. It should be noted that synchronisation may also be provide at the user level: the justification for providing it as part of the multicast is simplicity, efficiency, and possibility that not all synchronisation points may be programmed at the user level.

In the *MP* paradigm, it is possible to implement any of synchronisation points as the signal for the multicast to terminate, based on the specification of termination semantics as part of the call. We have taken the view that the call should succeed if the required number of replies/*ACKs* return: exceptions are ignored. Applying this principle to collector calls suggests that the multicast be regarded as a success if the given number of replies are delivered to the collector. This also guarantees synchronisation point 1: the delay over a scheme that directly records the termination of operations is small, being the time for one additional message. Note that each operation separately acknowledging its completion would require an extra (n-1) messages, where n is the number of receivers. Synchronisation point 3 is not lexically defined, and is thus best provided at the user level.

In the *RPC* scheme, exceptions were handled individually, along with genuine replies. This suggests that exceptions be forwarded to the collector and handled there. Replies will be processed by a block of code, the *get-reply-block*. Unlike the situation in the *MP* paradigm, there is no time at which all required replies have arrived: the

completion of the operations and the exit from the reply-processing are the only significant synchronisation points. As exceptions are forwarded to the collector, it makes sense to let the synchronisation point also be an attribute of the collector. We thus propose that a collector call statement terminate when the collector get-reply-block terminates.

In cases when the reliability of the multicast is not an issue, as when *dbases* are invoked by *report_status*, an asynchronous collector multicast may be adequate: the rare failures may be handled at the user level. In the *MP* paradigm, this functionality may be provided by using send, as in:

send *dbases.report_status()* **overto** *status_collector*

In the *RPC* paradigm, there is no obvious way to specify this; however, a simple notation could be provided.

4.10.2 Remote Reply Queues in the Message Passing Paradigm

Collector use requires changes to the syntax of the call-statement. It should be noted that the use of the collector precludes the call being part of an expression.

<call_statement> ::= [**call**] <denotation> [**overto** <collector_id>]

A major design question is what operations the collector should have, and what their semantics should be. Clearly there must be an operation to allow the user to remove replies from the queue: **get_reply**, which must block when the reply queue is empty. This necessitates the provision of a **qsize**. non-blocking primitive to check the queue size: This operation is particularly useful when the multicaster is unaware of the size of the receiver group, and hence does not know how many replies may be removed from the collector. We propose two other built-in operations: **release_coll** and **wait_for_coll**, which operate a status, which will signify *free* or not *free*. *Release_coll*

will set a collector's status to *free*: *wait_for_coll* will return if, or when, a collector is *free*, and set the status to not *free*. These operations may be used to provide synchronisation between the multicaster and consumer, as discussed in section 4.10.6. The use of these operations to program a collector pool manager is shown in Appendix A.

4.10.3 Collector Statements and Semantics

For direct user invocation, a collector could be programmed as a resource. A collector to handle replies to the operation *query_load* would be:

```
resource collector
        import server
        op qsize() returns q_length:int
        op release_coll()
        op wait_for_coll()
        op get_reply(RES id : integer) returns load : real
end
```

Note that the declaration of *get_reply* agrees with that of the operation *query_load.*

The true-resource collector, like the true-resource multicaster, does not meet our needs, as the user must perform customisation best done by the compiler. Thus a built-in pseudo-resource collector is needed. As with other pseudo-resources, a collector must be declared, have a capability, be capable of being created, destroyed, and invoked by a set of built-in operations. It is implicitly invoked by a call multicast.

4.10.4 Declaration, Creation and Destruction of Built-in Collectors

As with *MCNs*, we add a new capability definition to RR(3.1) :

<captype> ::= cap | net | collector

Noting that as a collector variable must correspond to an operation, we have declarations of the following type:

var *load_collector* : **collector** *server.query_load*

A collector will be created and destroyed using standard *SR*:

load_collector := **create collector**() [**on** *vmach*]
destroy *load_collector*

where the optional **on** *vmach* specifies the *VM* host of the collector, by default that of the resource creating the collector. Here the word *collector* plays the same role as did *net* in the declaration of *MCNs*.

A collector may be invoked by a set of built-in operations, of the form:

collector_identifier. collector_operation

which are denotations, as defined in **RR** 7.3. The operations are get_reply, qsize, release_coll, and wait_for_coll.

Get_reply

The effect of *get_reply* is to retrieve a reply, the result of a multicast invocation of some receiver, from the *collector*, blocking if the reply_queue is empty. The syntax for get_reply, as a built-in operation on a pseudo-resource conflicts with that for get_reply when used to access replies to a regular call. When no explicit reply-queue is given, the specific operation must be named in order to ensure that the replies are from the correct call. Since a collector is associated with a specific operation, no operation need be specified: the conflict between these two uses of get_reply can be seen from the following example:

Multicaster consuming its own replies

On multicaster:

```
call server.query_load(id)
fa i:= 1 to no_servers ->
        total_load := total_load + get_reply server.query_load(id)
af
```

Replies sent to collector

On multicaster:

call server.query_load(id) over to load_collector

On consumer:

```
fa i:= 1 to no_servers ->
        total_load := total_load + load_collector.get_reply(id)
af
```

The use of the parameters in a get-reply-statement needs clarifying. With operation invocations, the values of VAL variables are passed by value to the operation: they are unchanged by the operation, which returns to the invoker in the *invocation block* packet, but unlike the values of RES parameters, they are not copied back to local variables of the invoker. If a multicast has VAL parameters, the structure of the parameter list for the get_reply statement presents a dilemma: either the get_reply statement will have parameters corresponding to the VAL parameters of the multicast, local variables whose only function is to act as place-holders, or, by omitting them, will have a parameter list which differs from that of the corresponding operation and multicast statement. Neither option is desirable.

qsize

qsize returns the size of the reply_queue (of type integer), and is non blocking. It permits the consumer to determine whether the queue is non-empty before executing a blocking *get_reply*.

release_coll

release_coll frees a collector, and discards any remaining replies on its reply queue. A collector is created with status *free*. The successful invocation of *wait_for_coll* makes the status not *free*.

wait_for_coll

wait_for_coll is a blocking operation that returns when the collector is *free*, setting the collector's status to not *free*.

4.10.5 Remote Reply Queues in the RPC Paradigm

The use of remote reply-queues with the *RPC* paradigm is somewhat unnatural, resulting in multiple logical threads of control. The syntax employed for multicasting must specify the collector, rather than reply-processing:

<multicast_statement> ::= **mc** <denotation> [-> <reply_processing_block>] **cm** |
mc <denotation> **overto** <collector_identifier> **cm**

Note that the denotation is that defined in standard *SR*, with no termination semantics. The statement to invoke *servers* with the operation *query_load*, and have the replies go to the collector *load_collector* is:

mc *servers.query_load(load,id)* **overto** *load_collector* **cm**

As previously stated, we propose that the multicast-statement terminate when the collector's reply-processing statement exits. We also propose that exception-handling be the responsibility of the collector. We propose the following syntax for collector use:

<collector_statement> ::= **gr** <denotation>
[-> <reply_processing_block>]
[**gr_handler** -> <exception_handling_block>]
rg

Here the denotation must be a collector_identifier followed by an argument list corresponding to that of the operation invoked. The code to process the replies to *query_load* and create a table *load_list* of loads for each server is:

gr *load_collector(load, id)* ->
load_list(id) := load
rg

The semantics and syntax of the reply-processing-block are identical to that of the same block of code used by the multicaster. This means that the consumer need have no

knowledge of the receiver group size, as the reply_processing_block will automatically terminate when all replies have been used.

It should be noted that the use of collector within the *RPC* paradigm is somewhat awkward, and the collector, though a pseudo-resource, is invoked in a fashion that does not correspond to operations.

4.11 Other Options

Querying the Group

The *V* system [Cheriton85] provides a primitive QueryGroup, that allows the user to determine a number of facts about the group. QueryGroup(group-id,pid) returns a structure that tells whether the process could join the group, whether the process is already a group member, whether the group exists, and the size of the process group.

In our scheme, querying whether or not a receiver can join a group is lexically determined. Whether a process is currently a member of a group, whether a group is active and the size of the process group are all volatile, in the sense that each may change immediately the QueryGroup call returns. This is an inevitable feature of a decentralised process group. It thus appears that such information must be maintained at the user level.

Security

Within *V* one may chose to employ a certain level of security, by employing a *same user group*, allowing only processes belonging to the same user to access a process group. Our scheme, by its use of capabilities, already has a built in level of security, which allows permissions to be granted in a structured way, and has the virtue that several users may have a private MCN, employing a MCN capability that is known only to them.

5. DESIGN AND IMPLEMENTATION ISSUES

In this chapter we discuss the implementation of the scheme for multicasting and collectors described in chapter 4. Firstly, it is necessary to outline existing communications primitives and how the creation of *VMs* and resources is achieved in *SR*. The implementation described here is for the message passing paradigm. However, as we note in section 5.6, only minor changes are required for the *RPC* paradigm.

5.1 Implementation of Resource and Operations

Each resource and operation is represented by an entry in the active resource or operation table of its host *VM*: its capability is a pointer into one of these tables. Each newly created resource is assigned an entry in the *active resource* table. This entry includes the *VM* capability (an integer), a unique sequence number, and a pointer to the *operations table* for each operation. The operations table has an entry for each active operation, indicating whether the operation is serviced by a proc or process: in the former case, the table has a pointer to the code for the proc, in the latter, it has a pointer to the appropriate *invocation block* [Andrews88]. The capability to an operation is a pointer to its host *VM's* table of active operations.

The *SR* compiler produces C-language code, which is then compiled by the C-compiler and linked with the *RTS* object code. The linked body of code runs as a *UNIX* process, the *VM*. When a user runs the program, the *VM* starts to run on the user's host machine. The C-code generated by the *SR* compiler provides for a remote action, such as resource creation or operation invocation, by creating blocks of data, which are transmitted by the *RTS*, within a packet, to the appropriate *VM*: any replies or acknowledgements are received, and the values of parameters and return values are copied back to local variables.

The *RTS* implements a multi-threaded system for multiple *SR* processes, within the *VM*. After an *SR* process has sent a message to create, destroy, or invoke a remote *SR* process, it waits on a semaphore, whose identity is contained in the message packet. The arrival of the packet on a *VM* causes an *SR* process to be spawned. This process acknowledges completion, or sends back values in a return packet. When this arrives, the appropriate semaphore is signaled and the original *SR* process may continue.

The initial *VM* forks a separate *UNIX* process called *srx*, that acts as a name server for *VMs*, ensuring that each *VM* has a unique number, and allowing a *VM* to obtain the socket address of another. Only this one instance of *srx* exists. Secondary *VMs* may be created on chosen physical hosts via create-statements. In the C-code, the function *crevm* is called with two arguments, the id of the intended host physical machine and the *VM's* capability, the sequence number of which is filled in by the *RTS* before *crevm* returns. Each *VM* is created via a call to *srx*, which assigns a sequence number as the *VM* capability.

Each *VM* listens on a set of sockets, initially its *listening socket*, known to *srx*, and used by other *VMs* to establish contact. When contact is made a new socket is created to provide a channel between the two *VMs*. Messages received on these sockets are examined in a round-robin fashion, and used to invoke the appropriate part of the *RTS*. These messages include ones to create and destroy resources, invoke operations, as well as acknowledgements of earlier messages. When a *VM* is created, it is given the socket for *srx*. A *VM* (A) wishing to transmit a message to resource B on another *VM*, checks a local directory of *VMs* and sockets. If B is unknown, A sends a message to *srx* and receives back the *listening socket* of B.

The C-code to create a resource assembles a *creation block* and calls the function *create*. The creation block has fields for its size, a pointer to the entry for the resource in the pattern table, the id. of the intended *VM*, the initialisation values, and a pointer to the resource capability. When *create* returns, pointers to the active resource table entry

for the resource will have been inserted into the resource capability.

Invocations are performed by having the invoker create an *invocation block*, which holds the capability to the required operation and any arguments. The *RTS* is called by the C-function *invoke*. The invocation block is sent to the required operation, which may read the values of VAL and VAR parameters, and modify the values of those of type VAR or RES. The operation may also insert the value of the return value of the operation, if such exists. The invocation block is returned to its creator, which then copies the returned value and the values of VAR and RES parameters to local variables.

The C-code for an invocation generates an invocation block and calls *invoke*. The invocation block has fields for its size, the size of the operation's arguments, the operation type (such as SEND_IN), and the operation's capability. The *RTS*, if necessary, transmits the packet over the network to the appropriate *VM*. The *VM* of the invoked operation uses the capability to find the operation table entry for the operation: if serviced by a proc, a new *RTS* process is spawned; if by a process, the invocation block is queued, and may eventually be serviced via an input statement. If the invocation was a send, the *RTS* of the operation's *VM* will acknowledge the packet receipt, and *invoke* will return, causing the send to terminate. If the invocation was a call, the *RTS* of the operation's *VM* will await the return of invocation before acknowledging with a packet containing the modified invocation block.

Calls within a co-statement are made sequentially without waiting for completion of previous calls. The *RTS* sets up a structure to accept returning invocation blocks, and if, necessary, invoke the *post-processing block* associated with each call. The entire co-statement returns, either when all calls have returned and their post-processing is completed, or when a post-processing block performs an *exit-statement*. The order of returning calls is non-deterministic. Unread packets, and those arriving after the co-statement has terminated, are discarded.

5.2 Multicasting

Implementing multicasting in *SR* requires a reliable *VM-to-VM* communications scheme. If the multicast is to be efficient, each *VM* must be able to multicast to the remainder. We thus employ a two layered model: multicasts between *VMs* and multicasts to invoke operations. The latter are implemented via the former.

In the lower layer there is a process group of *VMs*, which must be created before multicasting commences, and updated when *VMs* are created or destroyed. Each *VM* will be required to know the size of the *VM* group, in order to know the number of expected acknowledgements.

At the upper level is a process group of operations or resources. Multicasts on an *MCN* (multicast network) are first received and acknowledged by each *VM*, which then demultiplexes the multicast to each of its local *MCN* receivers. This is the scheme described in section 2.2.1.

5.2.1 Broadcasting on the Sun Network

The Sun-Network [Leffler83] provides both stream and datagram communications, using sockets. Stream sockets provide reliable communications; datagrams are unreliable. Messages from process to process are implemented using addresses: machine addresses and port numbers. A process listening on a socket is waiting for a packet addressed to a particular port on a given machine. This is transparent to the user process, which acquires a port by the act of binding to a socket. Each port obtained in this way is unique and is held by the process as long as it executes. Unfortunately the port number assignment scheme is host-specific, implying that some *VMs* may not be able to bind to that port (it being in use), and also that some multicasts may be received by unsuspecting processes listening on that port. In a practical implementation some scheme for reserving a port number on a network-wide basis would be required.

Broadcasting over the network is achieved by using wildcard values for addresses, and a specific port number. A specific port is chosen by having the first receiver bind to a socket with port number set to zero. The number of the port may be passed as a parameter to other receivers and senders. Subsequent receivers will bind to a socket with this value for its port number, and INADDR_ANY as its socket address. The sender binds its socket using the port number and a wildcard value of zero for its socket address.

The two-level multicasting scheme has a major limitation in that it appears that only one *UNIX*, process may receive multicasts on a given host machine, as only one socket on a given processor may bind using a particular port number. Consequently only one *VM* per processor may belong to the *VM* process group and srx may not receive multicasts. The solution to this problem is to introduce a new layer of daemon processes one per machine. These would multicast to each other, as described for *VMs* above, and demultiplex the multicasts to the *VMs* and *srx*. *Srx* could easily be enhanced to create the **multicast managers** as required, and act as a name server for them.

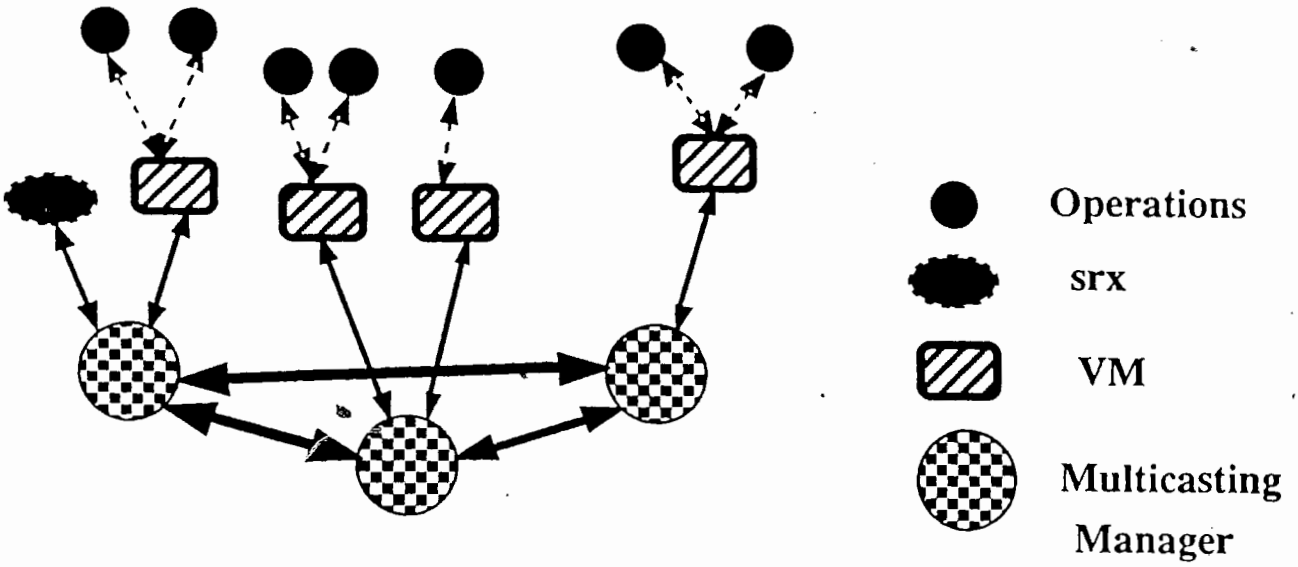

Figure 6 Three Layer Multicasting

In *UNIX* broadcasting works only with datagrams. This means that the multicasting system of *SR* must take steps to ensure reliability, based on acknowledgements, timeout and re-transmission. A problem is that datagram service requires that each datagram be read from the socket in its entirety, by a single read statement. This is problematical in a system in which message sizes vary. One solution is to require all multicast packets to be less than some fixed size. The sender would reject excessively large packets: the user would read any multicast packet into a fixed size buffer. A general solution to the problem would be to transmit overflow packets to carry the contents of packets exceeding the fixed size. It should be noted that Ahamad and Bernstein's multicast sockets could be employed here [Ahamad85].

5.2.2 Process Group Creation/Destruction and Modification

Multicasting at either level occurs in two modes, transmission and maintenance. **Transmission mode** refers to normal multicasting to provide communications, between *VMs* or to invoke operations. **Maintenance mode** refers to multicasts required to create, destroy, or modify the process group. For *VM-to-VM* multicasting, the addition or destruction of *VMs* requires maintenance multicasts. For *MCNs*, the addition or removal of receiving resources or operations requires maintenance multicasts. Note that all multicasts at the upper level are transmission multicasts at the lower level. At either level, maintenance multicasts must not overlap transmission multicasts, as reliability demands that the size of the multicast group, either the number of receivers or *VMs*, must be known. We prevent this overlap by suspending transmission multicasts while a maintenance transmission takes place.

5.2.3 VM-to-VM Multicasting

Multicasting Reliability

In *VM-to-VM* communication, the multicasting *VM* broadcasts a packet, and collects unicast acknowledgements (*ACKs*). If the number of *ACKs* fails to reach the known number of *VMs* before a timer times-out, a retransmission occurs. We choose to re-multicast, rather than unicast to the delinquent *VMs*, as this latter approach requires each *VM* to know the identity of other *VMs*, and to keep track of which *VMs* acknowledge. The use of a sequence number allows *VMs* to quickly discard duplicates. If retransmission a fixed number of times fails to produce all the required acknowledgements, *VM* must be regarded as unreachable, and an exception declared.

In order to perform both *transmission* and *maintenance* multicasts, protocols must be used. It should be noted that while some of the *ACKs* are required for synchronisation, others are only required at a lower level, to provide reliability. In the diagrams that illustrate the protocols, we employ dotted lines to indicate *ACKs* required for reliability only, thick lines for multicasts, and thin lines for *ACKs* required for synchronisation.

The VM Process Group

The protocol for reliable multicasting relies on each *VM* knowing the total number of *VMs*, the size of the *VM* process group. This implies that a protocol must be used when a *VM* is being created or destroyed. (In our prototype we did not implement this protocol. Its use is not essential if care is taken to create all *VMs* before creating any *MCNs*.) Our design implements this via a central *VM* controller within an enlarged *srx*, which must now have a separate socket on which to receive multicast acknowledgements and a separate *SR* process to deal with them. These are required because *srx* currently has no internal queue, but queues requests at its sockets: messages are read and processed in a round-robin fashion. If a request, such as a process group update,

were received while such actions were blocked by the update protocol, *srx* would be deadlocked. With the separate socket and *SR* process, multicast *ACKs* may be read and processed at all times.

Creating and Destroying VMs

When the first *VM* forks *srx*, both processes must execute code to initialise the *VM* group. For subsequent requests to create *VMs*, the following protocol must be executed:

1. **srx** performs an MCN_ADD_VM multicast and awaits MCN_SUSPEND_ALL_ACK acknowledgements.

2. Each **VM**, on receipt of a MCN_ADD_VM multicast, continues to serve any incoming multicasts, but will not initiate any. When all current outgoing multicasts complete, it acknowledges the *srx*.

3. **srx**, when it has received *ACKs* from all *VMs*, creates the new *VM*. Afterwards it will multicast a MC_CONTINUE message.

4. Each **VM**, on receipt of a MC_CONTINUE message, increments the local value for the number of *VMs* and acknowledges *srx* with a MCN_CONTINUE_ACK.

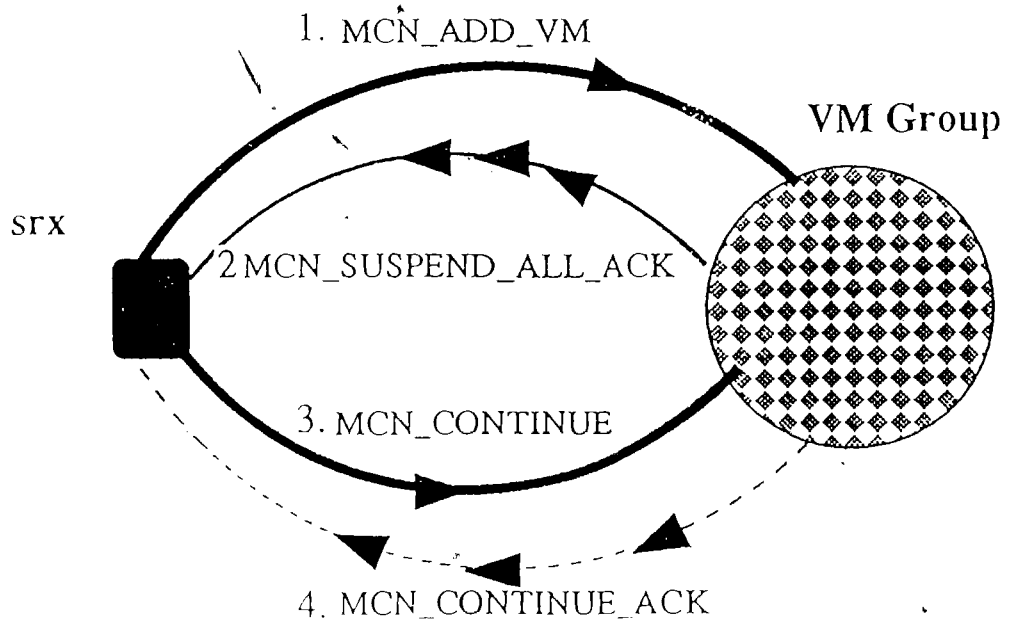

Figure 7 Protocol for Creating a VM

This protocol ensures that all multicasts are suspended while a *VM* is being created or destroyed. The MCN_CONTINUE_ACKS are not required by the protocol, but by the need to ensure that the continue message has been received by all *VMs*.

5.2.4 Multicast Networks

In order for a *VM* to be able to demultiplex a multicast to a *MCN*, it must maintain a list of the capabilities of local receivers for each *MCN*. The resource adding a receiver to the *MCN* will have the receiver's capability and will thus sent it to the receiver's *VM* to be added to the list. In the case of operation-based *MCNs*, the C-code for a multicast generates a multicast packet, containing an invocation block, as for a conventional invocation, but with no specified operation. Each *VM* must copy in turn the capability of each operation on that *MCN's* local receiver list into the invocation block, which may now be used to invoke these operations.

With resource-based *MCNs*, multicasting is more complex. A multicast packet must provide the information necessary for each *VM* to select the required operation capabilities from the locally stored resource capabilities. This information is the offset of the operation capability from the start of the resource capability, and its size. These quantities have no current significance in the implementation of *SR*, but are currently evaluated by the compiler.

Creating or Destroying a MCN

Creating and destroying a *MCN* is performed by a multicast MCN_CREATE (MCN_DESTROY) message from the creating *VM*, acknowledged by MCN_CREATE_ACKs (MCN_DESTROY_ACK).

The capability of a *MCN* must be a unique system wide identifier. Since *MCNs* may be created by any *VM* via a multicast, the capability's uniqueness is ensured by including the *VM's* capability and a unique sequence number. The alternative would be

to use the *srx* as a name server for *MCNs*, and have requests to create *MCNs* sent to it. This would require at least one extra unicast. The C-code for creating an *MCN* is the function *create_net*, which has two arguments: pointers to the capability of the MCN's host *VM*, and to the *MCN's* capability. At run-time the *VM's* capability is copied into the *VM* field of the *MCN* capability, and a sequence number is generated to complete the *MCN* capability. The protocol is illustrated in Figure 8, and works as follows:

1. **The creating VM** multicasts a MCN_CREATE packet, containing the *MCN* capability, to the *VMs*.

2. **Each VM** adds the new *MCN* to the local list of *MCNs*.

Destroy_net is implemented in a similar fashion. It may be noted that a *VM* may multicast on an *MCN* and have the multicast reach a *VM* before it has created the *VM*, due to the MCN_CREATE message getting lost. In this unlikely event the *VM* may ignore the multicast, as it will be resent after the multicasting *VM* times-out. Eventually the *VM* will get a resent MCN_CREATE multicast, and will subsequently be able to service the resent multicast on that *MCN*.

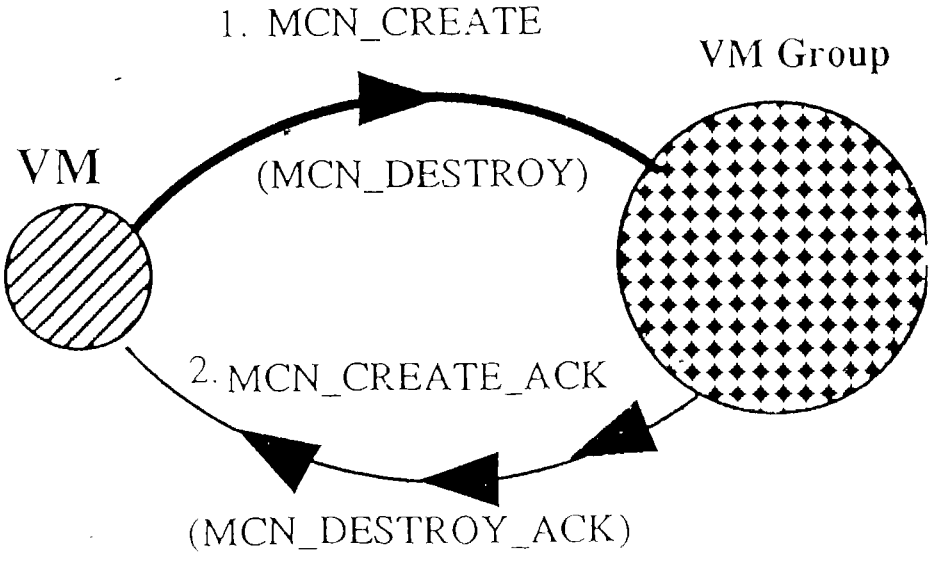

Figure 8 Protocol for Creating or Destroying an MCN

MCN Updates

A resource that has the capability to a suitable receiver may join it to an *MCN*, and later remove it. Joining (removing) an operation to an *MCN* is done in C-code by invoking the function *net_join (net_leave)* with two arguments: the *MCN* capability and receiver capability. The *RTS* then creates an *update block* which contains the type of update, MCN_JOIN (MCN_QUIT), and the capabilities of the receiver and *MCN*, and sends the packet to the host *VM* of the receiver.

Updates to a given *MCN* require the suspension of transmissions on that given *MCN*. This is achieved using a protocol similar to the one used for *VM* process group changes. In our prototype we have avoided this complication by requiring the user to keep a static process group, created before multicasting starts. The protocol is shown in Figure 9, and works as follows:

1. **The initiating VM** sends an MCN_UPDATE packet.

2. **The other VMs** in the group, on receipt of an MCN_UPDATE packet, suspend multicasting on the specific *MCN* and, when current multicasts terminate, acknowledge with a MCN_SUSPEND_ACK packet and update their local data-bases. The local update consists of incrementing or decrementing the size of the process group, and storing or removing the operation's capability on the local-receiver-list for that *MCN*, if the operation is a local one.

3. **The initiating VM**, when all acknowledgements have been received, multicasts a MCN_CONTINUE packet.

4. **The other VMs** issue an MCN_CONTINUE_ACK packet, and resume multicasting on that *MCN*.

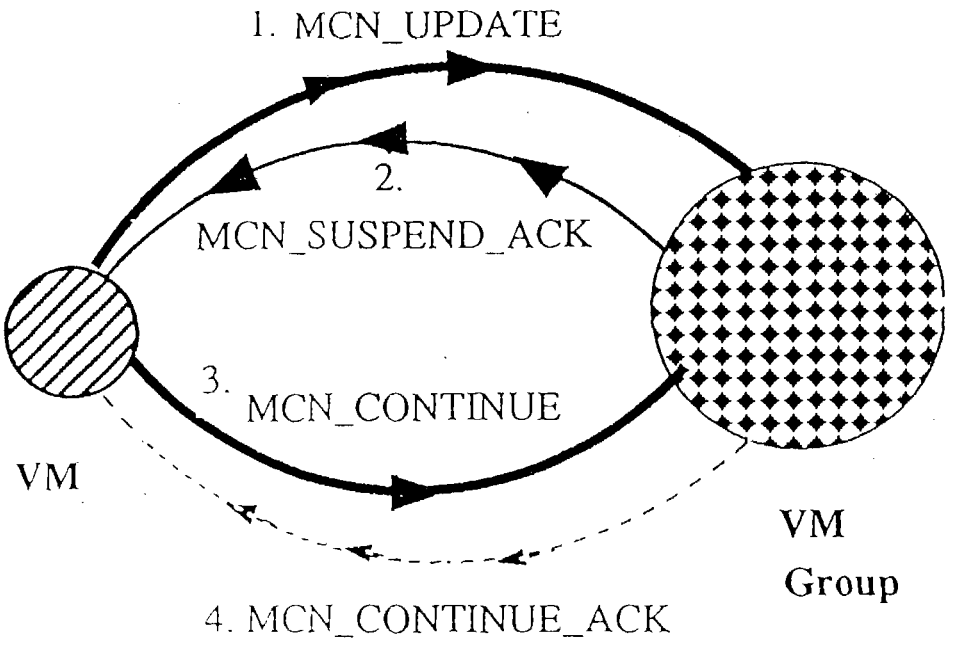

Figure 9 Protocol for MCN Updates

MCN Multicasting

As with unicasts, the values of VAL and VAR parameters must be copied into the invocation block, before it is multicast. In the case of calls, reply packets must be made available to the user. The first reply will have its values copied back to the user, those of RES and VAR parameters, and any return value, where permitted. *Get_reply* accesses the queue and copies back values in the same way.

The required protocol is simple. The packet is multicast to the *VMs*. Each *VM* acknowledges the receipt with a MCN_INVOKE_ACK packet. This indicates reliable delivery of the message, permitting a asynchronous multicast to terminate. In the case of asynchronous multicasts, each completed operation causes a return packet of type MCN_CALL_COMPLETE to be sent back to the multicasting *VM*. This packet will be placed on a queue by the *RTS*. The protocol is illustrated in Figure 10, and works as follows:

For MC_SEND

1. **The multicasting VM** sends a packet of type MCN_INVOKE.

2. **The receiving VM** replies with a packet of type MCN_INVOKE_ACK

3. **The multicasting VM,** upon the receipt of of MCN_INVOKE_ACKS from all *VMs*, then terminates the multicast.

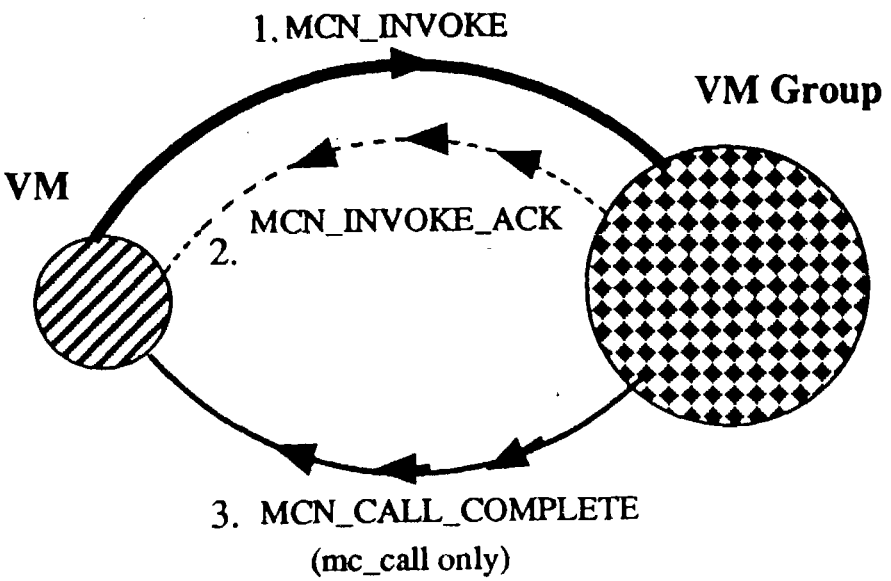

Figure 10 Multicasting Messages for Send and Call Multicasts

For MC_CALL (also illustrated in Figure 10)

1. **The multicasting VM** sends a packet of type MCN_INVOKE.

2. **The receiving VM** replies with a packet of type MCN_INVOKE_ACK

3. When each operation completes it sends a packet of type MCN_CALL_COMPLETE.

4. **The multicasting VM** uses the MCN_INVOKE_ACKS to check if a retransmission is required. After the required number of MCN_CALL_COMPLETEs (all or the number set by the termination semantics) has returned, the multicast terminates.

Same order multicasting may be implemented using *srx* as a central multicaster. REQ_MCN_SEND and REQ_MCN_CALL packets are sent to *srx*, which then performs the multicast.

5.3 Collectors

In our prototype we have not implemented collectors; however, the following describes how this may be achieved.

A collector has a capability like that of an *MCN*: the capability of its host *VM*, and a sequence number. When a collector is created in *SR*, a C-function *create_col* will be called to send a MCN_COLL_CREATE packet to the host *VM*. The host *VM* will initialise a data structure for the collector, and complete the collector's capability by assigning a sequence number. The *VM* will then acknowledge with a MCN_COLL_CREATE_ACK packet, containing the collector capability. The *VM* initiating the creation will copy back the value of the capability.

When a *VM* receives a reply packet of type COLL_INVOKE, it must find the collector's data-structure using the collector capability as key, and add the invocation block to the reply_queue.

It is envisaged that in most cases consumers will be located on the same *VM* as the collector. In other cases each collector invocation will require new *SR* packets being sent to the collector's *VM* to invoke the collector, and return packets carrying values and acknowledgements. This may be modelled directly on existing *SR* communications.

5.4 Implementing the Two Paradigms

Our prototype was designed to implement multicasting using the *MP* (Message Passing) paradigm (section 4.7): here we discuss the differences between implementing this paradigm and the *RPC* (Remote Procedure Call) paradigm, as introduced in section 4.8. With asynchronous multicasts, the implementations will be virtually identical,

since the same semantics apply. The one exception will be in the application of the termination semantics in the *MP* paradigm: here the semaphore (on which the send waits) will be signaled when the required number of *ACKs* have returned.

With synchronous multicasts, the two paradigms require very similar implementations. The difference is that in the *RPC* paradigm a reply queue is created for each multicast and destroyed when the statement ends; in the *MP* paradigm the queue is permanent and associated with a particular multicast operation for that particular process.

With the *MP* paradigm, the first reply is copied back to local variables before the call returns. Each time a *get_reply* is executed, the appropriate reply-queue is checked: if empty, the statement blocks; otherwise the reply values are copied back to local variables and the statement terminates. Each new call empties the queue for that particular multicast operation. The call_statement itself will not terminate until the required number of replies, by default ALL, have arrived.

In the *RPC* paradigm, a reply-queue is initialised for each multicast and deleted when the multicast-statement terminates. The multicast-statement transmits its message, and blocks until a reply arrives. The values in the reply are copied back to local variables, and the reply-processing-block executes, if it exists. The next reply is then copied back and reply processing initiated. Each reply decrements a counter, initialised to the number of receivers: this counter is checked to determine when to terminate the multicast.

Though we did not attempt to design the error handling scheme, as this has not yet been done for any *SR* statement, a few points are worth making. An exception may be detected remotely, resulting in the return of a special type of packet, or locally, with the failure of a *VM* to respond. In the latter case, the local *RTS* may prepare a special packet. In either case the packet may be passed to a higher level. At this higher level, the package type will be recognised and the handler executed, rather than the usual reply-processing-block.

It is impossible to be definitive about the relative speed of the two paradigms, as the costs depend on the application and implementation. There are, however, no major implementation differences that would appear to lead to a significant difference in performance

5.5 Implementation of the Prototype

As previously noted in chapter 5, several features of the design were not implemented in the prototype. Our view is that the only major issue that needs resolving by experiment is the relative efficiency of multicasts and unicasts within a co-statement. This perspective permitted us to simplify the protocols for updating the *VM* group and *MCN* membership, by assuming that both groups would be established before multicasting commenced, and would remain unchanged. Furthermore, the features required to support multiple replies and collectors were not implemented, as it was not expected that substantial gains in efficiency would result here.

However, designing a prototype gave us a thorough understanding of the structure of the invocation mechanism, and enabled us to refine and improve our initial designs. Implementing the prototype requires changes to both the *RTS* and the compiler. These changes are outlined in Appendices B and C. Appendix D contains the two files: mcn.c and bcutil.c, which contain the major portion of the extra *RTS* code required.

5.6 Potential Timesavings using Multicasts

The time taken to perform a set of invocations depends on the degree of parallelism, which is maximal when all processors are active, as well as the network. We consider three ways of invoking multiple operations:

(1) a sequence of unicast invocations

(2) the same set of unicast invocations within a co-statement,

(3) the same set of invocations effected by a multicast.

The co-statement allows calls (but not sends) to be performed in quasi-parallel. The difference between a sequence of calls, and the same sequence performed within a co-statement, is that within the co-statement the calls are non-blocking, and network packets may be sent one after another, without waiting for each invocation to complete. The co-statement collects the replies as they arrive and associates each with the correct call. The co-statement thus permits the invocations to be serviced in parallel, if the operations are on different hosts; however, the preparation and sending of the packets containing the invocation, and the receipt and processing of the replies must be performed sequentially by the host initiating the invocations.

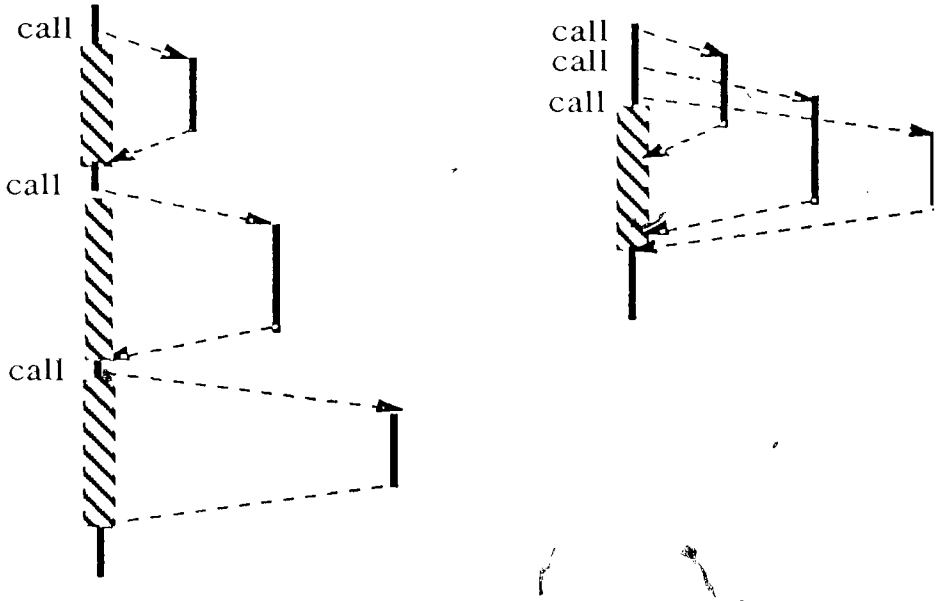

Figure 11 A sequence of calls, without and within a co-statement

Examination of Figure 11 shows how the use of a co-statement saves time. It is clear that the extent of the time savings depends on the time taken to service the invocations. In the case when only one or a few replies are required, the above applies, with the additional advantage that the quorum may be reached sooner. With a sequence of calls, it is unlikely that the user can attempt the fastest invocations first; however, within the co-statement, the fastest invocations returning allow the user to continue before the slower invocations have returned.

The major difference between a sequence of invocations and the use of multicast is that only one network packet need be prepared and sent by the initiating VM, thus saving processor and network time; reply or acknowledgement packets must be prepared, transmitted, and processed as before.

6. CONCLUSIONS AND FUTURE RESEARCH

6.1 Conclusions

We described a number of advantages of multicasts over a sequence of one-to-one multicasts, which our design realises:

- our semantics allow a client to request service from a server without knowing its identity: the client need only have the *MCN's* capability.

- network traffic is reduced, and the average time before each operation is invoked will be reduced.

- user code is simplified.

- the average time for a process group member to receive a message is reduced.

We detailed our design considerations for multicasting in *SR*, and the functionality required in sections 4.1 and 4.4. We claim these to be attainable: the scheme proposed is powerful, permitting a wide range of different synchronisation schemes, including early termination, reply handling, and reliability. In section 4.9 we demonstrated that multicasting based on the remote procedure call paradigm is more appropriate in *SR* than multicasting based on the message passing paradigm, particularly with regard to its reply handling and error-handling capabilities:

- the semantics of the *RPC* are consistent with standard *SR*.

- the number of new primitives required is small and their use is consistent with *SR*. The *RPC* syntax has an advantage over *MP*, in that only one multicasting primitive is required, and that synchronisation, early termination and post-processing are all handled in ways similar to those employed in standard *SR*.

- the changes required to the *SR RTS* and compiler are small, and appear unlikely to have much effect on performance.

- the new primitives are simple to use.

We have thus shown that multicasting can be introduced into a high level language in a way which is simple, both semantically and linguistically. Different forms of synchronisation and the processing of replies can be specified with minor changes to the grammar. The proposed scheme permits the user to create, modify, and destroy *MCNs*, and to invoke them by a single statement, which permits simple and concise user code.

In conclusion it should be noted that the existing co-statement and the multicast provide for the parallel execution of the invoked operations. The multicast has the additional advantage of reducing the time of communication, including that spent by the initiating RTS in preparing and sending the packet.

6.2 Future Research

One aspect of the research that we were unable to complete, due to a intractable bug in the communications scheme, was to compare the time taken for multicasting and sequences of unicasts, both within and without co-statements. Such comparisons must be done experimentally, as the times depend on the specific hosts, network, network traffic and the algorithm being executed. Thus, while the parallelism provided by either the co-statement or multicasting provides substantial benefits, whether the multicast is significantly more efficient than a co-statement in practice, it still an open question. Research of this nature could be performed in using various implementations, and in other-than-*UNIX* environments.

A second area for future research is a close examination of distributed algorithms to determine those for which multicasting is applicable, and to implement these in order to determine what speed-ups are attainable: in particular the relative merits of the two paradigms may be examined. Other high level languages may be examined to determine how multicasting may be incorporated. Lastly, the implementation of error-handling schemes for multicasting appears a substantial topic.

Appendix A : Collector Pool Manager

The *collector pool* manager creates *collectors* and, on request, passes their *capabilities* to users. After allocating a *collector*, the manager will invoke *wait_for_coll*, thus being alerted when the *collector* is freed.

It should be noted that in some cases *collectors* should not *home* on the *collector* manager, but on the multicaster, for re-use. For this to be the case, the multicaster should perform the *wait_for_operation*. However, only one *wait_for_coll* will succeed each time the *collector* is freed. This problem may be solved by giving the *collector-pool* manager two *get_collector operations*: *get_collector* and *get_homing_collector*. In the first case the *collector* will not execute *wait_for_coll*. The provision of non-homing *collectors* requires the provision of a *return_collector operation*, to permit the user to explicitly release a *collector* to the manager. The code for such a manager is presented. For simplicity we have assumed a static *pool* of n *collectors*.

```
resource collector_pool_manager
        import my_resource
        get_collector() returns coll: collector my_opn_type
        get_homing_collector() returns coll: collector my_opn_type
        return_collector(coll: collector my_opn_type)
end

body collector_manager(number_of_colls : int)

        op wait_til_free(my_coll: collector my_opn_type;n:int)

        var coll[1:number_of_colls] : collector my_opn_type
        var free[1:number_of_colls] : bool
        var num_free : int := number_of_colls

initial
        fa i: = 1 to number_of_colls ->
                coll[n] := create collector()
                free[n] := true
        af
end
```

```
var coll: collector my_opn_type
var n :int
do true ->
      in get_homing_collector() and (num_free>0) ->
            fa i:= 1 to number_of_colls    ->
                  if free[n]->
                        num_free--
                        free[n] := false
                        exit
                  fi
            af
            send wait_til_free(coll[n],n)
            return(coll[n])

      [] get_collector() and (num_free>0) ->
            fa i:= 1 to number_of_colls    ->
                  if free[n]->
                        num_free--
                        free[n] := false
                        exit
                  fi
            af
            return(coll[n])

# return_coll allows a user to return a non-homing collector

      [] return_collector(coll) ->
            coll.release_coll()  # just to make sure
            # find which number coll it is. n, say
            free[n] := true
            num_free++
      ni
od
end


# wait_til_free is a vulture proc that waits for the collector to be freed
# and then updates manager's data structures

proc wait_til_free(coll,n)
      coll.wait_for_coll()
      free[n]:=true
      num_free++
end
```

Appendix B : Run-time Support Changes

Changes to the *RTS* may be grouped into three categories: specific code to deal with *MCNs* (*mcn.c*), code to implement multicasting via datagram sockets (*bcutil.c*) and miscellaneous changes to the existing *SR RTS* (*main.c, net.c, socket.c, remote.c, srx.c*)

Broadcasting

Bcutil.c contains the code to customise sockets for sending and receiving multicasts over the Sun network. Datagram sockets must be used for multicasting, as against the stream sockets used for *VM-to-VM* communication in *SR* [Leffler83]. Hence each *VM* must add a new socket of type DGRAM to receive incoming multicasts; a second socket must be used for outgoing multicasts.

(1) The function *get_first_rec_sock* creates the first receiving socket by using INADDR_ANY as the s_addr and by zeroing the port number, before binding the socket. After binding, the socket has an assigned port number that may be found using *get_port*.

(2) *Get_rec_sock*, used by subsequent *VMs*, creates and binds additional receiving sockets. It is identical to *get_first_rec_sock* except that the port number's value is assigned by the user.

(3) *Get_send_sock* creates and connects sending sockets, using BC_WILDCARD (zero) for the s_impno, and the user specified port number.

VM-to-VM Multicasting

During initialisation, *VM* 1 calls *init_all_multicasts*: this provides sockets for sending and receiving multicasts. When *VM* 1 forks and execs *srx*, the port number is provided as an additional argument. When *srx* in turn forks other *VMs*, it also provides the port number as a argument. These additional *VMs* execute *init_multicasts*. to provide

the required sending and receiving sockets associated with this port.

The ability to receive multicasts must be incorporated into the network interface of each *VM*, which with *srx* has a daemon *SR* process (*net_recv*) which selects on a set of file descriptors. Rather than employ a second daemon process, we chose to implement multicast reception as part of *net_recv*. This posed a number of problems, caused by the differences between stream and datagram sockets, the principal one being that a datagram packet must be read once in its entirety, while a stream socket packet can be read piecemeal. A major complication in modifying the *RTS* is that *VMs* and *srx* share much of the communications code, and thus any modification to the code must be consistent with both uses.

SR packets are read in two parts: *net_recv* reads the fixed length packet header, which contains the size of the packet: *net_more* is called to read the rest. This is not possible with datagrams, which must be read in their entirety. As datagrams are limited in size they may be read into a fixed size buffer. Our solution does this, requiring *net_recv* to be able to distinguish between stream and datagram packets, and for *net_more* to be called only for unicast packets. To facilitate these changes the space for the packet is now allocated within *net_recv*, which returns the packet, rather than its type. Additional code was added to *net_recv* to trap multicasts received by the sending *VM*: a dummy packet of type MSG_NONE is returned, and discarded.

In order to receive multicasts the receiving socket must be added to the set of file descriptors selected on by *net_recv*. This is performed by *mcn_conn* which is called by each *VM* as part of its network initialising code. As noted above, *net_recv* must be able to distinguish between multicasts and unicasts. This is most easily done by reserving a particular file descriptor for incoming multicasts. As *VMs* and *srx* both use *net_recv*, this number must be common to both. For *VMs* the file descriptor is naturally 3; for *srx* the omission of a multicast socket leaves file descriptor 3 free for other use. This problem is solved by having *srx* duplicate a file descriptor as part of its initialisation: this is

3 and is thus never used by *srx*.

The fact that *SR* packets sent by datagram which may not exceed a fixed size, implies that *SR* multicast packets that exceed this size must be transmitted in several datagram packets. As we were constructing a prototype only, we decided to limit multicast packets so as to fit into a single datagram packet.

Note that the comparative rarity of *MCN* creation, destruction and updating implies that we need not be too concerned over the efficiency of these actions. These actions are also simpler in that only one action is taken on each *VM*, to update the local *MCN* database; with multicast invocations multiple operations may be invoked, each potentially requiring an acknowledgement.

As noted we did not implement ACKs required to ensure reliability. In the absence of such ACKs, asynchronous multicasts do not require any ACKs. The handling of ACKS requires the careful use of counting semaphores. After the initiating *VM* has initiated local action and sent the multicast, it waits on a counting semaphore: acknowledgements signal the semaphore, allowing the initiating process to continue when all the signals have been executed. In the case of multicast calls, the acknowledging must wait until the operations complete: each each invoked operation must acknowledge.

The above changes relate to an implementation of the *MP* paradigm. The difference between this and an implementations are minor, relating primarily to the manipulation of the reply-queue.

Appendix C : Compiler Changes

The *SR* code is compiled into *C* code by a two-pass compiler. The *C* code is then compiled into object code by the *C* compiler, and linked with the run-time support code. The first pass of the *SR* compiler performs the lexical analysis, generates the symbol table, and a list of intermediate code (i-code). The second pass processes the i-code, with reference to the symbol table, outputting *C* code.

Each variable in *SR* is typed by assigning it a signature. The signature gives the data-type of low-level objects: for high level objects such as operations, it gives the number and type of its parameters. The signature is employed to determine the compatibility of variables with respect to some binary operation, and in addition points to the memory allocated to the variable at run-time.

The first step in modifying the compiler to handle multicasts was to introduce a number of new tokens to capture certain new keywords: viz *net, join, quit, mc_send, mc_call*

In order to permit the declaration of *MCNs*, the compiler code was modified so that the keyword *net* would be accepted along with *cap*. This permits the declaration of an MCN associated with any object with a capability, such as a resource or operation. In declaring an *MCN* variable, the signature created is the same as that for a variable of the corresponding operation or resource, except for the type being T_NET. When a *MCN* declaration is parsed, run-time space is allocated to hold the *MCN* capability.

Statements for creating and destroying an *MCN* are parsed using the same code as for creating and destroying other objects. The parser recognises that an *MCN* is being created from the signature of the object being created, and thus calls a new function (*net_create_stmt*), instead of *create_stmt*, to parse the remainder of the statement. *Net_create_stmt* creates an empty list for initialisation arguments and uses TK_CREATE_NET in place of TK_CREATE, which is used for creating other objects .

Statements for updating the *MCN* group or invoking it begin with a new keyword (such as *join* or *mc_send*), permitting the parser to immediately identify the statement type and call the appropriate new function to complete the parsing. The update statements are easily parsed as they take two arguments, the *MCN* and the receiver. At present no signature check is made as to whether the two are associated with the same resource or operation type. The intermediate code uses TK_JOIN_NET and TK_QUIT_NET.

Multicasting statements require the parsing of an argument list. If the *MCN* is associated with a resource, the name of the required operation must precede the list. As the parsing problem here is identical with that for a regular invocation, we may make use of existing compiler code to parse the *denotation*. The intermediate code employs TK_SEND_NET and TK_CALL_NET. In the prototype we have not provided for the compiling of the optional termination semantics or the use of collectors. However, incorporating these features is straightforward.

The i-code generation phase is generally straightforward, as each new feature has a new type of i-code statement. *MCN* creation and destruction statements generate simply *C* function calls with a pointer to the appropriate capability. The code for *MCN* updates is similar, only in this case there are two arguments to the *C* function - the *MCN* and the receiver capabilities.

Multicasts require more complex C-code than creations or updates, due to the variable number and size of arguments. The compiler must generate the code to create a packet, fill in the values of arguments, and call a C function to effect the multicast. This closely resembles the actions required to perform a unicast, and hence most of the same code can be used.

In the prototype we have not incorporated most of the features required to support the use of collectors. The compiler code for the declaration, creation and destruction of collectors parallels that for *MCNs*, and is easily written. The invocation of collectors is

simple, as the C-code required is merely a function call with a single parameter, except for *get_reply*, which requires the use a variable size packet and the copying back of the value of VAR or RES type parameters to local variables. This is already performed by *calls*, and thus the same code can be used, or mimicked.

The above describes the changes required to implement the *MP* paradigm. In order to implement the *RPC* paradigm, some changes are required in both phases of the parse. The i-code node for a multicast-statement must include the reply-processing-block. The C-code generated must contain the multicast invocation and code to copy back values, as before, the code for the reply-processing-block, with suitable labels and gotos to implement the implicit reply-processing-loop, a counter for the replies, and a test-statement to terminate the loop after all replies have been processed.

Appendix D : RTS code

```
/************************** file: mcn.c ************************************/
/* contains major functions required for multicasting                      */

#include "rts.h"

/***************************************************************************/
/* data structures used to store identity of existing MCNs and their       */
/* local receivers                                                         */
/* The list of MCNs on a VM is kept as a link-list of mcn_blocks           */
/* Each mcn_block has a link-list of rec_blocks, one per local receiver    */


typedef struct mcn_block_st *mcn_block;   /* data-block for an mcn         */
typedef struct rec_block_st *rec_block;   /* data-block for a receiver     */


struct rec_block_st{                      /* data-block for a receiver     */
  struct ocap_st opn;                     /* operation's capability        */
  rec_block nextrec;                      /* pointer to next receiver      */
  };

struct mcn_block_st{                      /* data-block for a MCN          */

  struct mcn_st id;                       /* mcn_capability                */
  int numrecs;                            /* total number of receivers     */
  int numlocal;                           /* number of local receivers     */
  rec_block recs;                         /* pointer to list of local receivers */
  mcn_block nextmcn;                      /* pointer to next MCN block     */
  };

/********************** functions ******************************************/

mcn_block get_mcn_block();                /* returns a new MCN block       */
short get_seqno();                        /* returns a sequence number     */
mcn_block find_mcn_block();               /* searches for an MCN block     */
void mulcast();                           /* multicasts a packet           */
void mulcast_in();                        /* used in debugging without network */
void rem_mcn_block();                     /* remove a MCN block            */
void add_rec();                           /* add a receiver to local list  */
void rem_rec();                           /* remove a receiver from local list */
void local_invoke();                      /* invokes multicasts on each VM */
void mcn_ack();                           /* acknowledge multicast         */

/************************ globals **********************************************/
static mcn_block mcn_block_head = NULL;           /* pointer to mcn-list     */
static seq myseqno;                               /* sequence number         */
```

```
/*****************************************************************************/
/*                        user called functions                              */
/*****************************************************************************/

/*****************************************************************************/
/* mcn_create : creates an MCN                                               */
/*****************************************************************************/
void
mcn_create(netw)
int netw;                               /* pointer to MCN capability         */
{
  int packet;                           /* packet created                    */

                          /* initialise MCN cap. */

  ((struct mcn_st * )netw)->netvm = my_vm;
  ((struct mcn_st * )netw)->mcnseqn = get_seqno();

                          /* create and initialise packet */

  packet =    (int) mem_alloc(INVOCATION_HEADER_SIZE, RTS_OWN);

  ((pach)packet) ->type = MCN_CREATE;
  ((pach) packet)->net = *(struct mcn_st *) netw;
  ((pach) packet) ->size = INVOCATION_HEADER_SIZE;

  mulcast((pach)packet);

  return;
}

/*****************************************************************************/
/* mcn_destroy : destroys a mcn                                              */
/*****************************************************************************/
void
mcn_destroy(netw)
int netw;                               /* pointer to MCN capability         */
{
  int packet;                           /* packet created                    */

                          /* create and initialise packet */

  (pach)mem_alloc(INVOCATION_HEADER_SIZE, RTS_OWN);

  ((pach) packet)->net = *(struct mcn_st *) netw;
  ((pach) packet) ->type = MCN_DESTROY;
  ((pach) packet) ->size = INVOCATION_HEADER_SIZE;

  mulcast((pach)packet);

  return;
}
```

```
/*************************************************************************/
/* mcn_update: updates the process group membership                      */
/*************************************************************************/

void
mcn_update(packet)
int packet;                             /* packet created by user code */
{
  mulcast((pach) packet);
}

/*************************************************************************/
/* mcn_invoke : multicasts over a MCN                                    */
/*************************************************************************/

void
mcn_invoke(packet)
int packet;                             /* invocation packet created by user code */
{

  if (find_mcn_block(((pach)packet)->net)->numrecs ==0)  /* no such MCN */
    perror("no such mcn in mcn_invoke");

  ((pach) packet)->type = MCN_INVOKE;              /* could be done by compiler */

  mulcast((pach)packet);
return;
}

/************************** MULCAST ***********************************/
/* Mulcast uses the packet to effect local and remote actions          */
/* Locally: it spawns a process to take the actions                    */
/* It also multicasts the packet to all VMs. These also spawn          */
/* processes to take the actions.                                      */
/* The processes spawned to take the actions: rmcn_create,etc..        */
/* call mcn_ack to acknowledge their completion.                       */
/* Mulcast (meanwhile) has a loop containing a P, which executes       */
/* each time an ack returns, until all acks have returned.             */


void
mulcast(packet)
pach packet;
{
  int n, rest;          /* number of bytes sent and size of rest of message */
  char *addr;           /* pointer to remaining message                     */
  remd rem;             /* remote message descriptor                        */
  int numacks;          /* number of ACKs required                          */
  int i;
```

```
                                /* create semaphore */

P(rem_count);
P(rem_mutex);
rem = rem_free;
rem_free = rem_free->next;
V(rem_mutex);
                                /* find the number of acks */

switch(packet->type){
  case MCN_CREATE:
  case MCN_DESTROY:
  case MCN_UPDATE:
  case MCN_SEND:
    numacks = 2;                /* note: these are for reliability          */
  break;                        /* reliability is not provided elsewhere    */

  case MCN_CALL:
  {  mcn_block blk;             /* data-block for the given mcn             */
    blk = find_mcn_block(packet->net);
    numacks = blk->numrecs;
  }
  break;

}
                                /* initialise for return */

packet->origin = my_vm;
packet ->rem = rem;
rem->ph = packet;
rem->wait = create_sem(0);


                                /* local action */

switch (packet->type){
  case MCN_CREATE:   Activate(Spawn(rmcn_create,RTS_OWN,1,packet));break;
  case MCN_DESTROY:  Activate(Spawn(rmcn_destroy,RTS_OWN,1,packet));break;
  case MCN_UPDATE:   Activate(Spawn(rmcn_update,RTS_OWN,1,packet));break;
  case MCN_INVOKE:   Activate(Spawn(rmcn_invoke,RTS_OWN,1,packet));break;
  }

                                /* send packet */

rest = packet->size;
addr = (char *) packet;
while (rest>0) {
  n = write(mcast_send_sock,addr,rest);
  if (n<0) perror("mulcast");
  rest -= n;
  addr += n;
}
```

```
                                /* wait for replies */

    for(i=0;i<numacks;i++){
        P(rem->wait);
        }
    kill_sem(rem->wait);

                                /* free semaphore */

    P(rem_mutex);
    rem->next = rem_free;
    rem_free =rem;
    V(rem_mutex);
    V(rem_count);
}
/*****************************************************************************/
/* functions that are used to act on the contents of multicast packets       */
/* received by VMs.                                                          */
/*****************************************************************************/

/*****************************************************************************/
/* rmcn_update: adds or removes a receiver                                   */
/*****************************************************************************/
void
rmcn_update(packet)
pach packet;
{
    struct crep_st *reply;      /* acknowledgement packet          */
    mcn_updb upblk;             /* update block found in packet    */
    int size;                   /* size of ack packet              */

    upblk = (mcn_updb) packet;

                                /* join or remove receiver */

    if(upblk->type== MCN_JOIN){
        mcn_block mcnblk;
        if ((mcnblk= find_mcn_block(packet->net)) != NULL) {
            mcnblk->numrecs++;                          /* increment total receivers */
            if(upblk->receiver.vm==my_vm)               /* receiver is local         */
                add_rec(upblk->receiver,packet->net);
            }
        else perror("No such network to be joined0);
        }
    else{                                               /* type is MCN_QUIT          */
        mcn_block mcnblk;
        if ((mcnblk= find_mcn_block(packet->net)) != NULL) {
            mcnblk->numrecs--;                  /* decrements total receivers */
            if(upblk->receiver.vm==my_vm)
                rem_rec(upblk->receiver,packet->net);
        }
        else perror("No such network to be joined0);
    }
```

```
                        /* send acknowledgement packet */

    size = sizeof(struct crep_st) + sizeof(struct mcn_updb_st);
    reply = (struct crep_st *) mem_alloc(size,RTS_OWN);
    reply ->ph.rem = packet->rem;
    mcn_ack(packet->origin,ACK_MCN_UPDATE,&reply->ph,size);

    mfree((daddr)reply);
    Kill(cur_proc,FALSE);
}


/*************************************************************************/
/* rmcn_invoke                                                           */
/*************************************************************************/
void
rmcn_invoke(packet)
pach packet;
{
  invb invblk;              /* invocation block in packet */
  mcn_block mcnblk;         /* mcn_block for a receiver   */
  int size;                 /* size of ack packet         */
  struct crep_st *reply;    /* reply                      */

  invblk = (invb) packet;

  if((mcnblk =find_mcn_block(packet->net)) == NULL)
    perror("No network to invoke0);

                        /* if a CALL invoke receivers before acking */

  if (invblk->type == MCN_CALL) local_invoke(mcnblk,invblk);

                        /* send acknowledgement */

  size = sizeof(struct crep_st) + sizeof(struct mcn_updb_st);
  reply = (struct crep_st *) mem_alloc(size,RTS_OWN);
  reply ->ph.rem = packet->rem;
  mcn_ack(packet->origin,ACK_MCN_UPDATE,&reply->ph,size);

  if(invblk->type == MCN_SEND) local_invoke(mcnblk,invblk);

  mfree((daddr)packet);
  Kill(cur_proc,FALSE);
}
```

```
/*****************************************************************************/
/* rmcn_create: creates an MCN                                               */
/*****************************************************************************/
void
rmcn_create(packet)
pach packet;
{
  mcn_block newblock;                  /* new mcn_block          */
  int size;                            /* size of reply          */
  struct crep_st *reply;               /* reply block */

                          /* create and initialise new mcn-block */

  newblock = get_mcn_block();
  newblock->id = packet->net;

                          /* send ACK */

  size = sizeof(struct crep_st) + sizeof(struct mcn_updb_st);
  reply = (struct crep_st *) mem_alloc(size,RTS_OWN);
  reply ->ph.rem = packet->rem;
  mcn_ack(packet->origin,ACK_MCN_CREATE,&reply->ph,size);

  mfree((daddr)reply);
  Kill(cur_proc,FALSE);

}

/*****************************************************************************/
/* rmcn_destroy : destroys a MCN                                             */
/*****************************************************************************/
void
rmcn_destroy(packet)
pach packet;
{
  int size;                    /* size of reply */
  struct crep_st *reply;       /* reply block */

                          /* destroy MCN */

  rem_mcn_block(packet->net);

                          /* send acknowledgement */

  size = sizeof(struct crep_st) + sizeof(INVOCATION_HEADER_SIZE);
  reply = (struct crep_st *) mem_alloc(size,RTS_OWN);
  reply ->ph.rem = packet->rem;
  mcn_ack(packet->origin,ACK_MCN_DESTROY,&reply->ph,size);

  mfree((daddr)reply);
  Kill(cur_proc,FALSE);
}
```

```
/*****************************************************************************/
/* local_invoke : invokes local receivers for an mcn                         */
/*****************************************************************************/
void
local_invoke(mcnblk,invblk)
mcn_block mcnblk;          /* mcn_block                */
invb invblk;               /* invocation block         */
{
  rec_block recblk;                    /* receiver block          */

  recblk = mcnblk->recs;               /*  first receiver         */

  while (recblk !=NULL){
    invblk->opc = recblk->opn;   /* insert op. cap. in invocation block */

      /* set type to ensure proper ack. behaviour */

  if (invblk->type == MCN_SEND) invblk->type = SEND_IN;
  if (invblk->type == MCN_CALL) invblk->type = CALL_IN;

    invoke(invblk);                    /* invoke the operation    */
    recblk = recblk->nextrec;          /* next receiver                */
  }  /* end of while */
}

/*****************************************************************************/
/* mcn_ack: sends acknowledgement packet                                     */
/*****************************************************************************/
void
mcn_ack(dest,type,packet,size)
  tindex dest;             /* destination VM           */
  enum ms_type type;       /* message type                    */
  pach packet;             /* packet                   */
  int size;                /* size of packet                  */
{
  if (packet->origin == my_vm) {       /* locally initiated       */
    V(packet->rem->wait);
    }

  else{                    /* remotely initiated       */
    if(!(net_known(dest))) {           /* if unknown VM, get its socket */
      struct num_st mn;
      pach mph;

      mn.num =dest;
      mph = remote(SRX_VM,REQ_FINDVM,(pach)&mn,sizeof(mn));
      net_conn(dest,((struct saddr_st *)mph)->addr);
      }

    net_send(dest,type,packet,size);   /* send ack packet         */
  }
}
```

```
/*****************************************************************************/
/*                utilities                                                  */
/*****************************************************************************/

/************************** mcn_block-utilities ******************************/


/******************** get_mcn_block ******************************************/
/* returns a new initialised mcn_block inserted at the front of the mcn_list */

mcn_block
get_mcn_block()
{
mcn_block newblock;          /* new mcn_block returned by function */

                             /* get new block and initialise */

  newblock = (mcn_block) mem_alloc(sizeof(struct mcn_block_st),RTS_OWN);
  newblock->numrecs = 0;
  newblock->numlocal =0;
  newblock->recs =  NULL;

                             /* add block to list of blocks */

  newblock->nextmcn =  mcn_block_head;
  mcn_block_head =  newblock;

  return(newblock);
}


/************************* rem_mcn_block *************************************/
/* removes from the mcn_list the block corresponding to the given mcn id     */
void
rem_mcn_block(net)
struct mcn_st net;                   /* capability of mcn to be removed */
{
  mcn_block blk;                     /* two mcn_blocks used for search */
  mcn_block prev;

                             /* search for mcn block for the given MCN */

  blk = mcn_block_head;
  prev =mcn_block_head;
  while (blk !=NULL){
    if ((blk->id.netvm == net.netvm) && (blk->id.mcnseqn == net.mcnseqn) ){

                             /* MCN found */

      if (blk == mcn_block_head)   mcn_block_head = blk->nextmcn;
      else prev->nextmcn = blk->nextmcn;
      mem_free((int)blk);
      return;
      }
```

```
    else{           /* not found */
        prev = blk;
        blk = blk->nextmcn;
    }
  }
  return;
}
/************************ find_mcn_block *********************************/
/* returns a pointer to the mcn_block with the given mcn_id                */

mcn_block
find_mcn_block(net)
struct mcn_st net;                      /* capability of mcn we seek */
{
  mcn_block blk;                        /* mcn_block used for search */

  blk = mcn_block_head;
  blk->id =net;
  while (blk !=NULL){
    if ((blk->id.netvm == net.netvm) && (blk->id.mcnseqn == net.mcnseqn) )
      return(blk);                                  /* found it */
    else blk = blk->nextmcn ;
    }
  return(NULL);
}
/*************************************************************************/
/* general utilities                                                       */
/*************************************************************************/

/************************** printdb ***********************************/
/* prints out the entire mcn database */
void
printdb()
{
mcn_block blk;                          /* blocks used for search of database */
rec_block rec;

printf(" MCN_DB");
blk = mcn_block_head;

  while(blk != NULL){                   /* search all mcns */
    printf("mcn: vm: %d seq: %d0,blk->id.netvm,blk->id.mcnseqn);
    printf("numrecs: %d numlocal: %d 0,blk->numrecs,blk->numlocal);
    rec = blk->recs;
    while(rec != NULL){     /* search each individual mcn db */
      printf("rec: vm: %d op_index: %d seqn: %d0,rec->opn.vm,
              rec->opn.oper_index,rec->opn.seqn);
      rec=rec->nextrec;
    }
    blk=blk->nextmcn;
  }
}
```

```
/******************** receiver manipulation ************************/

/*************** add_rec : adds a receiver to the local receiver list *****/
void
add_rec(receiver,net)
struct ocap_st receiver;        /* new receiver                          */
struct mcn_st net;              /* mcn                        */
{
  mcn_block mcnblk;             /* mcn-block of mcn           */
  rec_block recblk;             /* new rec_block for receiver            */

  if ((mcnblk =find_mcn_block(net)) != NULL){          /* mcn exists */
                                /* create and initialise receiver block */
    recblk = (rec_block) mem_alloc(sizeof(struct rec_block_st),RTS_OWN);

    recblk->opn = receiver;
    recblk->nextrec = mcnblk->recs;
    mcnblk->recs = recblk;
    mcnblk->numlocal++;
  }
}
/************** rem_rec: remove local receiver from list ***************/
void
rem_rec(receiver,net)
struct ocap_st receiver;        /* receiver to be deleted */
struct mcn_st net;              /* mcn from which to delete it */
{
  rec_block recblk,prevrec;     /* receiver blocks used to search list */
  mcn_block mcnblk;             /* mcn block for given mcn */

                                /* find mcn_block */
  if ((mcnblk = find_mcn_block(net)) == NULL) return;

                                /* search for receiver */
  recblk = mcnblk->recs;
  prevrec = recblk;
  while (recblk !=NULL){    /* if found delete */

    if ((recblk->opn.vm == receiver.vm) && (recblk->opn.seqn == receiver.seqn)
        && (recblk->opn.oper_index == receiver.oper_index )){
      if (recblk ==mcnblk->recs) mcnblk->recs = recblk->nextrec;
      else prevrec->nextrec=recblk->nextrec;
      mcnblk->numlocal--;
      mem_free((int)recblk);
      return;
    }
    else{
       prevrec = recblk;
       recblk = recblk->nextrec;
    }
  } /* while */
  return;
}
```

```
/*****************************************************************************/
/************************ file:  bc.c ****************************************/
/*****************************************************************************/
/* a set of utility functions for multicasting over the Suns                 */

#include "rts.h"

int get_first_rec_sock();       /* gets a socket with unspecified port     */
int get_rec_sock();             /* gets a  receiving socket for given port */
int get_send_sock();            /* gets a sending socket of given port     */
u_short get_port();             /* gets to port number for a given socket  */


/********************** init_all_mcasts  ***********************************/
/* executed by initial VM to get multicast receiving sock and reserve port */

void
init_all_mcasts()
{
  mcast_rec_sock = get_first_rec_sock();                /* get rec. sock      */
  mcast_port = get_port(mcast_rec_sock);                /* find the port number */
  mcast_send_sock = get_send_sock(mcast_port);          /* get sending socket */
}

/*************************** init_mcasts ***************************************/
/* executed by later VMs to get a multicast receiving socket for given port */

void
init_mcasts()
{
  mcast_rec_sock = get_rec_sock(mcast_port);            /* get rec socket: given port */
  mcast_send_sock = get_send_sock(mcast_port);          /* get send socket */
}

/***************************** get_first_rec_sock ******************************/
/* reserves a port and returns a socket bound to it                          */

int
get_first_rec_sock()
{
int sock;                       /* socket                                    */
SOCK_ADDR_IN mysockname;        /* socket name                               */
int name_len;                   /* length of sender_address   */

                                /* set up socket for reading */

  sock = socket(AF_INET,SOCK_DGRAM,0);
  if (sock<0){
    perror("opening dg socket");
    exit(1);
  }
```

```
                        /* create name with wildcards */

    bzero((char *) &mysockname,ADDR_SIZE_IN);
    mysockname.sin_family = AF_INET;
    mysockname.sin_addr.s_addr = INADDR_ANY;
    if (bind(sock, (SOCK_ADDR_IN *) &mysockname, ADDR_SIZE_IN ) < 0 ){
      perror("binding first dg socket");
      exit(1);
    }

                        /* get port value */

      name_len = ADDR_SIZE_IN;
      if (getsockname(sock, (SOCK_ADDR_IN *) &mysockname, &name_len) < 0) {
        perror("getting socket name");
        exit(1);
      }

  return(sock);
}

/****************************** get_rec_sock ******************************/
/* get_rec_sock returns a broadcasting receiving socket bound to the given port     */

int
get_rec_sock(port)
u_short port;                           /* port number                          */
{
int sock;                               /* socket                   */
SOCK_ADDR_IN mysockname;                /* socket name                          */
int name_len;                           /* length of sender_address  */

                        /* set up socket for reading */

  sock = socket(AF_INET,SOCK_DGRAM,0);
  if (sock<0){
    perror("opening dg socket");
    exit(1);
  }

                        /* create name with wildcards */

  bzero( (char *) &mysockname,ADDR_SIZE_IN);
  mysockname.sin_family = AF_INET;
  mysockname.sin_addr.s_addr = INADDR_ANY;
  mysockname.sin_port = port;
  if (bind(sock, (SOCK_ADDR_IN *) &mysockname, ADDR_SIZE_IN ) < 0 ){
    perror("binding subsequent dg socket");
    exit(1);
  }

  return(sock);
}
```

```
/*************************** get_send_sock ***********************************/
/* get_send_sock returns a broadcast sending socket bound to the given port */


int
get_send_sock(port)
u_short port;
{
  int sock;                                  /* socket       */
  SOCK_ADDR_IN recsockname;                  /* address of socket          */
  struct hostent *host ;                                  /* host of sender*/
  char my_name[MAXHOSTNAMELEN];                           /* name of host               */

                                /* set up socket for sending */

  sock = socket(AF_INET,SOCK_DGRAM,0);
  if (sock <0){
    perror("opening dg socket");
    exit(1);
  }

                                /* get host  */

   gethostname(my_name,MAXHOSTNAMELEN);
   host = gethostbyname(my_name);

                                /* set up socketaddress */

   bzero( (char *) &recsockname,ADDR_SIZE_IN);
   bcopy((char *) host->h_addr,(char* ) &recsockname.sin_addr,host->h_length);
   recsockname.sin_family = AF_INET;
   recsockname.sin_port = port;
   recsockname.sin_addr.s_impno = (u_char) BC_WILDCARD ;

                                /* connect socket  */

  if (connect( sock, (SOCK_ADDR_IN *) &recsockname, ADDR_SIZE_IN))
    perror("connecting sock: get send sock [%d],my_vm");

  return(sock);
}
```

```
/******************************** get_port ********************************/
/* get_port  returns the port number for the given socket */

u_short
get_port(sock)
int sock;                               /* socket whose port number is sought */
{
  int name_len;                         /* length of socket name      */
  SOCK_ADDR_IN mysockname;                              /* socket name                */
  u_short port;                         /* port                       */

  name_len = ADDR_SIZE_IN;
  if (getsockname(sock, (SOCK_ADDR_IN *) &mysockname, &name_len) <0){
        perror("getting socket name");
        exit(1);
        }
  port = htons(mysockname.sin_port);

  return(port);
}
```

References

[Ahamad85] Ahamad,M. and Bernstein,A.J.,"Multicast Communication in UNIX 4.2BSD" Proceedings of the 5th International Conference on Distributed Computing Systems,pp.80-87, May 85.

[Andrews83] Andrews, G.R., and Schneider, F.B., "Concepts and Notations for Concurrent Programming", Computing Surveys, Vol. 15, No.1, March 1983.

[Andrews88] Andrews, G.R., et al., "An Overview of the SR Language and Implementation", A.C.M. Trans. on Programming Languages, Jan. 88.

[Andrews87] Andrews, G.R., and Olsson, R.A.,"Revised Report on the SR Programming Language". Dept. of Computer Science, University of Arizona. (TR 87-27)

[Atkins89] Atkins, M.S., Haftevani, G.B., and Luk, W.S., "An Efficient Kernel Level Dependable Multicast Protocol for Distributed Systems", 8th Symposium on Reliable Distributed Systems, Seattle, Oct. 89.

[Bernstein83] Bernstein, P. and Goodman,N. "The Failure and Recovery Problem for Distributed Databases" Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, Montreal, 83.

[Birman 87] Birman, K.P., and Joseph, T.A., "Reliable Communication in the Presence of Failures", A.C.M. Trans. on Computer Systems, Vol.5, No.1,87.

[Birkhard87] Birkhard, W.A., Martin, B.E., and Paris, J.F. "The Gemini Replicated File System Test-bed" Proc. of the Third International Conference on Data Engineering, L.A., California, 87

[Brich Hansen77] Brich Hansen, P, "The Architecture of Concurrent Processes", Prentice Hall, Eaglewood Cliffs, N.J., 77.

[Charlesworth87] Charlesworth, A., "The Multiway Rendezvous", ACM Transactions on Programming Languages and Systems, Vol 9, No. 2, July 1987, pp. 350-366.

[Cheriton85] Cheriton, D.R. and Zwanepoel, W, "Distributed Process Groups in the V Kernel", A.C.M. Transactions on Computer Systems, Vol. 3, No.2, May 1985.

[Cheriton 87] Cheriton, D.R. and Stumm, M. "The Multi-Satellite Star : Structuring Parallel Computations for a Workstation Cluster", Dept. of Computer Science, Stanford University.,87.

[Feldman] Feldman, J.A., "High Level Programming in distributed computing", Commun ACM 22,6 , June 79.

[Good79] Good, D.I., Cohen, R.M., and Keeton-Williams, J,
"Principles of proving concurrent programs in Gypsy". In Proc. 6th ACM Symp. Principles of Programming Languages", ACM, New York, 79.

[Hoare78] Hoare, C.A.R., "Communicating Sequential Processes", Commun. ACM, Aug. 78.

[Leffler83] Leffler, S, Joy, W., and Fabry, R.,"4.2BSD Interprocess Communication
Primer", Computer Systems Research Group, Univ. of Cal., Berkeley, July 83.

[Martin87] Martin, B., Bergan C., and Russ, B. "PARPC : A System for Parallel Procedure
Calls", Proc. of the 1987 International Conference on Parallel Processing, Penn. State Univ. Press.

[Navaratnam87] Navaratnam, S., "Reliable Group Communications in Distributed Systems", Master's Thesis, University of British Columbia, 1987.

[Navaratnam88] Navaratnam,S., Chanson, S., and Neufeld, G., "Reliable Group Communication in Distributed Systems", 8th International Conference on Distributed Computer Systems, June 88.

[Olsson86] Olsson, R.A., "Issues in Distributed Programming Languages: The Evolution of SR", Dept. of Computer Science, University of Arizona. (TR 86-21)

[USDD81] U.S.D.D. U.S. Department of Defence . "Programming Language Ada: Reference Manual, vol 106, Lecture Notes in Computer Science" Springer-Verlag, New York,81.

[Wirth77] Wirth, N, "Modula: A language for modular multi-programming", Softw. Pract. Exper. 7 , 77.