

A DOMAIN STRUCTURE FOR DISTRIBUTED COMPUTER SYSTEMS

by

L. Casey and N. Shelness

Dept. of Computer Science, Edinburgh University, EH9 3JZ.

ABSTRACT.

The successful implementation of generalized multiple computer systems will require attention both to the form of physical architecture and to the choice and implementation of a suitable systems environment in which to construct and run applications. This paper argues for the use of a multi-computer physical architecture in preference to a multi-processor architecture, and for dynamic distribution of functions and control as opposed to static allocation of functions and hierarchical control. A systems environment which is based on a domain structure is then described. The domain structure restricts sharing of items. This alleviates the main problem in implementing a capability mechanism to support domains in a system without shared memory, which is that a central table of capabilities is required. It also makes the management of the non shared items easier since they can be required only at one computer at a time. Essential sharing is also handled without central control but at the cost of some complexity. Considerable attention is paid to the handling of interdomain jumps as they provide the opportunity for the dynamic allocation of functions. It is conjectured that the resulting system would be capable of smooth expansion in size from one to twenty five computers. In operation it would exhibit dynamic load balancing as well as having the protection advantages of domain structure.

1. INTRODUCTION

Though it is fast becoming a cliché in papers such as this, developments in LSI technology have changed and are continuing to change the basic economic assumptions and framework in which computing systems are designed, constructed and sold. These changes are twofold. Firstly, a massive increase in component density has freed a number of cheap levels of component inter-connection, namely the printed circuit component carrier and the printed circuit mother board, or backplane, for use in the inter-connection of multiple processors and memories rather than the intra-connection of a single processor or memory. Secondly, the extremely low cost of mass produced LSI components may make economic the construction of a range of computing systems by combining a variable number (on the order of from one to twenty five) of identical processors and memories, rather than requiring the construction of a number of different processors and memories in different component technologies as has been the case historically. There are also potential advantages of reliability and incremental growth in this approach.

If we are to construct systems containing from one to twenty five processors, there are two basic architectural forms that we can employ. We can construct multi-processor systems in which multiple processors are coupled to a logically common memory by some form of inter-connection medium (figure 1), or we can construct multi-computer systems in which each of the multiple

processors is connected directly to its own private memory forming multiple computers which are then inter-connected by some form of inter-connection medium (figure 2). It is of course possible to build hybrids of these two forms. Each of the two forms introduces a separate set of problems and overheads. Hybrid forms tend to possess both sets of problems and overheads.

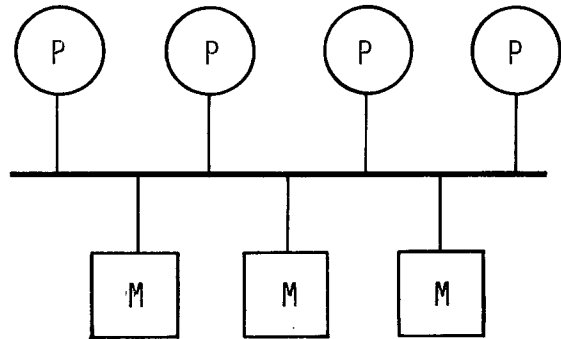


Figure 1

This work was supported in part by a study award from the New Zealand DSIR.

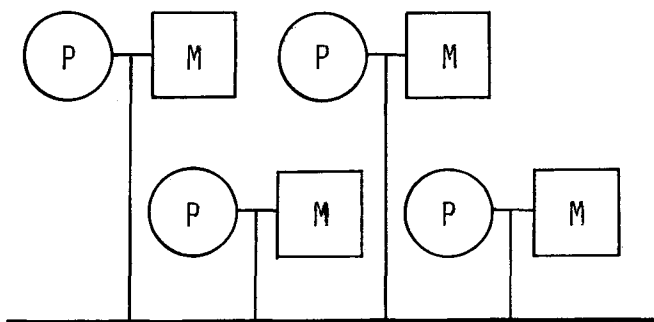


Figure 2

In multi-processor systems, each of the processors is contending for access to a single common memory. While it is likely that sufficient memory bandwidth can be achieved through interleaving (incorporating a number of autonomous memories capable of parallel operation), this merely transfers contention from the memory itself to the inter-connection medium. This bottleneck can only be overcome by the inclusion of additional paths between processors and memories. This will tend to take one of two forms: porting in which processors and memories are fully inter-connected or some form of crossbar switch. Porting is just not viable, even with the modest number of processors and memories envisaged. E.g. twenty processors and twenty memories would require the provision of four hundred address and data highways. Neither do we believe that a crossbar switch would be viable. For in the case of twenty processors and twenty memories, we would still require twenty processor highways and twenty memory highways and a switch with 400 switching elements. Even with LSI technology, such a switch is unlikely to be either small or cheap because of the large number of external connections it would require.

It is this problem of contention that mitigates against the construction of cheap and simple multi-processor systems, that has caused us to turn our attention to multi-computer architectures. In multi-computer system, memory contention has been eliminated but at the cost of introducing a number of other problems of which the most fundamental is assignment of function. In a multi-processor system, code can be executed or data accessed from any location in the logically common memory. In a multi-computer system, the code and data required by a computation must reside in the memory of the computer on which the computation is being performed. Therefore when we assign a function to a particular computer we also place the code and data required by the function in the computer's memory. There are two forms that function assignment may take. It may be static with functions and hence code and data assigned permanently to a particular computer or it may be dynamic with functions and hence code and data migrating between computers. The former

approach has tended to be employed in systems employing hierarchical clusters of function specific computers. For example in a large main-frame, a number of specialized computers perform tape, disc and communications control on behalf of the CPU. A similar approach, though involving many more computers, is employed in a distributed, process plant control system. Such systems can in no sense be considered general. They are tailored, at great expense, for a single un-altering application. It is difficult to see how to apply static function assignment in more general systems, nor is it clear that it would be desirable.

It is an absolute requirement, if we are producing a range of systems, that software be constructed independent of the number of computers, so as to be able to exploit the entire range of system configurations. It is possible, at least in theory, to achieve static assignment through compiling or linkage-editing the software for each configuration, but this still leaves us with an inability to exploit the inherent redundancy of the system in case of failure except through re-compiling or re-linking. In addition unless we have an extremely stable temporal relationship between functions, some computers will be overloaded at times that others are idle. As our reasons for rejecting a multi-processor system were essentially those of poor performance, we would not be willing to accept poor performance in a multi-computer system due to poor load distribution.

For all of these reasons we have attempted to design a system environment which is independent of the number of computers in a particular configuration and which allows for dynamic function assignment under de-centralized control. This system environment is the distributed domain structure. Its features and the means by which it is mapped onto the physical hardware are now described.

2. THE SYSTEMS ENVIRONMENT.

Interest in domains as a basis for the structuring of systems is growing [LIND76] and a number of domain structure proposals have appeared in the literature. Of these, several [SPIE73, SPIE74, COSS74, WULF74] relate in important aspects to the domain structure we are proposing in this paper.

2.1 The components.

The components of our systems environment are domains, segments and virtual processors. A domain consists of a group of segments. A segment is a vector of information items with identical access characteristics. An important property of a segment is the degree to which it may be shared. A segment may be part of many domains at any one time (shared simultaneously), it may be restricted to being part of one domain at a time (shared serially) or it may be only ever part of a single domain (unshared).

A virtual processor is that entity which, when multiplexed onto a physical processor executes instructions and accesses data from segments in a domain. The information required to represent and manage a virtual processor [DENN76] is kept in a special segment called the virtual processor segment. A virtual processor segment can only be part of one domain at a time. It is not directly accessible

in that domain. But apart from this special segment all the segments in the domain can be accessed. These are the only segments that can be accessed while the virtual processor is executing in the domain (that is while the virtual processor is part of the domain). These segments constitute the virtual memory of the virtual processor while it is in the domain. (Note that we are being slightly peculiar in our use of the term 'domain'. For us it denotes all the components of a calculation at a particular instant. The more usual definition is to denote all the segments directly accessible to a computation at a given instant. This is the same as is denoted by (segmented) virtual memory).

A computation is effected by the progress of a virtual processor through a number of domains. As a computation proceeds, different resources and services are required. The virtual processor transfers to other domains by invoking an inter-domain jump so as to be able to access the appropriate code and data.

There is a philosophical question concerning the nature of an interdomain jump. This question arises when some of the segments that constitute the new domain were also part of the old domain. In these circumstances a processor could be said merely to be altering the segments that make up its current domain rather than transferring to a new domain. We have adopted the philosophy of Needham who states that a change of domain is associated with a change of segment from which instructions are being executed [NEED72]. This approach puts an emphasis on code segments as identifying domains and leads to the following prescription for the contents of a domain, c.f. Spier [SPIE74].

A domain consists of a code segment, an optional public data segment, local data segments and, when code from the code segment is being executed, a virtual processor segment. The code segment will consist of one or a number of related routines. It is simultaneously shareable by any number of virtual processors. The public data segment is accessible by all virtual processors that execute the code segment of the domain. The public data structure is never accessible from another domain. When the public data structure exists we can view the routines of the code segment as managing the data structure. All access to the data structure is mediated by the routines of the domain. The code segment and public data segment are the only segments that can be simultaneously part of many domains. We call this simultaneously shareable part of a domain the generic domain. (It is called a procedure in Hydra terminology [WULF74].)

If the generic domain does not have any (write) shared data then we call the domains it belongs to pure code domains. Because of their affinity to the monitors introduced by Hoare and Brinch Hansen [HOAR74.HANS73], domains that have a public data segment are called monitors.

Local data segments are segments private to a virtual processor. They may be accessible to that virtual processor only when it is executing in a particular domain or they may be passed as parameters between domains so that the virtual processor can access them in a number of domains. Thus local data segments are either unshared or shared serially.

2.2 Summary of systems environment.

We can summarise the type of systems environment we think suitable for a distributed system as follows. The basic unit of storage organisation and access is the segment and segments are organised into domains. Figure 3 depicts the components of a domain. A domain derives its identity from its generic domain which defines the services it provides (by the nature of its code segment) and, if it is a monitor, the resources that it controls. There may exist many domains that share a generic domain but the remaining segments of these domains are, at any particular instant, unique. A domain may be looked upon as the incarnation of a generic domain by the addition of segments for the storage of data related to a particular virtual processor.

3. THE KERNEL

The function of a kernel is to map the systems environment onto the physical architecture. For a distributed system we speak of there being a kernel at each site (computer) and it is the combined actions of these kernels that map the systems environment onto the physical architecture. The kernels of a distributed system have the obvious functions of managing memory and handling interrupts and driving devices. They have also, in our system, to maintain the domain structure and transmit segments between sites. The kernels communicate with each other to perform these functions.

3.1 The implementation of domains.

The generally accepted method of implementing domains is to use capabilities [LIND76]. The capabilities for each segment that constitutes a domain are kept in a list unique to each domain, called the c-list. The current c-list of a virtual processor is the c-list of the domain in which it is executing. Addresses within the virtual memory of the virtual processor are specified as an offset into the current c-list to point to a segment followed by an offset within the segment. An interdomain jump causes the current c-list of the virtual processor to be altered to the c-list of the new domain.

3.2 Capability organization.

In a system that has common primary memory the capability mechanism is almost always implemented using indirection via a central table. This facilitates the handling of multiple capabilities for a segment. The central table holds the actual addresses of segments; capabilities in c-lists are pointers or hash keys into this table. If the location of a segment is changed then only the entry in the central table need be altered, rather than every capability for that segment.

In a distributed system it is not practical to use a single central table (nor indeed to split the table statically over all sites) as every use of a capability would involve a message to the site holding the address of the segment and a return message with the address. However multiple capabilities for a segment are only a necessity for segments that are simultaneously shareable. For the rest a single transfer only capability [GRAH72] can be used. The use of transfer only capabilities simplifies domain management in a distributed system.

THE COMPONENTS OF A DOMAIN

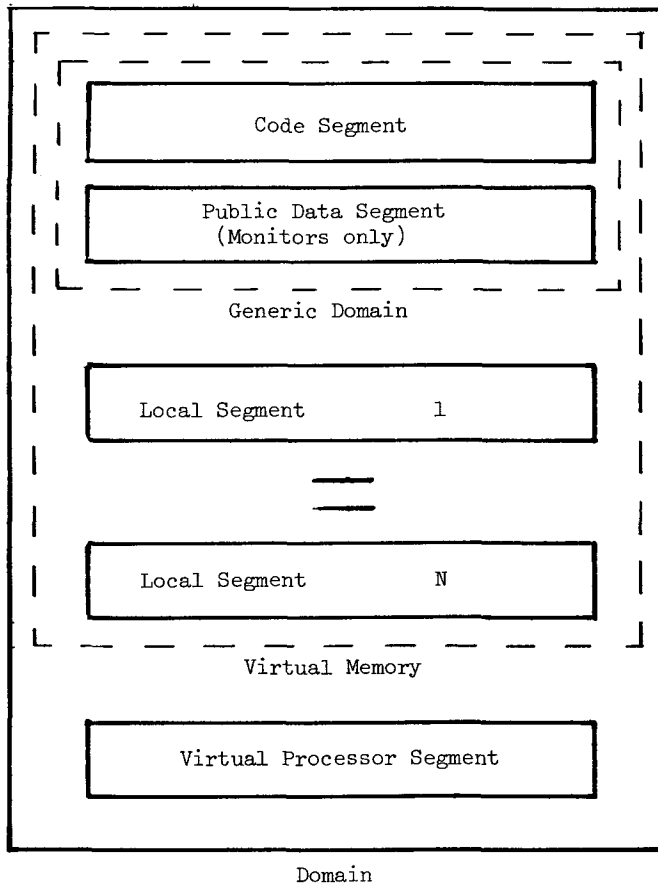


Figure 3

Figure 4 gives the form of a c-list for a domain. Local segments are either unique to a particular domain or, if they are passed as parameters, shared serially between domains. Therefore local segments are named by transfer only capabilities so that there exists exactly one capability for each segment. This capability is a type of pointer specifying, inter alia, the site where the segment resides. Virtual processor segments are shared serially between all domains of the virtual processor and so likewise can be named by a single transfer only capability. However as generic domains are (potentially) simultaneously shareable between all incarnations of the domain, a capability for a generic domain must be of a more general type. There can be many of these capabilities in existence for a single generic domain and they cannot specify directly which site the generic domain resides at because of the difficulty of updating them all should the generic domain change site. In our distributed system these capabilities, which we call global object capabilities, are interpreted by kernels which cooperate together to locate the

C-LIST FOR A DOMAIN

	Transfer only capability for Local Segment N

2	Transfer only capability for Local Segment 1
1	Global object capability for Generic Domain
	Transfer only capability for Virtual Processor Segment

Figure 4

generic domain.

3.3 Locating generic domain

The exact method of locating generic domains in a distributed system depends very much on the properties of the communication subsystem connecting the sites. For example, one could use some sort of broadcast or loop system that has associative recognition of addresses similar to that in DCS [FARB72a]. Each kernel would load the associative memory of the message reception unit with the names of all generic domains at its site. In this way the kernel would receive all control messages addressed to the generic domains resident at its site. We have surveyed the types of communication subsystem available and have developed alternative global object management mechanisms when associative recognition of addresses is not employed [CASE77].

4. THE INTERDOMAIN JUMP.

Just as the key to distributing processes in a system such as DCS lies in distributing the passing of interprocess messages [FARB72b], the key to distributing a domain based system lies in distributing the interdomain jump. Our approach to the mechanization of the distributed interdomain jump is as follows.

When a virtual processor initiates an interdomain jump it forces a trap to the kernel at the site at which it is currently executing. This kernel starts to build a c-list for the new domain. It enters the name of the new generic domain into

the c-list and transfers from the old to the new c-list the capabilities for the virtual processor segment and any local segments being passed as parameters. All the capabilities required for the new domain may be known (as in the case of an interdomain return jump) or specifications for new temporary segments may have to be extracted from a template [WULF74] associated with the generic domain. In either case the new c-list is transferred to the kernel of the site at which the generic domain resides (located by the global object management mechanism). At this site the decision is made as to where the new domain should be located. (We call this decision process the 'best' site calculation and describe it in more detail in section 4.2). The c-list is transmitted to this 'best' site whereupon the kernel there gathers to its site all the segments that make up the domain. Since it has the c-list it knows the location of the requisite local segments and the virtual processor segment from their transfer only capabilities. It therefore sends control messages to the kernels at these sites to transmit the segments to it. As the segments arrive the kernel alters their capabilities to reflect their new location. The fetching of the generic domain is more complex and is described below. When all the segments have arrived and any temporary ones, if required, have been created, the c-list is marked as being a valid c-list and the domain is queued for execution.

4.1 Movement of generic domains.

When a kernel receives a request to send off a local segment or virtual processor segment that is resident at its site, it can comply immediately because the segment cannot possibly be in use at its site. This is not the case with segments of a generic domain. The generic domain may be in use, or it may form part of a domain that is ready to run or is awaiting the arrival of other segments in order to run. Further there can be more than one outstanding request for a generic domain. Correct management of generic domains requires that they are not removed prematurely from a site and that all sites that have requested them will get access to them within a reasonable period of time. The strategies employed to ensure this goal depend on whether the domain is a monitor or a pure code domain.

4.1.1 Pure code domains.

In the case of a pure code domain a kernel can dispatch immediately the code segment that constitutes the generic domain and retain a copy at its own site. If space is tight the copy can be deleted as soon as it is no longer needed, otherwise it can be kept till another incarnation of the generic domain is required at that site. In this case there are no problems with premature removal of the generic domain. A scheme whereby each kernel accepts just one request for each generic domain and passes any other subsequent requests on to the site that made the original request can result in a chain of waiting sites being built up when demand for the generic domain is heavy. It can be shown that under certain general conditions the generic domain gets to every site in the chain. These conditions relate to the repeating of request for generic domains, the way global object management

is performed, and the relative speed of movement of segments and request for segments. They are discussed further in [CASE77] .

4.1.2 Monitors

For monitors a similar chain of sites waiting for the generic domain can be built up. But since copying of monitors is not practical each site must keep the monitor until it has finished using it. The problem is to make sure that a site does actually finish with a monitor and does not keep it in use indefinitely, thereby locking out all other sites. To this end we make a rule that a kernel ceases to perform the 'best' site calculation for a domain once it (the kernel) has been requested to send the generic domain to another site. Instead the kernel sends the c-list for the domain to the site on whose behalf it has reserved the generic domain. At this site the c-list is queued to await the arrival of the generic domain. Since part of the validation procedure of all new c-lists is the 'best' site calculation and since this can only be done by the kernel at the site where the generic domain resides, eventually there will be no more valid c-lists for the domain (assuming that a domain's execution is of finite duration) and the generic domain can be shifted safely to the next site in the chain of waiting sites.

4.2 The 'best' site calculation.

The optimum time to shift a computation to another site is at the time of an interdomain jump. It is optimum in the sense that a minimum context is involved; only the virtual processor segment and parameter segments form part of both the old and the new domain. Some or all of the remaining segments required for the new domain may already be at the new site. The efficacy of our distributed system lies in the determination of where the new domain will be located, that is, in the 'best' site calculation. If domains are moved around the system too frequently then computations will be subject to extra transmission time delays and the communication subsystem may become overloaded. If the movement is too infrequent then the loading at different sites can become seriously unbalanced; some sites idle while others are choked with work.

To make sensible decisions about where to send a domain, a kernel needs information about the status of other sites. There are a number of ways this information can be obtained: by a bidding procedure [FARB72b] , by exchanging status at regular intervals, or by appending status information to every message sent between kernels.

We describe a two stage procedure which is successfully employed to perform the 'best' site calculation in a simulation of a distributed system based on our model (See section 6.1). The simulated system exhibits stable behaviour over a range of system sizes varying from one to nine sites (the largest system we can simulate). In this simulated system status information is propagated by adding it to each message. Each kernel maintains a table of free memory and outstanding work at every site. Given suitable hardware this function could be performed by the communication interface units and not by the kernel.

4.2.1 The initial calculation.

The initial calculation at the generic domain site considers (normally) only three possible sites as candidates for 'best' site:

- 1) the site of the generic domain segments
- 2) the site of the virtual processor segment and any local segments that are parameters (since they were part of the domain being left, the parameters will be at the same site as the virtual processor segment)
- 3) the site of any other local segments which, since they were all last used in the previous incarnation of the domain being entered, will all be at one site.

If the generic domain cannot move because it is 'tied down' to drive a peripheral attached to a particular site then that site is chosen as 'best' site. Otherwise the basic policy is for the kernel at the site of the generic domain to consider the total size, at each distinct site, of segments for the domain. The site with the largest aggregate size is chosen as the 'best' site. Hence, when all the segments are at the same site, that site is chosen, and otherwise transmission on the communication links is minimized (assuming a homogeneous communication subsystem).

However minimization of communication bandwidth requirements to set up the domain is only one of a number of criteria that are considered. Before the aggregate sizes are compared there are a number of biases that are applied to them.

- 1) Since the calculation takes place at the generic domain site it is possible to know how many other virtual processors have incarnations of the same generic domain. If the generic domain were to go to another site then either these other domain incarnations might have to follow (if they are re-entered again) or the generic domain might have to return to the original site. Hence the size of the generic domain (the code and possible public segments) is biased upwards by a factor representing its outstanding work, favouring the generic domain site as the choice of 'best' site when the comparison of aggregate segment sizes is done. This encourages domains with a common generic domain to be all at the same site.
- 2) The load, that is the number of ready to run domains, at each of the three possible sites is taken into account. If the domain is sent to a lightly loaded site it will be executed quicker than at a heavily loaded site. And if transmission times are less than average execution times, substantial advantages accrue by choosing an idle site rather than a site with even one other domain to run. Thus sizes are biased downwards by a factor representing the overall load at a site. This encourages load balancing.
- 3) A kernel is likely to spend a lot of its time on memory management if it has very little free memory left. Not only should an attempt be made to balance loads at sites but also an attempt should be made to ensure that a site does not run out of memory space when other sites have plenty available. So the sizes of segments are biased upwards proportional to the

amount of free memory at their site. This will encourage migration away from sites with little free space.

- 4) A good part of a computation may consist of repeated calls between the same pair of domains, or, more generally, a repeated sequence of calls. If the domains reside at different sites and are all larger in size than the virtual processor segment then the virtual processor segment could continually travel between the sites concerned. Obviously the computation would proceed faster if all the domains were at one site. If a history stack is kept with the virtual processor segment it is easy to determine if interdomain jumps are limited to a tight cluster of domains. If the size of the virtual processor segment is biased upwards when this clustering occurs then eventually all the domains involved will come to the same site. In particular, a domain that calls a 'tied down' peripheral handler domain will tend to migrate to the site of the handler domain. (Of course, if a computation uses two or more peripherals controlled from different sites then the virtual processor segment, together with parameter segments, is doomed to traverse back and forward between sites).

The effects of the first stage calculation can be summarized as follows:

It effects at most three sites and it tends to balance the load and free memory of these sites.

It tends to aggregate all incarnations of a generic domain at one site.

It tends to localize domains that are used together to one site.

Other things being equal, it minimizes the load on the communication subsystem.

4.2.2 The second stage calculation:

The first stage calculation we have described above only takes into consideration a maximum of three sites, and nominates one of them 'best' site. If just this calculation were used a site that was completely idle would have no way of breaking into the circle of the elect. However we make it a rule that each 'best' site can decide whether or not it wants to run the domain. The criteria we use for accepting or rejecting the c-list at the 'best' site helps spread the load over all sites in the system.

Assuming that the generic domain is not 'tied down' at its site, the 'best' site kernel scans the table it has of the loads at other sites. If some other site is substantially less busy than itself (and, according to the free space table, has the space to accommodate the domain) then that site will be nominated as 'best' site. If the kernel does not find a substantially less busy site then it checks to see if its own site has enough space not only for the incoming segments but also for any temporary local segments that have to be created. If it does have the space it accepts the c-list, requests any segments not already at its site to be sent to it, creates any temporary segments, and eventually schedules the domain ready to run.

If the kernel does not have enough space then a second, less critical, scan of the load and free

memory tables is made to find a site that does have enough space. If no such site can be found, or the domain is 'tied down' to the present site, then the c-list is placed in a queue of c-lists waiting for space. Should sites in the distributed system reach this stage there is a danger of deadlock occurring. Some overall control on the number of 'active' virtual processors is required. This need not be a kernel function; for example it could be delegated to the command interpreter monitor domain.

The effects of the second stage calculation can be summarized as follows:

At low loading of the system it has little effect, there is no point in moving computations to idle sites if they leave an idle site behind.

At moderate loading of the system the policy is to try to keep all sites busy.

At heavy loading of the system the concern is more with finding space for domains.

5. FUNCTIONAL SPECIALIZATION

Frequently multiple computer systems are proposed where each computer carries out a specific function [FOST72, COLO76]. In contrast, except for the limitations imposed by specific peripherals being attached to specific sites, all functions can be carried out at all sites in our system. The distribution of functions is determined dynamically by the domain management mechanism in response to the applied load. Because functions are not tied to specific computers, down time in the event of a computer failure is minimized. The faulty site need only be isolated, not repaired or replaced, before the (less powerful) system is operational again.

However in our distributed system any site may be made into a functionally specialized site by having it contain a single generic domain that never moves from the site. The kernel at such a site need not be of the same form as that at a general site, so long as it presents the same external appearance to the rest of the system. We envisage this feature being useful for handling front end processors and back end processors, and for large systems where the failure of one specialized processor does not mean the failure of the whole system because there is more than one of each type of specialized processor.

6. RESEARCH

6.1 Simulation studies.

To help establish the viability of our model we have performed some computer simulation studies. The simulation program is written in Simula and allows us to simulate distributed systems with up to 9 sites before the 128K addressing limit of the DEC System 10 on which it is run, is reached. The characteristics of the applied workload were derived from those of our local time sharing system, suitably modified to reflect the lower power assumed of each site. The control mechanisms described before are fully detailed in the simulation and, as mentioned before stable behaviour was achieved (though not before a mechanism was added to avert

deadlock if the system became overloaded). A simulation run produces a wealth of statistics including the average response time to a terminal command. For a constant load per site this response time decreases slightly as the number of sites in the simulated system goes up (provided no resource goes into saturation). This is in accordance with queueing theory predictions of the effect of multiple servers.

Perhaps the most significant result, after the demonstration of stable behaviour, of the simulation studies is the indication they give that a bandwidth of around 3MHz for the communication subsystem is adequate to support up to 9 sites without the communication subsystem becoming saturated. Thus we conjecture that the available or imminent communication subsystems should be able to support systems with up to about 25 sites. More details of the simulation program and results are given in [CASE77].

Work is continuing on the simulation program to evaluate the effects of different types of communication subsystem, of refinements to the domain management strategy and of alterations in the status information content and handling. Further we would like to quantify the benefits of the very dynamic load balancing we allow compared with more static schemes.

6.2 Construction of a prototype.

We are also undertaking the construction of a three site prototype. Each site will consist of a minicomputer with another attached cpu to act as inter-site interface unit. The power of this interface unit will allow us to experiment with different kinds of communication subsystem as well as with moving some of the kernel functions to the interface unit. For example we hope that low level storage management might be performed by the interface unit so that transmission and reception of segments can be affected without interrupting the main processor too often.

Operation of the prototype will involve us writing domain structured software. We have not considered in this paper the question of domain sizes or how a programmer might go about constructing a program for execution in our system. We have ad hoc ideas on making domains large enough (and domain switching overheads small enough) so that interdomain jumps do not predominate over the performance of useful work. But obviously widespread adoption of the sort of architectural developments we are proposing is predicated on there being matching high level language developments. We hope to do some investigation of suitable language constructs.

7. CONCLUSION.

To achieve a domain orientated distributed system we proceeded in two stages. First we examined the infra-structure of domains to determine what components are not shared. These can be easily managed by the use of transfer only capabilities. Then we detailed how the remaining shared items, the generic domains, can be managed without the use of shared memory.

With these management schemes we can achieve the protection associated with domains [LIND76] in a distributed system. In the resulting domain

based system, fine grain load balancing is possible because a load balancing decision is taken at every interdomain jump. Load balancing at the time of interdomain jumps involves minimum context being shifted between sites. Our distributed system automatically shares code when primary memory space is short but otherwise multiple copies (of pure code domains) can exist so as to make the best use of available processing power. It could be employed in an interactive system for students and in many turn-key systems where there is not a requirement for a computation to be able to access a large amount of memory at once. It would be expandable without reprogramming as the requirements of its users grew.

SPIE74

SPIER M.J., HASTINGS T.N. and CUTLER D.N. 'A storage mapping technique for the implementation of protective domains' Software - Practice and Experience Vol 4 No 3 1974 pp 215-230.

WULF74

WULF W.A. et al. 'HYDRA: the kernel of a multiprocessor operating system' CACM Vol 17 No 6 Jun. 1974 pp 337-345.

BIBLIOGRAPHY:

- CASE77 CASEY L.M. 'Computer structures for distributed systems' Ph.D. Thesis, CST-2-77, Computer Science Dept. Edinburgh University, Jan. 1977.
- COLO76 COLON F.C. et al. 'Coupling small computers for performance enhancement' AFIPS NCC Vol 45 1976 pp 755-764.
- COSS74 COSSERAT D.C. 'A data model based on the capability protection mechanism' IRIA International Workshop on Protection in O.S. (Paris) August 1974 pp 35-53.
- DENN76 DENNING P.J. 'Fault tolerant operating systems' Computing Surveys Vol 8 No 4 Dec. 1976 pp 359-389.
- FARB72a FARBER D.J. & LARSON K.C. 'The system architecture of the distributed computer system - the communications system' Symp. Computer-Communications Networks and Teletraffic (New York) Apr. 1972 pp 21-27.
- FARB72b FARBER D.J. & LARSON K.C. 'The structure of a distributed computing system - software' Proc. Symp. Computer-Communications Networks and Teletraffic (New York) Apr. 1972 pp 539-545.
- FOST72 FOSTER C.C. 'A view of computer architecture' CACM Vol 15 No 7 Jul. 1972 pp 557-565.
- GRAH72 GRAHAM G.S. & DENNING P.J. 'Protection - principles and practice' AFIPS SJCC Vol 40 1972 pp 417-429.
- HANS73 HANSEN P.B. 'Operating System Principles' Prentice-Hall (Englewood Cliffs) 1973.
- HOAR74 HOARE C.A.R. 'Monitors: an operating system structuring concept' CACM Vol 17 No 10 Oct. 1974 pp 549-557.
- LIND76 LINDEN T.A. 'Operating system structures to support security and reliable software' Computing Surveys, Vol 8 No 4, Dec. 1976 pp
- NEED72 NEEDHAM R.M. 'Protection systems and protection implementations' Proc. AFIPS FJCC Vol 41 1972 pp 571-578.
- SPIE73 SPIER M.J. 'A model implementation for protective domains' Int. Journal of Computer and Information Sciences Vol 2 No 3 Sep. 1973 pp 201-229.