# Design and Implementation of an Object-Orientated 64-bit Single Address Space Microkernel

*Kevin Murray, Tim Wilkinson, Peter Osmon – SARC, City University*
*Ashley Saulsbury – Swedish Institute of Computer Science*
*Tom Stiemerling, Paul Kelly – Imperial College*

**Abstract**

In the mid eighties, the System Architecture Research Centre at City University developed a message-passing, UNIX compliant micro kernel (*Meshix*) for our own scalable distributed memory architecture (*Topsy*). Over the last two years we have been engaged in a research programme aimed at learning from this experience, and developing a new operating system based on these lessons. The result is the *Angel* microkernel. This paper sets out the lessons we have learnt from *Meshix*, how this has influenced the design of *Angel* and outlines our current design of *Angel* and its C++ implementation. We will also describe our future plans and hopes for *Angel*, and the lessons that we have learnt from the design and implementation process.

## 1 Introduction

Almost all modern operating systems are being designed using microkernels [1, 2]. The microkernel architecture can be said to encompass good software engineering practice: small code "units" that are insulated from each other together with a minimal amount of critical code (the microkernel). They also introduce an "open system architecture" due to the ease with which additional services can be provided and used.

Virtually without exception, however, microkernel architectures use message passing as the basis of communication, implementing the client-server paradigm upon this using remote procedure call techniques (RPC). Message passing offers an apparently ideal structuring mechanism — it isolates one "unit" from another, requires only a minimal microkernel (message passing and process control), and allows extra services to be provided simply by registering the service which then receives and processes the messages.

*Meshix* is typical of such microkernel based, message passing operating systems and was developed several years ago [3]. Over the last few years we have been looking at its structure and performance in a very critical manner to decide how to improve upon it — in essence we are trying to evaluate whether or not the message passing microkernel is as good as it seems. This has shown there are a number of issues that have not yet been addressed by most current message passing microkernel architectures, or which have only been addressed with limited success or requiring complex restructuring of the system. It is to tackle these issues that *Angel* has been designed.

This paper will outline the issues behind the the design of *Angel*, in addition to the actual design and implementation of *Angel*. Although *Angel* moves away from a message passing structure, we will show how its most important use — RPC — can be very efficiently implemented in *Angel* using LRPC techniques, and how it maintains the essential isolation of the systems into protected "units". We will also mention some of the other work that we are applying to *Angel*, principally in the areas of fault tolerance and scalable I/O systems.

## 2 Shortcomings of Meshix

The original goal of *Meshix* and the *Topsy* architecture was to produce a scalable, parallel multiprocessor[1]. To help achieve this goal a dedicated point to point network with custom, virtual cut-through routing chips were developed which supported a raw bandwidth of 10 Mbytes/sec. To a reasonable extent the scalability goal has been achieved. Unfortunately its communications performance is only about 100 K/sec, as seen by a user process, and there is limited support for parallel programming. The reason is largely to do with two factors: the nature of microkernels, and the adoption of UNIX. The adoption of System Vr3 UNIX as the primary interface to *Meshix* means there is no support for parallel programs, forces the use of UNIX heavyweight processes and limits the IPC mechanisms.

### 2.1 Microkernel

In a microkernel architecture, there is an inherent performance loss caused by information exchange between services and clients. Typically a client collects the information it needs in its own private address space, independently of the server. When it wishes to exchange information with a server (probably to obtain some service), it first creates a message containing this information and then requests the microkernel to convey this to the server. Usually this involves several context switches and some data copying, remapping or cross-machine transferral, all of which are known to be costly actions.

The Chorus group [4] (among others), has done much work to overcome this. The methods used include replacing context-dependent addresses with unique addresses, so speeding up message delivery whilst reducing security, combining mutually trusted servers into a single address space (and hence protection domain), so reducing context switches, and by placing all of the IPC management into the microkernel. In addition they use the lightweight RPC optimisation developed for the DEC Firefly system [5] to improve the speed of RPCs. All of these modifications have required non-trivial alterations to the operating system's structure and increased the complexity of the system. It is our belief that communication (or more generally co-operation), despite the above optimisations, is still slower than desirable and more complex an operation than need be.

We performed a set of detailed measurements of the speed of the *Meshix* communication system [6, 7] to help identify the causes of communications costs, to better understand these costs, to find ways to reduce or eliminate them and to help develop a simpler mechanism. This study concluded that many, though not all, of the costs are an inherent fact of using message passing in multiple protection domains and the numerous context switches and data copying or remapping that this caused. During this study, it also became apparent that much, often unmeasured, time was spent in *preparing* the data for transfer.

As a comparison, we modelled the behaviour of a very simple distributed shared memory (DSM [8]) scheme with an amount of hardware assistance comparable with the current *Meshix* message passing system. The conclusion was that this would easily outperform the current message passing system, used in *Meshix*. This lead us to believe that the shared memory paradigm should be at the base of future parallel operating systems, replacing the message passing that is currently in use. This is in agreement with several other researchers and manufacturers [9, 10, 11].

### 2.2 UNIX

The UNIX process model provides every process with the illusion of a complete computer for itself. But whenever the process tries to access anything other than the processor or the data currently residing in memory, it may suffer a context switch as another process is given the chance to run. A context switch involves the exchange of a large amount of information beyond the processor's context including the memory map and extensive operating system information. This is called a heavyweight process model.

1. It should be noted that this is a somewhat different goal from some other systems which seek to produce a distributed computing systems.

Since *Meshix* provides a UNIX programming model, the process model it implements is that of UNIX: distinct heavyweight processes. The heavyweight process model is costly [12] due to its extensive amounts of state, and this reduces the benefits of writing programs in parallel, although this may be overcome to some extent using threads packages. Unfortunately, these threads are not real first class objects in the operating system and certain system operations for one thread affect others (eg. blocking). Even then it is still nearly impossible to share these threads between processes, unlike such systems as Psyche [13], to achieve the required flexibility (such as cost effective load balancing). For efficient parallel programming a lightweight, flexible and extensible process model is needed. This is one where changing from one thread to another is exceptionally cheap, where the actions of one thread do not necessarily affect another and where exchanging processes should also be cheap.

### 2.3 Support for Parallel Programs

*Meshix* provides no synchronisation primitives other than those implicit in message passing. Any others are built using messages. This means that if co-operating processes need to use synchronisation other than messaging, it is slow and limited by the characteristics of the messaging system. In a scalable parallel machine, real parallel programs will require efficient and varied synchronisation mechanisms (e.g. barrier synchronisation) and better support for them must be provided.

Additionally, much work has been done on load balancing in many systems and support for this is important to parallel programs since it is necessary for them to distribute their work over the machine efficiently. With a general purpose computer, the load on various parts of the system can change, and to maintain the efficiency of the applications running on such a system, it is necessary to re-allocate work between available processors. This is at the heart of load balancing, and naturally in a scalable parallel system it is important that, at the very least, there is support to allow this to be done.

## 3 The Angel Design and Single Address Space Architectures

From our experience with *Meshix*, and as a result of our studies both of *Meshix* and other systems, it was decided that *Angel* should have the following characteristics:

- ▶ It should not support message passing, but use shared memory to support a single address space. This decision was taken to tackle two problems: the lack of speed of the message passing model, as outlined above, and to improve the context switch time by removing the need to flush various caches, which has been noted as the most costly part of the context switch operation.
- ▶ It should provide a protection mechanism which is not part of the process. This decision was taken to allow a far greater flexibility in protection scheme, and allow more than one process to operate within the same domain to increase speed when necessary. It is also a logical step following the above point in which we had divested address translation from the process.
- ▶ It should allow processes to be informed of the actions of the operating system on their behalf. This decision is aimed at allowing threads within a process to become first class citizens of the operating system and to allow the process to partake in scheduling decisions that may affect it.
- ▶ It should use a minimal microkernel. None of our studies of *Meshix* showed a flaw in the microkernel design; in fact many of our experiences with *Meshix* have shown how vital the microkernel design is. The problems identified have been tackled by the above alterations to the architecture, so *Angel* remains a microkernel. However, as the implementation section will show, there is even less in the *Angel* kernel than in many other microkernels.

The following sections will outline the main characteristics of the *Angel* design.

### 3.1 SASA

Most importantly, *Angel* is a Single Address Space Architecture (SASA), like such systems as Multics [14], Psyche [15], and Opal [16]. A SASA is one in which there is only one address space shared by the entire system (all the processes, servers and the kernel). This is in contrast to the UNIX approach whereby every process has its own unique address space. This has several benefits: it improves and simplifies data sharing, helps cache performance, and blurs the distinction between shared memory and distributed memory machines. The SASA is maintained between multiple processors using shared memory techniques. The SASA has become feasible with the appearance of large address space processors [17], enabling many processes to consume addresses from the same range without exhausting the supply.

This address space is managed as persistent objects (contiguous groups of pages). Not only does this remove the need for an explicit "file system" interface (with a different namespace and explicit system calls) but greatly simplifies the storage of complex structures, databases, etc.

## 3.2 Protection Issues

In Unix one process is protected from another by the use of separate address spaces. In a SASA all processes share the same address space, so separating protection from address translation, and hence a new scheme is needed to provide protection. This has also caused some researchers to propose alterations to the traditional memory and protection hardware with the addition of new hardware support [18]. The protection scheme must define two areas within which it works: the unit of protection, and the method used to specify and meet access requirements.

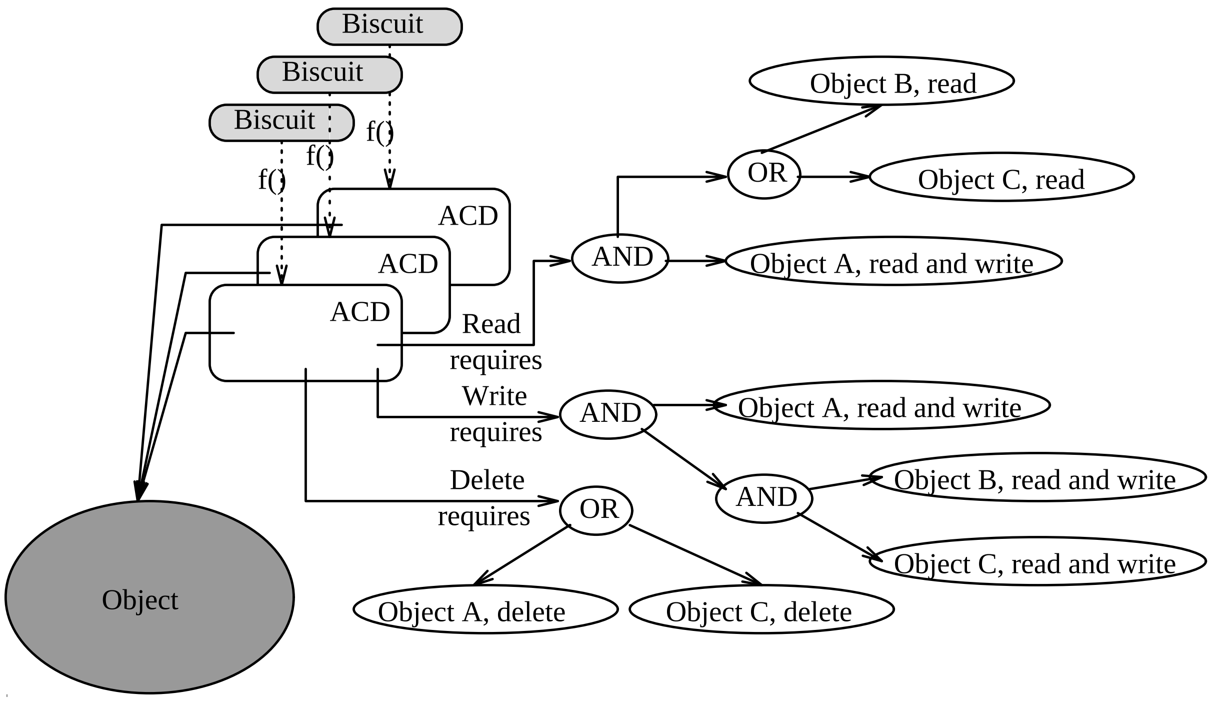

Figure 1: The structure of ACDs and biscuits

In *Angel*, protection is provided on objects which consist of one or more pages. Objects cannot overlap, nor may they be contained within other objects. A critical server in *Angel* is the object manager which is responsible for allocating addresses to objects and for validating access to objects. For every object, the object manager associated with it one or more *Access Control Descriptors (ACD)* which describe the other objects that must be accessible before this object may be accessed.

An example of this structure is shown in figure 1 in which one object has three ACDs associated with it; one of them has part of its permissions tree show. In this example, to gain write access to the object, the process must already have read and write access to objects A, B and C.

When an object is created, or when a new ACD is associated with an object, the object manager gives out a *biscuit* from which it can reliably identify the valid corresponding ACD. Conceptually there is only one biscuit per ACD despite processes being free to duplicate this as frequently as they like. When a process wishes to access an object, it presents this *biscuit* to the object manager. The *biscuit* is then used to determine if the process possesses the necessary objects to resolve the requested object. Consequently, the system does *not* have the concept of user identifiers. However, it is trivial to implement such a system by creating an object whose sole purpose is to act as a "user id" for access checking.

## 3.3 Support for Parallel Programs

*Angel* supports first class threads and uses upcalls for inter-process and kernel-process signalling (see section 4.2). Their purpose is to allow process to be informed external events in which they have declared an interest, eg. the release of locks, the arrival of new work, a page fault or a pending time slice. By passing such information onto the process, the process is able to make its own decisions on what to run and to take remedial action (e.g. release a lock) when decisions are imposed upon it.

The DSM supported by *Angel* allows the construction of locks such as spin locks with reasonable efficiency. When combined with the upcall mechanism, it is simple for a thread to "sleep" and be "woken" at some later date. This provides asynchronous systems, not possible with shared memory alone.

The SASA that lies at the heart of *Angel* makes implementing load balancing trivial. As all processes and threads on all processors exist within a single address space that also contains all the necessary kernel information, moving a process or thread from one physical processor to another simply involves loading the processor context for the thread into the new processor. The DSM that implements the SASA will then move any necessary data as it is accessed. The design of *Angel* as it stands will not automatically load balance work for a process, but this can easily be provided through library routines.

## 3.4 The Angel Model

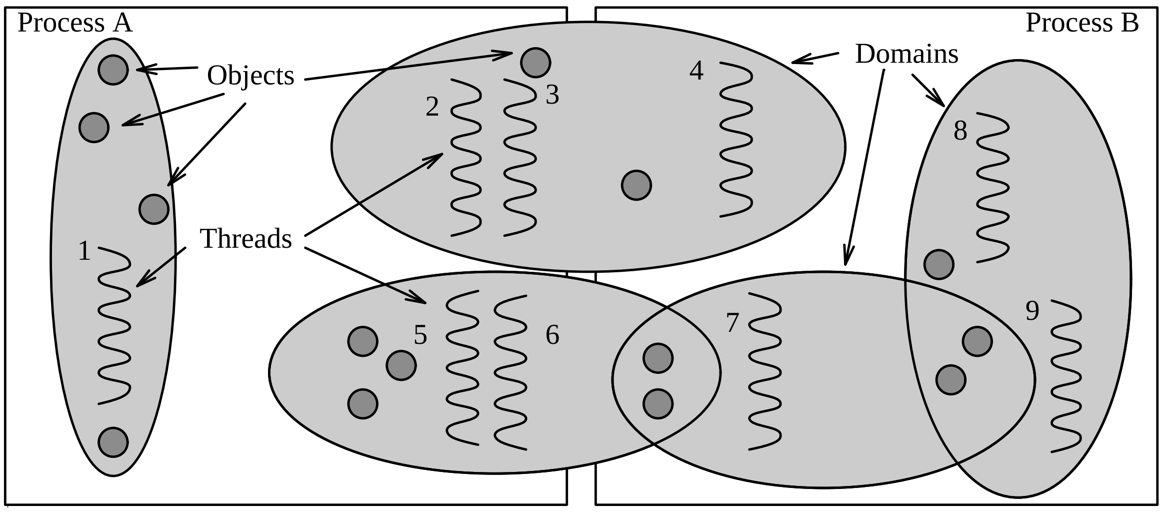

Figure 2: The Angel Process Model

Figure 2 shows how the above points are combined into the process model that *Angel* supports. Within any process there may be one or more threads. Threads may run in their own domain, as is the case with thread 1 in the diagram. This allows several threads in the same process to be protected from each other. Alternatively several threads may share a protection domain, potentially between different processes, as is the case with threads 2 to 4. Where threads do not wish to share a protection domain for security or trust reasons, they may have some mutually shared objects, as is the case with the remaining threads.

### 3.5 Fault tolerance

It is possible to build a scalable, efficient fault tolerance scheme in a SASA based operating system. This relies on the unification of resources to simplify the implementation, and the augmentation of the DSM system in order to capture the data interactions necessary to make distributed checkpoints. Unlike other schemes [19, 20] where excessive DSM activity can result in large number of checkpoints being made, we allow general data sharing without checkpoints, instead utilising the DSM state information to determine which data depends on which. This allows distributed checkpoints to be made which will only effect processes which are interacting, and also allows the DSM mechanism to be reused for checkpointing data to other machines' memories. Experiments indicate this costs only an additional 10% on an applications execution time. A full description can be found in [21].

## 4 The Angel implementation

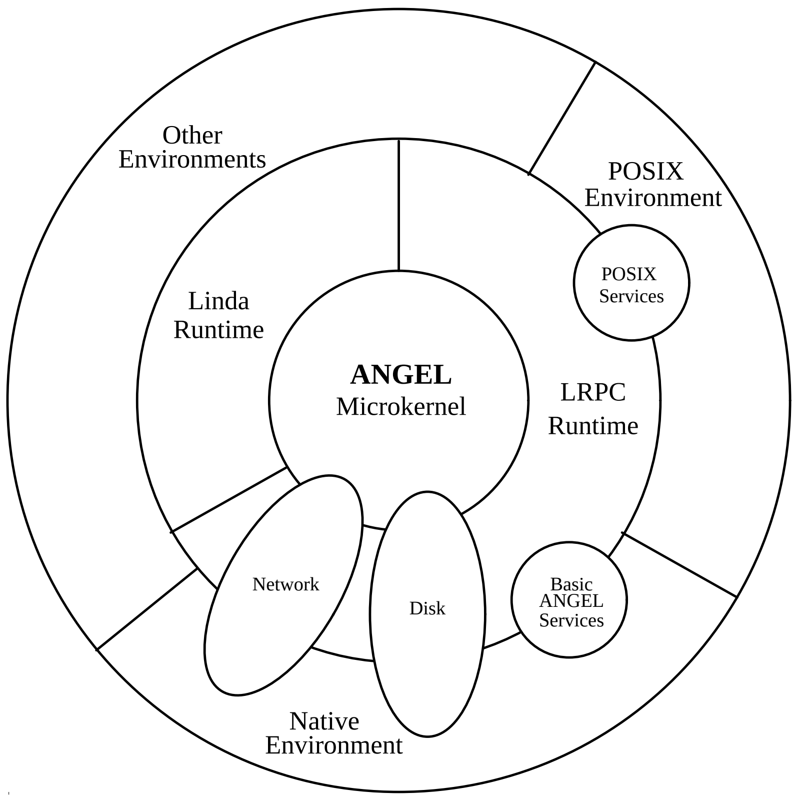

Figure 3: Angel structure

Figure 3 depicts the general structure of the *Angel* operating system. This structure has few differences from more conventional, message passing microkernel designs. However, the use of a single address space and shared memory for communications has significantly simplified the microkernel. Currently, the implementation consists of 2,500 lines of C++ code and 1,000 lines of include files. This constitutes the virtual memory, the distributed shared memory and the device management systems but not the device drivers themselves.

At time of writing, we have completed initial work on the microkernel and client/server communication system. The microkernel provides two major services:

1. Persistent virtual memory, and

2. Virtual processor management.

The client/server communications are implemented using "lightweight" RPCs.

## 4.1 Persistent virtual memory

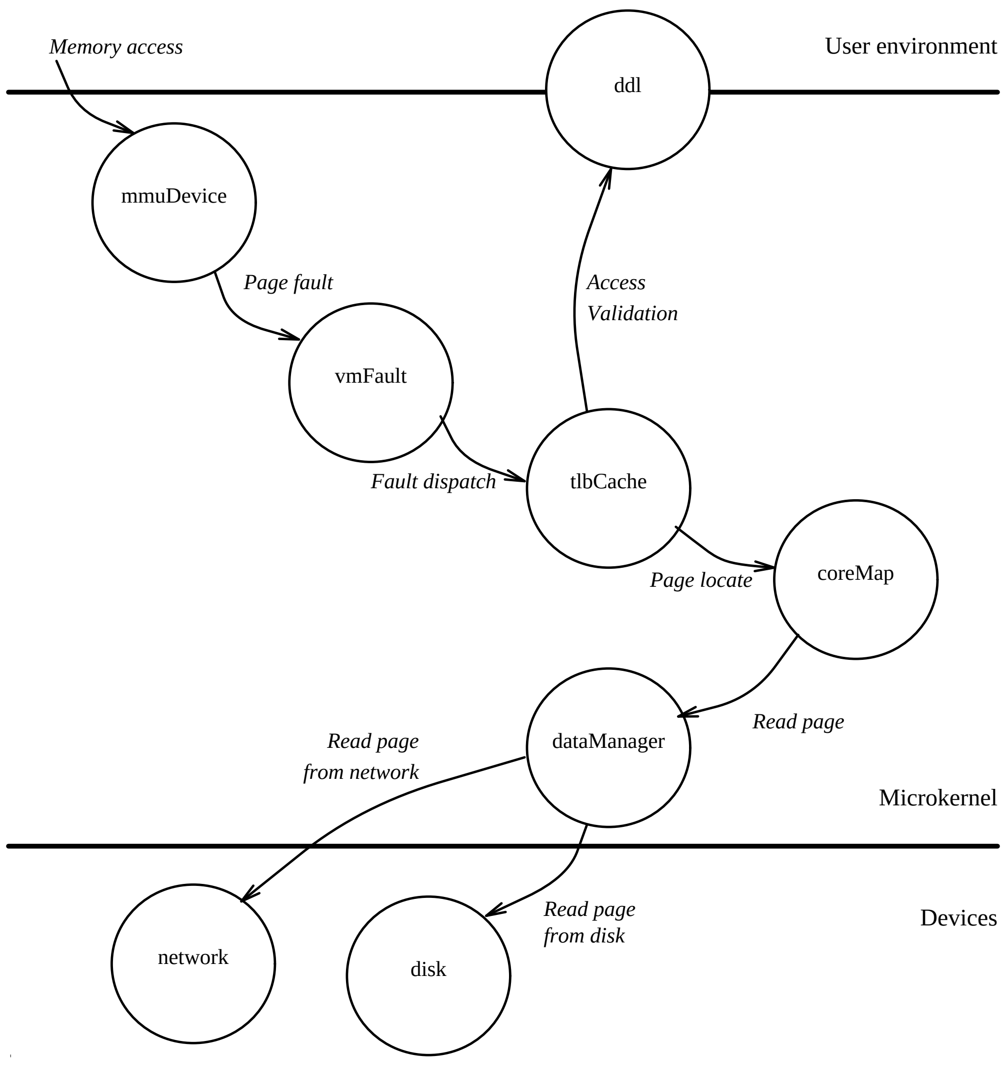

Figure 4: Object orientated VM system

The virtual memory (VM) system is the heart of *Angel* since it supports the persistent single address space. The single address space nature of the VM enables some simplifications of the structure to be made but the persistence introduces other complications.

Figure 4 demonstates the events in the VM system initiated by a page fault. Page faults are generated by the *mmuDevice*, a processor dependent object responsible for collecting all necessary information regarding the fault, and passed into the main, processor independent code, *vmFault*. This determines whether the fault is legitimate (user attempts to access supervisor data are caught here) and requests the relevant page from the *tlbCache*. The *tlbCache* first determines whether the access was to an object accessible by the virtual processor (using the *ddl* which describes this relationship). If it was not, a fault condition is returned. If it was, the accessed address is used to form a *pageID*, an unique identifier for data in time and space. These pageIDs are used to support data aliasing[2] necessary for the copy-on-write mechanism. The pageID is then used by the *coreMap* to locate the relevant data. The local coreMap memory is first searched for data corresponding to this ID. If found, the page is returned for installation by the mmuDevice. If not found, the coreMap allocates an empty core page and request the *dataManager* to find the data and install it. The dataManager does this by consulting both the *network* (which provides the DSM system) and the *disk*.

Several point in this VM system are worth special mention. First, the *ddl* is held in the user environment, so allowing it to be treated as any other object, sharable via the DSM and swappable onto disk. This prevents consumption of valuable kernel resources and allows the user to easily determine attributes of their envionment without the microkernel's assistance[3]. Second, the devices (network and disk) are accessed through an LRPC interface (see section 4.3). This allows them to be installed externally from the microkernel if desired although the LRPC mechanism will automatically optimise this interface when this is not the case. Currently, these devices are contained within the kernel but we are planning to make them loadable kernel-level device drivers in order to improve modularity and flexibility without compromising performance. Third, at various stages, the VM system may reach a point where it cannot continue immediately. This may be the result of a fatal error (eg. an access is made to an object not available to the user) or a temporary error (eg. the requested data must be fetched from disk). In these cases, the error is reported back to the virtual processor by use of an **upcall**. This enables the virtual processor to reschedule another thread.

## 4.2 Virtual processor management

The microkernel attempts to impose little process structure on the application or programmer. Unlike POSIX therefore, it does not implicitly provide such services as file descriptors, "death of child" signals or other heavyweight features. Consequently the process structure, termed a **virtual processor (VP)**, leaves much of the general management work to the application. This presents no additional problem since it can be encapsulated in libraries.

A virtual processor operates around two general data structures; its domain descriptor list (*ddl*) and its upcall list. The *ddl* holds information about all object the virtual processor has access to. As already mentioned, this object is used by the virtual memory system to determine the validity of memory accesses. However, it also holds information for processor management; such as which objects may be signalled using upcalls, and which object was initally executed.

The *upcall list* is the virtual processors' interrupt mechanism and is used by both kernel and other VPs for preempting each other when important events occurs. These events include:

- Alarms,
- Invalid memory accesses,
- Temporarily invalid memory accesses, and
- Lock releases.

2. This is where two or more virtual addresses reference the same physical data.
3. Natually, the user is prevented from direcly modifying the ddl.

The first three of these events are microkernel generated; the forth is generated by user level code associated with the release of mutual exclusion locks or conditional variables.

Upcalls are a fixed sized structure, convey little information, and will not be delivered if the recipient has insufficient resources to receive them. Each one identifies its sender, its type and two further type specific pieces of information (eg. Invalid memory accesses report the failed address and reason for the failure; lock releases report the address of the locking structure.). The VP can precisely control the effect each upcall has when it delivered, determining whether a handler is invoked immediately, whether the upcall is queued for later attention, or whether the upcall is ignored completely. By default, all upcalls are ignored unless the VP specifies otherwise. This generally means that upcalls are simply discarded without effect although "invalid memory accesses" will terminate the VP.

### 4.2.1 Threaded virtual processes

*Angel* does not explicitly support threaded processes, leaving this to user level code. However, through the use of kernel and user level upcalls, it still provides facility for a "first class citizen" thread model. For example, in the kernel, whenever a situation occurs where it should block, the VP is upcalled to allow another threads to be scheduled. Similarly, user level locks can use this facility in parallel programs or client/server relationships (we use this heavily in the LRPC mechanism). At the user level, a POSIX thread model [22] is provided. The operation of POSIX threads is well documented, but it is worth nothing how this model interfaces to *Angel*'s upcall system in order to provide "first class citizens".

All locks are implemented in shared objects. For mutual exclusion locks, if a lock is not obtained, the failed thread inserts itself into the lock's pending queue. The thread scheduler is then called to dispatch another, the VP blocking if there are no others ready to run. When the lock is released, the releasing thread examines the head of the pending queue and releases the top thread. If this thread is within the same protection domain, the operation can be accomplished locally. If not, a *lock release upcall* is dispatched to the appropriate VP. On receiving this, the thread is released locally. The mechanism used for conditional variables is similar to this except that the thread release is delayed until the associated lock is released. By placing locks in shared memory, the operations of obtaining and releasing locks is greatly simplified and the need to consider whether a thread is local or remote is hidden.

## 4.3 Client/Server Communications

Like many commercial and research operating systems, *Angel* uses the notion of clients and servers in order to improve the functional modularity of the system. However, unlike many of its predecessors, message passing is not used to implement RPC communication, instead this is done through shared memory regions. This approach enables a more "lighweight" RPC mechanism to be implemented (based on work by Bershad et al [5]).

*Angel*'s LRPC mechanism operates by the sharing of C++ objects in sections of shared memory. These objects are passed between client and server by manipulation of shared lists and the release of the associated locks. However, optimisations in this mechanism are possible if both client and server operate in the same protection domain. In such cases a direct subroutine call can be made from client to server so avoiding the need for locking altogether. This optimisation can be determined when the LRPC channel is established rather than at compile time so providing greater flexibility.

### 4.3.1 LRPC example

Figure 5 illustates a simple client/server interaction using a shared memory object for communication. This object constitues a private channel between parties, available in their protection domains only (although one-to-many channels are no more difficult to arrange).

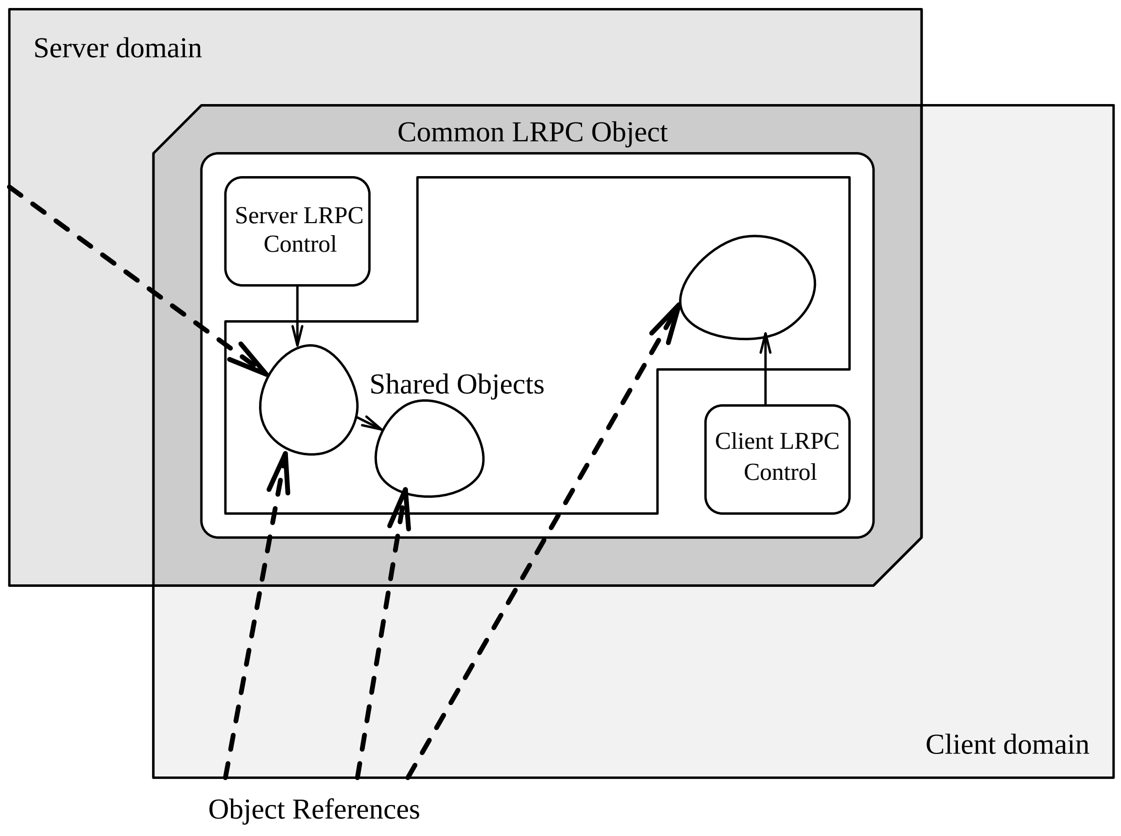

Figure 5: Lightweight RPC object shared between a client and server

In conventional RPC, a client makes a request of the server by packaging data to be transfered and then informing the server of its intentions. The server then unpackages the request, performs the work, and replies to the client using a similar RPC mechanism. LRPC in *Angel* benefits over such a system in two ways; first, the use of shared memory reduces the need to package data, in some cases removing it altogether; and second, implict encapsulation of the client/server relationship in C++ classes simplifies and hides access to the interface.

For example, the server in figure 5 maintains the private database holding users' information. A client wishing to search this database (such as `/bin/ls -l`) must make requests via an LRPC channel. However, rather than constructing and copying requests to the server, a `passwdEntry` object can be allocated which is already shared with the server using the C++ placement operators (eg. overloading of `operator new()`). This object can then be used as normal within the client, the interaction with the server happening transparently and without extra copying by either party.

## 4.4 Current status

The majority of development work has been done by operating the microkernel as an emulation under SunOS UNIX. However, in order to validate the system and determine whether our efforts to keep the dependent and independent code seperate have been successful, we recently ported the kernel to a Tadpole M88K system. This work took a week to complete despite the need to write a new two-level MMU system and although some restructuring has resulted, no major problems were encountered.

However, neither of these systems are appropriate to *Angel*'s needs due to the restricted address space. Currently we are investigating a port to either an SGI Indigo or DEC Alpha PC either of which is more appropriate.

## 5 Lessons and Further Work

The most "politically difficult" decision to make regarding *Angel* was to forego UNIX compatibility. It is acknowledged that if an SASA style operating system is to accepted, then it must provide support for UNIX and its existing software base. As a first step we have investigated modifying compilers to generate code that gave the appearance of UNIX memory semantics. This resulted in a performance penalty of only a few percent [23]. We are now investigating a full UNIX service under *Angel*. It appears that a reasonable degree of compatibility can be provided at low cost, without altering the SASA to provide a region of memory addresses with UNIX characteristics.

The fault tolerance mechanism described above has been designed, implemented and analysed on a simulator, rather than in the current *Angel* implementation. One, relatively simple, task is therefore to implement this scheme in the current microkernel. Once this has been done we hope to study the performance of the system and see if it can be further improved.

The main area of future work lies in dealing with the projected large I/O requirements that a parallel computer will generate. Many current parallel computers are badly I/O limited, and overcoming this bottleneck is extremely important in opening up new markets for parallel machines. There are several schemes we are currently investigated to perform this, the most hopeful is to make use of the algorithms from the fault tolerance scheme which generates a distributed log stream of data for storage on disk.

## 6 Conclusions

This research was conceived as an exercise in learning from *Meshix* (and other message passing micro-kernels); the result is the Angel operating system, which is still a micro-kernel, but is based around a SASA supported by DSM, and not around message passing. The current implementation is small, and has been easy to write, which leads us to believe that we have constructed a good design, and that a SASA is the way to build systems. There are other benefits from this approach which are important to scalability, for example in the areas of fault tolerance, data sharing and load balancing. Although we have not developed the system with UNIX support it mind, it appears that we can provide a simple version of this at very low overheads. All these points lead us to believe that SASAs are an important way of constructing operating systems, especially for scalable, parallel machines.

## 7 Authors Information

Dr Kevin Murray's thesis work at the University of York concerned the development of Wisdom, an operating system designed to support a high-level programming environment on a DMMP conforming to a subset of the ANSA transparency model. In addition, he contributed to the development of Wisdom's filesystem. He then worked at Imperial College, in collaboration with the Systems Architecture Research Centre, on the Angel operating system concentrating on its scheduling and inter-process communications aspects, before being appointed lecturer at City University, where he has remained heavily involved in the Angel work.

Tim Wilkinson has worked extensively on the Topsy project including work on the Meshix OS and Meshnet communications chips. His PhD work, now nearly completed, centres around the design of a reliable 64-bit distributed operating system using data dependent checkpoints. He is currently employed on the Angel operating system project.

Prof. Peter Osmon is head of the Systems Architecture Research Centre at City University. He was Principal Investigator on the Alvey-funded Cobweb project. He conceived and directed the Topsy Unix multicomputer project. He has a current IED grant with Phoenix VLSI and Texas Instruments concerned with the design of an interface device to support shared-memory over a serial interconnect (ICTVS, reference number GR/F99618), and he is Principal Investigator of the SERC funded project developing the Angel kernel (GR/G28277).

Dr. Tom Stiemerling has worked on implementing DVSM on Topsy, and the specification and implementation of the Angel kernel, and is supported by SERC research grant GR/G 28277. His doctoral work carried out at Edinburgh University involved the performance analysis by simulation of a shared memory multiprocessor architecture.

Dr. Paul Kelly is a lecturer in the Department of Computing at Imperial College. He was a researcher on the Alvey-funded Cobweb project. His doctoral work led in part to IED projects on functional programming of transputer networks, and exploitation of more general parallel hardware using functional languages and program transformation. More recently he has collaborated in the development of Paragon, an object-oriented graph-rewriting language, and is also an investigator on the related SERC-funded project developing the Angel kernel at Imperial (GR/G23562).

## Bibliography

[1] Open Software Foundation, "The OSF/1 operating system," in *Spring 1991 EurOpen Conference*, pp. 33–41, 1991.

[2] M. Rozier, "Overview of the CHORUS Distributed Operating Systems," Tech. Rep. CS-TR-90-25, Chorus Systemes, 1990.

[3] P. Winterbottom and P. Osmon, "Topsy: An Extensible Unix Multicomputer," in *UK IT90 Conference, Southampton University*, 1990.

[4] A. Bricker, "A new look at micro-kernel-based UNIX operating systems: Lessons in performance and compatability," in *EurOpen Spring'91 Conference, Tromsoe, Norway*, May 1991.

[5] B. Bershad, T. Anderson, E. Lazowska, and H. Levy, "Lightweight remote procedure call," *ACM Operating Systems Review*, vol. 23, pp. 102–113, December 1989.

[6] P. Osmon, T. Stiemerling, A. Whitcroft, Wilkinson.T., and N. Williams, "Evaluating Meshix – a Unix compatible micro-kernel Operating System," in *OpenForum'92*, November 1992.

[7] A. Whitcroft and P. Osmon, "The CBIC: Architectural Support for Message Passing or Shared Memory?," in *U.K. Performance Engineering Workshop*, September 1992.

[8] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, Department of Computer Science, 1986.

[9] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy, "Scalability study of the KSR-1," Tech. Rep. GIT-CC93/03, College of Computing, Georgia Institute of Computing, Atlanta, Georgia, 1993.

[10] E. Hagersten, A. Landin, and S. Haridi, "DDM – A Cache-only Memory Architecture," Tech. Rep. Research Report R91:19, SICS, Sweden, November 1991.

[11] M. Hill, J. Larus, S. Reinhardt, and D. Wood, "Cooperative shared memory: software and hardware for scalable multiprocessors," in *ASPLOS V*, pp. 262–273, September 1992.

[12] J. C. Mogul and A. Borg, "The Effect of Context Switches on Cache Performance," in *ASPLOS, International Conf. on Architectural Support for Programming Languages and Operating Systems*, (Santa Clara, CA (USA)), pp. 75–85, April 1991.

[13] B. Marsh, M. Scott, T. LeBlanc, and E. Markatos, "First-Class User-Level Threads," Tech. Rep., Computer Science Department, University of Rochester, NY, 1991.

[14] E. Organick, *The Multics system: an examination of its structure.* M.I.T. Press, 1972.

[15] M. Scott, T. LeBlanc, B. Marsh, T. Becker, C. Dubnicki, E. Markatos, and N. Smithline, "Implementation Issues for the Psyche Operating System," Tech. Rep., University or Rochester, Department of Computer Science, 1988.

[16] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska, "How to Use a 64-Bit Virtual Address Space," Tech. Rep. 92-03-02, Department of Computer Science and Engineering, University of Washington, March 1992.

[17] Dobberpuhl *et al.*, "A 200Mhz 64-bit Dual Issue CMOS Microprocessor," in *International Solid-State Circuits Conference*, February 1992.

[18] E. Koldinger, J. Chase, and S. Eggers, "Architectural support for single address space operating systems," in *ASPLOS V*, pp. 175–186, September 1992.

[19] K.-L. Wu and W. Fuchs, "Recoverable distributed shared virtual memory," *IEEE Transactions on Computing*, vol. 39, pp. 460–469, April 1990.

[20] B. Fleisch, "Reliable distributed shared memory," in *IEEE Workshop on Experimental Distributed Systems*, pp. 102–105, 1990.

[21] T. Wilkinson, "Implementing Fault Tolerance in a 64-bit Distributed Operating System," Tech. Rep., City University, 1993.

[22] POSIX 1003.4a, "Threads Extension." IEEE Draft.

[23] T. Wilkinson *et al.*, "Compiling for a 64-Bit Single Address Space Architecture," Tech. Rep. TCU/SARC/1993/1, SARC, City University Computer Science Department, March 1993.