

Architectural Principles and Techniques for Distributed Multimedia Application Support in Operating Systems

Geoff Coulson and Gordon Blair

Distributed Multimedia Research Group,
Department of Computing,
Lancaster University,
Lancaster LA1 4YR,
UK.

telephone: +44 (0)524 65201
e-mail: [geoff, gordon]@comp.lancs.ac.uk

ABSTRACT We propose some architectural principles we have found useful for the support of continuous media applications in a microkernel environment. In particular, we discuss i) the principle of upcall-driven application structuring whereby communications events are system rather than application initiated, ii) the principle of split-level system structuring whereby key system functions are carried out co-operatively between kernel and user level components and iii) the principle of decoupling of control transfer and data transfer. Under these general headings a number of particular mechanisms and techniques are discussed. Our suggestions arise from experiences in implementing a Chorus based real-time and multimedia support infrastructure within the SUMO project.

1. Introduction

Over the past two years members of the SUMO¹ team at Lancaster University and CNET (France Telecom) have been designing and implementing a microkernel based system with facilities to support distributed real-time and multimedia applications. In this paper, we take a retrospective look at our design and revisit some of the abstract architectural principles we have applied.

We are interested in both communications and processing support for distributed real-time/multimedia applications in end systems, and believe that such applications require *thread-to-thread* real-time support according to user supplied quality of service (QoS) parameters. Such support, depending on the level of QoS commitment required, may require dedicated, per-connection, resource allocation

in the CPU scheduler, virtual memory system and communication system. It may also require ongoing dynamic QoS management in all these areas. Another important requirement we have imposed on ourselves is to support standard UNIX applications on the same machine as our real-time/ multimedia support infrastructure; we do not want to build a specialist real-time system that is isolated from the standard application environment. Finally, *efficiency* is a prime consideration in our work. In particular, we are interested in minimising system imposed overheads by reducing the cost and number of system calls, context switches and copy operations.

To achieve these ends we use the Chorus microkernel [Bricker,91] as a vehicle for our research. Chorus lets us run UNIX applications through a SVR4 compatible UNIX 'personality' known as Chorus/MiX, and also provides rudimentary real-time support to native Chorus applications. We have designed our distributed real-time/ multimedia support system as a Chorus 'personality' implemented partly in kernel space and partly as a user level library to be linked with native Chorus applications. The personality provides a QoS driven application programmer's interface (API), connection oriented communications with dedicated, per-connection, resources, and facilities for monitoring and maintaining ongoing QoS levels.

This paper is structured as three main sections, each of which describes a key architectural principle of our design. The three principles are: i) *upcall-driven application structuring* whereby communications events are system rather than application initiated, ii) *split-level system structuring* whereby key system functions are carried out co-operatively between kernel and user level components and iii) *decoupling of control transfer and data transfer* whereby the transfer of control is carried out

¹ The SUMO project is funded by CNET, France Telecom. Aspects of the research are also funded by the UK EPSRC.

asynchronously with respect to the transfer of data. Under these headings a number of innovative techniques are discussed.

We restrict ourselves in this paper to a fairly general and abstract treatment of the above architectural principles. A more detailed description of our design can be found in the literature. In particular, the API is comprehensively described in [Coulson,94a] and [Coulson,94b], and the underlying infrastructure in [Coulson,93] and [Coulson,94c]. The latter paper also describes a number of novel aspects of the system not discussed in this paper such as the support of multiple levels of QoS commitment and their associated admission testing algorithms.

2. Upcall-driven Application Structuring

The system infrastructure, rather than the application, is responsible for the initiation of communication events (both sending and receiving).

In conventional designs, system APIs are mostly *passive* and applications are mostly *active*. For example, when an application needs to send or receive data, it typically invokes a system call such as *send()* or *recv()*. It also provides the buffer from/ to which data is to be sent/ received. In contrast, our continuous media API is structured so that the *system infrastructure* is active and *applications* are passive. Application programmers attach *rhandlers*, which are C functions containing application code to process the real-time media, to *rports*, which are globally unique units of addressing. Then, programmers establish *connections* with a given QoS between *rports*. At connect time, the *system*, rather than the application, allocates buffers for connections and provides the thread on which the *rhandlers* will be executed. At data transfer time, the system decides to upcall the application to obtain/ deliver data at instants determined by the QoS specification (in terms of rate, jitter, delay etc.) provided by the application at connect time. When an application *rhandler* is upcalled, the address of the associated *rport's* buffer is passed as an argument so that application code in the *rhandler* can access the buffer. Source *rhandlers* are expected to fill buffers with data to be sent, and sink *rhandlers* to use the data as provided.

We believe there are three major benefits of this style of application/ system interaction in our

context¹. First, it relieves the application of the burden of explicitly creating threads and allocating buffers. Second, the *system*, rather than the application, can choose the timing of application code execution, and thus can optimally monitor and manage the QoS of the connection, *including the execution of application code*, to provide the required thread-to-thread QoS support². Third, we contend that structuring the API with *rhandlers* is a natural and effective model for real-time programming. Real-time programming is considerably simplified when programmers can structure applications to react to events and delegate to the system the responsibility for initiating communication events. The programmer is still ultimately in control of event initiation but this control is expressed declaratively through the provision of QoS parameters at connect time and need not be explicitly programmed in a procedural style.

Along with these benefits, an *efficiency gain* potentially results from upcall-driven application structuring, because a single thread can be used for both protocol and application processing. In conventional systems, applications interface with communications by performing system calls which block and reschedule if the communications system is not ready to send, or if data has not yet arrived. With infrastructure initiated communication, on the other hand, it is not necessary for the application and communications system to wait for each other, and thus no context switch is incurred, as the communications system always initiates the exchange and the application code is (or should be) always be ready to run.

To our knowledge, upcall-driven application structuring was first discussed by Andrew Black [Black,83] at the 1983 ACM Symposium on Operating System Principles. However, Black's reason for exploring this structure was not related to continuous media. Rather, he was interested in accommodating UNIX pipelines in the object oriented programming paradigm.

3. Split-level System Structuring

The performance of key system functions is shared co-operatively between kernel and user space

¹ Of course, this model is also used in other contexts such as event driven GUIs like X-Windows.

² For example, the fact that physical memory buffers are allocated by the infrastructure eases the implementation of an efficient buffer management scheme and zero-copy communications data path (see section 4).

managers with asynchronous communication of management information between the two.

We assume that distributed multimedia applications will typically require a high degree of internal concurrency. For example, it is likely that each media stream will require at least one thread of execution and it is also likely that applications will be structured as *pipelines* of processing stages on streams of media. We further assume that it will be convenient to encapsulate this (per application) concurrency within a single address space (or a small number of address spaces) to ease and optimise communication and synchronisation between concurrent activities.

Given these assumptions, it is reasonable to place as much as possible of the required system functions *in the same address space as the application itself*. This has the benefit of minimising communication overheads between the application and its support infrastructure and thus enabling tight coupling for management purposes. Unfortunately, this approach has the corresponding drawback that the kernel loses global awareness of resource usage across the whole machine. Split level structuring is designed to maintain the advantages of user space management while mitigating its disadvantages.

3.1 Split Level Scheduling

3.1.1 The Basic Scheme

The above mentioned advantages and disadvantages are particularly evident in the case of CPU resource management through user level threads. Here the benefit is cheap user level concurrency and the drawback is that the relative urgencies of threads in different address spaces are not visible to the kernel scheduler.

The solution is *split level scheduling* which was originally devised at the University of California at Berkeley; a UNIX implementation was reported by Govindan and Anderson at the 1991 ACM Symposium on Operating Systems Principles [Govindan,91].

In split level scheduling, a small number of *virtual processors* (VPs) execute user threads in each address space (typically, one VP per physical CPU is used¹). The split level scheduling scheme maintains the invariant that:

- each *user level scheduler* (ULS) always runs its most urgent user thread, and
- the *kernel level scheduler* (KLS) always runs the VP supporting the *globally* most urgent user thread.

Split level scheduling allows many context switches to take place cheaply in the same address space but also ensures that the relative urgencies of threads across the whole machine are appropriately taken into account.

In our implementation, which uses the *earliest deadline first* scheduling policy [Liu,73], VPs are realised as Chorus kernel threads. To enable the KLS and ULSs to co-operate, an asynchronous communication mechanism is used which consists of i) a segment of memory shared between the KLS and all the ULSs, which serves as an asynchronous ‘bulletin board’ area, and ii) an asynchronous software interrupt mechanism² for kernel-to-user space event notification. The ULSs place in the bulletin board area the urgency (i.e. the deadline in our case) of their most urgent runnable thread, and the KLS inspects these urgencies each time it runs and chooses to run the VP that is supporting the user thread with the globally greatest urgency.

3.1.2 Conditional Deadlines

We have extended Govindan’s original design with *conditional deadlines*. Conditional deadlines enable the urgency of external events (such as the arrival of a network packet, or a timeout) to be taken into account when scheduling decisions are made.

A conditional deadline is of the form `<event, deadline>` and has the intuitive interpretation: “when `event` has arrived this thread will be runnable and will have a deadline of `deadline`”. Conditional deadlines are placed in the shared ULS/KLS memory area for consideration by the KLS as above. When the KLS runs, if it can match a current event (e.g. network packet arrival) with a conditional deadline, *and* the `deadline` field of the conditional deadline is the globally earliest, then the KLS will choose to run the associated thread’s VP. The KLS will also deliver an asynchronous software interrupt (see section 4.3) to enable the VP’s ULS to gain control when it next runs so that the appropriate user thread can be immediately scheduled.

Note that without conditional deadlines, the KLS

¹ In this paper, we assume a single CPU and thus a single VP per address space. However, the design can easily be

² generalised to shared memory multiprocessor architectures. See section 4 for details of the implementation of our software interrupt mechanism.

would have to *immediately* notify a ULS on the occurrence of an event to ensure timeliness of response. However, if the event turned out to be non urgent, the context switch to the VP receiving the software interrupt may have been a waste of time - particularly if the user thread with the globally earliest deadline happened to reside in a different address space.

3.2 Split Level Communications

The strategy of split level communications structuring is to leave the kernel responsible for multiplexing and demultiplexing network packets to application address spaces, but to let application address spaces perform transport level processing. In this way, transport protocol processing can automatically take advantage of the split level scheduling infrastructure and thus exploit cheap user level context switches.

Split level communications structuring also allows *meaningful* deadlines to be placed on (transport level) protocol processing activities, as the ultimate deadline of the final packet delivery is easily available in the application context. Thus, the scheduling of protocol processing need not be performed 'blind' as it is in typical kernel implementations. A further advantage is that multiple transport protocols can easily be dynamically configured in and out of applications according to their particular requirements [Thekkath,93]. This is important in a multimedia context where different protocols may be appropriate for different media types.

3.3 Split Level Buffer Management

The strategy of split level buffer management is for the kernel level manager to 'loan' physical¹, locked, buffers to per-address space managers, but to always reserve the right to reclaim the buffers if memory is more urgently required elsewhere (according to global connection priorities or deadlines) or if the per-address space manager retains the buffer longer than it has agreed to. The policy adopted in our current design is that the application is allowed to keep the buffer for at least the normal duration of transport protocol processing time *plus* rhandler execution time. If, however, this period has elapsed and the application address space has not

¹ It is often necessary to use physical memory buffers in time critical real-time and multimedia systems as the access latency to virtual memory is at the mercy of the paging system.

returned ownership of the buffer to the kernel², the kernel may reclaim the buffer.

The semantics of 'reclaiming' locked buffers is to convert locked memory into standard swappable virtual memory. In this way, applications do not lose their data although they do lose guaranteed access latency to that data as the memory region is subject to being paged out. If the kernel does not need to reclaim buffers at the end of an rhandler execution, the user space manager may re-use buffers for other connections (e.g. in user level pipelines; see section 4.4).

4. Decoupling of Control Transfer and Data Transfer

Transfer of control is carried out asynchronously with respect to transfer of data to permit the use of separately optimised pathways for both.

In traditional systems, the transfer of control and the transfer of data are usually tightly coupled. For example, the execution of a UNIX system call passes data to the kernel and simultaneously transfers control to the kernel. Similarly, the return of a call such as *recv()* transfers control back to the application and simultaneously transfers the received data to the application. However, there are well-known advantages to be gained from decoupling control transfer and data transfer. For example, asynchronous message passing in distributed systems yields additional concurrency, and copy-on-write based IPC as used by Mach [Accetta,86] and Chorus defers data copying, and avoids it altogether if the receiver only needs to read the data.

In the following sections we show, by means of four examples from our Chorus based design, how the principle of decoupling of control transfer and data transfer can be usefully exploited in a number of situations in a real-time/ multimedia environment.

4.1 Direct Connections

In distributed multimedia applications it is often required to receive continuous media data from the network and directly play it out on a device such as an audio card or a frame buffer (which is probably managed by kernel level code). The application may or may not require to keep track of the transfer of

² This is achieved by setting a flag in the shared memory bulletin board area; see *asynchronous system calls* in section 4.

individual buffers of data for synchronisation purposes. The opposite scenario, where data from a local device is to be put directly onto the network, is equally common. In conventional operating systems, the only way to achieve such a data flow is to route the data through an intermediate user process. Unfortunately, this involves significant per-buffer overheads. For example, when receiving data from the network which is to be played out on a local device, two system calls per buffer (one *recv()*, and one *write()* to place data into the device), a context switch to the user address space and probably a number of copy operations are involved.

In a *direct connection*, data that is to be passed directly between the network and a local device does not pass into user space at all; it is processed entirely within kernel space. In API terms, the application associates an *rport* with the device rather than its own address space (thus resulting in an identical API for 'conventional' connections and direct connections). Note that direct connections require us to support an in-kernel instance of the transport protocol *in addition to* the user level implementation discussed above. When a direct connection is established, the infrastructure pre-maps the buffer associated with the connection into the output device's memory (assuming a 'receiving' scenario). Then, data can be directly copied off the network card on to the device without leaving kernel space. The only significant in-line overhead is incurred by the fragmentation/ re-assembly functions of the in-kernel transport protocol.

If the user application does not need to synchronise with the delivery of buffers, no further overhead is incurred. However, if it is required to synchronise, the application can attach an *rhandler* to the *rport* (as described in section 2). This is upcalled on each buffer transfer with the usual *rhandler* semantic. The only difference in the API between this case and the normal case described in section 2 is that the buffer pointer passed as an argument to the *rhandler* upcall will be a *null* pointer as the application context will not have the rights to directly access the kernel managed buffer.

4.2 Asynchronous System Calls

For continuous media connections, *asynchronous system calls* exploit the *predictable periodicity* of the transfer of control and data between application address spaces and the kernel. To issue an asynchronous system call (e.g. an asynchronous version of *send()*), user level library code:

- i) places an operation identifier and parameters

in the shared KLS/ULS memory bulletin board area and then

- ii) sets an 'operation request' bit, also in the bulletin board area.

The KLS, when it runs at the next system clock tick, notices that an operation request bit is set and consequently passes the user's parameters to a kernel server thread which carries out the system call on behalf of the ULS. This avoids a special domain crossing for the system call at the expense of a couple of instructions to examine a bitmap on each clock interrupt. As long as domain crossings are frequent (as will be the case when a number of continuous media connections are running) this is likely to reduce the overall system call overhead.

An additional benefit is that the inherently non blocking semantics of asynchronous system calls allow an important optimisation of the split level scheduling design (see section 3.1). The problem with standard, blocking, system calls in a split level scheduling context is that a user level thread performing a blocking system blocks its underlying VP [Marsh,91]. This means that any other user thread in that address space is unable to run until the blocking system call returns, even if one of them has a globally highest urgency. Non blocking asynchronous system calls eliminate this problem by *immediately* releasing the VP so that it can execute another user thread (n.b. the calling user thread is blocked at the user level while this is happening to preserve the expected blocking semantics for applications). When the system call is completed, the kernel delivers the result to the ULS through the conditional deadline mechanism (see section 3.1.2). The ULS can then re-schedule the blocked user thread. Note that applications using user threads see only standard blocking system calls.

4.3 Asynchronous Software Interrupts

Our implementation of software interrupts is similar to that of asynchronous system calls and similarly avoids a special domain crossing. The mechanism for kernel-to-VP control transfer is as follows:

- i) the KLS places an event identifier and parameters in the KLS/ULS bulletin board area;
- ii) the KLS alters the program counter field of the target VP's context structure (also kept in the bulletin board area) to point to a standard entry point in the ULS.

Thus, when the VP is next scheduled, the VP immediately enters its ULS, which picks up the event identifier and parameters, and schedules a user thread to deal with the event. Note that the actual transfer of control only occurs when the target VP is next scheduled as determined by the split level scheduling system. The original contents of the program counter field are stored in the bulletin board area so that the interrupted user thread can be resumed by the ULS at some later time.

Asynchronous software interrupts are also provided as a service accessible from user level code. This service enables library code in one address space to cheaply notify an event to another address space on the same machine. The service also allows the sender to name a pre-existing memory segment shared between the sender and receiver address spaces so that data can be optionally transferred in the same call.

4.4 User Level Pipelines

Our API for *pipelines* of processing stages is very similar to the connection abstraction described in section 2. But rather than passing a *pair* of rports as arguments to the *connect()* primitive, we pass a *list* of rports. Also, in the case of pipelines, the delay QoS parameter applies end-to-end over the entire chain of rports.

Intermediate processing stages in pipelines are also realised in a similar way to that described above: when data arrives at an intermediate processing stage, the rhandler associated with the rport is upcalled. When the rhandler returns, it is assumed that the rhandler's application code has performed some appropriate processing on the buffer whose address was passed up to it, and the data can be passed on to the next stage.

As the various stages of a pipeline form part of the same application, it is typically the case that pipelines (or large sections of them) are implemented in a single address space. The data transfer mechanism in this case is as follows: when an rhandler implementing one stage of a pipeline returns, having operated on a buffer, the next stage in the pipeline is simply passed the address of the same buffer. Meanwhile, the first stage sets to work on a second buffer; and so on. At the end of the pipeline, when buffers are finished with, they are returned to a user level pool from which they can be reused by the first pipeline stage. With this implementation, intra-address space pipelines incur only user level control transfers between the threads dedicated to each pipeline stage, and zero copy operations between stages.

Note that the API is the identical regardless of whether intra-address space, inter-address space or inter-machine connections or pipelines are used. Inter-address space communication on the same machine uses buffers that are statically mapped into both the source and sink address spaces for data transfer, and use asynchronous software interrupts, as described above, for control transfer. Inter-machine connections use the split-level communications system described in section 3.2.

5. Illustrative Scenario

To further illustrate the integrated use of the principles and techniques described above, let us examine their application in a thread-to-thread continuous media scenario in the context of our Chorus based system. The scenario, illustrated in figure 1, involves the transfer of compressed video from a frame grabber card on a source machine to a decompress/ display application on a sink machine. In figure 1, the large ovals represent user address spaces with library code below the horizontal line and application code above. The rectangles represent kernel space with the enclosed shaded regions representing devices.

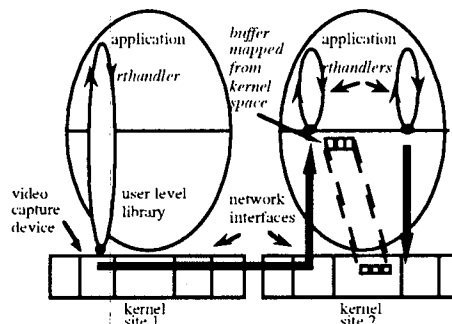


Figure 1: An Illustrative Scenario

The send side features a *direct connection*, involving the video capture device and the network interface, which avoids the need for data to pass into user space. It also features the (optional) use of an *rhandler* to allow the sender, which is structured as an *upcall-driven application*, to monitor and synchronise with the progress of the connection. If the rhandler is used, the ULS of the *split level scheduling system* is notified via an *asynchronous software interrupt* each time a frame of video has been transmitted, and schedules a user thread to execute the application code.

On the receive side, the *split level buffer management system* allocates a physical buffer from the kernel buffer pool to hold incoming network

packets associated with the connection. This buffer is statically mapped into both kernel space and the application address space (to eliminate the need for copying).

In the *split-level communications system*, when a complete network level packet has been received, the application address space's ULS is notified via the *conditional deadline* mechanism and initiates transport level processing. This may involve the receipt of further network packets to build a complete user level buffer. When a complete user buffer has been built, and the receiving thread has the globally earliest deadline, the ULS runs a thread which upcalls the application's rhandler with the address of the buffer as a parameter.

The receive side features a *user level pipeline* which involves one user thread performing decompression and another displaying uncompressed video in a window. The display is achieved by means of *asynchronous system calls* to a display device (not shown). Context switches between the two pipeline threads are achieved at user level costs and the transmission of data from the decompressor to the display does not involve data copying.

6. Conclusions

We have discussed three architectural principles useful for the support of distributed real-time/multimedia applications in operating systems, and have illustrated the applicability of the principles with specific techniques from our Chorus based distributed real-time/multimedia support infrastructure.

Firstly, we contended that the principle of *upcall-driven application structuring* leads to well-structured real-time applications, relieves applications of the burden of explicit thread creation and buffer allocation, and leads to potential efficiency gains because of reduced context switches.

Secondly, we argued for the principle of *split-level system structuring*. We suggested that this can improve efficiency by exploiting application specific knowledge (e.g. scheduling deadlines or buffer requirements) in a local, user level, context where application/manager interaction is cheap, while relying on a kernel level manager to 'bias' resources to application address spaces of the basis of their relative needs. Active co-operation of management information between user and kernel level managers is key, but as long as an asynchronous style of communication between managers is acceptable, this can be achieved cheaply by means of a shared

memory 'bulletin board'.

Thirdly, we suggested that the principle of *decoupling of control transfer and data transfer* can be widely applied in a multimedia environment. In support of this, we described four techniques from our implementation: direct connections, asynchronous system calls, asynchronous software interrupts and user level pipelines.

A final point is that, although we have shown the three principles working together in this paper, they are largely orthogonal and should be capable of being exploited in a range of operating system environments. Similarly, many of the individual techniques we have described can usefully be implemented in a stand-alone fashion. We are currently implementing and evaluating the chorus based design described in this paper and look forward to validating our principles in terms of direct performance measurements.

Acknowledgement

We would like to gratefully acknowledge our colleagues Jean-Bernard Stefani, Francois Horn and Laurent Hazard of CNET, France Telecom for many profitable discussions around the issues of this paper.

References

- [Accetta,86] Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., and M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Technical Report* Department of Computer Science, Carnegie Mellon University, August 1986.
- [Black,83] Black, A.P., "An Asymmetric Stream Communication System", *Proc. 9th ACM Symposium on Operating System Principles*, Mount Washington Hotel, Bretton Woods, New Hampshire, USA, 10-13th October, 1983.
- [Bricker,91] Bricker, A., Gien, M., Guillemont, M., Lipkis, J., Orr, D., and M. Rozier, "Architectural Issues in Microkernel-based Operating Systems: the CHORUS Experience", *Computer Communications*, Vol 14, No 6, pp 347-357, July 1991.

- [**Coulson,93**] Coulson, G., Blair, G.S., Robin, P. and Shepherd, D., "Extending the Chorus Micro-kernel to Support Continuous Media Applications", *Proc. Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, Lancaster House Hotel, Lancaster, UK, published by Springer Verlag, ISBN 3-540-58404-8, October 93.
- [**Coulson,94a**] Coulson, G., and G.S. Blair. "Micro-kernel Support for Continuous Media in Distributed Systems", *Computer Networks and ISDN Systems*, Vol 26 (1994), pp 1323-1341, Special Issue on Multimedia, 1994.
- [**Coulson,94b**] Coulson, G., G.S. Blair, P. Robin, and D. Shepherd, "Supporting Continuous Media Applications in a Micro-Kernel Environment." in *Architecture and Protocols for High-Speed Networks*, Editor: Otto Spaniol. Kluwer Academic Publishers, 1994.
- [**Coulson,94c**] Coulson, G., Campbell, A., P. Robin, Blair, G.S., Papathomas, M., and Shepherd, D., "The Design of a QoS Controlled ATM Based Communications System in Chorus" to appear in *IEEE Journal on Selected Areas in Communications*, Special issue on ATM LANs, 1994.
- [**Govindan,91**] Govindan, R., and D.P. Anderson, "Scheduling and IPC Mechanisms for Continuous Media", *Thirteenth ACM Symposium on Operating Systems Principles*, Asilomar Conference Center, Pacific Grove, California, USA, SIGOPS, Vol 25, pp 68-80, 1991.
- [**Liu,73**] Liu, C.L. and Layland, J.W., "Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment", *Journal of the Association for Computing Machinery*, Vol. 20, No. 1, pp 46-61, February 1973.
- [**Marsh,91**] Marsh, B.D., Scott, M.L., LeBlanc, T.J. and Markatos, E.P., "First class user-level threads", *Proc. Symposium on Operating Systems Principles (SOSP)*, Asilomar Conference Center, ACM, pp 110-121, October 1991.
- [**Thekkath,93**] Thekkath, C.A., T.D. Nguyen, E. Moy, and E. Lazowska, "Implementing Network Protocols at User Level." *IEEE Transactions on Networking*, Vol 5, No 1, pp 554-565, October, 1993.