A QUALITY OF SERVICE DRIVEN CONCURRENCY FRAMEWORK FOR OBJECT-BASED MIDDLEWARE

Geoff Coulson and Oveeyen Moonian

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

e-mail: geoff@comp.lancs.ac.uk

ABSTRACT

Threads play a key role in object-based middleware platforms. Implementers of such platforms can select either kernel or user-level threads, but neither of these options are ideal. In this paper we introduce Application Scheduler Contexts (ASCs) which flexibly combine both types of thread and thereby attempt to exploit the advantages of each. Multiple ASCs can co-exist, each with their own concurrency semantics and scheduling policy. ASCs also support quality of service (QoS) configurability, and define their own QoS schema. We show how ASCs can be efficiently implemented and how they can usefully be exploited in middleware environments. We also provide a quantitative evaluation that demonstrates the feasibility of the ASC concept in performance terms.

1. Introduction

Threads play a key role in object-based middleware platforms (also referred to as Object Request Brokers or ORBs) such as the OMG's CORBA [OMG,00] or Microsoft's DCOM [Microsoft,99]. Essentially, threads fulfill *three* main roles in such platforms: First and foremost, they allow objects to execute concurrently with remote invocations to (and from) other objects in the same address space (aka *capsule*). Second, they facilitate the structuring of autonomous distributed objects inside a single capsule. Third, they allow CPU resources to be selectively targeted at logically separate activities in a capsule. For example, separate threads can be dedicated to incoming message handling, user interface processing and media stream processing. Furthermore, by dedicating varying numbers of threads with varying priorities, deadlines etc., to such activities, the *quality of service* (QoS) of activities can be individually managed and controlled.

There are two types of thread in common use: *Kernel threads* are implemented and scheduled by the operating system (OS). *User-level threads*, on the other hand, are implemented in a user-level library and are transparent to the OS. The user-level library allocates stack memory for each thread, and switches threads by saving and restoring per-thread CPU state such as the stack pointer and instruction pointer registers. While early middleware platforms (e.g. ANSAware [ANSA,91] and DCE [OG,99]) tended to incorporate user-level threads, recent platforms are far more likely to employ kernel threads. This is partly because kernel threads are now more widely available than

before, and partly because there are a number of perceived disadvantages of user-level threads relative to kernel threads:

- user-level threads are associated with a performance bottleneck (referred to as the 'whole-capsule blocking problem') when threads perform blocking OS calls such as 'sleep', or input/ output (I/O) calls; because the thread implementation is transparent to the OS (which provides only a single execution context to the capsule), a single thread performing such a call blocks all other threads in the capsule;

- user-level threads cannot exploit multiple CPUs (where available); again, this is because the thread implementation is transparent to the OS, which provides only a single execution context to the capsule;

- user-level threads are said to be less portable because they require small sections of code that are CPU specific: i.e. code to initialise a run-time stack and to save and restore CPU state.

In other respects, however, user-level threads enjoy significant advantages over kernel threads:

- their context switch overhead is an order of magnitude less than that of kernel threads [Anderson,91],

- it is often possible to support a far higher degree of concurrency than with kernel threads, which tend to be subject to OS imposed resource limits,

- because they are implemented at the user level, they offer far higher degrees of *flexibility*.

The flexibility advantage is crucial, and comprises two main aspects: *concurrency semantics* and *scheduling policy*. Regarding concurrency semantics, user-level implementations can select from, for example, *non-preemptive*, *preemptive* and *timesliced* options[1], along with numerous variants and combinations of these. Kernel threads, on the other hand, are almost always preemptive and timesliced. Regarding scheduling policy, it is straightforward in user-level thread packages to implement application tailored policies such as *earliest deadline first* (EDF) for media stream processing, or *shortest job first* for transaction processing [Stankovic,95]. It is equally straightforward to modify and adapt these policies dynamically should the need arise [Blair,98]. Kernel threads, on the other hand, typically offer only a small fixed set of alternative policies.

This paper discusses a novel framework known as *application scheduler contexts* (ASCs) which, by combining both user-level and kernel threads, attempts to maximise

---

[1] With *non-preemptive* semantics, context switches are only invoked explicitly; e.g. in response to a 'yield' call or similar. The *preemptive* semantic is additionally able to switch context in response to asynchronous events such as alarms. The *timesliced* semantic does not need a 'yield' call because the system implicitly ensures that threads are time division multiplexed over the CPU(s). Each semantic has unique advantages. For example, non-preemptive is maximally efficient and may not require explicit mutual exclusion in critical sections, preemptive is good for responsiveness in soft real-time applications, and timesliced offers an intuitive programming model in which all threads continually make progress without the need for strategically placed 'yield' calls. These semantics may also be combined. For example, standard operating system processes/ kernel threads typically conflate preemptive and timesliced semantics.

the advantages and minimise the disadvantages of each type of thread. In the ASC framework, user-level threads are referred to simply as *threads*, and kernel-threads are referred to as *virtual processors* (this is abbreviated to '*VP*' in the remainder of the paper). The basic modus operandi of the framework is to multiplex each ASC's threads over its VPs in an ASC-specific manner. While threads are visible to applications, VPs are transparent and spend their lives in an endless loop, repeatedly taking up a thread context and executing it until the thread either yields or is preempted, and then taking up another thread, and so on.

The goals of the ASC framework transcend those of existing two-level thread architectures (see section 6). For example, an important goal of the framework is to support multiple co-existing scheduling environments (i.e. ASCs), each of which defines its own concurrency semantic and scheduling policy. Another key goal is to facilitate flexible and extensible *QoS management* for threads. To achieve QoS management, each ASC defines its own schema for QoS specification, and maintains a dedicated pool of processing resources (VPs). Threads then specify their QoS requirements in terms of this schema and the ASC maps these specifications onto a subset of its VPs according to its scheduling policy.

The remainder of this paper is structured as follows. Section 2 defines and presents the ASC framework in abstract terms; section 3 then discusses its implementation. Subsequently, section 4 considers the application of the ASC framework in middleware environments and section 5 provides a quantitative evaluation of the framework. Finally, section 6 discusses related work and section 7 offers concluding remarks.

2. The ASC Framework

2.1 Overview

ASCs are defined as a 7-tuple:

> *<thread-set, concurrency-semantic, scheduling-policy, VP-set, VP-activation-policy, QoS-schema, QoS-specification-set>*

The task of each ASC is to multiplex its *thread set* over its *VP set* according to its associated *concurrency semantic*, *scheduling policy* and *VP activation policy*, while honouring each of its thread's individual *QoS specifications*, each of which is expressed according to the ASC's *QoS-schema* (see Fig. 1).

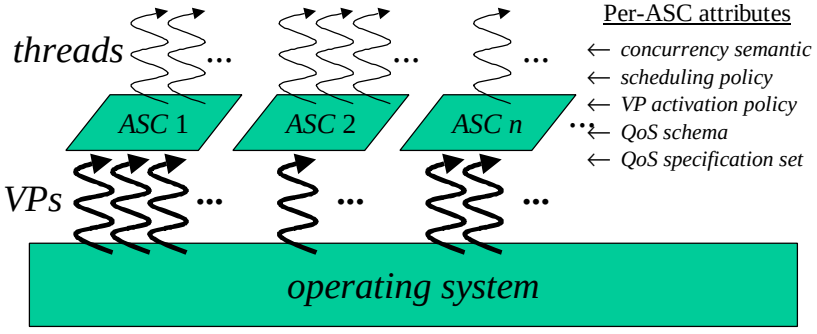

Figure 1: The ASC concept

The *VP activation policy* determines when/ whether the ASC should activate/ passivate VPs, or create/ destroy VPs, in response to factors such as loading, or any

other ASC-specific performance/ QoS indicators. The form of the *QoS-schema* corresponds to the nature of the scheduling policy (section 4.1.2 has a simple example). For example, a QoS-schema associated with an EDF policy may include an integer 'period' parameter. Each ASC maps its threads' QoS specifications onto its processing resources in an ASC specific way, and is free to decide on the *stringency* with which it treats these specifications. This can range from best effort through guaranteed, and will typically be a function of the stringency of the guarantees provided to the ASC's VPs by the underlying OS (see section 3.7). A related point is that ASCs may *admission test* thread creation requests in case they need to be able to 'guarantee' the requested QoS of new threads, or if admitting a new thread may compromise 'guarantees' already given to other threads.

We view ASCs as highly dynamic objects. Instances can be created and destroyed dynamically, and both threads and VPs can be added/ removed at will. Furthermore, it is possible to change the QoS of a running thread, and to migrate running threads from one ASC instance to another. It is even possible to migrate threads between ASCs of different classes; this is useful when it is desired to change the QoS-schema available to a thread.

2.2 API Level Interfaces

The ASC framework, as viewed by application-level users of the framework, is defined by the following (minimal) interfaces, which are expressed in C++/ Java inspired pseudo-code:

```
class Thread {
  Thread(Runnable f, String argv[]); // constructor
  ASC asc();
  static int yield(); // class method
  static Thread current_thread(); // class method
  …
}

class QoS {…} // abstract class

interface ASC {
  int thread_put(Thread *thread, QoS *qos);
  static int thread_change(ASC *asc, QoS *qos); // class method
  …
}
```

The *Thread* class encapsulates a user-level thread (which is, essentially, a stack plus associated CPU state). Its constructor takes a *Runnable* object containing an 'entry-point' operation, to which the arguments specified in *argv* are passed. The *Thread* class also contains an operation to return the thread's current host ASC (*asc()*), a per-class operation, *yield()*, by means of which an executing thread can explicitly yield its VP, and another per-class operation, *current_thread()*, which returns the caller's *Thread* object. The *Thread* class additionally contains various ancillary operations (not shown here) associated with timers and concurrency control objects such as semaphores [Coulson,99a].

Threads are managed with respect to their hosting ASC by means of the *thread_put()* and *thread_change()* operations in the *ASC* interface. Each installed ASC must implement this interface. *Thread_put()* inserts a new thread into the target ASC. Its *qos* argument specifies the thread's QoS requirements in terms of the target ASC's

QoS schema (each ASC defines its own QoS specification class which inherits from the abstract *QoS* class; see section 4.1.2 for a simple example). *Thread_change()*, on the other hand, allows the calling thread to either change its QoS in its current ASC, or to migrate to a new ASC (the *qos* argument in this latter case must refer to the new ASC's QoS schema). Both *thread_put()* and *thread_change()* may fail if the target ASC implements admission testing and cannot support the requested QoS.

2.3 Internal Interfaces

In addition to the above application visible interfaces, the ASC framework relies on the following interfaces which are visible only to the run-time environment to be described in section 3:

```
class VP {
  VP(ASC asc); // constructor
  ASC asc();
  boolean idling();
  static int idle(); // class method
  int activate();
  …
}

interface ASC_internal {
  boolean preemptive();
  boolean activation_required();
  Thread *schedule(Thread *old);
}
```

The *VP* class encapsulates a kernel thread that acts as a VP. Its constructor associates the new VP with a particular ASC and the *asc()* operation returns the VP's associated ASC. The *idling()*, *idle()* and *activate()* operations are used to manage the run-status of the VP and are discussed in section 3.3. In addition to these operations, the VP class would be expected to give access to any OS level QoS configurability (e.g. priorities) that were available to the underlying kernel thread.

The *ASC_internal* interface's *preemptive()* operation returns *true* if and only if the ASC supports preemptive threads, and the *activation_required()* operation returns *true* whenever the ASC has runnable threads but no currently running VPs. These operations are further discussed in section 3. The *schedule()* operation is responsible for updating any dynamic state associated with the descheduled thread *old* (e.g. deadlines in an EDF ASC), storing *old* inside the ASC (e.g. in a run queue data structure), and returning the thread context that the ASC's scheduling policy has determined should be the next to run (assuming there is at least one runnable thread; it returns *null* otherwise). As discussed in section 3.3, *schedule()*, in cooperation with *Thread.yield()*, is also responsible for implementing the ASC's VP activation policy.

As with the *ASC* interface, all installed ASCs must implement *ASC_internal*.

3. Run-time Environment

3.1 Overview

In this section we describe the run-time environment required to support ASCs. The (interlocking) aims of this environment are as follows:

- to provide a generic implementation of *Thread.yield()* that is responsible for interfacing to the calling thread's 'home' ASC;
- to ensure, in collaboration with per-ASC VP activation policies, that VPs idle and are reactivated as appropriate;
- to provide support for preemptive and timesliced ASCs through a *signal handling* framework;
- to provide a *concurrency control regime* which maintains safety while avoiding bottlenecks and minimising locking overhead.

In addition, we briefly discuss, in section 3.7, the provision of *proportional share VPs* as part of the supporting environment. As detailed below, these are not essential but are useful, where available, when it is required to build ASCs that support highly predictable QoS.

3.2 Thread.yield() Implementation

Thread.yield() is either called explicitly by a thread that wants to yield to another thread, or when a preemption takes place (see section 3.4). Its main role is to interact with each ASC's *schedule()* operation and to dispatch threads as follows:

- it freezes the context of its calling VP's currently executing thread,
- it determines the calling VP's 'home' ASC (via an internal mapping from the VP's unique identifier),
- it passes the frozen thread context to the calling VP's home ASC by calling the latter's *ASC_internal.schedule()* operation, and receives a new thread context as this operation's return value,
- it resumes this new thread context on the calling VP.

In addition, *Thread.yield()* has various auxiliary 'housekeeping' responsibilities as described below.

3.3 VP Management

If an ASC's *schedule()* operation returns *null* (i.e. the ASC currently has no runnable threads), the VP that called *Thread.yield()* must be made to *idle* until work becomes available. To achieve this, it invokes *VP.idle()* which causes its underlying kernel thread to wait on an OS-level semaphore.

Idling VPs are subsequently reactivated by one of two mechanisms:

- First, any VP may be explicitly reactivated by its host ASC during a subsequent call of *schedule()*, according to the ASC's VP activation policy. ASCs use the VP class's *idling()* and *activate()* operations to (respectively) determine how many VPs are currently idling and to reactivate some appropriate number of VPs as determined by its specific policy (see 4.1.2 for examples).
- Second, to prevent entire ASCs from becoming permanently and irretrievably dormant, *Thread.yield()* itself takes responsibility for ensuring that each ASC with runnable threads has at least one active VP. To do this, it loops over all installed ASCs each time it is called and, for each ASC whose

activation_required() operation returns true, it activates one of that ASC's currently idling VPs.

It is also necessary to deal with the exceptional case that *schedule()* has returned *null* and all other ASCs are also without runnable threads. In this case, the calling VP runs a distinguished 'sentinel' thread not associated with any particular ASC. This thread invokes a generic *sentinel driver* object with which *sentinel callbacks* can be registered. Sentinel callbacks typically block waiting for incoming signals (see next section); and when a signal occurs, they wake threads that have been waiting for the signal and then call *Thread.yield()* so that VPs associated with these threads can be reactivated.

3.4 Signal Handling, Preemption and Timeslicing

Signals are asynchronous calls issued by the OS to notify user-level code of events such as alarm expiry or I/O availability. In the ASC framework, they are an essential underpinning of the preemptive concurrency semantic. The run-time environment employs a generic and extensible framework for signal handling that is closely integrated with *Thread.yield()* and the environment's concurrency control scheme (see section 3.5).

The objective of the signal handling framework is to safely dispatch signals to the appropriate signal handler routine with minimum latency while honouring per-ASC concurrency semantics and maintaining low locking overhead. These signal handlers are typically used to build low-level services (e.g. the timer module discussed below); they are transparent to applications at the thread level.

For each type of signal[1] to be handled (e.g. alarm expiry), an object of the following class is provided:

```
class SignalType {
      int signal; // OS designated signal identifier
      boolean sigflag;
      int sighandler();
}
```

On signal occurrence, the OS invokes a *generic signal handler*, in the context of the interrupted VP, which starts by setting the *sigflag* member in the appropriate *SignalType* object. In case the interrupted VP happens to belong to a preemptive ASC (i.e. the ASC's *preemptive()* operation returns *true*) *and* the run-time environment is currently unlocked (see section 3.5), the generic handler continues by directly calling *Thread.yield()*, which completes the signal handling process by testing and clearing all registered sigflags and calling the associated *sighandler()* of any sigflag that was set[2]. This case results in the lowest possible signal handling latency.

---

1 Some OSs support a per-kernel-thread semantic for signal delivery (i.e. signals are delivered to the kernel thread that 'caused' the signal), while others, like POSIX, support a per-process semantic (i.e. signals are delivered to any kernel thread, or to a single designated one). Our framework makes no assumption either way.

2 To prevent the possibility of a race condition between the setting of *sigflag* in the generic handler, and its testing and clearing in *Thread.yield()*, the latter employs an atomic *test-and-set* routine to test and clear the flag in one uninterruptible action. Without this, it would be possible for signals to be missed if a *sigflag* was set (i.e. a new signal occurred) between testing and clearing the flag prior to invoking *sighandler()*.

If, however, the interrupted VP does *not* belong to a preemptive ASC *or* if the run-time environment is currently locked, the generic handler must return directly having set *sigflag*, relying on an alternative route for the subsequent invocation of *Thread.yield()* (and consequent completion of the signal handling process):

- in the case of a non-preemptive VP, the generic handler directs a secondary signal (user software interrupt) toward an alternative VP owned by a preemptive ASC;
- the case of the run-time environment having been locked is dealt with by a post-critical-section test at the end of *Thread.yield()*; this takes any sigflags that have just become set as an indication that it should *re-invoke itself* to deal with the newly arrived signal(s).

Alternatively, if it is not important to respond immediately to signals, the signal handling framework can be configured to simply wait until the next time *Thread.yield()* is invoked for any reason.

As an example of the use of the signal-handling framework, the *timer module* provides timing services for threads (the associated operations are exported by the *Thread* class). The timer module implements these services by multiplexing 'virtual' timers over the SIGALRM signal (assuming a UNIX environment), using a delta queue scheme [Silberschatz,98]. Each thread offers the user a fixed number of independent timers, each of which is associated with a user-provided handler that is called each time the timer fires. The timer module's *sighandler()* implementation unblocks threads whose timers have fired, directs a user signal to a running VP in the ASC(s) whose threads' timers have fired, and resets the OS's alarm according to the firing time of the next alarm in its queue. Eventually, when their host ASC reschedules them, *Thread.yield()* will invoke these threads' user-provided handlers before dispatching the threads from the point where they had previously left off.

Given the availability of preemption and user-level timers, ASCs can easily implement *timesliced* threads simply by setting a user-level periodic timer; *Thread.yield()* will be invoked on each tick of this timer (in the manner described above), giving the ASC the opportunity to run a different thread. For efficiency, the timer should be owned by a 'slave' thread that never actually runs, thus saving an unnecessary context switch (e.g. in a priority based ASC, its priority could be set below that of all other threads)[1].

3.5 Critical Sections and Concurrency Control

In any concurrent system, an optimal delineation of critical sections is crucial in delivering performance and predictability while maintaining safety and correctness. In general, performance is enhanced by appropriately trading off the number and length of critical sections, and by minimising the overhead of the associated locking operations.

---

[1] In environments with relaxed soft real-time requirements, efficiency can be further increased by 'rounding up' non-periodic timers to the nearest multiple of such a periodic timer so as to 'piggy back' on the periodic timer and thus minimise the number of SIGALRMs. An alternative implementation, with a different accuracy/ overhead trade-off, could employ a separate *SignalType* object and a separate periodic alarm signal such as SIGVTALRM.

The run-time environment employs *two* levels of concurrency control. The first level protects the run-time environment itself, plus all per-ASC code[1], inside a single critical section, from the effects of preemption and multiple VPs. We are currently experimenting with refining this critical section into multiple, per ASC, sections. However, the trade-offs involved are subtle and this work remains ongoing.

The function of second level concurrency control is used to ensure that `libc` and other library calls remain atomic in the face of preemption. More specifically, it prevents the signal handling framework's generic signal handler from calling *Thread.yield()* if the interrupted VP was inside a library call when a signal occurred. If *Thread.yield()* were to be called under these circumstances, the run-time environment might subsequently dispatch another thread on the interrupted VP which attempts to make the same library call, resulting in deadlock in cases where the library call employs an internal mutex lock.

Second-level concurrency control is logically implemented as a flag (see section 3.6 for more detail) that is set before entering a library call, cleared on exiting the library call. The flag is inspected by the generic signal handler (as described in section 3.4) which will return immediately if it finds the flag set. Note that this flag based solution avoids the use of OS-level signal masking calls. This is an important benefit as these calls carry a significant overhead in frequently executed sections of code.

3.6 Adaptive Concurrency Control

The run-time environment must operate with maximal efficiency while supporting dynamically varying capsule configurations involving many different mixes of installed ASCs. In seeking maximal efficiency, we observe that a significantly reduced overhead can be incurred in cases where all currently installed ASCs are non-preemptive (as can be determined by calling each ASC's *preemptive()* operation), or where only a single ASC/VP is present. This is especially true in the case of locking. First level locking (as discussed in section 3.5), for example, is entirely redundant in a non-preemptive, single ASC/VP configuration and can be inexpensively implemented as a simple boolean flag in the preemptive single ASC/VP case. In multi-VP configurations, however, a more expensive OS-level mutex is required which, furthermore, must support a '*trylock()*' operation so that the generic signal handler can test and obtain the mutex in a single atomic action.

Similarly, second level locking is redundant in the non-preemptive, single ASC/VP case, and is sufficiently implemented as a boolean variable in the preemptive, single ASC/VP case. In multi-VP configurations, however, the logical flag must be implemented as a counter that is incremented on entering a library call wrapper and decremented on leaving. Furthermore, the increment/ decrement operations must be protected, in a short critical section, by a spinlock[2] to prevent the flag being cleared by a VP leaving a library call just after another has set the flag and entered.

To exploit the above observations, the run-time environment dynamically determines which case applies and employs the appropriate lock implementation. For simplicity, it combines the preemptive/ non-preemptive single ASC/VP cases, and thus

---

1 This design choice considerably eases the design and implementation of ASCs which do not themselves need to be concerned with concurrency issues.

2 For efficiency and portability, we implement this spinlock in user-level software using a variant of Peterson's algorithm for mutual exclusion [Silberschatz,98].

only distinguishes between single VP and multiple VP configurations. The switch-over between these two states (as determined by monitoring VP creation/ destruction) can therefore be effected by simply inverting a boolean variable which controls run-time selection of the appropriate lock implementation.

3.7 Proportional Share VPs

The usefulness of the VP concept as a building block for QoS provision is at least partially dependent on the degree to which VPs, like real CPUs, can offer a predictable execution rate of *x* instructions per second. Unfortunately, this property conflicts with traditional goals of standard kernel threads such as fairness and responsiveness to user interaction [Nieh,94].

In searching for a compromise between these conflicting requirements, we are experimenting with *proportional share* policies for the OS scheduler. One such policy is *lottery scheduling* [Waldspurger,94] which works by assigning notional *lottery tickets* to VPs and then holding a 'lottery' at each scheduling point; the winning VP claims the right to run on the CPU until the next scheduling point. The beneficial properties of lottery scheduling are:

- if a process has *x*% of the available lottery tickets it should probabilistically emulate a CPU with *physical CPU speed* ÷ *x* cycles per second;
- starvation cannot occur, as every process will win some lotteries as long as it holds tickets.

However, lottery scheduling has its drawbacks. In particular, although it shares CPU resources proportionately over the long term, its predictability is limited in the short term. It also incurs a non trivial run-time overhead. We are therefore currently evaluating an alternative scheme influenced by *stride scheduling* [Waldspurger,94] and *weighted round robin* [Silberschatz,98]. This scheme retains the benefits of lottery scheduling, but enhances it in terms of predictability and efficiency. Furthermore, it attempts to accommodate I/O-bound processes and standard workstation processes as well as VPs that can execute at a strictly predictable rate. Our results to date in this area are reported in [Moonian,00].

Of course the disadvantage of proportional share VPs is that ASCs that depend on them are not portable. Nevertheless, we believe that the ASC concept together with proportional share support has great potential in soft real-time environments such as multimedia systems and high-load servers.

4. Applying ASCs in the Middleware Environment

4.1 ASCs in GOPI

4.1.1 The GOPI Middleware Platform

The ASC concept was first developed and implemented as part of the GOPI middleware platform [Coulson,98], [Coulson,99a]. GOPI is a C-language 'micro-ORB' which provides generic middleware services (called GOPI-core) over which different middleware 'personalities' can be implemented. GOPI currently supports a CORBA personality [Coulson,00], and a multimedia personality that provides high-level media streaming and synchronisation services [Coulson,99b].

GOPI was designed with configurable resource management as a central requirement, and the ASC framework plays a key role in this. GOPI's other main QoS enabler is a configurable communications framework of 'pluggable' protocols. Protocols in this framework are responsible for mapping QoS specifications from layer to layer and, ultimately, to raw GOPI-core resources such as buffer pools, network connections and ASCs/ threads.

4.1.2 Currently Implemented ASCs

To date, we have implemented three ASCs in the GOPI environment: First, *PRIO* is a simple non-preemptive ASC that consists of less than 200 lines of C code and employs a priority based scheduling policy with round-robin scheduling of threads with equal priority. It defines the following simple QoS schema:

```
class PRIO_QoS extends QoS {
  PRIO_QoS(int priority); // constructor
  int get_priority(void);
  void set_priority(int priority);
}
```

PRIO is initially configured with a single VP but, of course, additional VPs can be added/removed as desired using the *VP* class. PRIO's VP activation policy is simply to activate one idling VP (where available) whenever a call of *schedule()* results in an additional thread being made runnable. PRIO is the only ASC configured in a newly created capsule.

Second, the *EDF* ASC provides a (slightly) richer QoS schema that supports the specification of period as well as priority. Its concurrency semantic is preemptive, and its scheduling policy is to favour threads of a higher priority, while running threads of equal priority in earliest-deadline-first order. In addition, its threads are *periodic*; i.e., EDF arranges that the *Runnable* entry-point of each of its threads is re-executed repeatedly, once per period. Furthermore, EDF monitors the extent to which its threads meet their per-period deadlines and imposes admission control when the proportion of missed deadlines exceeds a threshold. Additionally, this monitoring drives a VP activation policy that attempts to adjust the number of active VPs to maximise QoS delivery.

Finally, the *DIRECT* ASC is built around a straightforward VP activation policy which creates/ destroys a new VP every time a new thread is created/ destroyed, and reactivates idling VPs whenever threads become runnable. Thus it essentially provides kernel threads directly to its users. Like the other ASC implementations, DIRECT avoids memory wastage by creating VPs with tiny stacks. These are all that are required, as VPs spend all their time running on user-thread stacks.

We have found multiple co-existing ASCs (primarily PRIO and EDF) to be particularly beneficial in supporting GOPI's multimedia personality. More specifically, we have employed PRIO for request handling (see next section) alongside EDF for media stream processing, and have managed, even using standard kernel threads as VPs, to achieve a reasonably clean separation between the resource demands of these two distinct activities.

4.1.3 Message Handling in GOPI

In any middleware context, the communications and concurrency sub-systems must work closely together to detect, receive and process incoming messages (invocations)

in a timely and efficient way. The flexibility of the ASC framework opens up numerous ways in which this integration can be achieved, each of which offers trade-offs that will be more or less attractive to different application scenarios. Furthermore, because ASCs can be dynamically instantiated, these various possibilities can be selected and/or altered at run-time, and multiple instances can co-exist.

The default mechanism in GOPI employs a sentinel callback (see section 3.3) that checks for incoming message availability using the *poll()* system call. When the callback detects a message, it wakes a receiver thread (or one of a pre-configured pool of threads) to receive and process it. Although it has the virtue of simplicity, this default scheme suffers from a number of drawbacks. In particular, *i*) incoming messages are only detected when the capsule has no other work to do, *ii*) it incurs the overhead of a *poll()* call for each incoming message (or batch of messages), and *iii*) it additionally incurs an OS-level context switch if the receiver thread runs on a different VP to the one that executed the sentinel callback. Nevertheless, the mechanism has proved adequate in simple configurations involving one or few ASCs/VPs. GOPI configured with this mechanism and a single ASC/VP performs at least as well as state of the art research ORBs like TAO and OmniORB [Coulson,00]. It performs even better under conditions of high load because *poll()* calls are amortised over multiple requests.

Other possible mechanisms are as follows:

- The DIRECT ASC can be employed to implement standard ORB policies such as thread-per-connection or kernel-thread pools [Gokhale,98]. These solutions do not need to *poll()* as kernel threads can block in the OS on a *recv()* call without blocking the whole capsule; however, thread-per-connection does not scale well to large numbers of connections, and kernel-thread pools involve significant numbers of costly OS-level context switches.

- The signal-handling framework can be employed to request a SIGPOLL signal (or similar) when incoming requests are detected at the OS level. This results in timely responsiveness to messages although at the cost of the signal handling overhead.

- Specialised, self contained, ASCs can be developed that encapsulate the detection of incoming requests and the provision of threads for their execution. Such ASCs would provide operations to register network sockets and use specialised policies to decide how many VPs to deploy and in what configuration. In design pattern terms [Gokhale,98], such ASCs could be viewed as a Reactor combined with a Threadpool.

Finally, we have found the ASC framework's thread migration capability to be extremely useful in handling incoming messages: Objects can deal with time consuming and resource intensive invocations simply by migrating the receiver thread from the default message handling ASC to one better matched to the task in hand.

4.2 ASCs in OpenORB

Besides GOPI, we have experimented with the ASC framework in the *OpenORB* middleware environment [Blair,98]. OpenORB is an experimental platform dedicated to prototyping and evaluating *reflective* ORB structures [Blair,99], that facilitate run-time reconfiguration and management.

Importantly, OpenORB includes a flexible resource management architecture that subsumes the ASC framework [Blair,99]. This architecture supports a *task* abstraction that relates to some logical activity carried out by a set of cooperating objects. For example, the activity of incoming message handling considered above could be designated as a 'task' which is handled by distinct detector, transport and IIOP objects. Tasks are represented as objects that contain collections of resources, including ASCs/ threads. These resources are exclusively accessible to the set of objects participating in the task; for example, when an object requests resources from the platform, these are implicitly allocated from its associated task.

The resources in a task can be inspected via a hierarchical data structure of alternately layered *resources* and *resource managers*. This meta-level data structure links resources and managers with *managed-by* and *builds-on* relationships: resources are *managed by* managers, and managers *build on* lower level resources. For example, threads are *managed by* ASCs which *build on* VPs which are *managed by* the OS scheduler which *builds on* physical CPUs. Similar hierarchies are provided for other resource types. Interfaces are available which support both inspection and adaptation of resources and their managers at any level of the hierarchy. For example, the scheduling policy of an ASC can be modified or replaced. In addition, resources can be moved from one resource manager to another (this is a generalisation of the notion of migrating threads between ASCs). Full details are available in [Blair,99].

Our implementation work in OpenORB has involved two versions. The first was written in Python, and linked with the C-language implementation of the ASC framework from GOPI. The second version, currently underway, is written in C++ and employs a *component model* [Szyperski,98] that is an extension of Microsoft's COM. In this version, we are re-implementing ASCs in terms of finer grained components. This means that new ASCs can be loaded into a running capsule on the fly and that the internals of ASCs, like the scheduling policy, VP activation policy or the run queue, can also be treated as independently (un)loadable/ adaptable entities.

5. Evaluation

5.1 Scope of Evaluation

This section offers a quantitative and qualitative evaluation of the ASC framework. First, we evaluate the ASC framework's basic context switch performance and compare this to that of user-level and kernel threads in a representative commercial OS environment. This comparison yields a direct indication of the fundamental performance characteristics of the ASC framework and also determines the extent to which a gain in flexibility is being purchased at the cost of efficiency. Second, we evaluate the degree to which the whole-capsule blocking problem (disadvantage #1 in the introduction) is/ can be addressed by the ASC framework. Finally, we comment on a qualitative aspect: the portability of the framework. In sum, the evaluation attempts to determine the extent to which the ASC framework is able to deliver a useful compromise between user-level and kernel threads as outlined in the introduction.

The following experiments were carried out on a 360MHz Sun SPARC Ultra 5 with 64MB of main memory and running SunOS 5.7. All programs were compiled with GNU gcc/g++ version 2.8.1 using the -O1 optimization flag. In addition to threads implemented in the ASC framework, our experiments involved SunOS's *pthreads* implementation [Sun,94]. We used 'system scoped' pthreads as kernel

threads and ‘process scoped’ pthreads as user-level threads. Sun’s default priority based SCHED_OTHER scheduling policy was used in both cases. In all cases, VPs were realised in terms of ‘system scoped’ pthreads. All threads were created with a stack size of 32768 bytes.

Note that Sun’s user-level thread implementation, like the ASC framework, employs a two-level architecture in which user-level threads are multiplexed over kernel threads (see section 6). This means that it does not always behave like a traditional user-level thread package. Some implications of this are brought out below.

5.2 Context Switch Performance

We wrote a trivial program to evaluate the context switch overhead of the ASC framework as compared with standard user-level and kernel threads. The program creates two threads of equal priority, each of which executes 1,000,000 iterations of a tight loop that involves a context switch from one thread to the other (the loop contains nothing but a call that yields to another thread). Three versions of the program were written that respectively employed kernel threads, user-level threads and PRIO threads. The PRIO ASC was used because its inherent overhead (i.e. the overhead of its *ASC_internal.schedule()* method, which only needs to remove the first *Thread* entry from its top priority internal queue) is minimal, with the implication that the overhead incurred is almost entirely due to the ASC framework itself. Clearly, however, more complex ASCs may add additional overhead.

To evaluate the effects of the adaptive concurrency control scheme described in section 3.6, two configurations of the PRIO version of the program were written: one with a single PRIO instance containing a single VP, and another that additionally contained an instance of the EDF ASC. The purpose of the second configuration was to force the run-time environment to use the more general but less efficient locking implementation. However, since the EDF ASC contained no threads, its VPs were dormant and therefore it incurred no overhead in itself.

PRIO threads (single ASC/VP only)	4.5 secs
PRIO threads (with extra ASC/VP)	6.8 secs
SunOS user-level pthreads	20.2 secs
SunOS kernel pthreads	94.4 secs

Table 1: Relative performance of PRIO and SunOS threads.

Table 1 shows a basic speedup for PRIO threads around an order of magnitude over kernel threads (as expected) and also shows a significant speedup against SunOS user-level threads. This confirms that the ASC framework comfortably achieves the performance expected of a user-level thread implementation, despite its additional functionality. The results also confirm that the reduced locking overhead of the single-VP case represents a worthwhile optimisation.

5.3 Whole-Capsule Blocking

As explained in the introduction, the whole-capsule blocking problem is a deficiency of traditional user-level thread implementations wherein the whole capsule is blocked when a single thread performs a blocking OS call. In this section, we

investigate the extent to which the ASC framework, and in particular the PRIO ASC, is competent to address this problem.

A simple program was written which creates a number of threads, each of which carry out a two-phase task. A synchronisation barrier is inserted between the two phases so that no threads start the second phase until all have completed the first. The first phase, *blocking*, performs a 1 second OS-level *sleep()* call that would be expected to induce the whole-capsule blocking problem and thus favour kernel threads over user-level threads[1]. The second phase, *synchronising*, is an *n*-thread generalisation of the 'synchronisation loop' described in section 5.2 (the *n* threads are arranged in a logical ring such that each thread *t*, $1 \le t \le n$, yields to thread $t+1$ modulo *n*). As is clear from section 5.2, this phase would be expected to favour user-level threads over kernel threads.

Timings for an eight-thread instance of the program are shown in Fig. 2. A similar pattern was observed for other numbers of threads. The first two columns (*User 1* and *User 2*) are for SunOS user-level threads (see below), while the last (*Kernel*) is for SunOS kernel threads. The remaining columns represent the PRIO ASC with varying numbers of VPs ranging from 1 through 8. *User 1* is seen to be the fastest configuration overall, followed by PRIO in the 2 VP case. It can also be observed that, as expected, the various PRIO versions generally speed up on the blocking phase as VPs are added (because adding VPs results in more 'sleep' calls running in parallel), but slow down on the synchronisation phase (because adding VPs leads to more OS-level context switches). "User 2" is the slowest configuration overall.

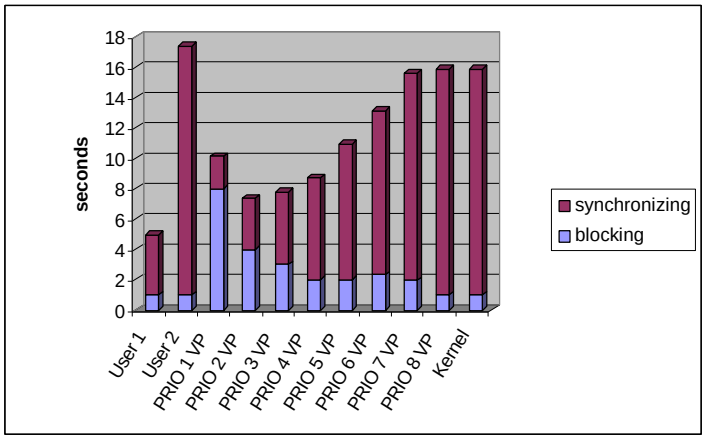

Figure 2: Whole capsule blocking experiment

Regarding the two user-level thread columns, whereas *User 2* runs the same program as the configurations discussed above, *User 1* reverses the order of the two phases; i.e., it performs *synchronisation* before *blocking*. The dramatic difference in performance between these two cases yields insight into the SunOS pthread library's

[1] Normally, of course, threads that need to block for a specified time would *not* use an OS-level *sleep()* call (precisely because this incurs the whole capsule blocking problem; it blocks the calling thread's VP which could otherwise be usefully employed running some other thread). Rather they would use an ASC framework call that allows their VP to execute some other thread in the meantime. We use OS-level *sleep()* in this experiment only as a convenient way of arranging for VP-level blocking (i.e., it is simpler to use than blocking I/O calls such as OS-level *recv()*).

approach to the whole-capsule blocking problem[1]. Its VP activation policy, which can be deduced from the close similarity between the *User 2* and *Kernel* columns, is evidently to create a new underlying kernel thread whenever a user-level thread performs a blocking system call (cf. *scheduler activations* as discussed in section 6). While this works well in the *synchronisation*-before-*blocking* case, it is clearly problematic in the reverse case, as the 8 kernel threads created during the blocking phase remain in place and become a liability in the synchronisation phase.

The first conclusion to be drawn from this experiment is that the whole-capsule blocking problem is effectively addressed by appropriately multiplexing user-level threads over kernel thread VPs. This is seen in the fact that both the SunOS user-level thread implementation and the ASC framework perform better than either pure kernel-threads or 'pure' user-level threads (the latter would presumably perform similarly to the PRIO-with-1-VP case).

Beyond this, we can further conclude that *flexibility* is of the essence. Adopting a fixed, system wide, VP activation policy (in the SunOS case, one that creates too many VPs that later may become a liability) can be significantly sub-optimal under the wrong conditions. The ASC framework is able to avoid such problems by supporting multiple ASCs, each of which is free to adopt distinct VP activation policies appropriate to its intended application domain.

5.4 Portability Issues

The minimum OS level facilities needed to implement the ASC framework are: *i*) OS provided kernel threads for VP support, *ii*) OS-level mutexes that support a '*trylock()*' call, and *iii*) a mechanism for event notification (either UNIX-like signals or a Win32-like *WaitForMultipleObjects()* call). We would expect these facilities to be available in almost any OS environment.

Beyond this, user-level thread packages have traditionally employed CPU or OS dependent code for the following user-thread related functions: *i*) initialising user-level stacks, *ii*) creating initial contexts for new user-level threads, and *iii*) saving and restoring contexts when switching user-level threads. Although this practice appears to seriously compromise portability, we follow [Engelschall,99] in arguing that machine dependent code can be eliminated on almost all platforms (at least on UNIX derived platforms) through the use of widely deployed standard facilities. For example, the POSIX `ucontext(3)` facilities address all three of the above functions. Full details are given in [Engelschall,99].

Unfortunately, in addition to the above, our implementation additionally requires atomic test-and-set (see section 3.4), and suggests the use of proportional share VPs which require specialised OS support. Atomic test-and-set must be based on a machine specific instruction such as 'test-and-set' or 'compare-and-swap', so there is no machine independent solution in this case. However, we would argue that the provision of a single assembly language function that wraps a single instruction should not constitute an unacceptably high barrier to portability. Regarding proportional share VPs, we hope to have demonstrated that the ASC concept is a useful abstraction with or without such support.

---

[1] Unlike the user-level thread case, the performance of the various PRIO configurations does not significantly change when *synchronisation* is performed before *blocking*.

A final portability related issue is the need to protect `libc` and other system library calls with a lock, as discussed in section 3.5. In a self-standing thread package it may seem a serious inconvenience to have to wrap all such calls. However, we argue that it is perfectly acceptable to do so in the context of a middleware ORB. This is because it is already standard practice in ORBs (e.g. TAO [Gokhale,98]) to export system facilities via wrapper classes, and to discourage the direct use of platform dependent facilities in the interests of making ORB application code more portable.

6. Related Work

In terms of its two-level kernel/ user-level thread structure, the ASC framework is influenced by SunOS 5.7's thread architecture [Sun,94], and by similar architectures in, for example, IBM's AIX, DEC's OSF/1 and SGI's IRIX [Engelschall,99]. In addition, the POSIX.1c-1996 standard has minimal support for a two-level thread architecture through its *pthread_setconcurrency()* call, and some Java virtual machine (JVM) implementations map language level threads onto kernel threads in a similar manner (which is designed to minimise kernel level context switches). The ASC framework is distinguished from these systems mainly in terms of its generality and flexibility. For example, unlike SunOS et al, the ASC framework allows schedulers (ASCs) to be dynamically (un)loaded, supports the co-existence of multiple scheduler instances of either the same or different types, provides QoS management facilities, and supports the use of application specific VP activation policies (see section 5.3).

The ASC framework can similarly be compared to *scheduling classes* in UNIX SVR4. A key difference here, in addition to the flexibility issue, is that ASCs should suffer less from mutual interference between classes (see [Nieh,94]) because they are implemented above the OS[1]. Govindan's work on *split-level scheduling* [Govindan,91] is also related. The goal of Govindan's work is to efficiently support EDF scheduling by minimising OS-level context switches. The ASC framework differs from split-level scheduling both in its overall goals and in its lack of dependence on specialised OS support.

In the area of OS-level research on concurrency mechanisms, Anderson's work on *scheduler activations* [Anderson,91] is of relevance, although the scope of this work is far less general than that of the ASC framework. The purpose of scheduler activations is to address the whole capsule blocking problem by 'activating' (calling) a standard entry point (usually a user-level thread scheduler) with a new kernel thread whenever a blocking system call is issued. They could thus be usefully employed in the ASC framework to provide a timely notification to VP activation policies whenever a VP blocks. Unfortunately, scheduler activations are not widely implemented in commercial OS environments, so the ASC framework must rely on more portable solutions to this problem (see, e.g., sections 4.1.2 and 5.3).

Also in the OS-level research area, the ASC framework is related to *resource containers* [Druschel,99] and *thread aggregates* [Kvalnes,00]. The goal of these two mechanisms is to support applications that use multiple threads and require predictable QoS (e.g. web servers). The approach of both is similar: to collect threads (and potentially other resources) into ring-fenced 'aggregates' which are given dedicated CPU cycles. These aggregates have notional similarities to ASCs. For example, Kvalnes's thread aggregates share a proportion of the available CPU(s), include an

[1] This will be particularly true when the underlying VPs are scheduled according to a proportional share mechanism (see section 3.5).

associated scheduling policy, and permit migration of threads between aggregates. The main differences are that resource containers and thread aggregates are implemented at the OS level and therefore forego the advantages of user-level concurrency in terms of efficiency, scalability and flexibility. In addition, these mechanisms do not support QoS specification or admission control and only allow selection of scheduling policies from a statically compiled set.

In terms of work on user-level threads, the ASC framework can be compared with the GNU Portable Threads package [Engelschall,99] and, more directly related to middleware thread support, the Cool Jazz thread package [Kramp,99]. The ASC framework borrows from Portable Threads in its approach to portability (see section 5.4) but is otherwise unrelated in its goals (e.g. the GNU package does not aim to support multiple VPs and QoS). The Cool Jazz package supports an event-based programming model in which programs are structured as finite state machines and threads execute event handlers. A user-level thread implementation is chosen for similar reasons to ours (i.e. efficiency, scalability and flexibility). Again, the ASC framework is more general in scope than Cool Jazz and we believe it should be possible to design an ASC that specifically supports the Cool Jazz programming model. Such an ASC would also, of course, be able to co-exist with other ASCs and could additionally benefit from the ASC framework's resource management capabilities.

7. Conclusions

This paper has introduced the notion of *application scheduler contexts*, and has argued that an ASC-based concurrency framework offers significant benefits in the middleware environment (we have discussed the deployment of the framework in two separate experimental middleware platforms). In particular, such a framework allows multiple scheduling 'contexts' to be defined, each with their own concurrency semantic, scheduling policy, QoS schema and VP activation policy. In addition, multiple contexts of the same or different types can co-exist at run-time and can be loaded/ unloaded as required. Finally, it is possible at run-time to change the QoS of threads and the resourcing of scheduling contexts, and to migrate threads from one to another.

Furthermore, referring back to section 1 of the paper, we hope to have substantiated our claim that the ASC framework is capable of delivering the advantages of user-level threads while largely avoiding their disadvantages. More specifically, *i*) section 5.2 has documented the *efficiency* of ASC-threads, *ii*) good *scaling* is inherent in the framework's use of user-level concurrency, and *iii*) the *flexibility* benefit is summarily addressed in the paragraph above. On the other hand, *i*) section 5.3 has shown how ASC-threads can work around the *whole capsule blocking problem*, *ii*) the *exploitation of multiple CPUs* is possible simply by virtue of the use of multiple VPs, and *iii*) as we have argued in section 5.4, the *portability issues* are significantly less serious than is often claimed.

To date, we have implemented the ASC framework in C on SunOS, Linux and Win32. We have also implemented it in C++ on Win32. As the latter platform supports an entirely different programming model for asynchronous event handing (based on kernel threads and the *WaitForMultipleObjects()* call), this version additionally demonstrates that the applicability of our signal handling framework is not restricted to UNIX-like systems that directly support signals.

In the future, we plan to exercise the ASC framework by implementing a wider range of ASCs in the GOPI environment. In particular, we hope to investigate the provision of specialised ASCs to support new CORBA features like audio/ visual streams, asynchronous method invocation and real-time events. We are also pursuing the deployment of the ASC framework in the OpenORB environment mentioned in section 4.2. Because OpenORB is far more configurable than GOPI, we expect this environment to form the basis of most of our future exploration of the ASC concept.

References

[Anderson,91] Anderson, T.E., Bershad, B.N., Lazowska, E.D. and H.M. Levy, "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism", Proc. Thirteenth ACM Symposium on Operating Systems Principles, Asilomar Conference Center, Pacific Grove, CA, USA, pp 95-109, October 1991.

[ANSA,91] The Advanced Networked Systems Architecture, "ANSA Reference Manual, Release 01.00", APM (Cambridge) Ltd. Poseidon House, Cambridge, UK, 1991.

[Blair,98] Blair G.S., Coulson G., Robin P. and Papathomas M., "An Architecture for Next Generation Middleware", Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Davies N.A.J., Raymond K. & Seitz J. (Eds.), The Lake District, UK, pp. 191-206, 15-18 September 1998.

[Blair,99] Blair, G.S., Costa, F., Coulson, G., Duran, H., Parlavantzas, N., Delpiano, F., Dumant, B., Horn, F., and Stefani, J.B., "The Design of a Resource-Aware Reflective Middleware Architecture", Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99), St-Malo, France, Springer-Verlag, LNCS, Vol. 1616, pp115-134, 1999.

[Coulson,00] Coulson, G., and Baichoo, S., "Implementing the CORBA GIOP in a High Performance Object Request Broker Environment", to appear in Springer-Verlag's Distributed Computing, early 2001.

[Coulson,98] Coulson, G and Clarke, M.W., "A Distributed Object Platform Infrastructure for Multimedia Applications", Computer Communications, Vol 21, No 9, pp 802-818, July 1998.

[Coulson,99a] Coulson, G., "A Configurable Multimedia Middleware Platform", IEEE Multimedia, Vol 6, pp 62-76, No 1, January - March 1999.

[Coulson,99b] Coulson, G., and Baichoo, S., "A Distributed Object Platform for Multimedia Applications", Proc. IEEE Multimedia Systems, Florence, Italy, ISBN 0-7695-0253-9, pp 122-126, June 1999.

[Druschel,99] Druschel, P., Banga, G., and Mogul, J.C., "Resource Containers: A New Facility for Resource Management in Server Systems", Proc. Third Symposium on Operating Systems Design and Implementation (OSDI'99), http://www.cs.rice.edu/~druschel/osdi99rc.ps.gz, New Orleans, LA, February 1999.

[Engelschall,99] Engelschall, R., "Portable Multithreading: the Signal Stack Trick for User-Space Thread Creation", paper included with *GNU Portable Threads* distribution, http://www.gnu.org/software/pth.

[Gokhale,98] Gokhale, A. and Schmidt, D.C., "Principles for Optimising CORBA Internet Inter-ORB Protocol Performance", Proc. HICSS '98, Hawaii, Jan 9th 1998, http://www.cs.wustl.edu/~schmidt/HICSS-97.ps.gz.

[Govindan,91] Govindan, R., and Anderson, D.P., "Scheduling and IPC Mechanisms for Continuous Media", *Proc 13th ACM Symposium on Operating Systems Principles*, Asilomar Conference Center, Pacific Grove, California, USA, SIGOPS, Vol 25, pp 68-80, 1991.

[Kramp,99] Kramp, T. and Koster, R., "Flexible Event-Based Threading for QoS-Supporting Middleware", Proc. Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS), IFIP, July 1999.

[Kvalnes,00] Kvalnes, A., The Vortex Project, University of Tromso, Norway, http://www.vortex.cs.uit.no/vortex.html, 2000.

[Microsoft,99] Microsoft DCOM web page: http://windows.microsoft.com/com/tech/dcom.asp.

[Moonian,00] Moonian, O and Coulson, G., "A Deterministic Adaptive Fair Share Scheduling Scheme", Internal Report, Computing Dept., Faculty of Engineering, University of Mauritius, Le Reduit, Mauritius, 2000.

[Nieh,94] Nieh, J., et al., "SVR4 UNIX Scheduler Unacceptable for Multimedia Applications", Proc, 4th Workshop on Network and Operating System Support for Digital Audio and Video, Lancaster, England, April 1994.

[OG,99] Open Group DCE Documentation web page: http://www.opengroup.org/pubs/catalog/dz.htm

[OMG,00] The Common Object Request Broker: Architecture and Specification, available at http://www.omg.org/

[Silberschatz,98] Silberschatz, A., and Galvin, P., "Operating Systems Concepts, Fifth Edition", Addison-Wesley, ISBN 0-201-54262-5, p135,1998.

[Szyperski,98] Szyperski, C., "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1998.

[Stankovic,95] Stankovic, J., "Implications of Classical Scheduling Results for Real-Time Systems", IEEE Computer, 1995.

[Sun,94] SunOS 5.7 Programmer's Manual, Pthreads entry, Sun Microsystems, 1994.

[Waldspurger,94] Waldspurger, C.A. and Weihl, W.E., "Lottery Scheduling: Flexible Proportional-Share Resource Management", Proc. First USENIX Symposium on Operating Systems Design and Implementation, November 1994. Proceedings at: http://www.usenix.org/publications/library/proceedings/osdi/index.html.