A Configurable Multimedia Middleware Platform

Geoff Coulson
Lancaster University

Experience demonstrates the benefits and feasibility of supporting multimedia applications in distributed middleware architectures. However, deployment of multimedia-capable middleware platforms has not yet occurred on a large scale. This article describes designing such a platform and its attempts to maximize performance, predictability, and configurability in a standard workstation operating-system environment.

Experience indicates both the benefits and feasibility of supporting multimedia applications in distributed middleware architectures such as the Object Management Group's (OMG) Corba (Common Object Request Broker Architecture).[1] The potential benefits of such support to application developers include

- provision of high-level programming abstractions for multimedia communications, including quality-of-service (QoS) specification and management, and
- seamless integration of multimedia with conventional distributed interactions.

In addition, the research community has demonstrated the feasibility of such support through implementation of a number of Corba-based, multimedia-capable, experimental middleware platforms (discussed later).[2-5]

Regardless, the large-scale deployment of such platforms seems unlikely in the short term. One major obstacle to deployment is the lack of standardization of appropriate programming abstractions. Despite guidelines available in the International Standardization Organization's (ISO) Reference Model for Open Distributed Processing, the OMG has not yet seriously addressed the issue. The OMG's response to demands for multimedia support in Corba (mainly from the telecommunications community) has been the Telecom SIG's "Control and Management of Audio/Video Streams" standard.[1] In my view, this represents a short-term solution to the problem because it doesn't treat continuous media as first-class data types.[6] In the long run, I believe a more integrated approach to multimedia support will be required. The other major obstacle lies in the engineering of multimedia middleware platforms. How best to structure middleware to achieve optimal QoS management remains poorly understood, particularly in terms of performance, predictability (in terms of timeliness), and configurability.[6]

It helps to distinguish static and dynamic aspects of QoS management.[7] Static aspects involve QoS specification, mapping, negotiation, and resource allocation at connection setup time. Dynamic aspects concern managing allocated resources at data transfer time to ensure continuous maintenance of required QoS levels.

This article focuses primarily on dynamic QoS management issues. In particular, I describe low-level concurrency and communications mechanisms designed to maximize performance and predictability in multimedia middleware. I implemented these mechanisms in a multimedia middleware platform called GOPI (Generic Object Platform Infrastructure). GOPI attempts to deliver (as best it can) performance and predictability in a configurable manner in a standard operating system environment. I also designed it for backward compatibility with Corba.

System support for multimedia middleware

Modern network-capable operating systems provide a system call interface rich in functionality but difficult to use and prone to error. Multimedia middleware therefore builds higher level abstractions on top of the operating system-level application programmer's interface (API) to ease the application developer's task. The key challenge involves providing useful abstractions without compromising performance and predictability. This requires careful use of the operating system-level facilities. I'll discuss the ways in which GOPI addresses this challenge later. First, I briefly review the facilities typically found in modern operating systems.

From a middleware implementation perspective, the most important system services fall under the headings of concurrency, communications, and event (or signal) handling. In addition, multimedia middleware heavily exploits calls to manage memory resources and optimized local

1070-986X/99/$10.00 © 1999 IEEE

interprocess communications (IPC) services such as shared memory or FIFOs.

The main concurrency-related services are those associated with *kernel threads* (threads supported natively by the operating system). These include calls to create kernel threads, synchronization calls on mutexes and semaphores, and thread-safe libraries (libraries whose non-reentrant routines are implicitly protected with mutexes). In addition, system designers often provide primitive facilities to support *user-level threads* (threads supported in a user-level library outside the operating system). In SunOS 5.5, for example, the `makecontext()` and `swapcontext()` calls allow a user-level thread library to manage the CPU state of user-level threads. Machine instructions such as test-and-set or compare-and-swap also prove useful as a building block for synchronization primitives in user-level thread packages.

Communications center on the socket abstraction. All modern operating systems have calls to create network sockets, to bind them to addresses, and to write and read data on them in either a blocking or nonblocking fashion. In addition, a scatter/gather I/O facility (often but not always provided) permits writing or reading structured data to or from a socket in a single call when the data resides in noncontiguous memory. Finally, system designers usually provide an I/O multiplexing call to determine which sockets can accept I/O or to block (with an optional timeout) until one or more sockets are ready.

Event-handling services inform user software of asynchronous events in the operating system, such as alarm expiration, periodic timing signals, or I/O availability on sockets. Event-masking calls protect critical user-level sections from interruption by asynchronous events. Although notoriously obtuse, event semantics provide a potentially useful service because they let the middleware learn of events in a (relatively) timely fashion without the overhead of polling with system calls.

GOPI's structure

At the coarsest level of granularity GOPI separates into two subsystems, both realized as runtime libraries linked with applications: the API personality and the GOPI core. The API personality presents GOPI core services to the application in terms of some standard interface such as Corba, with appropriate extensions for the multimedia functionality. Since the API personality does not significantly affect performance and predictability, I don't consider it further in this article; see elsewhere[2] for more details. The core services present the following low-level API (http://www.comp.lancs.ac.uk/computing/users/geoff/GOPI/index.html):

- Calls to manage location-independent communication endpoints, called irefs, and to bind these (using first- or third-party binding) according to a specified QoS, using a stack of user-selected application-specific protocols (ASPs).
- Calls to send and receive data in both request/reply and streaming mode, on bound irefs; the API optionally uses upcalls of application-defined handlers to obtain and deliver data in addition to the traditional downcall-based style of call.
- Calls to create and manage threads in association with application scheduler contexts (ASCs); calls to create and manage semaphores and timers; and calls to create and manage buffers and communicate them between threads in the same address space via channels.

I implemented the GOPI core as a set of independent modules: *base*, a collection of generally useful foundation programming classes; *thread*, a

Glossary of Terms

API	application programmer's interface
ASC	application scheduler context
ASP	application specific protocol
Corba	Common Object Request Broker Architecture
Dimma	APM Ltd. project addressing multimedia support in distributed object platorms
FIFO	first in, first out buffer
GOPI	Generic Object Platform Infrastructure
IIOP	Internet inter-ORB protocol
IP	Internet protocol
IPC	interprocess communications
ISO	International Standardization Organization
MPEG	Moving Pictures Experts Group
mutex	locking primitive implementing mutual exclusion
OMG	Object Management Group
QoS	quality-of-service
Retina	European Commision-funded project to design a Corba platform featuring streams and QoS extensions
TAO	Corba 2.0-compliant platform running on real-time operating systems
TCP	transport control protocol
UDP	user datagram protocol

"real-time" concurrency package; *msg*, a "real-time" interthread message-passing and buffer package; *comm*, an architecture for accommodating ASPs; and *bind*, a module supporting irefs, plus a binding, or connection management, protocol with QoS negotiation capabilities. Each module constitutes a separate library with minimal dependencies. For example, the thread module works as a stand-alone package, and you could replace the bind module without affecting the comm module.

Written mainly in C, GOPI runs on a variety of Unix platforms. The core consists of approximately 12,000 lines of code. A detailed description of the system including the API and the functionality of the various modules appears elsewhere.[2]

Concurrency

Next I'll consider the elements supporting concurrency in GOPI.

API overview

The following call—provided by the core GOPI API—creates threads:

```
int thread_create(Func f, int arg,
    ASC scheduler, CharSeq *params);
```

The `f` argument specifies a function for the newly created thread to execute, and the `arg` argument specifies a parameter to pass to this function. The remaining arguments let the caller select an ASC for the thread and assign per-ASC scheduling parameters (such as priority, deadline, and period). Calling an ASC-specific routine prespecifies the scheduling parameters by marshalling ASC-specific parameters p_1, p_2, ..., p_n into a `CharSeq` (character sequence) data structure:

```
void <ascname>_buildparams(CharSeq
    *params, <p1>, <p2>, ..., <pn>);
```

Another call modifies the current thread's ASC and scheduling parameters:

```
bool thread_change(ASC newscheduler,
    CharSeq *newparams);
```

For synchronization, the threads module provides semaphores and associated `thread_P()` and `thread_V()` operations. A timeout variant of the `thread_P()` operation (similar to the Posix `cond_timedwait()` service) assists in controlling temporal behavior:

```
bool thread_P_timeout(SemId sem,
    long uS);
```

This returns FALSE if the return resulted from a timeout and TRUE if the return occurred as a consequence of a `thread_V()` operation invoked by another thread. The call `thread_yield()` explicitly yields the CPU and causes a context switch. The `thread_exit()` call destroys the calling thread.

The thread module additionally supports an arbitrary number of per-thread timers. The call to set a timer follows:

```
int thread_timerset(Function
    timerfunc, long uS);
```

This call invokes the function `timerfunc`, in the context of the calling thread, after `uS` microseconds have elapsed. The system returns the timer's identifier as a reference to pass to related calls that stop the timer or read its elapsed time.

Threads and virtual processors

Internally, GOPI uses a two-level concurrency architecture that multiplexes user-level threads on top of kernel threads. I call kernel threads *virtual processors* in the GOPI context. Virtual processors spend their lives in an endless loop, repeatedly taking a user-level thread context from an ASC and executing it until it either yields or is preempted (see below). The number of virtual processors per capsule potentially affects performance and predictability (again, see below), but not the degree of user-thread concurrency available to GOPI or its applications. In particular, a capsule with a single virtual processor can support an arbitrary number of user-level threads.

I had two motives for supporting user-level threads: cheap context switches mean you can use high levels of concurrency, and the choice of scheduling policy is not restricted to whatever the operating system provides. The motive for using virtual processors arises from intraprocess parallelism on multiprocessors and more flexible scheduling options.

You can configure the thread module (actually, each ASC) according to one of the following options:

- *No preemption*. A thread only yields to another thread either explicitly or when the thread blocks.

- *Preemption*. Asynchronous events delivered to the thread module by the operating system can preempt threads.

You can alter the configuration dynamically at runtime. An option also exists to timeslice threads. I simply built this on top of the preemption option using a periodic alarm event that calls `thread_yield()` on each invocation.

ASCs

A user-level thread implementation has the significant benefit of considerable flexibility in scheduling, permitting easy implementation of a wide range of scheduling policies (such as priority based, rate monotonic, earliest deadline first, least laxity first, and so forth[8]). The GOPI thread module provides a framework of ASCs for implementing scheduling policies.

Each ASC supports the following interface:

```
int      asc_init(void);
bool     asc_admit(Thread t, CharSeq
         *params);
Thread  *asc_expel(int thread_id);
void     asc_deschedule(Thread t);
Thread  *asc_schedule(void);
```

Not directly accessible to the user, these ASC operations respond exclusively to code internal to the thread module. The user can create and install new ASCs and create threads managed by installed ASCs. ASCs can be installed dynamically at any time, and multiple instances of the same type of ASC can operate simultaneously.

Each ASC instance encapsulates a scheduling policy, a run queue, and one or more virtual processors. The `asc_init()` command creates the run queue and virtual-processor resources when called at ASC installation time. It creates virtual processors with the preemption, no preemption, or timeslicing options already described. The `asc_admit()` call tests a thread creation request for admission and, if the admission test suceeds, registers the given thread context with the ASC (which will place the thread on its run queue).

The `params` argument passed to `asc_admit()` is the same `params` argument the user originally passed to `thread_create()`. This argument remains completely opaque to the thread module and only the ASC can interpret it. The `asc_expel()` call removes a thread from an ASC; the `thread_exit()` and `thread_change()` routines use it as described later.

Virtual processors executing the thread module's context-switch code (inside `thread_yield()`) call `asc_schedule()` and `asc_deschedule()`. This code determines the calling virtual processor's "home" ASC and calls `asc_deschedule()` on the appropriate ASC, passing the user thread context of the currently executing thread as an argument. Internally, `asc_deschedule()` updates any dynamic scheduling state associated with the descheduled thread (for example, deadlines) and stores the descriptor on its private run queue. The `thread_yield()` code then calls `asc_schedule()`, which returns the user thread context that the ASC's policy has determined should run next. Finally, `thread_yield()` switches context to this newly selected thread.

The `thread_change()` routine attempts to migrate the current thread into a newly specified ASC by first removing it from its current ASC using `asc_expel()`. It then calls `asc_admit()` on the new ASC using the scheduling parameters provided. `thread_change()` succeeds or fails depending on the outcome of this call. The caller of `thread_change()` may also specify the same ASC, but with new scheduling parameters.

Locking issues

In any concurrent system, the implementation of locking proves crucial in maximizing performance and predictability. Maximum performance comes from minimizing the inherent overhead of the locking operations and appropriately trading off the number and length of critical sections. Similarly, both maximum performance and predictability result from minimizing the number of context switches incurred. In GOPI, the degree and type of locking, and thus the overhead involved, are configurable depending on whether or not any currently installed ASC has the preemption option enabled and whether just one or multiple virtual processors currently exist. GOPI implements three levels of locking as follows.

The first level of locking ensures that `libc` and other library calls are atomic in the face of preemption. Therefore, this level of locking isn't required unless some ASC has configured the preemption option (the rest of this paragraph assumes the preemption option). A configuration with more than one virtual processor further requires appropriate multithread-safe versions of the standard libraries; these include their own internal, virtual processor-level locking based on operating system-level mutexes. For a configuration with only a single virtual processor, howev-

er, a more efficient solution suffices. In such a situation, concurrency problems can only arise if the thread module preempts a thread while executing inside a standard library routine. Preventing such problems in the single-virtual-processor case just requires a simple Boolean flag; a thread making a library call sets this flag on entry and clears it on exit. Asynchronous event handlers can then decide whether to evoke a preemption (by calling `thread_yield()`) on the basis of the flag's value. Note that this solution doesn't need to use operating system-level signal masking calls. This provides an important benefit, as these calls carry significant overhead in frequently executed sections of code.

The second level of locking in the thread module concerns protecting the thread module's and ASCs' data structures (such as run queues and semaphore queues) in the face of preemption. A configuration of more than one virtual processor requires an operating system-level mutex to protect these data structures from operating system-level preemptions. Again, however, the far more efficient flag-based solution already described suffices in the special case of a single-virtual-processor configuration.

The third level of locking concerns protecting higher level GOPI modules and applications. At this level, of course, the thread module's semaphore implementation becomes available to implement locking. Each GOPI module attempts to maximize concurrency through the use of multiple, per-resource critical sections as opposed to single, per-module locks. Performance, as well as predictability, is further enhanced in that `thread_P()` doesn't cause a context switch unless it blocks, and `thread_V()` doesn't cause a context switch unless other threads block on the semaphore.

Event handling framework

The thread module employs a generic and extensible framework for operating-system-level event handling that's closely integrated with the locking scheme. The framework's objective targets handling events with minimum latency while maintaining low locking overhead. For each type of event handled, the framework user provides

- the address of a Boolean event flag variable,
- the address of an operating-system-level event handler (such as a Unix signal-handler routine) that, among other things, sets the event flag, and
- the address of a callback routine conditionally called from the thread module's context-switch code depending on the state of the associated event flag.

In operation, triggering an event (the operating system calls the event handler) prompts the event handler to first set the associated event flag. Its subsequent action then depends on whether the ASC associated with the interrupted virtual processor is configured for preemption and, if so, on the state of the level one and level two locks. If the no-preemption option is selected or if one or both of these locks are set, the handler simply returns. The callback routine will be invoked later by the thread module's context-switch code when the latter inspects the event flag and notices that it's set. If, however, the preemption option is selected and both locks are clear, the event handler itself calls `thread_yield()` before returning. This causes execution to immediately enter the context-switch code (thus preempting the currently executing thread) and directly results in invoking the callback. In either case, the callback is invoked with the minimum latency possible given the circumstances (immediately in the case of preemption or as soon as the context-switch code notices that the flag is set on the next context switch in the case of nonpreemption).

A further issue to consider is the possibility of a race condition between the setting of an event flag in the event handler and the reading and clearing of the flag by the context-switch code. Without any preventive measures, events might be missed if the event handler set the flag (that is, a new event came in) between the context-switch code detecting that the flag was set and it clearing the flag prior to invoking the callback routine. To prevent this potential pathology, the context-switch code uses a test-and-set instruction to read and clear the flag in a single atomic action.

One prominent user of the event handling framework, the timer implementation code, multiplexes virtual timers using a delta queue scheme, on top of (assuming a Unix environment) the SIGALRM event. On each invocation of the timer callback routine, the callback fires ripe timers and also resets the operating system alarm signal according to the firing time of the next alarm in its queue. These timers form the basis of the timed semaphores and also of message-passing routines in the msg module, which feature timeout options.

Other clients of the event-handling framework

include the periodic alarm handler, which uses the framework to implement timeslicing (a preemption takes place on each periodic alarm), and the comm module, which uses it to monitor I/O availability. In the Unix environment, the periodic alarm handler uses the SIGVTALRM signal, and the comm module uses the SIGPOLL signal.

Communications

Communications in GOPI involves multiple elements. We'll look at them next, from the API overview to the details of buffer management.

API overview

The primary abstraction employed by the communications system from the API user's point of view is the iref—a location-independent communication endpoint. The following call creates irefs:

```
Iref *bind_irefcreate(FlowType
    cust_flowtype, FlowType
    prov_flowtype, Function h, Aspname
    asp, CharSeq *qos);
```

The two **FlowType** arguments refer to the directionality (stream or request/reply) of the iref in each of its two roles, customer and provider (see below). The **h** argument specifies a handler: a C function pointer upcalled when data arrives at or leaves an iref. Handlers are typically written to call stub or skeleton code provided by the API personality. Data pass to or from handlers in buffers. The **asp** and **qos** arguments respectively select and configure an ASP stack associated with this iref in subsequent bindings.

Irefs are bound (connected) using the following call:

```
Iref *bind_bindreq(Iref *cust, Iref
    *prov, CharSeq *qos);
```

This call binds iref **cust** to iref **prov** (the call works regardless of the irefs' location) using the ASP associated with **prov** at creation time. **bind_bindreq()** works by invoking a generic binding protocol that negotiates a mutually acceptable QoS for the binding. If a non-null **qos** argument is given, then this is used as the target QoS; otherwise, the default QoS associated with **prov** at creation time is used. **bind_bindreq()** returns an iref on which invocations can be made to control the newly created binding (for example, start, stop, renegotiate QoS, and destroy). These control invocations result from the following routine:

```
int bind_invoke(Iref *iref, int
    operation, Buffer *args);
```

A general-purpose service, **bind_invoke()** uses the OMG's Internet Inter-ORB Protocol (IIOP)[1] to invoke an iref without requiring a prior explicit binding. To renegotiate the QoS of a binding requires calling **bind_invoke()** with the following arguments: the control iref returned from **bind_bindreq()**, an int constant RENEG, and a buffer into which a new target QoS has been marshalled from an ASP-specific **CharSeq** specification.

You can see that ASPs play a role in the communications module very similar to that of ASCs in the thread module; both provide application-specific behavior and are configured and reconfigured using specifications defined according to their own schema. As with ASCs, per-ASP routines marshall QoS parameters into **CharSeq**s for passing to **bind_irefcreate()** and **bind_bindreq()**. Besides configuring ASP behavior, QoS specifications determine the requested binding's resource requirement (transport sockets, threads, and buffers). Furthermore, ASPs are often written in terms of other ASPs that provide a lower level service. In such cases, the QoS specification determines the lower level ASPs selected and their QoS parameters. This process, recursively applied, leads to an arbitrarily deep protocol stack, the form of which is a function of the QoS specification passed to the top-level ASP.

ASPs

The comm module—a multilevel protocol framework—enables building bindings from stacks of ASPs in turn stacked on top of transport protocols. (The current implementation makes TCP/IP, UDP/IP, and Unix FIFOs available). To date, I have implemented ASPs for IIOP, adaptive audio, and intercapsule shared memory communication.[2] Each ASP supports the following interface (slightly simplified):

```
int  asp_listen(Profile *pf, FlowType
     flowtype);
int  asp_connect(Profile *pf, FlowType
     flowtype);
int  asp_accept(Profile *pf, int
     listening_sap);
int  asp_relisten(int sap, Profile
     *pf);
int  asp_reconnect(int sap, Profile
     *pf);
```

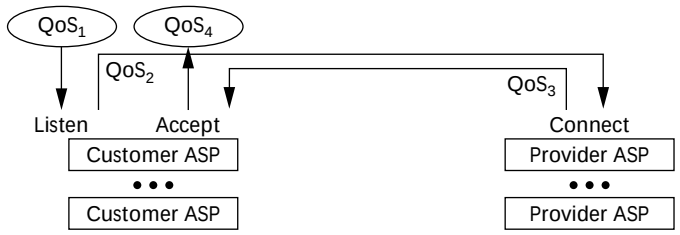

Figure 1. The binding protocol.

```
int  asp_reaccept(int sap, Profile
     *pf);
int  asp_close(int sap);
int  asp_hdrsize(int sap);
int  asp_flipqos(CharSeq *cs_qos);
int  asp_send(int sap, Buffer *buf);
int  asp_receive(int sap, Buffer
     **outbuf);
int  asp_call(int sap, Buffer *inbuf,
     Buffer **outbuf);
```

The binding protocol (see Figure 1) uses `asp_listen()`, `asp_connect()`, and `asp_accept()` to establish a mutually acceptable level of QoS between the customer and provider using the specified ASP. In the first instance, the target QoS from `bind_bindreq()` passes (in a `Profile` data type, which also contains addressing information) to `asp_listen()` at the customer side. `asp_listen()` tentatively allocates resources based on the given `Profile` and returns a local identifier for the binding that will later go to `asp_accept()`. `asp_listen()` may also modify the QoS information in its `Profile` argument if it would like to propose a different QoS from the initial target. The binding protocol then sends this (possibly modified) `Profile` to its peer, which passes it to the provider's `asp_connect()` routine. This works similarly to `asp_listen()`, again returning a local identifier and a possibly modified `Profile`. Finally, this `Profile` passes to `asp_accept()` back at the customer side, which makes the final decision as to the binding's viability.

`asp_relisten()`, `asp_reconnect()`, and `asp_reaccept()` operate similarly to `asp_listen()`, `asp_connect()`, and `asp_accept()`, but when the binding protocol is invoked on a current binding to renegotiate its QoS. As with the initial binding procedure, the ASP involved entirely determines the semantics of renegotiation. For example, some ASPs may ignore renegotiation requests, while others may only renegotiate if the binding is currently in a stopped state.

The remaining ASP routines are conceptually straightforward. `asp_close()` destroys an existing binding, `asp_hdrsize()` returns the size of header used by this ASP, and `asp_flipqos()` reverses the byte ordering of words in a QoS specification when a `CharSeq` QoS specification arrives from an opposite-end machine. Finally, the ASP supports the data transfer operations `asp_send()`, `asp_receive()`, and `asp_call()`. ASP implementations can assume that their data transfer operations are only called when the underlying system can accept or deliver data. For example, when an ASP's `receive()` entry point is called, it knows data awaits reading from the ASP or transport protocol below it without blocking. In general, programmers write ASPs for use in both request/reply or streaming mode interactions; the bind module chooses the interaction style (as determined by the `FlowType` argument passed to `asp_listen()` and `asp_connect()`). Full details of the ASP architecture and its relationship to the bind module appear elsewhere.[2]

Maximizing performance in a middleware-based communications system requires involving as few context switches, locking operations, and copy operations as possible. Maximizing predictability further requires isolating bindings from each other to the extent possible. To help achieve both these ends, the GOPI design employs nonmultiplexed bindings, each with its own operating sytem-level socket, its own thread, and its own state in each ASP (all allocated during binding protocol execution). Nonmultiplexed bindings eliminate the QoS crosstalk incurred when bindings with different QoS requirements are multiplexed onto some common layer.

You can configure each binding in the comm module according to one of the following options:

- *I/O polling* (the default option), in which the sentinel thread (that is, a distinguished thread that runs when no other thread is runnable) explicitly checks I/O availability every so often by means of an operating system primitive such as Unix's `poll()`, or
- *I/O notification*, in which the thread module's event handling framework lets the operating system asynchronously inform the comm module of I/O availability on the binding as soon as it happens.

The I/O notification option provides maximally timely responsiveness to incoming packets from the network—particularly if the thread mod-

ule's preemption option is selected. The I/O polling option, on the other hand, avoids the overhead of event handling (both options require `poll()`, as will become clear below) and thus may prove more efficient in a capsule with few threads and little else to do other than receive packets from the network.

The I/O data path

At the bottom level of the communications module lies a routine called `io_handler()`. When configuring bindings for I/O notification, `io_handler()` serves as the callback routine in the thread module's event handling framework. Thus it's called as soon as the operating system notifies its associated event (for example, SIGPOLL, assuming a Unix environment). On the other hand, selecting the I/O polling option requires calling `io_handler()` explicitly. This opens up a range of possibilities as to how often and under what circumstances to actually call `io_handler()`. The current implementation calls it from the sentinel thread referred to above.

When `io_handler()` executes, it first calls the `poll()` system call (or equivalent) to determine which currently open sockets await I/O. `io_handler()` then sends a notification message, using the services of the msg module, to the per-binding thread associated with each ready socket. On receiving this message, the per-binding threads call their associated ASP stacks to read/write data from/to the network as appropriate. A per-binding masking scheme employed in the receive direction prevents notifying per-binding threads more than once for the same incoming packet. Messages go to per-binding threads only if the mask remains clear. If the mask is clear, `io_handler()` sets it on sending a message, and the per-binding thread clears it when the corresponding packet has arrived.

Note that, because each binding has its own thread, the thread's associated ASC (whose scheduling decisions are, in turn, controlled by individual bindings' QoS specifications) determines the time and order in which bindings actually get to read/write from/to the network. This means that GOPI avoids processing low-urgency messages in favor of high-urgency messages. It also means GOPI avoids situations in which the steady flow of packets on one or more stream bindings indefinitely delays one-shot control messages (such as video stop/start). This pathology commonly afflicts many existing ORBs when used to support multimedia applications.

I/O notification issues

The previous description of the I/O data path omitted certain issues arising from I/O notification. The often complex semantics of event-driven I/O in operating systems require careful design to exploit effectively.

First, events such as Unix's SIGPOLL do not carry any information about which sockets are ready for I/O given multiple open sockets. Therefore, the middleware must explicitly issue a `poll()` call following receipt of the event to obtain this information. (GOPI calls `io_handler()`, which in turn calls `poll()`.) This implies the overhead of an event, a `poll()` call and a read/write for each network packet received via the I/O notification option.

Second, in many systems the middleware should not assume that each packet arrival generates exactly one event. On the one hand, when a number of packets addressed to different sockets all arrive from the network at around the same time, their arrival might evoke only a single event. On the other hand, many operating systems only generate an event for incoming data when the state of one or more sockets changes from "no data available on this socket" to "some data available on this socket." Hence, while the middleware prepares to read a packet following notification of its arrival by an event, another packet can arrive that does not evoke an event (because the condition "some data available on this socket" still holds). This means that relying exclusively on events to signal arriving packets may result in packets received at the operating-system level going unperceived and thus unreceived at the middleware level.

GOPI adopts the default solution of receiving a single packet for each event and additionally relying on a subsequent `io_handler()` call to detect any further packets that arrived without evoking an event. A subsequent `io_handler()` call occurs sooner or later—guaranteed—either because a further event occurs when a packet arrives on some other socket or because the sentinel thread will eventually run and issue an `io_handler()` call.

GOPI includes two additional possibilities. The first configures the system to always issue an explicit `io_handler()` call after reading each packet. This buys possibly improved predictability in receiving unperceived packets at the expense of a not inconsiderable performance hit. The second designs ASPs to read not just a single packet, but as many packets as the binding has buffer

space available. A nonblocking read means that the call returns immediately despite the number of packets actually available. This scheme looks attractive in that a single system call can receive possibly multiple packets. However, it works only where the ASP knows in advance the size of expected packets. Assuming noncontiguous buffers, the byte offsets of the unperceived packets—as required by a scatter/gather mode read—cannot otherwise be known at the time the read is issued.

Buffer management

Buffers in GOPI are allocated from preallocated pools of memory owned by the msg module. A "buddy system" allocation scheme[9] efficiently allocates and deallocates power-of-two sized buffers. A user-level buffer manager has the benefit that you can tune its performance according to expected usage patterns. In addition, it avoids the overhead of frequent `malloc()` calls, which become particularly expensive when using operating system-level locking in a multiple-virtual-processor environment. Buffers pass by reference between ASPs in a stacked binding; no copying occurs.

GOPI requires that all ASPs use fixed-size packet headers. Before allocating a buffer, an `asp_hdrsize()` call takes place on the top-level ASP in a binding to determine the required header size for the whole stack. (This call propagates down to lower level ASPs to obtain the total header-size requirement.) Buffers have an associated "current header pointer" that ASPs "push" and "pop" as the buffer passes up and down the stack.

The calls to allocate buffers, together with their headers, and to free buffers follow:

```
Buffer *msg_bufget(int hdrsize, int
    size);
void msg_bufput(Buffer *b);
Buffer *msg_bufmake(int hdrsize,
    char *header, int size, char
    *data);
void msg_bufgetfrags(Buffer
    *origbuffer, int fragsize, char
    *fraghdr,int fraghdrsize; Buffer
    *frags[], int *numfrags);
```

`msg_bufget()` allocates a buffer of size `size` from the buffer pool. If a nonzero `hdrsize` argument goes to `msg_bufget()`, a header area of size `hdrsize` is additionally allocated contiguously with the start of the buffer's data area. `msg_bufput()` decrements the given buffer's reference count and deallocates the buffer if the reference count has become zero. `msg_bufmake()` differs from `msg_bufget()` in that

- the caller (in the `data` argument) supplies the buffer's data area rather than taking it from the buffer pool, and
- given a null `header`, then a header area of size `hdrsize` is taken from the buffer pool, otherwise the given memory serves as a header.

`msg_bufmake()` proves useful in situations where ASPs manage a device such as an audio card or an area of frame buffer mapped to an address in the user virtual address space. In such a case, a buffer built using this address would permit receiving/sending data straight from/to the device without incurring any copy overhead.

`msg_bufgetfrags()` performs logical fragmentation on buffers. The call takes a buffer to fragment and returns an array of new buffers, each of which points into the original buffer's data area at an appropriate offset. `fraghdrsize` specifies the fragment header size, and `numfrags` returns the number of fragments produced. Given a nonnull `fraghdr` argument, this same piece of memory serves as the header for all the fragments. Otherwise, a new area of pool memory is allocated for each fragment header.

To prevent deallocation of the original buffer before deallocation of all the fragments, `numfrags` increments the original buffer's reference count. Also, the fragment buffers keep a reference to it so that they can decrement its reference count when they are deallocated. `msg_bufgetfrags()` is implemented in terms of `msg_bufmake()`. The data argument to `msg_bufmake()` consists of some address within the original buffer's data area, and the header-related arguments depend on the value of `fraghdr`.

Ideally, an ASP stack design will mandate performing fragmentation only once, by the bottom level ASP that interfaces directly with the transport layer. In such cases a technique called *header borrowing* offers an alternative to fragmenting with `msg_bufgetfrags()`. In header borrowing, fragment headers are written into header-sized pieces of memory within the packet buffer's data area so that the headers are contiguous with their payloads. For the first fragment of a packet, the ASP uses the header area specified in the original `msg_bufget()` call. For the remaining fragments, the ASP saves into temporary storage the header-

sized piece of data directly in front of the fragment and uses this memory for the fragment's header. Following transmission of each fragment, the borrowed memory is restored. (A similar scheme in the reverse direction handles defragmentation.) The relative costs of using header borrowing and `msg_bufgetfrags()`-based fragmentation follow:

- Header borrowing involves two copy operations of a header-sized piece of memory per fragment. Scatter/gather I/O isn't required, as the headers and payloads remain contiguous.
- `msg_bufgetfrags()` involves allocating memory for fragment buffer structs, headers, and scatter/gather data structures, and assigning the associated pointers.

The trade-off will likely favor header borrowing with relatively small headers and relatively many fragments.

Finally, when implementing ASPs having timeliness as a prime requirement, it's advisable to use fixed-size packets so that you can employ the scheme described for maximizing the I/O notification's timeliness. Using fragmentation, this implies that each packet's final fragment should be padded with garbage data to make it the same size as all the others. (To ensure space for this garbage data, overallocate the buffer to a multiple of the fragment size.)

Such a policy's effectiveness depends heavily on the fragment size chosen. Large fragments work well in that many packets won't need fragmenting, but not so well in that the last fragment may contain a lot of garbage data. Most ASPs developed so far employ the approach of letting the user set the fragmentation size as a QoS parameter. This assumes that the user has prior knowledge of likely packet size distribution and can thus make an informed trade-off.

Performance evaluation

This section describes experiments evaluating the GOPI platform's performance, specifically the extent to which GOPI can perform well despite offering high-level abstractions and a high degree of configurability. I did not design the experiments to evaluate the configurability properties themselves. These are more appropriately evaluated through API personalities and applications that exploit the platform's configurability in meaningful ways—an issue for future work.

Table 1. Relative performance of GOPI and SunOS threads.

GOPI threads	12.1 seconds
SunOS threads	24.7 seconds

Table 2. Relative performance of `msg_bufget()` and `malloc()`.

`msg_bufget()`, `msg_bufput()`	2.159 seconds
`malloc()`, `free()`	5.352 seconds

Furthermore, the experiments do not explicitly evaluate predictability because predictability is primarily a function of specific policies and only secondarily a function of the middleware architecture itself.

Threads

I wrote a simple program to evaluate the user-level thread package's performance. The program creates two threads, each of which execute a tight loop containing nothing but a yield call. Each thread loops 500,000 times. I ran two versions of this program on a Sparcserver-1000 running SunOS 5.5. One program uses GOPI threads, `thread_yield()`, and a simple, single-virtual-processor ASC that implements priority-based scheduling with earliest-deadline-first scheduling within priority bands. The other program uses SunOS detached processor-scoped threads (user threads), the SunOS `thr_yield()` call, and Sun's SCHED_OTHER scheduling policy (a priority-based policy).

The results, shown in Table 1, show a speedup for GOPI threads of 24.706/12.095 = 2.04 over SunOS 5.5 user-level threads.

Buffer management

I wrote a simple program to evaluate the user-level buffer management scheme's performance. The program repeatedly allocates and deallocates buffers over 1,000,000 iterations. The size of each allocated buffer comes from a pseudorandom sequence. One version of the program uses GOPI's `msg_bufget()` and `msg_bufput()` and the other uses the Unix `malloc()` and `free()` calls. Both versions used the same pseudorandom sequence of buffer sizes.

The results, shown in Table 2, show a speedup

Table 3. End-to-end round trips per second.

Packet Size (bytes)	1	1,024	8,192
Socket (no `poll()`)	1,015	861	484
Socket (with `poll()`)	999	839	442
GOPI (I/O polling)	743	734	402
GOPI (I/O notification)	664	522	350
GOPI versus Sockets Overhead	25 percent	12 percent	9 percent
Cool-ORB	464	420	260

Table 4. Stream performance with varying numbers of bindings (in seconds).

Number of bindings	1	2	4	8
Time for 160,000 packets	15.99s	8.55s	7.97s	7.52s

for the GOPI buffer management scheme of 5.352/2.159 = 2.48 over `malloc()` and `free()`. This occurs despite the fact that the GOPI buffer management routines allocate and initialize a `Buffer` struct (allocated from a separate dedicated pool) in addition to simply allocating memory.

End-to-end communications

To evaluate the end-to-end performance of the GOPI communications architecture, I carried out two experiments: one to measure performance relative to comparable systems and the other to evaluate the scalability of the nonmultiplexed communications architecture.

The first experiment compared GOPI to a minimal baseline Berkeley sockets program and to a commercial ORB, Chorus Systems' Cool-ORB 4.1. In fact, I used two Berkeley sockets programs—one with a null `poll()` system call inserted before each read and another without the `poll()`—and two GOPI programs—one configured to use I/O polling and the other, I/O notification. To obtain a fair comparison with the GOPI core, I modified the Cool-ORB program to avoid marshalling overhead.

I configured all five test programs as client-server pairs in which the client made repeated request/reply calls on the server. TCP served as the underlying transport protocol for all the test programs, none of which touched the data sent/received in any way. Both the clients and servers ran on the same machine to minimize the effects of load fluctuation.

Table 3 shows the number of round-trip server invocations measured per second for each program (averaged over 1,000,000 calls). The table gives figures for three packet sizes: small, medium, and large. The GOPI versus Sockets Overhead figures give the percentage overhead incurred by GOPI (with the I/O polling configuration) versus the minimal socket program (with `poll()`). It seems reasonable to use the socket program with `poll()` rather than the simplest and fastest socket program as a baseline reference for this performance comparison because all user-level multithreaded systems are inevitably required to use I/O polling of some kind.

Clearly, GOPI's performance compares very favorably with that of Cool-ORB. It also—as expected—performs worse than the baseline sockets program, although the overhead is probably not unacceptable when using larger sized packets (particularly as large packets are the norm for multimedia data).

In comparing GOPI with the sockets program, observe that GOPI carries the following overheads: thread context switching and message passing, buffer (de)allocation, ASP layer execution and header processing, and multiplexing/demultiplexing to/from irefs. Also note that I made no serious attempt to optimize GOPI. The I/O notification performance, as expected, proved inferior to I/O polling due to the additional overhead of event handling. However, I/O notification in GOPI still easily outperformed Cool-ORB and has the additional benefit of increased timeliness and predictability.

The second experiment evaluated the scalability of GOPI's nonmultiplexed communications architecture. I wrote a GOPI program that established varying numbers of bindings to transmit a given total number of packets. The two parts of the program again ran on the same machine. I used packet sizes of 1,024 bytes over a stream binding using the shared memory ASP. (I did this to avoid either packet loss, possible with a UDP/IP-based ASP, or slow start/buffering effects, possible with TCP/IP- and FIFO-based ASPs.) The binding ran as fast as the system would allow. I measured the time to receive 160,000 packets at the receiver (multiple averaged runs obtained the given figures).

The results in Table 4 showed an increase in performance for more than one binding over the single binding case. This counterintuitive result probably happens because fewer than one `poll()` call per receive occurs with multiple receiving bindings in place (`poll()` reports more than one socket with data waiting). There was then little appreciable difference in overall throughput for $n = 2$ and $n = 8$ bindings. This result provides evidence that GOPI's improved QoS predictability

(achieved through nonmultiplexing) doesn't come at the expense of an unacceptable degradation in performance.

ASP stacking overhead

I performed a final experiment to assess the ASP stacking framework's overhead using as the baseline the I/O polling configuration. To assess the ASP stacking's overhead, I wrote a minimal ASP that could configure multiple instances of itself in a stack, with the actual number of instances determined by a QoS parameter. I configured varying numbers of instances of this ASP above the baseline configuration. In terms of overhead, each instance contributed only a four-byte header and the minimum possible amount of processing.

The results in Table 5 show an acceptable overhead for relatively small numbers of stacked ASPs. Implementing stacks composed of large numbers of fine-grained micro-protocols would require optimization, however.

Table 5. ASP stacking overhead.

Configuration	Calls per Second	Percent Overhead
Baseline	734	-
Baseline + 1 null layer	716	2.6 percent
Baseline + 8 null layers	607	17.4 percent
Baseline + 16 null layers	531	27.7 percent

Implications for the operating system

Despite GOPI's main goal of operating in a standard operating-system environment, my experience suggests that a few localized and relatively straightforward modifications to the operating system could considerably enhance the performance and predictability of multimedia middleware. The first such modification, and the simplest to effect, would be to reimplement key informational system calls, making the information available to the middleware without the overhead of a system call. Examples include the Unix calls `gettimeofday()`, used to read the system clock, and `pthread_self()`, used to obtain a virtual processor's id. Such calls could be reimplemented by mapping operating-system virtual memory areas—to be written by the operating system and read by the application—into application address spaces.

The second suggested modification would provide per-socket I/O readiness information with the SIGPOLL signal, probably as an extension of the existing `siginfo` mechanism. The availability of such information would eliminate the need for a `poll()` call after each signal and would thus considerably reduce the I/O notification overhead.

Note that I don't recommend changing the semantics of SIGPOLL so that each packet arrival generates a SIGPOLL. Although the fact that some packets may fail to evoke a SIGPOLL leads to complexity in the middleware, good reasons exist to live with this semantic. First, a per-packet SIGPOLL would only work for datagram sockets. With stream sockets using TCP, the operating system has no way to identify a user-level packet, so the middleware would still require the additional complexity. Second, per-packet SIGPOLLs would add overhead. It's not at all clear that this would be less than the overhead of the user-level schemes already described.

My third suggestion: consider replacing the traditional priority-based scheduler with a fair-share policy such as lottery scheduling.[10] Clearly more wide-ranging and controversial than the other suggestions, this proposal nevertheless merits investigation. Fair-share scheduling offers a far superior basis on which to build predictable user-level ASCs while simultaneously providing obligatory operating system-level properties such as fairness. Lottery scheduling works by assigning notional lottery tickets to processes (or kernel threads) and then holding a lottery at each scheduling point; the winning process runs on the CPU until the next scheduling point. Key properties include

- starvation doesn't occur, as every process will win some lotteries as long as it holds at least some tickets, and
- if a process has x percent of the available lottery tickets, it will probabilistically take x percent of the available CPU time.

Thus, virtual processors implemented as lottery-scheduled kernel threads come much closer to "real" CPUs. In fact, each virtual processor approximates a CPU with a speed of S/x where S represents the speed of the physical CPU and x the percentage of lottery tickets it holds. Such support would allow user-level ASCs to operate in a considerably more predictable manner and to perform meaningful admission tests at the user level.

As a refinement, if the operating system had to vary the percentage of lottery tickets assigned to a given virtual processor, it could deliver an event

to the thread module. This would let ASCs adapt appropriately (for example, by changing scheduling policy or admission test criteria).

Related work

Recent work in the OMG's Corba forum has addressed the need for streams and real-time services in distributed object platforms. In particular, the Telecom Special Interest Group's (SIG's) specification for the "Control and Management of Audio/Visual Streams"[11] addressed the need for streams in Corba, and the Real-time SIG's (ongoing) "Real-time Corba" specification[12] addresses the need for more general real-time functionality.

As stated in the introduction, GOPI's approach to supporting streams differs fundamentally from the OMG's. The GOPI approach treats stream data as far as possible in the same manner and in the same environment as conventional data, whereas the OMG approach transports stream data out-of-band and only uses the ORB for stream control and management.

The Real-time Corba activity defines a number of concepts also found in GOPI (such as flexible threads and scheduling, pluggable transports, and higher level protocols). However, I can't yet make a detailed comparison because, although the various contributors have implemented aspects of the Real-time Corba specification, no integrated implementation of the whole design yet exists.

The European Commission-funded Retina project is designing a Corba platform featuring streams and QoS extensions.[3] The architecture cleanly separates between

- ORB support mechanisms such as interface reference management, threads, buffers, and so forth, and
- binding classes that provide communications services tailored to particular applications.

While comparable in terms of their overall goals, Retina focuses more on static QoS management issues such as binding establishment where GOPI emphasises dynamic QoS management issues.

The Dimma project[4] at APM in Cambridge, UK also addresses issues of multimedia support in distributed object platforms. Dimma builds on the ANSAware distributed systems platform enhanced with abstractions for resource management and a flexible multiplexing structure. Like GOPI, the degree of per-binding internal multiplexing can be configured according to the needs of different applications. Compared to GOPI, Dimma focuses more on global QoS configurability and less on the fine-grained QoS configurability of individual bindings. The finer grained configurability in GOPI arises largely through the ASC and ASP frameworks, lacking in Dimma.

TAO,[5] a Corba 2.0-compliant platform, runs on real-time operating systems. Although TAO's designers considered multimedia support, and recently implemented the OMG's Control and Management of A/V Streams specification in TAO,[13] the system appears primarily designed for hard real-time applications such as avionics. TAO features a real-time message scheduling service that can provide deterministic temporal guarantees (given a real-time operating-system infrastructure) by avoiding priority inversion and nondeterminism. In contrast to TAO, I designed GOPI to provide integrated multimedia support in a conventional operating-system/network environment and to emphasize flexible user services support rather than hard deterministic performance.

Related work in more specific areas also influenced GOPI. In terms of scheduling, split-level scheduling[14] and SunOS 5.5's two-level thread architecture influenced GOPI's two-level thread architecture. The GOPI architecture differs from split-level scheduling in that it does not depend on specialized operating-system support for user-level threads. It differs from the SunOS design mainly in terms of flexibility. For example, GOPI allows adding new schedulers (both preemptive or nonpreemptive) dynamically, and the event handling framework allows close integration between external events and scheduling.

GOPI's ASC framework also compares to scheduling classes in Unix SVR4, except that ASCs are user level and scheduling classes are kernel level. As a result, ASCs suffer less from mutual interference between classes,[15] particularly when the underlying virtual processors are lottery scheduled.

In terms of communications, flexible protocol frameworks such as the x-kernel[16] and Ensemble[17] clearly influenced GOPI's ASP framework. GOPI differs from these earlier systems primarily in its approach to QoS configurability. In most previous designs, the builder of a stack would attempt to realize a given QoS requirement by explicitly selecting layers and individually configuring them in an appropriate way. Although superficially attractive, this approach has the drawback that the stack-builder code must "know" how to map

the given QoS specification to the configuration parameters of each selected layer. Furthermore, it must recognize dependencies between layers with various configurations. (For example, the degree of compression supported by an MPEG layer may affect the choice of throughput and delay parameters given to a transport layer.)

Although small numbers of layer types keep these problems surmountable, the approach does not scale well for many layer types and when new QoS parameters must be introduced. To avoid these QoS mapping problems and to better support extensibility, GOPI builds stacks in an implicit, encapsulated manner. The stack builder specifies a single top-level ASP and configures it using the ASP's own QoS schema. The ASP instance itself then autonomously realizes the required QoS by mapping its QoS parameters to those of further, encapsulated, ASP instances (and so on recursively).

A second way in which GOPI differs from most previous protocol frameworks lies in its approach to scheduling. The x-kernel and Ensemble, in common with most other frameworks, execute stacks by dedicating a single thread to all stacks and using an external scheduler. In other words, they divide the total workload into discrete steps (such as the execution of individual layers in individual stacks) and serially execute these steps according to some policy (like first-come, first-served or priority order) as they become ready. Although this approach allows some limited control over each individual stack's QoS, typically by assigning different priorities to different stacks' steps, the GOPI thread-per-stack approach proves much more configurable and fine grained.

Conclusion

The GOPI middleware platform supports multimedia applications in a standard operating-system environment. Designed in a modular fashion, GOPI implements a number of configurable mechanisms to enhance performance and predictability. Its performance demonstrates that despite its high degree of configurability, GOPI compares favorably with related systems and can adequately meet multimedia applications' performance requirements. Moreover, because audio and video stream bindings typically require zero or minimal marshalling, a GOPI-based Corba implementation should incur little if any performance penalty above that already discussed.

The main dimensions of configurability in GOPI follow:

Threads

- Configure by choosing appropriate ASC and scheduling parameters
- Change ASC and scheduling parameters of threads at runtime
- Add new ASCs dynamically
- Configure ASCs with preemption, no preemption, or timeslicing

Bindings/communications

- Configure by choosing appropriate ASPs and QoS
- Renegotiate ASPs and binding QoS at runtime (not addressed in this article)
- Add new ASPs dynamically
- Configure ASPs with I/O polling or I/O notification

The ASC and ASP frameworks prove central to this configurability. These frameworks make it straightforward to customize middleware for specific applications by designing and installing specialized plug-in modules. Furthermore, these frameworks effectively sidestep the QoS mapping problem in which QoS specifications in some general-purpose high-level notation must be somehow mapped to low-level resources such as transport sockets, virtual processors, and memory. This mapping becomes trivial in GOPI because ASCs and ASPs define their own specialized QoS specification schema, and the mapping from this schema to generic resource managers (provided by GOPI) is hard wired into their code.

In future work, I intend to further develop support for configurability. One direction would port GOPI to the Windows NT environment to exploit NT's flexible support for dynamic loading to load new ASCs and ASPs at runtime. I also want to design appropriate APIs to permit maximal configurability while not overburdening the programmer with confusing detail. To this end, I am investigating the concept of reflection,[18] which separates basic behavior from metabehavior (that is, configurability). A report on early work in this direction appears elsewhere.[19] MM

References

1. *The Common Object Request Broker: Architecture and Specification 2.2*, available at http://www.omg.org/.
2. G. Coulson and M.W. Clarke, "A Distributed Object Platform Infrastructure for Multimedia Applications," *Computer Comm.*, Vol. 21, No. 9, pp. 802-818, July 1998.
3. F. Dang Tran, "Binding and Streams: The Retina Approach," *Proc. TINA 96*, 1996, available at http://www.uk.infowin.org/ACTS/RUS/PROJECTS/ac948.htm.
4. D. Donaldson, "Dimma—A Multimedia ORB," *Proc. Middleware 98*, Springer Verlag, London, UK, Sept. 1998, pp. 141-156.
5. D.C. Schmidt, *The Design of the TAO Real-Time Object Request Broker*, http://www.cs.wustl.edu/~schmidt/new.html#corba.
6. G.S. Blair and J.B. Stefani, *Open Distributed Processing and Multimedia*, Addison-Wesley, Reading, Mass., 1997.
7. G. Coulson and J. de Meer, "Guest Editor's Introduction, Special Issue on Quality of Service," *Distributed Systems Engineering J.*, Vol. 4, No. 1, Mar. 1997, pp. 1-3.
8. J. Stankovic et al., "Implications of Classical Scheduling Results for Real-Time Systems," *Computer*, Vol. 28, No. 6, June 1995, pp. 16-25.
9. D.E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, Mass., 1973.
10. C.A. Waldspurger and W.E Weihl, "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. First Symp. on Operating Systems Design and Implementation*, Nov. 1994, available at http://www.usenix.org/publications/library/proceedings/osdi/index.html.
11. "Control and Management of Audio/Visual Streams," OMG Document number telecom/97-05-07; available from http://www.omg.org/library/schedule/AV_Streams_RTF.htm.
12. "Real-time Corba V1.1," Initial Submission to Real-time SIG's RFP on Realtime Corba, OMG Document number orbos/98-01-08, available at http://www.omg.org.
13. S. Mungee, N. Surendran, and D. Schmidt, *The Design and Implementation of a Corba Audio/Video Streaming Service*, Washington Univ. Tech. Report WUCS-98-15, Dept. of Computer Science, Washington University at St Louis, Mo., May 1998.
14. R. Govindan and D.P. Anderson, "Scheduling and IPC Mechanisms for Continuous Media," *Proc 13th ACM Symp. on Operating Systems Principles*, SIGOPS, Vol. 25, ACM Press, New York, 1991, pp. 68-80.
15. J. Nieh et al., "SVR4 Unix Scheduler Unacceptable for Multimedia Applications," *Proc. 4th Workshop on Network and Operating System Support for Digital Audio and Video*, Springer Verlag Lecture Notes in Computer Science, Springer Verlag, Berlin, April 1994, pp. 41-53.
16. N. Hutchinson and L. Peterson, "The x-kernel: An Architecture for Implementing Network Protocols," *IEEE Trans. on Software Engineering*, Vol. 17, No. 1, Jan. 1991, pp. 64-76.
17. R. van Renesse, "Masking the Overhead of Protocol Layering," *Proc. 1996 ACM SIGCOMM Conf.*, Stanford, ACM Press, New York, Sept. 1996, pp. 96-105.
18. P. Maes, "Concepts and Experiments in Computational Reflection," *Proc. of OOPSLA 87*, ACM SIGPLAN Notices, Vol. 22, ACM Press, New York, 1987, pp. 147-155.
19. G.S. Blair, "An Architecture for Next-Generation Middleware," *Proc. Middleware 98*, Springer Verlag, London, UK, Nov. 1998, pp. 191-206.

Geoff Coulson is a lecturer at the University of Lancaster, UK. He is involved in a number of middleware-related research projects, including the OpenORB project investigating next-generation middleware based on reflective techniques. His main research interests are distributed systems architectures, multimedia communications, and operating system support for continuous media. He received his PhD in systems support for multimedia applications from the Computing Department, University of Lancaster.

Contact Coulson at Distributed Multimedia Research Group, Computing Dept., Lancaster University, Lancaster LA1 4LA, UK, e-mail geoff@comp.lancs.ac.uk.