SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **3**:1–2 *Prepared using* speauth.cls *[Version: 2000/03/16 v2.12]*

SP&E

Thread Transparency in Information Flow Middleware

Rainer Koster[1], Andrew P. Black[2], Jie Huang[2], Jonathan Walpole[2,*], and Calton Pu[3]

[1] *Fachbereich Informatik, Universität Kaiserslautern, Postfach 3049, 67653 Kaiserslautern, Germany*
[2] *Department of Computer Science and Engineering, OGI School of Science and Engineering, Oregon Health and Science University, 20000 NW Walker Road, Beaverton, OR 97006-8921, USA*
[3] *College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280, USA*

KEY WORDS: Middleware, Threads, Multimedia, Streaming

SUMMARY

Applications that process continuous information flows are challenging to write because the application programmer must deal with flow-specific concurrency and timing requirements, necessitating the explicit management of threads, synchronization, scheduling and timing. We believe that middleware can ease this burden, but many middleware platforms do not match the structure of these applications, because focus on control-flow centric interaction models such as remote method invocation. Indeed, they abstract away from the very things that the information-flow centric programmer must control.

This paper describes Infopipes – a new high-level abstraction for information flow applications – and a middleware framework that supports them. Infopipes handle the complexities associated with control flow and multi-threading, relieving the programmer of these tasks. Starting from a high-level description of an information flow pipeline, the framework determines which parts of a pipeline require separate threads or coroutines, and handles synchronization transparently to the application programmer. The framework also gives the programmer the freedom to write or reuse components in a passive style, even though the configuration will actually require the use of a thread or coroutine. Conversely, it is possible to write a component using a thread and know that the thread will be eliminated if it is not needed in a pipeline. This allows the most appropriate programming model to be chosen for a given task, and existing code to be reused irrespective of its activity model.

Contract/grant sponsor: This work is partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs, by NFS awards CDA-9703218 and EIA-0130344, and by Intel.
*Correspondence to: Jonathan Walpole; E-mail: walpole@cse.ogi.edu

Copyright © 2003 John Wiley & Sons, Ltd.

INTRODUCTION

The benefit of middleware platforms is that they handle application-independent problems transparently to the programmer and hide underlying complexity. CORBA and RPC, for instance, provide location transparency by hiding message passing and marshaling. Hiding of complexity relieves programmers from tedious and error-prone tasks and allows them to focus on the important aspects of their applications.

The way that a middleware platform can hide complexity without hiding power is to provide higher-level abstractions that are appropriate for the supported class of applications. In order to choose a suitable abstraction, it is necessary to make some assumptions about the functionality that typical applications require. For example, a common abstraction provided by current middleware is the client-server architecture and request-response interaction, where control flows to the server and back to the client.

However, this model is inappropriate for an emerging class of information-flow applications that pass continuous streams of data among producers and consumers. Building these applications on existing middleware requires programmers to specify control flow, which is not key aspects of the application. Moreover, existing middleware has inadequate abstractions for specifying data flow, including quality of service and timing, which *are* key aspects of the application.

We propose a new middleware platform for information-flow applications that is based on a producer-consumer architectural model, the *Infopipe*. Infopipes simplify the task of building distributed streaming applications by providing basic components such as pipes, filters, buffers, and pumps [1, 2]. Each component specifies the properties of the flows that it can support, including data formats and QoS parameters. When stages of a pipeline are connected, flow properties for the composite can be derived, facilitating the composition of larger building blocks and the construction of incremental pipelines.

The need for concurrently active pipeline stages introduces significant complexity in the area of thread management. However, this complexity can be hidden by the middleware. For example, the Infopipe platform frees the programmer from the need to deal with thread creation, destruction, and synchronization. Moreover, the control flow is managed by the middleware and is decoupled from the way pipeline components are implemented, be they programmed like threads or like functions. We call this approach *thread transparency*. It simplifies programs and allows reuse of Infopipe components. In the same way that RPC systems automatically handle communication and generate code for parameter marshaling, our middleware automatically manages threads and generates glue code that allows Infopipe components to be reused in different activity contexts.

The remainder of this paper describes the Infopipe middleware platform and explains in detail how it simplifies programming tasks in the general area of activity management. First the basic Infopipe concepts are introduced and an overview of the Infopipe middleware platform is presented. Then the concept of thread transparency is introduced and the relationship among various different component implementation styles is discussed. We show how Infopipes can

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

support component reuse, even when the original style in which a component was written does not match its intended new use. We also illustrate how timing control and synchronization are supported using Infopipes. Later sections of the paper present details of the middleware platform's implementation, including its underlying user-level threads package. Finally, several example applications that have been built using Infopipes are presented before the paper closes with a discussion of related work and areas for future research.

INFOPIPE MIDDLEWARE

The Infopipe abstraction has emerged from our experience building continuous media applications [3–6]. We have built a middleware framework in C++ based on these concepts and have reimplemented video applications using it [7].

Overview

Infopipes let us build information flow pipelines from pre-defined components in a way that is analogous to the way that a plumber builds a water flow system from off-the-shelf parts.

The most common components have one input port (*inport*) and one output port (*outport*). Such pipes can *transport* information, *filter* certain information items, or *transform* the information. *Buffers* provide temporary storage and reduce rate fluctuations. *Pumps* are used to keep the information flowing. Corresponding to these roles each port has an appropriate *polarity*: A *positive outport* pushes items downstream, while items must be pulled from a *negative outport*. Similarly, a *positive inport* pulls items from upstream, while items must be pushed into a *negative inport*. Hence, pumps have two positive ports, buffers have two negative ports, and filters and transformers have two ports of opposite polarity [1,8]. *Sources* and *sinks* have only one port, which can be either positive or negative.

More complex components have more ports. Examples are *tees* for splitting and merging information flows. Splitting includes partitioning an information item into parts that are sent different ways, duplicating items to each output (multicast), and selecting an output for each item (routing). Merge tees can combine items from different sources into one item or pass on information to the output in the order in which it arrives at any input.

In connecting components into a pipeline it is important to check the compatibility of supported flows and to evaluate the characteristics of the composite Infopipe. Each basic or composite Infopipe has a *Typespec* that describes the flows that it supports. Typespecs provide information about supported formats of data items, interaction properties such as port polarities, and ranges of QoS parameters that can be handled.

Transport protocols can be integrated into the Infopipe framework by encapsulating them as *Netpipes*. These Netpipes support plain data flows and may manage low-level properties such as bandwidth and latency. Marshaling filters on either side translate the raw data flow to and from a higher-level information flow.

In building an Infopipe, an application developer needs to combine appropriate filters, buffers, pumps, network pipes, feedback sensors and actuators as well as control components. To facilitate this task, our framework provides a set of basic components to control the timing,

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

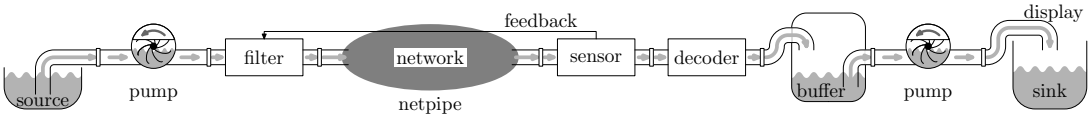


Figure 1. Infopipe example

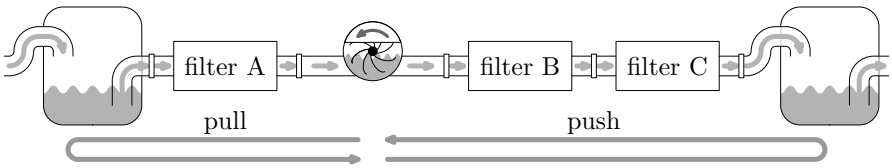


Figure 2. Polarity

which can be supplemented by a feedback toolkit for adaptation control [9]. Components for processing specific types of flow need to be developed by application programmers, but can easily be reused in various applications. For instance, developers of video on demand, video conferencing, and surveillance tools can use any available video codec.

Figure 1 shows a simple video pipeline from a source producing compressed data to a display. At the producer side, frames are pumped through a filter into a Netpipe that encapsulates a best-effort transport protocol. The filter drops frames when the network is congested. The dropping is controlled by a feedback mechanism triggered by a sensor on the consumer side. This lets us control which frames are dropped rather than suffering arbitrary dropping by the network. After decoding the frames, they are buffered to reduce jitter. A second pump releases the frames to the display at the appropriate rate.

Interaction

With respect to polarity, there are three basic classes of components:

- *Positive components* have only positive ports and cause information to flow in the pipeline. Pumps and active sources and sinks belong to this class.
- *Negative components* have only negative ports. Buffers and passive sources and sinks belong to this class.
- *Neutral components* have positive and negative ports. They do not initiate any activity but may pass it on to other components. Filters and transformers with one positive and one negative port belong to this class.

The processing of the information items is driven by a thread that originates from a positive component as shown in Figure 2. Negative and neutral objects can be implemented as objects with methods that are called through a negative port and may call out through a positive

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

port, making inter-object communication particularly efficient. Because pumps originate the threads, they regulate the timing of the data flow and can themselves be controlled by timers or feedback mechanisms. Each thread is responsible for calling through all the neutral pipeline stages as far as the next negative component up- or downstream as indicated in Figure 2. Hence, Pumps encapsulate the interaction with the underlying scheduler.

Besides exchanging data items, Infopipe components can exchange control messages. These messages are used for interaction between adjacent components as well as for global broadcast events. As an example of local interaction, consider a video resizing component that needs to be informed by the video display whenever the user changes the window size. Control interaction between remote components of a pipeline includes communication between feedback sensors, controllers, and actuators. Other events such as user commands to start or stop playing need to be broadcast to many components.

The current approach to handling control events is based on the assumption that handling these events does not require much time. Hence, there is no explicit control of timing or buffering of these events, and their handlers are executed with higher priority than data processing tasks, which are potentially long-running.

Infopipe Typespecs

The ability to construct composite pipes from simpler components is an important feature of the Infopipe platform. Automatic inference of flow properties, glue code for joining different types of components, and automatic allocation of threads help the application programmer and simplify the process of setting up an Infopipe.

A Typespec describes the properties of an information flow. Typespecs are extensible and new properties can be added as needed. Undefined properties may be interpreted as meaning either *don't know* or *don't care* as discussed below. The following list describes some parts of a Typespec.

- The *item type* describes the format of the information items and the flow.
- The *polarity* of ports in the information flow determines whether items are pushed or pulled. Polarity is represented in the Typespec by labeling each port as positive or negative. A positive outport will make calls to the `push` method of the downstream components, while a negative outport will understand and respond to a `pull`. Correspondingly, a positive inport will make calls to `pull`, while a negative inport represents the willingness to receive a `push`. With this representation, ports with opposite polarity may be connected, but an attempt to connect two ports with the same polarity is an error.
  Some components do not have a fixed polarity, but are polymorphic. For example, filters can operate in push or pull mode, as can chains of filters. When one port is connected to a port with a fixed polarity, the other port of the filter or filter chain acquires the opposite "induced" polarity [1, 10].
- A third property specifies the *blocking behavior* if an operation cannot be performed immediately. For instance, if a buffer is full, the push operation can either be blocked or

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

can drop the pushed item. Likewise, if a buffer is empty, a pull operation can either be blocked or return a nil item.
- While push and pull are the only data transmission functions, *control events between connected components* may be needed to exchange meta-data of the flow. The capability of components to send or react to these control events is included in the Typespec to ensure that the resulting pipeline is operational.
- Depending on the application, there may be a variety of *QoS parameters*. Processing a flow with specific values for these parameters requires elaborate resource management and binding protocols. If guarantees are not available, QoS parameters may nonetheless provide valuable hints to the rest of the pipeline by specifying possible ranges, for instance. For a distributed video player, these parameters may include frame rate and size, latency, and jitter. Guaranteeing a fixed frame rate, for example, requires the reservation of sufficient network and CPU bandwidth. Independently of guarantees, knowing frame sizes and maximum frame rate can help to allocate adequate buffering, for instance.
- For distributed pipelines, the *location* indicates that a flow must be produced or consumed at a particular node.

Properties can originate from sources, sinks, and intermediate pipes. Sources typically supply one or more possible data formats along with information on the achievable QoS. Likewise, sinks support certain data formats and ranges of QoS parameters reflecting user preferences. Hence, source properties indicate what can be produced, sink properties indicate what the user is willing to consume.

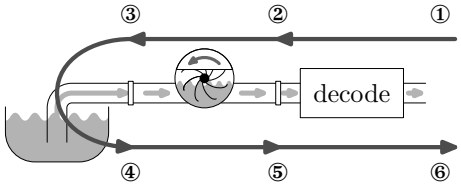
If for any stage in a pipeline a Typespec for an input or output port is given, Typespecs for other ports can be derived from that information. The derived Typespecs reflects restrictions imposed by that stage. These restrictions might originate because the stage supports only pull-interaction, fewer data types, or a smaller range for a QoS parameter. Moreover, stages can add or update properties. For example, a buffer can lower the jitter value and increase the latency value, and a decoder can increase the computation time estimate according to the resolution of the stream. The last example illustrates that components need to derive properties from others to provide specifications that are not overly general.

Because of this incremental nature of Typespecs, we do not associate a fixed Typespec with each component, but let each pipeline component transform a Typespec on each port to Typespecs on its other ports. That is, the component analyzes the information about the flow at one port and derives information about flows at other ports. These Typespec transformations are the basis for dynamic type-checking and evaluation of possible compositions. A Typespec can be applied as a query to any unconnected port of an Infopipe. Every component updates the Typespec and passes it on through all of its connected ends. If there is no other connected end the Typespec travels back to its origin becoming the reply to the query. If any component detects a type incompatibility it sets an error flag in the reply and the process is aborted.

Figure 3 gives an example for this process querying the Typespec of a flow at the output of an MPEG decoder ①. This decoder requires an MPEG stream at its input, it sends a Typespec with this item format ② to the pump. Here, the property of operating in pull mode is added ③. The file source adds several properties: The flow is an MPEG-video with a specific resolution

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls



① –
② mpeg
③ mpeg, pull
④ mpeg, pull, res. 320 x 240, 25 frames/second
⑤ mpeg, push, res. 320 x 240, 25 frames/second
⑥ YUV, push, res. 320 x 240, 25 frames/second

Figure 3. Typespec query

Figure 4. Distributed Infopipe

and frame rate. Since the source does not have other ports, the Typespec is propagated back ④. At the pump, the interaction is switched to push mode ⑤. The output of the decoder are video frames in YUV format ⑥.

Distribution

Any single protocol built into a middleware platform is inadequate for remote transmission of information flows with a variety of QoS requirements. However, different transport protocols, can be easily integrated into the Infopipe framework as *Netpipes*. These Netpipes support plain data flows and may manage low-level properties such as bandwidth and latency. Marshaling filters on either side translate the raw data flow to and from a higher-level information flow. These components also encapsulate the QoS mapping, translating between Netpipe properties and flow-specific properties.

In addition to Netpipes, the Infopipe platform provides protocols and factories for the creation of remote Infopipe components, which allow the construction of distributed pipelines. For checking flow compatibility in these cases, the middleware has an internal protocol for

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

sending Typespecs across the network. The location itself can be integrated into the type checking by adding a location property that is changed only by Netpipes. Finally, control events are delivered to remote components through the platform.

TRANSPARENT THREAD MANAGEMENT

Different timing requirements and computation times at different stages of a pipeline require multiple asynchronous threads. Unfortunately, handling multithreading and synchronization is difficult for many programmers and frequently leads to errors [11, 12]. However, because the interaction between components in an Infopipe framework is restricted to well known interfaces, it is possible to hide the complexity of low-level concurrency control in the middleware platform. This is similar to the way in which RPC or CORBA hide the complexity of low-level remote communication from the programmer.

While some aspects of the application such as timing behavior need to be exposed to the programmer, other aspects such as interaction with schedulers, inter-thread synchronization, and the adaptation of implementation styles can largely be hidden in the middleware platform. We describe how we achieve this support in the next three subsections.

Timing Control and Scheduling

Pumps encapsulate the timing control of the data stream. Each pump has a thread that operates the pipeline as far as the next negative component up- and downstream. Interaction with the underlying scheduler is also implemented in pumps. At setup, pumps can make CPU reservations, if supported, according to estimated or worst case execution time of the pipeline stages they control. Moreover, they can select and adjust thread scheduling parameters as the pipeline runs.

From our experience building multimedia pipelines we can identify at least two classes of pumps. *Clock-driven pumps* typically operate at a constant rate and are often used with passive sinks and sources. Both pumps in Figure 1 belong to this category. Audio output devices that have their own timing control can be implemented as clock-driven active sinks. *Environment-sensitive pumps* adjust their speed according to the state of other pipeline components. The simplest version does not limit its rate at all and relies on other components to block the thread when a buffer is full or empty. More elaborate approaches adjust CPU allocations among pipeline stages according to feedback from buffer fill levels [13]. Another kind of environment-sensitive pump is used on the producer node of a distributed pipeline [3, 4]. Its speed is adjusted by a feedback mechanism to compensate for clock drift and variation in network latency between producer and consumer.

The choice of the right pump depends on application requirements as well as the capabilities of the scheduler. While it is not yet clear to what extent pump selection and placement can be automated, pumps do automate thread creation and scheduling. The programmer does not need to deal with these low-level details but can benefit from appropriate timing and scheduling policies by choosing pumps and by setting appropriate parameters.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

If existing pumps do not provide the required functionality, it can be cleanly added by implementing new pumps. While a pump developer needs to deal with threads and scheduling, the pump encapsulates threading mechanisms in a way similar to that in which a decoder encapsulates compression mechanisms. In both cases the complexity is hidden from application programmers who use the new components.

Synchronization

Infopipe components need to process information (possibly from different ports) and control events. While information items and control events may arrive in any order, the middleware ensures synchronized access to shared data in its high-level communication mechanisms. The component developer does not need to deal with thread synchronization explicitly, but just provides data processing and event handling functions. Hence, thread synchronization is based on passing on data items and control events rather than on more error-prone primitive mechanisms such as locks and semaphores.

The pipeline components are implemented as monitors, also known as synchronized objects [14]: each component may contain at most one active thread at any time. However, we allow threads to be preempted because because otherwise long-running functions such as video decoders can introduce unacceptable delay. A data processing function of one component is never called before its previous invocation completes or while a control event handler of the same component is running. Control events that arrive while data processing is in progress are queued and delivered as soon as the data processing is done. Note, however, that control events can be delivered while threads are blocked in a `push` or `pull`. Hence, the programmer needs to make sure that the component is in a consistent state with respect to control handlers when these operations are called.

Implementation Styles in Pipeline Components

For implementing neutral pipeline components there are several styles with respect to activity. The main distinction is between active objects that have an associated thread and passive objects that are called by external threads [14]. On our platform, only positive components introduce threads and other components do not increase concurrency. Nonetheless, there is support for different implementation styles for components, decoupling concurrency control from the programming model. In particular, components may be programmed as if they had a thread of their own or as passive functions. To make a clear distinction between the polarity of a component and its implementation style, we call implementations as active objects *thread-style components* and implementations as passive objects *function-style components*. We will now examine these implementation styles by focussing on neutral components with one input and one output port, which are most common.

The middleware platform assumes components such as filters to be neutral, having a positive and a negative port. The external interface is an `item pull()` operation that can be called by downstream components and `void push(item)` operation that can be called by upstream components. Which of these is used in a particular pipeline component depends on the position of the component relative to pumps and buffers. Components between buffer and pump operate

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

in pull mode, whereas components between pump and buffer operate in push mode, as shown in Figure 2.

Independently of this external interface, the middleware provides a variety of internal interfaces to the component developer. He may implement *either* the `push` *or* the `pull` function – whichever is more suitable for the required functionality – and the platform will derive the other one. Alternatively, for components that produce exactly one output for each input, a conversion function `item convert(item x)` can be programmed. Finally, the developer can implement the component as the main function of a thread, using commands for receiving information items from upstream and sending information items downstream. Independently of this diversity, the middleware generates the glue code for providing the uniform external `push` and `pull` interface. The required mechanisms and examples are discussed in the following section in more detail.

Supporting multiple implementation styles has two advantages: Firstly, some styles are more suitable than others for implementing the functionality of a component; the developer can choose the one that fits best in a given situation. Secondly, it is important to be able to reuse existing code and to integrate it into the framework. Existing functionality can be wrapped into Infopipe components efficiently, if the implementation style can be retained. To make this case common, the platform supports several styles rather than imposing a particular one.

SUPPORT FOR MULTIPLE IMPLEMENTATION STYLES

In this section, we first demonstrate the supported implementation styles using an incrementor as a simple example. At the same time, we present the mechanisms and glue code used by the middleware to map the respective styles to the uniform external behavior as a neutral component. Then, a defragmenter is used as a slightly more complicated example to illustrate the importance of supporting multiple styles.

The core functionality of the first example is simply incrementing a value associated with each information item `x` passing through the component: `x.value++`. Its implementation as a conversion function is straightforward:

```
item convert(item x) {
  x.value++;
  return x;
}
```

Note that `convert` returns an output item for each input item, restricting this implementation style to components that comprise such a one-to-one mapping. In this case, the following `push` and `pull` methods are derived by the middleware:

```
void push(item x) {next->push(convert(x));}
item pull() {return convert(prev->pull());}
```

A second, more general approach is using a consumer or producer style by implementing the component functionality directly as `push` or `pull` function. By convention, the programmer does not directly call `push` or `pull` methods on other components. He instead uses `put` and

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

```
void push(item x) {
  x.value++;
  this->put(x);
  }
```

a) consumer style

```
item pull() {
  x=this->get();
  x.value++;
  return x;
  }
```

b) producer style

```
void main() {
  while (running) {
    x=this->get();
    x.value++;
    this->put(x);
  }
}
```

c) thread style

Figure 5. Incrementor in different styles

Figure 6. Thread-style incrementor

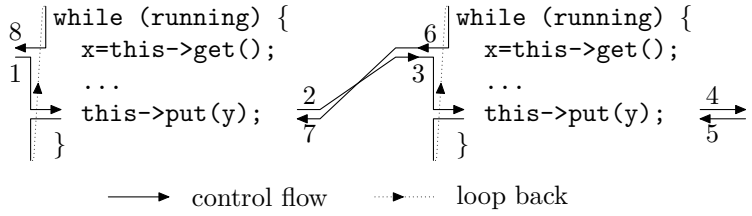


Figure 7. Synchronous threads

`get` methods, which are inherited from a base class provided by the middleware platform. In this case, the inherited implementation of `put` and `get` is as follows:

```
void put(item x) {next->push(x);}
item get() {return prev->pull();}
```

Figures 5a and b show `push` and `pull` methods for the incrementor example. Note that in these styles `put` or `get` may be invoked multiple times or not at all in one invocation of `push` or `pull`, making this approach more general than `convert`, which allows only components that produce exactly one output item for each input item.

The next option, a thread-style implementation is shown in Figure 5c. This way of programming provides the most flexibility, because statements for sending and receiving data items (`put` and `get`) can be mixed freely as is most convenient for a given component. The way

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

```
while (running) {
  x=this->pull();
  this->put(x);
  }
```

a) Push-mode wrapper for `pull`

```
while (running) {
  x=this->get();
  this->push(x);
  }
```

b) Pull-mode wrapper for `push`

Figure 8. Coroutine wrappers

to give these thread-style components the facade of a neutral component is to use *coroutines*, that is, threads interacting synchronously in such a way that they provide suspendable control flow but are not a unit of scheduling [15]. The communication mechanism between the threads does not buffer data; instead the activity travels with the data. All but one of the coroutines in a given set are blocked at any time. Figure 7 gives an example of two coroutines interacting in push mode. An item is pushed into the first component unblocking it from a `this->get` call (1). The component then processes the data and calls `this->put`, which blocks the first component and passes the item to the next component (2), which unblocks from its `get` (3). It again does some processing and a `put` (4). When `put` returns (5), the control flow loops back to the `get` call. This blocks the second component (6) and unblocks the first component from its `put` (7). Finally the control flow reaches a `get` call again and returns to the upstream component (8).

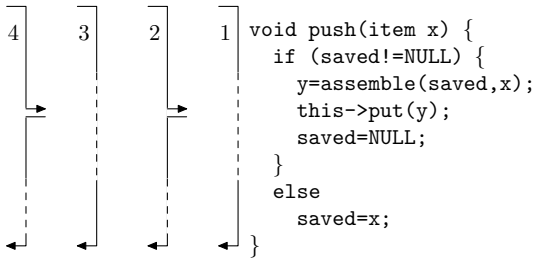
The coroutine behavior described above is implemented by inheriting different `put` and `get` methods from appropriate superclasses. The behavior of these methods depends on the implementation style of the adjacent component. Consider a pipeline running in push mode. If the target of a `put` is a function-style component providing a `push`-method, then `put` can simply call `next->push`. However, if the target is a thread-style component, then `put` performs a switch to the coroutine of the target component, which is waiting blocked in its `get` method. Pull mode is handled analogously.

The producer- and consumer-style implementations shown in Figures 5a and b have a major drawback. Components have to provide both a `push` and a `pull` operation that implement the same functionality. Alternatively, components could provide only one of these operations, but then could be used in one mode only, making building the pipeline more difficult. These restrictions can be avoided with middleware support that allows `push` functions to be used in pull mode and vice-versa. Our Infopipe middleware generates glue code for this purpose and converts the functions into coroutines as illustrated in Figure 8.

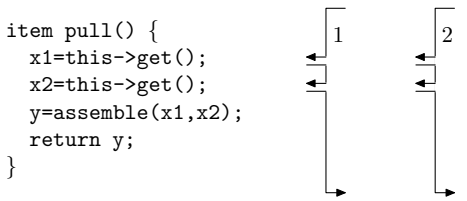
For components such as the incrementor that maintain a one-to-one relation between input and output items, the implementations in different styles are so similar that there seems to be no point in distinguishing them. The distinction is more obvious with component like a defragmenter that combines two data items into one. In this second example component, the actual merging is performed by the function `y=assemble(x1,x2)`.

Copyright © 2003 John Wiley & Sons, Ltd.

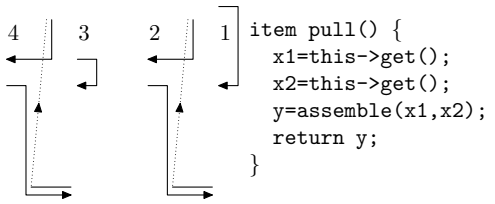
Prepared using speauth.cls



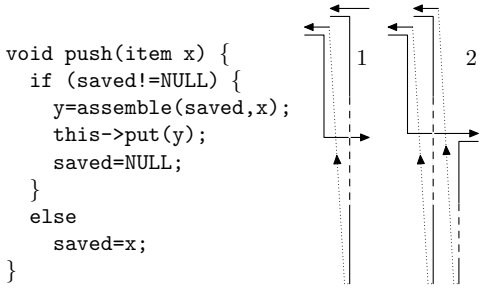
a) `push` in push-mode



b) `pull` in pull-mode



c) `pull` in push-mode



d) `push` in pull-mode

Figure 9. Function-style defragmenter

The `push` and `pull` methods of an implementation in function style, are shown in Figure 9. Each numbered group of arrows shows the control flow for one call to the method that it annotates. In Figure 9b, each invocation of `pull` travels all the way through the code triggering two `get` calls and, hence, two `pull` calls to the upstream pipeline component. For `push` in Figure 9a every other call (2 and 4) causes a `put` and, hence, a downstream `push`. If no output item can be produced the call returns directly. Figure 9c and d indicate the control flow for the pull implementation used in push mode and vice versa, using the coroutine wrappers shown in Figure 8.

This example shows that the `pull` operation for the defragmenter can be implemented more easily than `push`. The latter requires the programmer to explicitly maintain state between two invocations, which is done in this example using the variable `saved`. Conversely, for a fragmenter, `push` would be the simpler operation. Hence, choosing the appropriate style can reduce the complexity of the implementation. Also note that the `convert`-style does not work in this case, because two output items need to be produced for each input item.

Figure 10 shows a thread-style implementation of the defragmenter example. Here again, each numbered group of arrows denotes the control flow for one `push` call (in Figure 10a) or one `pull` call (in Figure 10b) to the component. When operating in push mode, upstream `get` calls block the defragmenter and each invocation executes from `get` to `get`. As an exception,

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

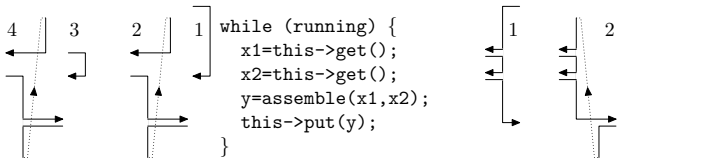


Figure 10. Thread-style defragmenter

Table I. Implementation styles

implementation style	push mode	pull mode
conversion function	direct call	direct call
push function	direct call	coroutine
pull function	coroutine	direct call
thread	coroutine	coroutine

the first `push` call invokes the main function of the component, enters its loop, and satisfies the first call to `get`. Again, the pull mode works analogously.

Supporting such thread-style components that are written as active objects is important for several reasons. One reason is the reuse of code from older pipeline implementations that used an active object model or implemented each stage as a process. Another reason is the flexibility that thread-style implementations provide: The programmer can freely mix statements for sending and receiving data items as is most convenient for a given component. Finally, more programmers are familiar with the thread-style model than with the function-style model.

Note that the information flow is the same in Figures 9 and 10. In each mode, the number of incoming and outgoing arrows is the same for each invocation and for all three implementations. Every other push triggers a downstream push in Parts a and c of the figure and every pull triggers two upstream pulls in Parts b and d.

While we have used an incrementor and a defragmenter as examples, the different ways of implementing components that we have described also apply to fragmenters, decoders, filters, and transformers. By supporting all these styles, we provide flexibility in developing and reusing components, but for efficiency it is nonetheless important to avoid context switches and use direct calls whenever possible. Hence, the framework detects which components can share a thread and for which ones additional coroutines are needed.

Figure 11 shows several pipelines between a passive source and a passive sink with the associated threads and coroutines depicted as dashed boxes. The same coroutine boundaries would apply to pipeline sections between two negative components. Altogether, there are

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

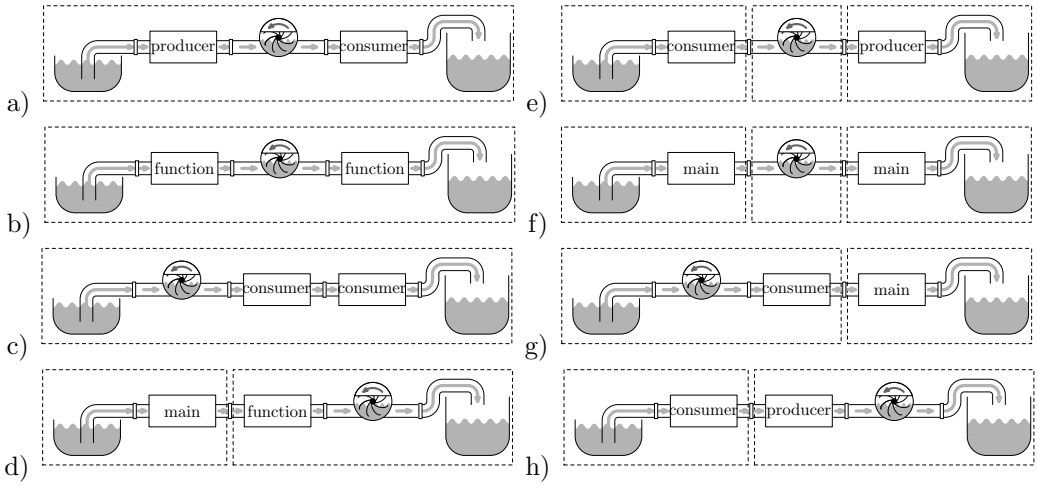


Figure 11. Pipelines and coroutines

four styles of neutral components, as shown in Table I. Thread-style implementations provide a thread-like main function. Function-style components are consumers implementing `push`, producers implementing `pull`, or are based on a conversion function. In push mode, consumers and conversion functions are called directly, and in pull mode producers and conversion functions are called directly. Otherwise, a coroutine is required. In each case, all threads operate synchronously as one coroutine set and the pump controls timing and scheduling in all components.

The behavior of components with more than two ports is more complex. Not all styles of implementation can be supported for all components. Sometimes the functionality of the component makes a particular style inappropriate. To see this, consider a switch with one inport and two outports. Incoming packets are routed to one of the outports depending on the data in the packet. Now consider this switch in pull mode, that is, packets are pulled from either outport. A pull request arrives at outport 1 triggering an upstream pull-request at the inport. Suppose that the incoming packet is routed to outport 2. Now there is a pending call without a reply packet and a packet nobody asked for. Suspending the call would require buffering potentially many requests on outport 1 and buffering packets at outport 2 until all packets at outport 2 are pulled. This approach leads to unpredictable implicit buffering behavior and complex dependencies.

To avoid these problems the Infopipe framework generally allows only one negative port in a non-buffering component. However, there are exceptions. For instance a different type of switch may route the packet not according to the value of the packet, but based on the activity. A pull on either outport triggers an upstream pull and returns the item to the caller. In this

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

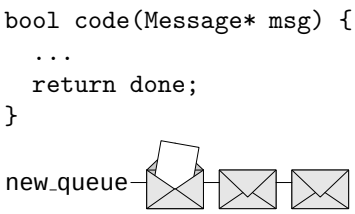


Figure 12. Thread structure

case, the outports must both be negative and the inport must be positive. This component could not work in push mode.

IMPLEMENTATION

To support the Infopipe middleware, an appropriate underlying multithreading infrastructure is needed to manage concurrency and synchronization. A user-level thread package is more appropriate than kernel-level threads, because the lower context switching overhead allows for finer-grained components, and scheduling policies can be customized. The Cool Jazz package developed in the Distributed Systems Group at the University of Kaiserslautern [6, 16, 17] has been used as a basis for the Infopipe middleware. It is a message-based user-level thread package, which provides flexible support for timing control as well as for plugging in application-specific schedulers. The package is implemented in C++.

The underlying thread package

Cool Jazz uses a message-based threading model that facilitates inter-thread communication and synchronization. While Infopipes hide the difficulties of concurrent programming by providing higher-level interfaces to the application developer, advanced low-level mechanisms still support the development of the middleware platform itself. In Cool Jazz, each thread consists of a code function and a *new-queue* for incoming messages as shown in Figure 12. Unlike conventional threads, the code function is not called at thread creation time but each time a message is received. After processing a message, the code function returns, but the thread is terminated only when the return value is -1. In this way, code functions resemble event handlers, but may be suspended waiting for other messages, or may be preempted. Inter-thread communication is performed by sending messages to other threads, either synchronously if there remains nothing for a thread to do until a reply is received, or asynchronously whenever a reply is not needed immediately, or no reply is required at all.

Information flow applications have timing requirements. The QoS provided to the user depends on tasks to be performed in time. Hence, managing concurrent activities must take

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

their relative importance and urgency into account and schedule them appropriately. Because Infopipes encapsulate policies for timing and scheduling in pumps, it is important for the thread package to provide flexible mechanisms that support pump development without restricting possible policies.

Cool Jazz provides such flexible mechanisms for supporting timing constraints. In addition to thread priorities, constraints may be attached to messages. The priority of a thread is derived from the constraint on the message it is currently processing or, if the thread is waiting for the CPU, on the constraint of the first message in its new-queue. If no constraint is specified for the respective message, the priority statically assigned to the thread is used. To create subtasks that will be processed by other threads in parallel, new messages with the same constraint can be created, while passing on a message also transfers the priority derived from its constraint. This approach supports scheduling based on static priorities by using thread priorities only, on dynamic priorities by using message constraints only, or on a mixture of both. To bound priority inversion, priority inheritance is supported: If a thread is processing a message at a priority lower than that mandated by the constraint of the first message in its new-queue, it inherits this priority.

To support a wide range of applications and environments, the threading system should also be adaptable to specific needs. Hence, not only timing parameters but also the scheduler itself should be customizable. Pipeline developers can choose an appropriate scheduler for their application and use pumps that encapsulate the interaction with the scheduler and utilize its capabilities. Other pipeline components are independent of the underlying scheduler.

While loadable and hierarchical operating system schedulers [18–20] provide some flexibility, scheduling among threads of one application can be done a lot more efficiently than among threads in the kernel. Threads may rely on each other to work cooperatively. Moreover, application-specific semantic information can easily be exploited in scheduling decisions. Consider, for instance, a feedback-driven real-rate scheduler [13], which adaptively schedules threads based on their progress. While providing semantic information about progress to the kernel scheduler via fixed system interfaces in general is not simple, this algorithm can be efficiently used for scheduling threads of one application. A user-level scheduler can have a clear notion of progress and have detailed information about inter-thread dependencies.

When designing a user-level thread package there still is no optimal scheduler for all scenarios. Even providing a set of predefined policies cannot avoid all design dilemmas. Cool Jazz therefore applies the *open implementation* design methodology [21]. All scheduling code has been separated from the rest of the thread package allowing the user to plug-in a specific scheduler via a well-defined interface. This scheduler is called whenever some event might induce a scheduling decision. In this way, policies are encapsulated in the exchangeable scheduler, while mechanisms for concurrency are built into the Cool Jazz core. This flexible approach is beneficial whether or not there are guarantees from the system. The user-level scheduler has detailed knowledge about timing constraints, slack times, and inter-thread dependencies, and can easily monitor the progress of all subtasks. Hence, it can efficiently use reserved CPU capacity as well as adapt to resource fluctuations.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

```
class pipe {
public:
  virtual void push(item x)=0;
  virtual item pull()=0;
  virtual void receive(event e)=0;
};
```

```
class fct_convert: public pipe {
public:
  virtual void push(item x);
  virtual item pull();
protected:
  virtual item convert(item)=0;
};

class thread_style: public pipe {
public:
  virtual void push(item x);
  virtual item pull();
protected:
  void put(item x);
  item get();
  virtual void run()=0;
};
```

```
class fct_push: public pipe {
public:
  virtual void push(item)=0;
  virtual item pull();
protected:
  void put(item x);
};

class fct_pull: public pipe {
public:
  virtual void push(item x);
  virtual item pull()=0;
protected:
  item get();
};
```

Figure 13. Simplified base class declarations

Infopipes on Cool Jazz

The Infopipe platform creates a thread for each pump. If there is no need for coroutines in the section of a pipeline that is controlled by a particular pump, the thread calls the `pull` methods of all components upstream of the pump, then calls `push` with the returned item on the components downstream of the pump, and finally returns to the pump, which schedules the next pull. This is the situation in configurations a), b), and c) in Figure 11. For configurations d), g), and h) there are two coroutines and for configurations e) and f) there are three coroutines associated with the pump. If such coroutines are needed, each of them is implemented by an additional thread of the underlying thread package. Their synchronous interaction is implemented on top of it.

Infopipe `push` and `pull` calls between coroutines and control events are mapped to asynchronous inter-thread messages. Although `push` and `pull` are synchronous to the Infopipe programmer, synchronous messages cannot be used, because the thread would not then be responsive to control events. Instead, the thread blocks waiting for either a control event or the data reply message. A control event is dispatched to the appropriate handler and then the thread blocks again. After receiving the reply message the code function of the thread is resumed. In this way, the middleware establishes synchronous exchange of data items between

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

coroutines, while control events can be handled even if the component is blocked in a `pull` or `push`.

The component developer indicates his choice of implementation style by inheriting from the appropriate base class as shown in Figure 13 and by overriding a `run` method for a thread-style component, a `push` method for a consumer, a `pull` method for a producer, and a `convert` method for a component based on a conversion function. Additionally, a handler for control events needs to be provided. For pipeline components that change the Typespec of flows the inherited implementation of the type query must be overridden.

The time for a mere function call is two orders of magnitude shorter than a context switch between the user-level threads. Hence, the approach that we have sketched, in which threads and coroutines are introduced only when necessary, is important for the efficiency of pipelines that handle many control events or many small data items, such as a MIDI mixer. For these applications, allocating a thread for each pipeline component would introduce a significant context switching overhead, even for user-level threads. If kernel-level threads are used, this overhead would be even larger.

Figure 14a) shows the implementation of a simple clock-driven pump. Once started, it emits items at a fixed frequency, which is specified at pump instantiation and can later be changed with `set_freq` When the pump is started with `on_start`, `reftime` is initialized with the result of `stamp()`, which returns the current time. For each item pumped, `reftime` is increased by the item period. At startup and whenever the alarm timer goes off, `stroke` is invoked and items are forwarded until the time for the next item is in the future. Then, a new alarm is requested for that point in time. Until then, the pending item is saved in `stored` and the pump becomes idle. The timestamp-driven pump in Figure 14b) works similarly. Items are processed according to monotonically increasing timestamps. At initialization, `reftime` is set to the real-time equivalent of timestamp 0. At each alarm, `stroke` emits all items with a timestamp in the past before setting the next alarm.

While in these simple examples the pumps do not interact with the scheduler, pumps may also assign timing constraints to information items flowing through them. These constraints can be used for EDF scheduling, for instance. Messages between coroutines inherit the constraint from the message sent by the pump, applying the constraint to the entire coroutine set. In this way, the pump controls the scheduling in its part of the pipeline across coroutine boundaries. For EDF, the pump in Figure 14b) could simply be extended by `set_constraint(x,calc_deadline(x.timestamp));`

Pipeline composition

Pipelines can be configured by a high-level C++ interface or by a simple Infopipe composition language, as illustrated using the video player in Figure 15a) as an example. Figure 15b) shows how it can be constructed in C++. The default outports and inports of pipeline components are simply connected by the operator `>>`. The resulting pipeline is a composite Infopipe without external ports.

Alternatively, pipelines can be specified in a simple composition language. It allows configurations to be described more concisely than in C++ and provides textual representations of pipeline configurations, which can easily be transmitted for setting up remote parts of a

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

```
class pump {
public:
  void on_start();
  void on_stop();
  void on_alarm();
protected:
  void stroke();
  virtual void calc_init(item x)=0;
  virtual time calc_next(item x)=0;
};

void pump::on_alarm() {
  stroke();
}

void pump::on_stop() {
  cancel_alarm();
}
```

a) fixed-rate pump

```
void pump::on_start() {
  stored=get();
  calc_init(stored);
  stroke();
}

void pump::stroke() {
  while (1) {
    if (stored) {put(stored);stored=0;}
    item x=get();
    time nexttime=calc_next(x);
    time now=stamp();
    if (nexttime>now) {
      stored=x;
      set_alarm(nexttime-now);
      return;
      }
    put(x);
  }
}
```

b) timestamp-driven pump

```
class fpump {
public:
  fpump(int f): freq(f) {}
  void set_freq(int f) {freq=f;}
protected:
  int freq;  // pumping frequency
  virtual void calc_init(item x);
  virtual time calc_next(item x);
};

void fpump::calc_init(item x) {
  reftime=stamp();
}

time fpump::calc_next(item x) {
  reftime+=MINUTE/freq;
  return reftime;
}
```

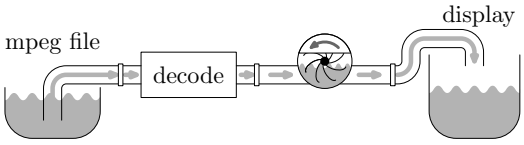
```
class tpump {
public:
  tpump() {}
protected:
  virtual void calc_init(item x);
  virtual time calc_next(item x);
};


void tpump::calc_init(item x) {
  reftime=stamp()-x.timestamp;
}

time tpump::calc_next(item x) {
  return reftime+x.timestamp;
}
```

Figure 14. Pump implementations

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

a) simple video player



b) configuration in C++

```
mpeg_video_file source("test.mpg");
mpeg_video_decoder decode;
clocked_pump pump(30);     // 30 frames/second
video_display sink;
composite pipeline=source>>decode>>pump>>sink;
```

c) configuration in the composition language

```
mpeg_video_file test.mpg>>mpeg_video_decoder>>clocked_pump 30
  >>video_display;
```

Figure 15. Simple composition example

```
pipeline.receive(event(START));
...
pipeline.receive(event(STOP));
```

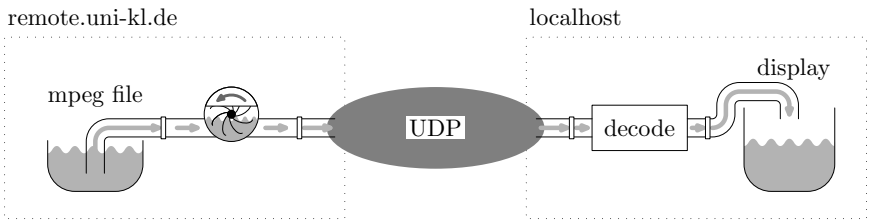
Figure 16. Sending control events

pipeline. New components are allocated by the name of their class and a space-separated list of parameters for the instantiation. The instances are then connected by the overloaded operator `>>`, which connects the default outport and default inport of its operands. Figure 15c) shows an example.

Once a pipeline is setup, it can be controlled by sending events to it as in Figure 16. The `pipeline` composite forwards the events to its components and, in this case, the pump reacts to the `START` and `STOP` commands. This hierarchical forwarding of events allows the addition of control functionality to composite components, which is an important feature for efficient pipeline composition and a common approach in graphical window systems. While regular composite components simply forward all events to their constituents, specialized composites can also drop, delay, modify, or generate events.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls




```
mpeg_video_file test.mpg>>clocked_pump>>@remote.uni-kl.de udp
    >>mpeg_video_decoder>>video_display;
```

Figure 17. Example remote configuration

As introduced above, transport protocols are encapsulated in Netpipes. These special components are specified in the composition language by `@host netpipe` or `netpipe @host`. In the former case, the partial pipeline up to the Netpipe is run on the remote machine, while in the latter case the partial pipeline from the Netpipe on is built on the remote host. Figure 17 gives an example configuration, which creates a source and a pump on a remote machine. An MPEG video is streamed from the host `remote.uni-kl.de` through a UDP Netpipe to a decoder and a sink on the local machine. Internally, Netpipes are implemented as two components, a sink on the upstream side and a source on the downstream side. In the the example in Figure 17, the sink is passive and the source is active.

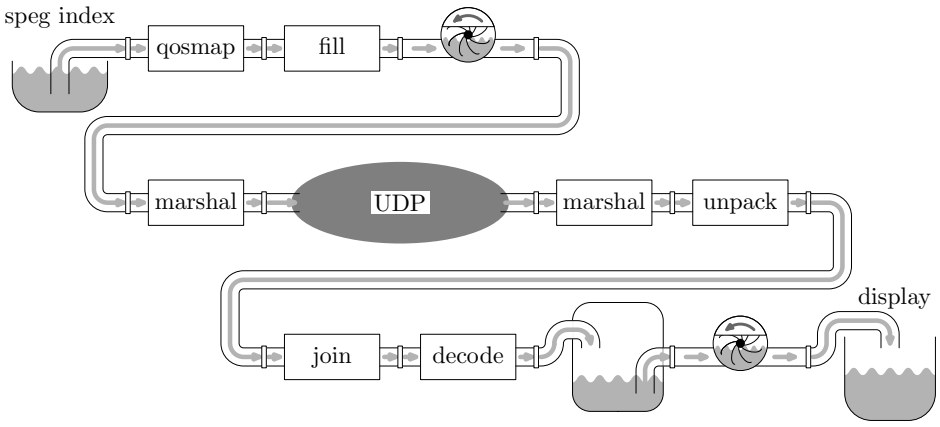
APPLICATIONS

To test the Infopipe platform and to gain experience in using it, several versions of a video player have been built on top of this middleware. In order not to reimplement existing functionality and to check the difficulty of reusing code in Infopipe components, a player developed at OGI was ported to the Infopipe platform. The communication between the stages was wrapped to use Infopipe ports. Timing control was extracted and built into a pump. The structure of this player is shown in Figure 15. An MPEG stream is read from a file, decoded, and displayed.

The components based on code from OGI, that is file source, decoder, and display sink, were originally run as threads and coroutines. They could be easily incorporated into Infopipes with a thread-style implementation by mapping the original communication mechanism to corresponding calls to `put` and `get` and by adding appropriate Typespec checking. However, passing on information items is not the only interaction between components. In this player, MPEG decoder and video display both operate on shared representations of decoded frames; the former uses them as reference frames, the latter sends them to the screen. Hence, these pipeline stages need to communicate via control messages when the frames can be deleted.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls




```
speg_index_file test.mpg>>qos_map>>fill>>clocked_pump>>marshal
   >>@oo7 udp>>unmarshal>>unpack>>join>>mpeg_video_decoder
   >>buffer 50>>clocked_pump>>video_display
```

Figure 18. Distributed video player

Untangling the dependencies and encapsulating this interaction in control events required some effort. Another modification was the extraction of timing control into pumps, which lead to a cleaner structure than including this functionality in the display stage. Since no frames are dropped in this setup, the pump simply passes on the video frames at the specified rate of 30 frames per second.

A distributed player is shown in Figure 18. It uses the SPEG scalable video codec developed at OGI [5], which supports fine-grained control over video quality and resource consumption. On the host `oo7`, packet indices of an SPEG stream are read from the file `test.mpg.idx`, are assigned priorities for potential packet dropping (`qos_map`), are supplemented by the actual packet data (`fill`), marshaled, and transmitted over UDP. On the local node, the packets are unmarshaled, merged into an SPEG stream (`unpack`), translated into MPEG (`join`), decoded, and displayed. The flow is controlled by pumps that stream the video in real time. A buffer after the decoder is used to remove jitter introduced by the network and varying decoding times.

As with the local player, it was easy to wrap the SPEG components into Infopipes. For remote communication, a generic UDP Netpipe is used, which receives and sends data items consisting of contiguous buffers as UDP packets. Since this item format is different from that used by the OGI components, marshaling Infopipes were added. They were quickly built as light-weight pipes with a function-style implementation. Since the number of SPEG packets

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

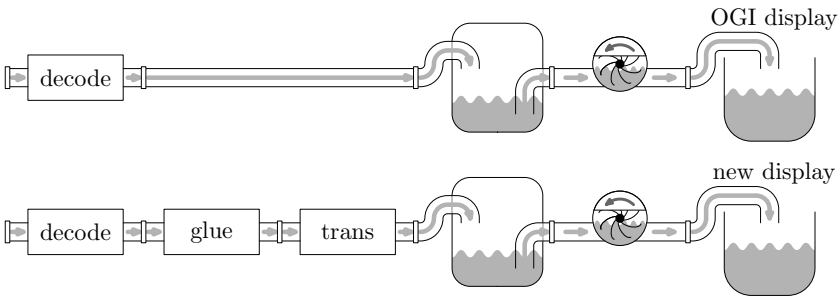


Figure 19. Alternate video display sink

per video frame is variable, the first pump cannot simply run at a fixed item rate. Neither can the second pump, because frames may be lost in the network. Hence, the pumps use time stamps on the information items to regulate the flow. To utilize the buffer, the second pump starts only when the buffer is half full. A more advanced version of this player would use synchronization feedback for controlling the fill level of this buffer, and QoS feedback for adapting resource consumption by dropping low-priority packets [3, 4].

To gain more experience in developing Infopipe components, a second video display component has been built. The OGI sink uses the *Simple DirectMedia Layer* (SDL) library [22] to display video frames in YUV format as produced by the MPEG decoder. It automatically uses hardware for processing this format if available, and translates the video format in software if not. In the latter case, timing could be improved by moving this computationally intensive translation before the buffer. Moreover, SDL does not support multiple windows, which we wished to use to display multiple views of a scene. An alternative display sink provides this functionality and can be used as shown in Figure 19. The translation stage is built as a function-style component and the actual display as a consumer-style passive sink. Because the MPEG decoder now uses only control events for managing decoded frames, the translation pipe can simply comply with this interface. Since the new components do not use the OGI item format, an additional `glue` component is needed to adapt it. The fact that we could simply insert a light-weight pipeline stage to deal with the mismatch illustrates the versatility of the platform.

Finally, a live camera source and JPEG components have been developed. The same consumer-side components can be used for accessing a live video and a stored video source as shown in Figure 20.[†] While the characteristics of the sources result in different formats (MPEG and JPEG) for transmission, the decoders provide the same high-level interface: they both produce raw video frames. This simple example illustrates that pipeline components and sections can be developed independently and can easily be composed into pipelines later.

[†]Auxiliary components such as marshalers are omitted.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

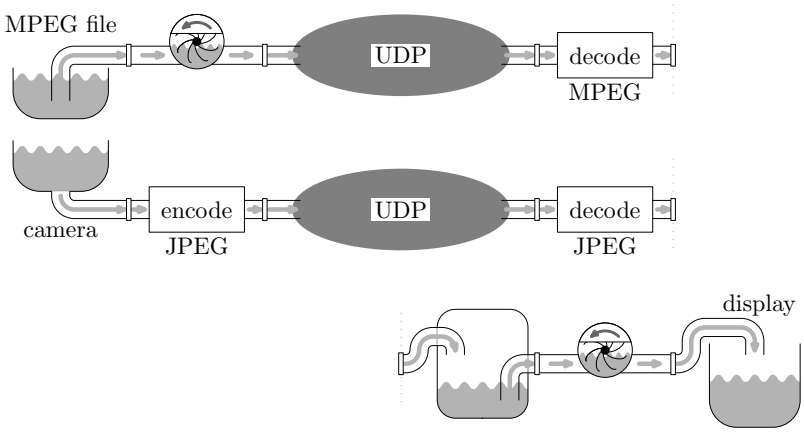


Figure 20. Two video services

RELATED WORK

Because most common middleware platforms are based on objects and RMI, their basic functionality is not very suitable for information flows. However, many extensions have been proposed to better support streams and real-time properties. More easily, event-based middleware can be used as a basis for flows. Other platforms support the concepts of openness and reflection, which are also applicable to an information-flow middleware. Useful functionality and concepts for building multimedia applications can be found in a variety of free, commercial, and research frameworks and toolkits. Related to the setup of pipelines are approaches to facilitating the composition of components and checking the validity of configurations. On a lower level, processing flows of packets has been investigated for protocol frameworks and routers.

There are several approaches for integrating streaming services with middleware platforms based on remote method invocations such as CORBA [23]. The CORBA telecoms specification [24] defines stream management interfaces, but not the data transmission. Only extensions to CORBA such as TAO's pluggable protocol framework allow the use of different transport protocols and, hence, the efficient implementation of audio and video applications [25]. Asynchronous messaging and event channels supplement synchronous RMI-based interaction and introduce the concurrency needed for information flow pipelines. Extensions of the event service providing real-time functionality [26] and stream management [27] have also been proposed. Real-time CORBA, finally, adds priority-based mechanisms to support predictable end-to-end QoS. As extensions of an RMI-based architecture these mechanisms facilitate the integration of streams into a distributed object system. There is, however, no concept of pipelines, which can be built from flow processing

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

stages from source to sink. Infopipes, in contrast, provide a high-level interface tailored to information flows and more flexibility in controlling concurrency and pipeline setup. Moreover, Infopipes are designed to support application-specific communication mechanisms, while CORBA was built to make distribution transparent.

Event-based middleware such as ECho [28, 29] provides a type-safe and efficient way of communicating data and control information in a distributed and heterogeneous environment. While such event platforms are primarily used for scientific computations, they also can be employed to transport information flows if complemented by appropriate timing control. A higher-level Infopipe layer is being built on top of a distributed event middleware [30], but the implementation discussed in this paper tries to get by without another platform layer.

Open middleware platforms and communications frameworks such as OpenORB [31–33] and Bossa Nova [34] offer a flexible infrastructure that supports QoS-aware composition and reflection. Management components use reflective mechanisms to dynamically restructure an application when changes in the environment degrade QoS [31]. Moreover, there are some similarities between Infopipes and the Open-ORB Python Prototype (OOPP) in creating composite components, and OOPPs operational bindings resemble Netpipes [32]. In contrast to these platforms, Infopipes provide more specific support for flows with respect to interfaces between pipeline stages and scheduling. Nonetheless, their reflection and QoS management functionality is applicable to information flow middleware and integrating it would be worthwhile.

The MULTE middleware project features open bindings [35, 36] and supports flexible QoS [37]. It provides applications with several ways to specify QoS using a mapping or negotiation in advance to translate among different levels of QoS specification. In our approach we typically use dynamic monitoring and adaptation of QoS at the application-level to implicitly manage resource-level QoS. Moreover, MULTE focuses on the integration of QoS and stream support with an RMI-based architecture such as CORBA rather than providing a flow-based pipeline abstraction.

Structuring data processing applications as components that run asynchronously and communicate by passing on streams of data items is a common pattern in concurrent programming [e.g. 38]. *Flow-Based Programming* applies this concept to the development of business applications [39]. While the flow-based structure is well-suited for building multimedia applications, it must be supplemented by support for timing requirements. Besides integrating this timing control via pumps and buffers, Infopipes facilitate component development and pipeline setup by providing a framework for communication and threading.

The VuSystem [40] is a multimedia platform that has several similarities to Infopipes: applications are structured as pipeline components processing information flows, there are interfaces for flow and control communication, and no particular real-time support from the operating system is needed. VuSystem, however, is single-threaded and timing and flow are controlled by the data processing components themselves. Infopipes, in contrast, support multiple threads, preemptive scheduling, and a choice of several programming styles for components and more elaborate consistency checks for pipeline setup.

QoSDREAM uses a two-layer representation to construct multimedia applications [41]. On the *model layer*, the programmer builds the application by combining abstract components and specifying their QoS properties. The system then maps this description to the *active layer*

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

consisting of the actual executable components. The setup procedure includes integrity checks and admission tests. The active-layer representation may be more fine-grained than the model specification, introducing additional components such as filters, if needed. In this way, the system supports partially automatic configuration. While the current Infopipe implementation provides less sophisticated QoS control, it provides a better modeling of flow properties by explicitly using pumps and buffers. Moreover, complex Infopipe components can contain additional control functionality for coordinating the subcomponents, which is not possible if composites only exist on the model layer.

For constructing streaming applications from components, there are also free and commercial frameworks [42–44]. GStreamer and DirectShow support setup of local pipelines without timing and QoS control. They provide services to automatically configure components for the conversion of data formats. GStreamer supports function-style push and thread-style component implementations, but does not have pumps to encapsulate timing control. RealSystem is a distributed framework that allows file source components to be used in servers as well as in local clients. The actual transmission is hardcoded into the RealServer and may be configured by adaptation rules.

Similarly to Infopipes, the *Regis* environment [45] separates the configuration of distributed programs from the implementation of the program components. The *Darwin* language is used to describe and verify the configurations. Components, which execute as threads or processes, are implemented in C++ with headers generated from Darwin declarations. While the Infopipe implementation described here also uses C++ for pipeline setup and definition of Typespec properties, a rudimentary construction language is also supported. A more comprehensive Infopipe Composition and Restructuring Microlanguage is planned as part of the Infosphere project [2].

DeLine proposes a *Flexible Packaging* method [46] to use application functionality, the *ware*, in different environments. Reusable *packagers* provide access to these environments. For instance, the same ware can be used on data from a database or a text-file parser. Wares and packagers communicate via coroutines and asynchronous channels. Compatibility can be checked and glue code can be generated based on channel signatures. A key distinction is whether the ware or the packager drives the computation. In contrast to this approach, Infopipes extract the functionality of driving the computation into particular components, the pumps. Since controlling the timing of information flows may be complex, it is beneficial to perform this task in special-purpose components. While neutral Infopipe components behave reactively, components written in different ways can be transparently integrated.

Ensemble [47] and Da CaPo [48, 49] are protocol frameworks that support the composition and reconfiguration of protocol stacks from modules. Both provide mechanisms to check the usability of configurations and use heuristics to build the stacks. Unlike these frameworks for local protocols, Infopipes use a uniform abstraction for handling information flows from source to sink, possibly across several network nodes. While in this way all protocol functionality can be built from rudimentary Netpipes and low-level Infopipe components, established protocol frameworks can also be used to build advanced Netpipes that encapsulate complex protocols.

The *x-kernel* protocol architecture [50] associates processes with messages rather than protocols. In this way, messages can be shepherded through the entire protocol stack without incurring any context switch overhead. For Infopipes, negative components isolate pipeline

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

sections that need to be scheduled independently. Within these sections, information items can travel through several components without a context switch, resembling the thread-per-packet approach.

The Scout operating system [51] generalizes from the x-kernel by combining linear flows of data into *paths*. Paths provide an abstraction to which the invariants associated with the flow can be attached. These invariants represent information that is true of the path as a whole, but which may not be apparent to any particular component acting only on local information. This idea — providing an abstraction that can be used to transmit non-local information — is applicable to many aspects of information flows, and is one of the principles that Infopipes seek to exploit. For instance, paths are the unit of scheduling in Scout, and a path, representing all of the processing steps along its length, makes information about all of those steps available to the scheduler. This is similar to the way in that a section of an Infopipe between two negative components is scheduled by one pump.

Routers based on the Click architecture [52] are assembled from packet processing modules similarly to Infopipes. In Click, there is also a notion of push and pull mode. It even supports polymorphic components and checks port compatibility with respect to push and pull. Buffering is made explicit by queue components and network devices work as active sources and sinks, but there is no equivalent of pumps. In contrast to Click, the Infopipe abstraction extends to the application level and can handle information items, which are more complex than packets. Because scheduling characteristics are more diverse for Infopipe components, more elaborate concurrency mechanisms and preemption are supported. Moreover, the pipelines can be distributed.

CONCLUSIONS AND FUTURE WORK

Infopipes provide a framework for building information flow pipelines from components. This abstraction extends uniformly from source to sink. The application controls the setup of the pipeline, configuring its behavior based on QoS parameters and other properties exposed by the components.

The Infopipe platform manages concurrent activity in the pipeline and encapsulates synchronization in high-level communication mechanisms. To specify scheduling policies, the application programmer needs only to choose appropriate pumps, which interact with the underlying scheduler and control the actual timing. Neutral components such as filters can be implemented as active objects, passive consumers, passive producers, or conversion functions, whichever is most suitable for a given task, and existing code can be reused regardless of its implementation style with respect to threading. The Infopipe platform also handles creation of and communication between threads and coroutines. This is very much like the way in which CORBA handles marshaling and remote communication.

We have implemented a prototype Infopipe platform, and thus have shown that the Infopipe approach is feasible. We have built distributed video applications to test and evaluate individual mechanisms as well as the final platform. Our experience in implementing them has shown that Infopipe functionality greatly facilitates the development of information-flow applications.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

As for future work, implementing additional schedulers and communication mechanisms helps to increase confidence in the current abstractions and interfaces. While flexibility was a design goal in building the middleware, more experience using it will allow assumptions and invariants to be identified more exactly.

For synchronization and scheduling, the current implementation assumes one thread per positive component. This precludes any parallelism inside a pipeline section between two negative components. To integrate multiprocessor support, positive components need to be enabled to control multiple threads. This capability would permit pipeline-style parallelism among components in the same pipeline section. Nonetheless, invocations of each component could still be synchronized at the component boundaries, avoiding the necessity of reentrant implementations. Then, the development of information flow applications on multiprocessor machines would also benefit from using pumps as dedicated components for concurrency control.

Acknowledgements

The Infopipe video player is based on a video pipeline built by Ashvin Goel and Charles 'Buck' Krasic using coroutines and POSIX threads.

REFERENCES

1. Black AP, Huang J, Walpole J. Reifying communication at the application level. In *Proceedings of the International Workshop on Multimedia Middleware*. ACM, 2001; Also available as OGI technical report CSE-01-006.
2. Pu C, Schwan K, Walpole J. Infosphere project: System support for information flow applications. *ACM SIGMOD Record* 2001; **30**(1).
3. Cen S, Pu C, Staehli R, Cowan C, Walpole J. A distributed real-time MPEG video audio player. In *Proceedings of the Fifth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, volume 1018 of *Lecture Notes in Computer Science*. Springer Verlag, 1995; 142–153.
4. Walpole J, Koster R, Cen S, Cowan C, Maier D, McNamee D, Pu C, Steere D, Yu L. A player for adaptive mpeg video streaming over the internet. In *Proceedings of the 26th Applied Imagery Pattern Recognition Workshop (AIPR-97)*. SPIE, 1997; .
5. Krasic C, Walpole J. QoS scalability for streamed media delivery. Technical Report CSE-99-011, Oregon Graduate Institute, 1999.
6. Koster R, Kramp T. Using message-based threading for multimedia applications. In *Proceedings of the International Conference on Multimedia and Expo (ICME)*. IEEE, 2001; .
7. Koster R. *A Middleware Platform for Information Flows*. Ph.D. thesis, Department of Computer Science, University of Kaiserslautern, 2002.
8. Black AP. An asymmetric stream communication system. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*. 1983; 4–10.
9. Goel A, Steere D, Pu C, Walpole J. Adaptive resource management via modular feedback control. Technical Report CSE-99-003, Oregon Graduate Institute, 1999.
10. Huang J, Black AP, Walpole J, Pu C. Infopipes – an abstraction for information flow. In *ECOOP 2001 Workshop on The Next 700 Distributed Object Systems*. 2001; Also available as OGI technical report CSE-01-007.
11. Ousterhout J. Why threads are a bad idea (for most purposes), 1996. Invited talk given at USENIX Technical Conference, available at `http://www.scriptics.com/people/john.ousterhout/threads.ps`.
12. van Renesse R. Goal-oriented programming, or composition using events, or threads considered harmful. In *Proceeding of the 8th ACM SIGOPS European Workshop*. 1998; .

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

13. Steere D, Goel A, Gruenberg J, McNamee D, Pu C, Walpole J. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. 1999; 145–158.
14. Briot JP, Guearraoui R, Löhr KP. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys* 1998; **30**(3).
15. Buhr P, Ditchfield G, Stroobosscher R, Younger B, Zarnke C. μC++: Concurrency in the object oriented language C++. *Software – Practice and Experience* 1992; **20**(2):137–172.
16. Koster R, Kramp T. A multithreading platform for multimedia applications. In *Proceedings of Multimedia Computing and Networking 2001*. SPIE, 2001; .
17. Kramp T, Koster R. Flexible event-based threading for QoS-supporting middleware. In *Proceedings of the Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS)*. IFIP, 1999; .
18. Goyal P, Guo X, Vin HM. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI)*. 1996; .
19. Shih C, Liu J, Qian J, Jonnalagadda M, Li J. Open real-time Linux. In *Proceedings of the Second Real-time Linux Workshop*. 2000; .
20. Candea GM, Jones MB. Vassal: Loadable scheduler support for multi-policy scheduling. In *Proceedings of the Second Usenix Windows NT Symposium*. 1998; .
21. Kiczales G, des Rivieres J, Bobrow DG. *The Art of the Metaobject Protocol*. MIT Press, 1991.
22. Lantinga S. Simple DirectMedia Layer. `http://www.libsdl.org`.
23. OMG. *The Common Object Request Broker: Architecture and Specification (Release 2.5)*. OMG, 2001. `http://www.omg.org/cgi-bin/doc?formal/01-09-34`.
24. OMG. CORBA telecoms specification. `http://www.omg.org/corba/ctfull.html`, 1998. Formal/98-07-12.
25. Mungee S, Surendran N, Schmidt DC. The design and performance of a CORBA audio/video streaming service. In *HICSS-32 International Conference on System Sciences, minitrack on Multimedia DBMS and WWW*. 1999; .
26. Harrison TH, Levine DL, Schmidt DC. The design and performance of a real-time CORBA event service. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 1997; .
27. Chambers D, Lyons G, Duggan J. Stream enhancements for the CORBA event service. In *Proceedings of ACM Multimedia Conference*. ACM, 2001; 61–69.
28. Eisenhauer G, Bustamante F, Schwan K. Event services for high performance computing. In *International Conference on High Performance Distributed Computing (HPDC)*. 2000; .
29. Isert C, Schwan K. ACDS: Adapting computational data streams for high performance computing. In *International Parallel and Distributed Processing Symposium (IPDPS)*. 2000; .
30. Pu C, Swint G, Consel C, Koh Y, Liu L, Moriyama K, Walpole J, Yan W. Implementing Infopipes: The SIP/XIP experiment. Technical Report GIT-CC-02-31, Georgia Institute of Technology, 2002.
31. Blair GS, Andersen A, Blair L, Coulson G, Sánchez Gancedo D. Supporting dynamic QoS management functions in a reflective middleware platform. *IEE Proceedings – Software* 2000; **147**(1):13–21.
32. Andersen A, Blair GS, Eliassen F. A reflective component-based middleware with quality of service management. In *Protocols for Multimedia Systems (PROMS)*. Cracow, Poland, 2000; .
33. Blair GS, Coulson G, Andersen A, Blair L, Clarke M, Costa F, Duran-Limon H, Fitzpatrick T, Johnston L, Moreira R, Parlavantzas N, Saikoski KB. The design and implementation of Open ORB 2. *IEEE Distributed Systems Online* 2001; **2**(6).
34. Kramp T, Coulson G. The design of a flexible communications framework for next-generation middleware. In *Proceedings of the Second International Symposium on Distributed Objects and Applications (DOA)*. IEEE, 2000; .
35. Eliassen F, Kristensen T, Plagemann T, Rafaelsen HO. MULTE-ORB: Adaptive QoS aware binding. In *Workshop on Reflective Middleware (RM 2000)*. New York, USA, 2000; .
36. Plagemann T, Eliassen F, Hafskjold B, Kristensen T, Macdonald RH, Rafaelsen HO. Managing cross-cutting qos issues in multe middleware. In *Proceedings of the ECOOP Workshop on Quality of Service in Distributed Object Systems*. Sophia Antipolis and Cannes,France, 2000; .
37. Kristensen T, Plagemann T. Enabling flexible qos support in the object request broker COOL. In *Proceedings of the ICDCS International Workshop on Distributed Real-Time Systems (IWDRS 2000)*. IEEE, Taipei,Taiwan, 2000; .
38. Lea D. *Concurrent Programming in Java*. Addison-Wesley, 1997.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

39. Morrison JP. *Flow-Based Programming : A New Approach to Application Development*. Van Nostrand Reinhold, 1994.
40. Lindblad CJ, Tennenhouse DL. The vusystem: A programming system for compute-intensive multimedia. *IEEE Journal of Selected Areas in Communications* 1996; **14**(7):1298–1313.
41. Naguib H, Coulouris G. Towards automatically configurable multimedia applications. In *Proceedings of the International Workshop on Multimedia Middleware*. ACM, 2001; 28–31.
42. Mircosoft. DirectX 8.0: DirectShow overview. `http://msdn.microsoft.com/library/psdk/directx/dx8_c/ds/Oview/about_dshow.htm`, 2001.
43. Taymans W. GStreamer application development manual. `http://www.gstreamer.net/documentation.shtml`, 2001.
44. RealNetworks. Documentation of RealSystem G2 SDK, gold r4 release. `http://www.realnetworks.com/devzone/tools/index.html`, 2000.
45. Magee J, Dulay N, Kramer J. Regis: A constructive development environment for distributed programs. *Distributed Systems Engineering Journal* 1994; **1**(5).
46. DeLine R. Avoiding packaging mismatch with flexible packaging. *IEEE Transactions on Software Engineering* 2001; **27**(2):124–143.
47. van Renesse R, Birman K, Hayden M, Vaysburd A, Karr D. Building adaptive systems using Ensemble. Technical Report TR97-1638, Computer Science Department, Cornell University, 1997.
48. Plagemann T, Plattner B. CoRA: A heuristic for protocol configuration and resource allocation. In *Proceedings of the Workshop on Protocols for High-Speed Networks*. IFIP, 1994; .
49. Vogt M, Plagemann T, Plattner B, Walter T. A run-time environment for Da CaPo. In *Proceedings of INET'93, International Networking Conference*. Internet Society, San Francisco, 1993; .
50. Hutchinson NC, Peterson LL. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering* 1991; **17**(1):64–76.
51. Mosberger D, Peterson LL. Making paths explicit in the Scout operating system. In *Proceedings of the second USENIX symposium on Operating systems design and implementation (OSDI)*. USENIX, 1996; .
52. Kohler E, Morris R, Chen B, Jannotti J, Kaashoek MF. The Click modular router. *ACM Transactions on Computer Systems* 2000; **18**(3):263–297.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls