Thread Transparency in Information Flow Middleware

Rainer Koster
Andrew P. Black
Jie Huang
Jonathan Walpole
Calton Pu

Technical Report CSE-01-005

First issued May 2001, Revised August 2001

OGI School of Science and Engineering
Oregon Health and Science University
Beaverton, Oregon.

Thread Transparency in Information Flow Middleware

Rainer Koster
University of Kaiserslautern, koster@informatik.uni-kl.de
Andrew P. Black, Jie Huang, Jonathan Walpole
OGI School of Science and Engineering, Oregon Health and Science University,
{black,jiehuang,walpole}@cse.ogi.edu
Calton Pu
Georgia Institute of Technology, calton@cc.gatech.edu

Abstract

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 middleware that supports control-flow centric interaction models such as remote method invocation does not match the structure of these applications. Indeed, it abstracts away from the very things that the information-flow centric programmer must control.

We are defining Infopipes as a high-level abstraction for information flow, and we are developing a middleware framework that supports this abstraction directly. Infopipes handle the complexities associated with control flow and multi-threading, relieving the programmer of this task. Starting from a high-level description of an information flow configuration, 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.

1 Introduction

The benefit of middleware platforms is that they handle application-independent problems transparently to the programmer and hide underlying complexity. CORBA or RPC, for instance, provide location transparency by hiding message passing and marshalling. Hiding of complexity relieves programmers from tedious 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.

This work is partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs, by NFS award CDA-9703218, and by Intel.

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 behaviors, which are not key aspects of the application. Moreover, existing middleware has inadequate abstractions for specifying data-flow behaviors, 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 and the Infopipe abstraction. Infopipes simplify the task of building distributed streaming applications by providing basic components such as pipes, filters, buffers, and pumps [2, 28]. 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 that can be hidden in the middleware. Hence, our platform frees the programmer from the need to deal with thread creation, destruction, and synchronization. Moreover, the actual control flow is managed by the middleware and is decoupled from the way pipeline components are implemented, be they active or passive objects. We call this approach *thread transparency*. It simplifies programs and allows reuse of infopipe components. In the same way that RPC systems automatically generate code for parameter marshalling and message handling, our middleware handles thread management and generates glue code that allows Infopipe components to be reused in different activity contexts.

Section 2 describes the Infopipe middleware platform we are developing. Thread transparency is discussed in Section 3. Section 4 describes the current implementation. Related work is summarized in Section 5 before the conclusions in Section 6.

2 Infopipe Middleware

The Infopipe abstraction has emerged from our experience building continuous media applications [6, 13, 16, 36]. Currently we are building a middleware framework in C++ based on these concepts. On top of this platform we are reimplementing our video pipelines to facilitate further experimentation.

2.1 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 and one output port. Such pipes can *transport* information, *filter* certain information items, or *transform* the information. *Buffers* provide temporary storage and remove rate fluctuations. *Pumps* are used to keep the information flowing. Corresponding to these roles each port has a appropriate *polaritiy*. Pumps have two ports with *positive* polarity, one pulling items from upstream and one pushing them downstream. Buffers, in contrast, have two *negative* ports being pulled from or pushed into. Filters and transformers have two ports of opposite polarity [1,2]. *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 splitting an information item into parts that are sent different ways, copying 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 combining components of 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 the

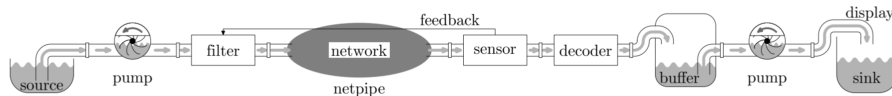

Figure 1: Infopipe example

capability of operating in push or pull mode, 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. Marshalling filters on either side translate the raw data flow to and from a higher-level information flow. These components also encapsulate the mapping of QoS properties, which is described in more detail in Section 2.4.

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 and a feedback toolkit for adaptation control [8]. 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 all use any available video codec components.

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 encapsulating a best-effort transport protocol. The filter drops frames when the network is congested. The dropping is controlled by a feedback mechanism using a sensor on the consumer side. This lets us control which data is dropped rather than suffering arbitrary dropping in the network. After decoding the frames, they are buffered to reduce jitter. A second pump controlling the output timing finally releases the frames to the display.

2.2 Interaction

With respect to polarity, there are three 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. Common components 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 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 components up- or downstream. Hence, Pumps encapsulate the interaction with the underlying scheduler.

Besides exchanging data items, Infopipe components can exchange control messages. These messages are used to represent local interaction between adjacent components as well as global broadcast events. As an example of local interaction, consider an MPEG-decoder that passes on decoded

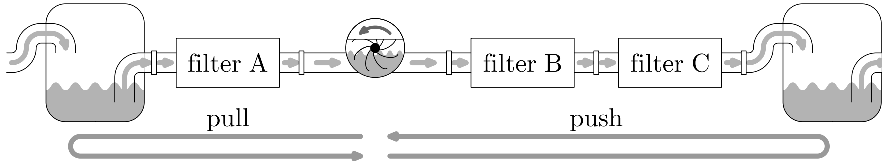

Figure 2: Polarity

video frames but must still keep them as reference frames. Communication between the decoder and downstream components must be used to determine when the shared frames can be deleted. Another case is 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 potentially many components. While control events to adjacent components can easily be sent directly, we use a simple event service to facilitate global distribution of control events.

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 and buffering of these events and their handlers are executed with higher priority than potentially long-running data processing.

2.3 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 binding protocols for 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 the flow determines whether items are pushed or pulled. Polarity is represented in the Typespec by assigning each port a positive or negative polarity. A positive out-port will make calls to the `push` method of the downstream components, while a negative out-port has the ability to receive a `pull`. Correspondingly, a positive in-port will make calls to `pull`, while a negative in-port 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. For example, filters can operate in push or pull mode, as can chains of filters. These components are given the polymorphic polarity $\alpha \to \overline{\alpha}$. 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 [2, 9].

- 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 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.

- *QoS parameters* may include video frame rates and sizes, latency, or jitter. While processing a flow with specific values for these parameters requires elaborate resource management and binding protocols, QoS parameters may provide valuable hints to the rest of the pipeline even if guarantees are not available. For instance, feedback mechanisms can trade one quality dimension for another, for instance, trade frame rate for timely delivery, which again can be reflected in the Typespec.
- For distributed pipelines, the *location* indicates that a flow, from a source or to a sink, for instance, 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 likes 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 may support only a subset of the flow types in the given Typespec, reflecting 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.

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.

2.4 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. Marshalling filters on either side translate the raw data flow to a higher-level information flow and vice-versa. 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. Remote Typespec queries also require a middleware protocol as well as a mechanism for property marshalling. The location itself can be integrated in 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.

3 Transparent Thread Management

Different timing requirements and computation times at different stages of a pipeline require multiple asynchronous threads. Unfortunately, handling multi-threading and synchronization mechanisms is difficult for many programmers and frequently leads to errors [26, 33]. 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 such as timing behavior need to be exposed to the programmer, as described in Section 3.1, other aspects such as scheduler interfaces,

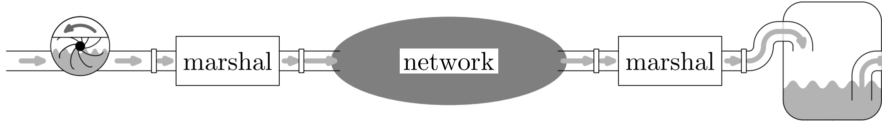

Figure 3: Distributed Infopipe

inter-thread synchronization, wrappers and the adaptation of implementation styles can largely be hidden in the middleware platform, as described in the following three subsections.

3.1 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, they can make reservations, if supported, according to estimated or worst case execution time of the pipeline stages they run. 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 [31]. Another kind of environment-sensitive pump is used on the producer node of a distributed pipeline [6, 36]. 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 hide thread creation and scheduling mechanisms. The programmer does not need to deal with these low-level details but can choose timing and scheduling policies by choosing pumps and by setting appropriate parameters.

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 similarly to the way that a decoder encapsulates compression mechanisms. In both cases, the complexity is hidden from the application programmers who use the new components.

3.2 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 inter-thread synchronization explicitly, but just provides data processing and event handling functions. Hence, inter-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 [4]: each component may contain at most one active thread at any time. However, we allow threads to be preempted because running functions such as video de-

coders non-preemptively can introduce unacceptable delay. A data processing function of one component is never called before the 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.

3.3 Implementation Styles in Pipeline Components

In this section we discuss several styles with respect to activity that can be used in implementing pipeline components. The main distinction is between active objects that have an associated thread and passive objects that are called by external threads [4]. To make a clear distinction between the polarity of a component as introduced in Section 2.2 and its implementation style, we call implementations as active objects *thread-style components* and implementations as passive objects *function-style components*. We focus on neutral components with one input and one output port, which are most common. As a simple example we use a defragmenter that combines two data items into one. The actual merging is performed by the function `y=assemble(x1,x2)`.

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

To implement these components in function style, `push` or `pull` must be provided by the programmer. By convention, the programmer does not directly call `push` or `pull` methods on other components. He instead uses `put` and `get` methods, which are inherited from a base class provided by the middleware platform. In this case, the implementation of `put` and `get` is as follows:

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

For the defragmenter example, the `push` and `pull` methods are shown in Figure 4. Each numbered group of arrows shows the control flow for one call to the method that it annotates. In Figure 4b, 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 4a 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. 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.

There are several reasons for integrating thread-style components that are written as active objects into this framework. 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.

The way 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 [5]. 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 5 gives an example of two coroutines interacting in push mode. An item is pushed into

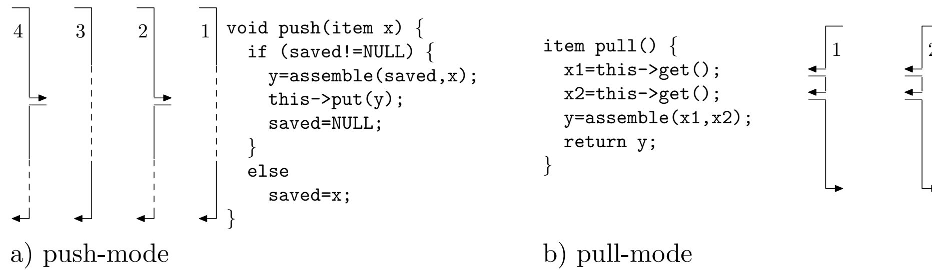

Figure 4: Function-style defragmenter

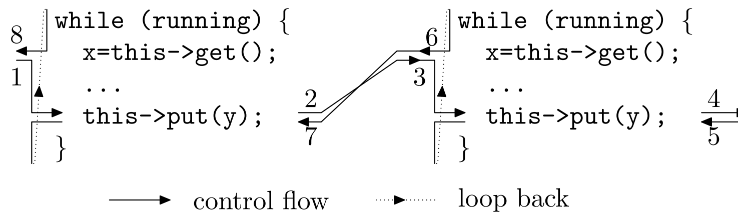

Figure 5: Synchronous threads

the first component deblocking it from a `this->get` call (1). The component then processes the data and calls `this->put`, which passes the item to the next component (2), which deblocks 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. 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 blocked in its `get` method. Pull mode is handled analogously.

Figure 6 shows an thread-style implementation of the defragmenter example. Here again, each numbered group of arrows denotes the control flow for one `push` call (in Figure 6a) or one `pull` call (in Figure 6b) 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, 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.

The function-style implementation shown in Figure 4 has 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 either pull or push 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 7. Figure 8 shows the resulting control flow for the defragmenter example.

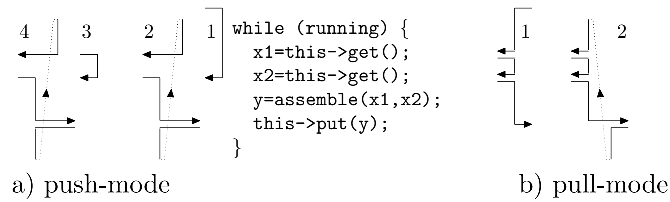

Figure 6: Thread-style defragmenter

Note that the information flow is the same in Figures 4, 6, and 8. 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 Part a of the figure and every pull triggers two upstream pulls in Part b.

In the common case of a component that produces exactly one output for each input, an additional, particularly simple, implementation is possible. All the programmer needs to do is provide a conversion function: `item convert(item x)`. While the functionality is restricted by this one-to-one mapping, this type of component can easily be used in pull as well as push mode.

The `push` and `pull` methods are provided by the middleware:

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

While we have used a defragmenter as an example, 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 9 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 four styles of neutral components. 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.

3.4 Complex 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 in-port and two out-ports. Incoming packets are routed to one of the out-ports depending on the data in the packet. Now consider this switch in pull mode, that is, packets are pulled from either out-port. A pull request arrives at out-port 1 triggering an upstream pull-request at the in-port. Suppose that the incoming packet is routed to out-port 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 out-port 1 and buffering packets at out-port 2 until all packets at out-port 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

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

a) Push-mode wrapper for `pull`

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

b) Push-mode wrapper for `push`

Figure 7: Coroutine wrappers

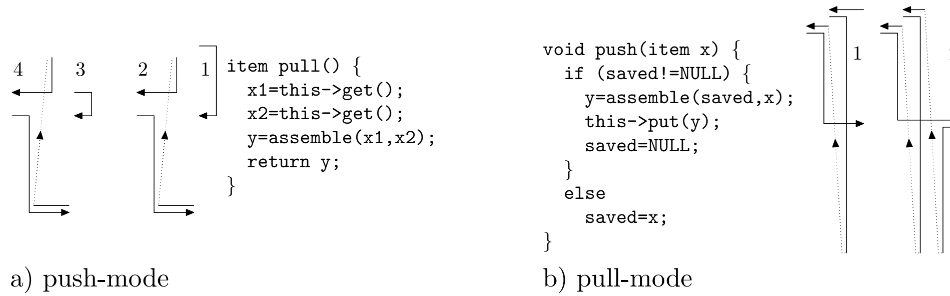

Figure 8: Function-style defragmenters, used other way

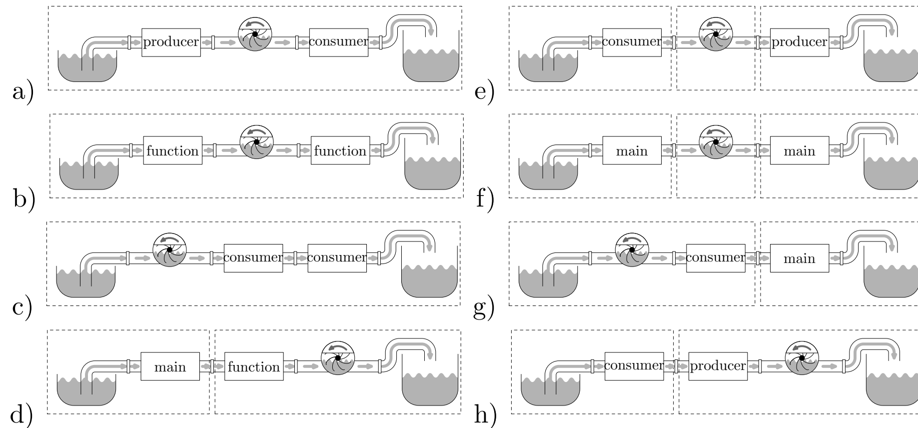

Figure 9: Pipelines and coroutines

route the packet not according to the value of the packet, but based on the activity. A pull on either out-port triggers an upstream pull and returns the item to the caller. In this case, the out-ports must both be negative and the in-port must be positive. This component could not work in push mode.

4 Implementation

The development of the Infopipe middleware described in Section 2 is still in progress. We have implemented the activity-related functionality discussed in the previous section and part of the Typespec processing. A local video player has been built on top of it.

The platform is built on a message-based user-level thread package [12,13,15] implemented in C++. Each thread consists of a code function and a queue for incoming messages. 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 to do for a thread until a reply is received, or asynchronously whenever a reply is not needed immediately, or no reply is required at all. Network packets and signals from the operating system are mapped to messages by the platform, allowing all kinds of events to be handled by a uniform message interface.

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 9. 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 then the thread would not be responsive to control events. Instead, the thread blocks waiting for either a control message 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 implementation establishes synchronous communication of data items between coroutines, while control events can be handled even if the component is blocked in a `pull` or `push`.

The thread package supports scheduling by attaching priorities to threads as well as by attaching constraints to messages. In the latter case, the effective priority of a thread is derived by the scheduler from the constraint of the message that the thread is currently processing or, if the thread is waiting for the CPU, on the constraint of the first message in its queue. If no constraint is specified for the message, a static priority is used that is assigned to the thread when it is created. The package provides a priority inheritance scheme that modifies this behavior as necessary to avoid priority inversion, for instance, when a thread receives a message with a higher priority than that of the message it is currently processing.

In the Infopipe framework, message constraints are assigned by the pumps. Messages between coroutines inherit the constraint from the message received by the sending component, 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.

While other systems for concurrency such as μC++ provide coroutines directly [5], this message-based approach facilitates the processing of control events and the scheduling of concurrent activities according to different timing constraints [13].

The component developer indicates his choice of implementation style by inheriting from the appropriate base class 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 function-style component. 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.

Pipelines are configured by a high-level C++ interface. Composition and start of a simple video player could be implemented by

```
mpeg_file source("test.mpg");
mpeg_decoder decode;
clocked_pump pump(30);  // 30 Hz
video_display sink;
source>>decode>>pump>>sink;
main_channel.send_event(START);
```

If the components were not compatible, the composition operator `>>` would throw an exception. This simple example does not compensate for jitter caused by varying decoding times. The last line starts the pipeline by broadcasting a control event, to which the pump reacts. In a video player for complex presentations consisting of several streams, an additional control component would register for global control events such as START and in response dispatch START events to individual pipelines at the start time of their stream relative to the start time of the overall presentation.

A context switch between the user level threads takes about $1\,\mu$s; the time for a mere function call is two orders of magnitude shorter. Hence, the approach that we have presented in which threads and coroutines are introduced only when necessary is mostly important for pipelines that handle many control events or many small data items, such as a MIDI mixer. For these applications, and if kernel-level threads are used, allocating a thread for each pipeline component would introduce a significant context switching overhead.

5 Related Work

Some related work aims at integrating streaming services with middleware platforms based on remote method invocations such as CORBA. The CORBA telecoms specification [25] defines stream management interfaces, but not the data transmission. Only extensions to CORBA such as TAO's pluggable protocol framework [17] allow the efficient implementation of audio and video applications [24].

One approach for adding quality of service support to CORBA has been introduced by the QuO architecture [35]. It complements the IDL descriptions with specifications of QoS parameters and adaptive behavior in domain specific languages. From these declarative descriptions so called delegates are generated and linked to the client application in a similar way to that in which stubs are generated from an IDL. QuO, however, has not been built for streaming applications and interaction is based on remote method invocations.

A model for specifying flow quality and interfaces has been proposed as part of the MULTE project [29]. Compatibility and conformance rules are used for type checking and stream binding. This model is more formal, but less flexible, than our current approach using Typespecs.

Similarly to Infopipes, the Regis environment [20] 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, there are plans for developing an Infopipe Composition and Restructuring Microlanguage [28].

Open middleware platforms and communications frameworks such as OpenORB [3] and Bossa Nova [14] offer a flexible infrastructure that supports QoS-aware composition and reflection. While these frameworks do not provide specific streaming support, they can serve as a basis for building information flow middleware.

Event-based middleware such as Echo [7, 11] provides a type-safe and efficient way of communicating data and control information in a distributed and heterogeneous environment. A higher-level Infopipe layer can also be built on top of these platforms.

Ensemble [34] and Da CaPo [27] 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 flow from source to sink, possibly across several network nodes.

The *x-kernel* protocol architecture [10] 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. We support this thread-per-packet approach for Infopipe components that are implemented in a way that allows direct method calls. Alternatively, developers may choose to program in an thread-like style if this simplifies the program structure.

The Scout operating system [23] 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, in Scout paths are the unit of scheduling, 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 that a section of an Infopipe between two passive components is scheduled by one pump.

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. 18]. *Flow-Based Programming* applies this concept to the development of business applications [22]. 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 [19] 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.

For constructing streaming applications from components, there are also free and commercial frameworks [21, 30, 32]. 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 component function-style push and thread-style 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.

6 Conclusions

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 implemenented 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 transparently handles creation of and communication between threads and coroutines. This is very much like the way in which CORBA transparently handles marshalling and remote communication.

We have implemented most middleware functionality for local pipelines. Using this platform, we have built several video processing components and configured a simple video player application. The supported functionality is being extended by distributed setup, resource reservations, and feedback mechanisms.

Acknowledgements

The Infopipe video player is based on a video pipeline built by Ashvin Goel and Charles 'Buck' Krasic using coroutines and POSIX threads. We would like to thank the anonymous reviewers for their help in improving this paper.

References

[1] A. P. Black. An asymmetric stream communication system. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 4–10, October 1983.

[2] A. P. Black, J. Huang, and J. Walpole. Reifying communication at the application level. In *Proceedings of the International Workshop on Multimedia Middleware*. ACM, October 2001. Also available as OGI technical report CSE-01-006.

[3] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next-generation middleware. In *International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, pages 191–206. IFIP, September 1998.

[4] J.-P. Briot, R. Guearraoui, and K.-P. Löhr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3), September 1998.

[5] P. Buhr, G. Ditchfield, R. Stroobosscher, B. Younger, and C. Zarnke. μC++: Concurrency in the object oriented language C++. *Software – Practice and Experience*, 20(2):137–172, February 1992.

[6] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole. 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*, pages 142–153. Springer Verlag, April 1995.

[7] G. Eisenhauer, F. Bustamante, and K. Schwan. Event services for high performance computing. In *International Conference on High Performance Distributed Computing (HPDC)*, August 2000.

[8] A. Goel, D. Steere, C. Pu, and J. Walpole. Adaptive resource management via modular feedback control. Technical Report CSE-99-003, Oregon Graduate Institute, January 1999.

[9] J. Huang, A. P. Black, J. Walpole, and C. Pu. Infopipes – an abstraction for information flow. In *ECOOP 2001 Workshop on The Next 700 Distributed Object Systems*, June 2001. Also available as OGI technical report CSE-01-007.

[10] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[11] C. Isert and K. Schwan. ACDS: Adapting computational data streams for high performance

computing. In *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.

[12] R. Koster and T. Kramp. A multithreading platform for multimedia applications. In *Proceedings of Multimedia Computing and Networking 2001*. SPIE, January 2001.

[13] R. Koster and T. Kramp. Using message-based threading for multimedia applications. In *Proceedings of the International Conference on Multimedia and Expo (ICME)*. IEEE, August 2001.

[14] T. Kramp and G. Coulson. 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, September 2000.

[15] T. Kramp and R. Koster. Flexible event-based threading for QoS-supporting middleware. In *Proceedings of the Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS)*. IFIP, July 1999.

[16] C. Krasic and J. Walpole. QoS scalability for streamed media delivery. Technical Report CSE-99-011, Oregon Graduate Institute, September 1999.

[17] F. Kuhns, C. O'Ryan, D. C. Schmidt, O. Othman, and J. Parsons. The design and performance of a pluggable protocols framework for object request broker middleware. In *Proceedings of the sixth IFIP International Workshop on Protocols for High-Speed Networks (PfHSN)*, August 1999.

[18] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.

[19] C. J. Lindblad and D. L. Tennenhouse. The vusystem: A programming system for compute-intensive multimedia. *IEEE Journal of Selected Areas in Communications*, 14(7):1298–1313, 1996.

[20] J. Magee, N. Dulay, and J. Kramer. Regis: A constructive development environment for distributed programs. *Distributed Systems Engineering Journal*, 1(5), September 1994.

[21] Mircosoft. DirectX 8.0: DirectShow overview. `http://msdn.microsoft.com/library/psdk/directx/dx8_c/ds/Oview/about_dshow.htm`, January 2001.

[22] J. P. Morrison. *Flow-Based Programming : A New Approach to Application Development*. Van Nostrand Reinhold, July 1994.

[23] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the second USENIX symposium on Operating systems design and implementation (OSDI)*. USENIX, October 1996.

[24] S. Mungee, N. Surendran, and D. C. Schmidt. 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*, January 1999.

[25] OMG. CORBA telecoms specification. `http://www.omg.org/corba/ctfull.html`, June 1998. formal/98-07-12.

[26] J. Ousterhout. 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`.

[27] T. Plagemann and B. Plattner. CoRA: A heuristic for protocol configuration and resource allocation. In *Proceedings of the Workshop on Protocols for High-Speed Networks*. IFIP, August 1994.

[28] C. Pu, K. Schwan, and J. Walpole. Infosphere project: System support for information flow applications. *ACM SIGMOD Record*, 30(1), March 2001.

[29] H. O. Rafaelsen and F. Eliassen. Trading and negotiating stream bindings. In *Proceedings of the Second International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware), LNCS 1795*, pages 273–288. IFIP/ACM, Springer, April 2000.

[30] RealNetworks. Documentation of RealSystem G2 SDK, gold r4 release. `http://www.realnetworks.com/devzone/tools/index.html`, May 2000.

[31] D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 145–158, February 1999.

[32] W. Taymans. GStreamer application development manual. `http://www.gstreamer.net/documentation.shtml`, January 2001.

[33] R. van Renesse. Goal-oriented programming, or composition using events, or threads considered harmful. In *Proceeding of the 8th ACM SIGOPS European Workshop*, September 1998.

[34] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. Technical Report TR97-1638, Computer Science Department, Cornell University, 1997.

[35] R. Vanegas, J. A. Zinky, J. P. Loyall, D. A. Karr, R. E. Schantz, and D. E. Bakken. QuO's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*. Springer Verlag, September 1998.

[36] J. Walpole, R. Koster, S. Cen, C. Cowan, D. Maier, D. McNamee, C. Pu, D. Steere, and L. Yu. A player for adaptive mpeg video streaming over the internet. In *Proceedings of the 26th Applied Imagery Pattern Recognition Workshop (AIPR-97)*. SPIE, October 1997.