2002

Thread Transparency in Information Flow Middleware

Rainer Koster
University of Kaiserslautern

Andrew P. Black
Oregon Graduate Institute of Science & Technology, black@cs.pdx.edu

Jie Huang
Oregon Graduate Institute of Science & Technology

Jonathan Walpole
Oregon Graduate Institute of Science & Technology

Calton Pu
Georgia Institute of Technology - Main Campus

Citation Details

Koster, Rainer, Andrew P. Black, Jie Huang, Jonathan Walpole, and Calton Pu. "Thread transparency in information flow middleware." Software: Practice and Experience 33, no. 4 (2003): 321-349.

Thread Transparency in Information Flow Middleware

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

[1] University of Kaiserslautern, koster@informatik.uni-kl.de
[2] Oregon Graduate Institute, {black,jiehuang,walpole}@cse.ogi.edu
[3] Georgia Institute of Technology, calton@cc.gatech.edu

Abstract. Existing middleware is based on control-flow centric interaction models such as remote method invocations, poorly matching the structure of applications that process continuous information flows. Difficulties in building this kind of application on conventional platforms include flow-specific concurrency and timing requirements, necessitating explicit management of threads, synchronization, and timing by the application programmer. We propose *Infopipes* as a high-level abstraction for information flows, and we are developing a middleware framework that supports this abstraction. Infopipes transparently handle complexities associated with control flow and multi-threading. From high-level configuration descriptions the platform determines what parts of a pipeline require separate threads or coroutines and handles synchronization transparently to the application programmer. Independently of the actual activity of pipeline components, they may be programmed like passive or active objects. In this way, the most appropriate programming model can be chosen for a given task and exisiting code can be reused regardless 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.

To provide any useful functionality, however, middleware needs to assume abstractions that are in common for the applications it supports. One assumption, on which exisiting middleware platforms are based, 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.

This model, however, is inappropriate for emerging information-flow applications that pass continuous streams of data among producers and consumers. Building these applications on exisiting middleware requires programmers to specify control-flow behaviors, which are not key aspects of the application. Morever, 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, 24]. Each component specifies the properties of the flows 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.

In addition to providing new abstractions that are useful to the application programmer, the middleware platform hides complexity that is common to information flow 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 supports the concept of thread transparency by freeing the programmer from the need to deal with thread creation and destruction and synchronization. Moreover, the actual control flow is transparently managed by the middleware and is decoupled from the way pipeline components are implemented, be they active or passive objects. This approach simplifies programs and allows widespread 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 [5, 12, 15, 32]. 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 development.

2.1 Overview

Infopipes model pipeline components for information flow analogously to plumbing for water flow. The goal is supporting a similarly simple composition of pipelines from components.

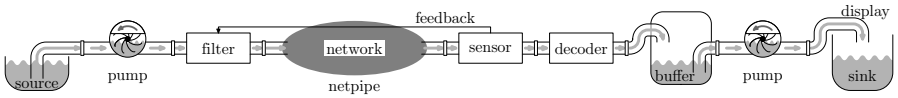


Fig. 1. Infopipe Example

The most common components have one input and one output. Such pipes can *transport* information, *filter* certain information items, or *transform* the information. *Buffers* provide temporary storage and remove rate fluctuations. There are *pumps* to keep the information flowing, pulling items from upstream and pushing them downstream. Hence, pumps have two active ends and buffers have two passive ones, while filters and transformers have two ends of opposite polarity but can typically be used in either push or pull mode [1,2]. *Sources* and *sinks* have only one end, and can be either active or passive.

More complex pieces 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 compatibilty of supported flows and to evaluate the characteristics of the composite Infopipe. From each basic or composite Infopipe a *Typespec* can be queried that describes the flows that it supports. These types include supported formats of data items, interaction properties such as the capability of operating in push or pull mode, and ranges of QoS parameters that can be handled.

To integrate different transport protocols into the Infopipe framework, they can be encapsulated in *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 QoS mapping of netpipe properties and information flow properties.

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 including pumps and buffers to control the timing and a feedback toolkit for adaptation control [7]. Components for processing specific types of flow need to be developed by application programmers, but can easiliy be reused in various applications. For instance, developers of video on demand, video conferencing, and surveillance tools all can 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 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 incurring 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 sink.

2.2 Interaction

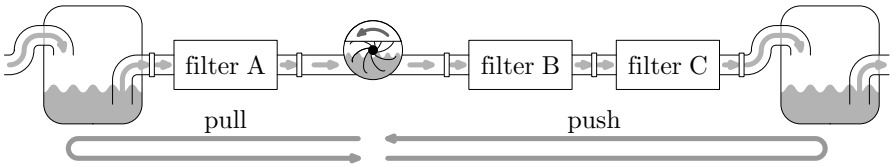


Fig. 2. Activity

In the basic model, pumps have two active ends, buffers have two passive ends, and filters an active and passive end. In this way, any activity in the Infopipe originates from a pump, as shown in Figure 2. Hence, pumps regulate the timing of the data flow and can themselves be controlled by timers or feedback mechanisms. Each pump has an associated thread that calls all other pipeline stages up to the next buffer up- or downstream and encapsulates the interaction with the underlying scheduler. Sinks and sources may be active or passive and, hence, operate similarly to pumps and buffers in this respect.

Besides exchanging data items, Infopipe components can exchange control messages. They include local interaction between adjacent components as well as global broadcast events. To exemplify the local control, consider an MPEG-decoder that passes on decoded video frames and at the same time still needs them as reference frames itself. Communication between the decoder and downstream components must 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 there directly, we are building an event service to facilitate global distribution of control events.

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

2.3 Infopipe Typespec

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 as well as 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 *activity* of ports in the information the flow determines whether items are pushed or pulled. Activity is represented in the Typespec by assigning each port a positive or negative polarity. A positive out-port will make calls to `push`, 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 $\alpha \rightarrow \overline{\alpha}$. When one end is connected to a port with a fixed polarity, the other end of the filter or filter chain acquires an "induced" polarity [2, 8].
- 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, which again can be reflected in the Typespec.

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 support certain data formats and ranges of QoS parameters. The latter can be restricted by the user to indicate preferences.

For any stage in a pipeline, an input or output Typespec can be a subset of a given output or input Typespec, because that stage supports only a subset

of flow types, for instance because it 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 analyses 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.

Fig. 3. Distributed Infopipe

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 at different stages of a pipeline require asynchronously operating threads. However, handling multithreading and synchronization mechanisms is difficult for many programmers and frequently leads to errors [22, 29]. But since 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, inter-thread synchronization, and 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 passive components up- and downstream. Interaction with the underlying scheduler is also implemented in these pipeline components. At setup, they can make reservations, if supported, according to estimated or worst case execution times of the pipeline stages they run. Moreover, they can assign and readjust thread scheduling parameters as the pipeline runs.

From our experience with 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 categorie. Audio devices that have their own timing control can be implemented as a clock-driven active sink. The second class of pumps adjusts its speed according to the state of other pipeline components. The simplest version does not limit its rate at all and relies on buffers 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 [27]. Another kind of pump is used on the producer node of a distributed pipeline [5, 32]. 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 automatic, 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 that encapsulates compression mechanisms. In either case, the complexity is hidden from application programmers using the new components.

3.2 Synchronization

The Infopipe 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 low-level primitives such as locks and semaphores.

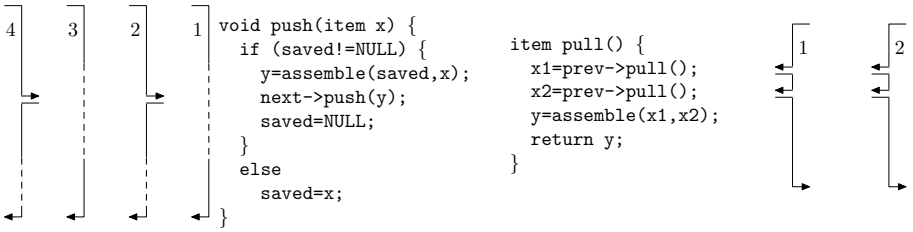
The pipeline components are implemented as synchronized objects [4], that is only one thread is active in one component at any time. However, threads can be preempted in favor of threads driven by other pumps, because running data processing functions such as video decoders non-preemptively can introduce unacceptable delay in more time-critical components such as writing samples to the audio device. A data processing function 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 Activity Styles in Pipeline Components

In this section we discuss several styles of activity that can be used in implementing pipeline components. The main distinction is between active objects that have an associated thread and passive or passive objects that are called by external threads [4]. We focus on the most common pipeline components with one input and one output end. As a simple example we use a defragmenter that combines two data items into one. The actual merging is performed by function `y=assemble(x1,x2)`.

The middleware platform assumes transformation components to be passive. 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 the components in this passive style, `push` or `pull` must be provided by the programmer. For the defragmenter example, the functions are shown in Figure 4. Each enumerated groups of arrows denotes the control flow for one call to the operation it annotates. In Figure 4b, each invocation of `pull` travels all the way through the code triggering two `pull` calls to the upstream pipeline component. For `push` in Figure 4a every other call (2 and 4) causes a downstream `push`. If no output item can be produced the call returns directly.



a) push-mode b) pull-mode

Fig. 4. Passive defragmenter

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.

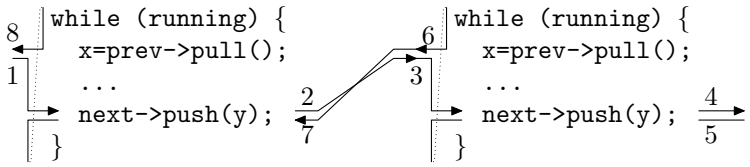


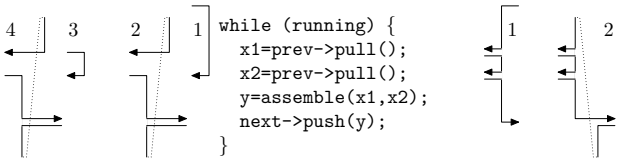
Fig. 5. Synchronous Threads

While components that have been built as active objects with a thread of their own do not match the style of component usage in the pipeline, there are several reasons for supporting them. 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 the model provides. 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 active model than with the passive model.

The way to integrate these active components in the facade of a passive component is to use coroutines, that is, threads interacting synchronously in a coroutine set. These coroutines merely provide a suspendable control flow, but are not a unit of scheduling. The communication mechanism between them does not buffer data; instead the activity travels with the data. All but one coroutines in a given set are blocked at any time. Figure 5 gives an example of two coroutines interacting in this way in push mode. An item is pushed into the first component deblocking it from a `pull` call (1). It then processes the data and pushes the result to the next component (2), which deblocks from its `pull` (3). It again does some processing and a downstream `push` (4). When this call returns (5),

the control flow loops back to the `pull` call blocking the second component (6) and deblocking the first component from its `push` (7). Finally the control flow reaches a `pull` call again and returns to the upstream component (8).

Note that now `prev->pull()` and `next->push()` cannot be direct method invocations to another component. Instead, `get()` or `put()` methods are called, which are provided by the platform. They either are mapped to a `push` or `pull` call or perform a coroutine switch depending on the implementation of the called object. For simplicity and uniformity, we nonetheless maintain the same notation in all examples.



a) push-mode b) pull-mode

Fig. 6. Active defragmenter

Figure 6 shows an active implementation of the defragmenter example. Here again, each enumerated 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 `pull` calls block the defragmenter and each invocation executes from `pull` to `pull`. As an exception, the first `push` call invokes the main function of the component and enters its loop. The `pull` mode works analogously.

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

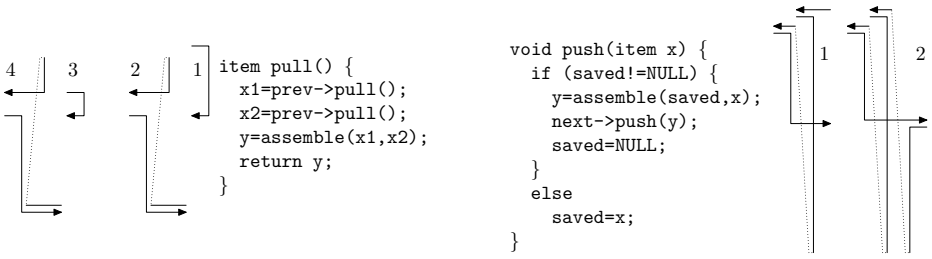
a) Push-mode wrapper for `pull`

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

b) Push-mode wrapper for `push`

Fig. 7. Coroutine wrappers

The passive implementation shown in Figure 4 has a major drawback. Components have to provide both a `push` and a `pull` operation for 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 mid-



a) push-mode b) pull-mode

Fig. 8. Passive defragmenter, used other way

dleware 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.

Note that the external activity 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.

There is one more way of implementing components that produce exactly one outgoing packet for each incoming packet. In this case, a conversion function can be provided: `item fct(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 glue code for the respective functions is simple:

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

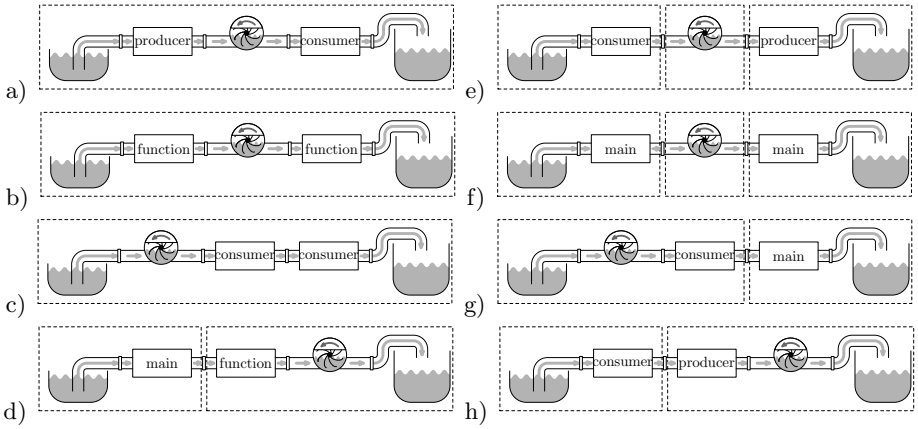


Fig. 9. Pipelines and coroutines

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, the introduction of coroutines provides flexibility in developing and reusing components, but for efficiency it is nonetheless important to avoid context switches and use direct function calls whenever possible. Hence, the framework automatically 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 depicted as dashed boxes. The same applies to pipeline sections between two buffers. Altogether, there are four styles of components. Active object implementations provide a tread-like main function. Passive objects are consumers implementing `push`, producers implementing `pull`, or are based on a conversion function. In push mode, consumers and functions are called directly, and in pull mode producers and 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 and supported interaction styles depend on their functionality. 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-style, 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 passive 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 out-port triggers an upstream pull and returns the item to the caller. In this case, the out-ports must both be passive and the in-port must be active. This component could not work in push-style.

4 Implementation

The developement 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 [11,12,14] 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 only terminated when indicated by the return code. In this way, code functions resemble event handlers, but may be suspended waiting for other messages or may be preempted. Threads work like extended finite state machines. 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 types 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 pipeline section a pump controls, the thread calls the `pull` functions of all components upstream of the pump, then calls `push` with the returned item to the components downstream of the pump, and finally returns to the pump, which schedules the next pull. This case applies to the configurations a), b), and c) in Figure 9. For configurations d), g), and h) there is a set of two coroutines and for configurations e) and f) there is a set of 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` as described in Section 3 are blocking, 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`. Moreover, because a set of pipeline components that communicate directly via function calls share a thread, each thread needs to internally dispatch data and events to the respective components.

The thread package supports scheduling control 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 taht the thread is currently processing or, if the thread is waiting for the CPU, on the constraint of the first message in its incoming queue. If no constraint is specified for the message, the priority statically assigned to the thread is used. The package provides a priority inheritance scheme that modifies this behaviour if needed for avoiding priority inversion, for instance, when a thread receives a message with a higher priority than that of the message it currently processes.

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.

The component developer indicates his choice of activity style by inheriting from the appropriate base class and by overriding a `run` method for an active object, 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.

Pipeline setup is configured by an high-level C++ interface. Composition and start of a simple video player is basically implemented by

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

If the components were not compatible, the composition operator `>>` would throw an exception. The last line starts the pipeline by broadcasting a control event, to which the pump reacts. This simple example does not compensate for jitter caused by varying decoding times.

A context switch between the user level threads takes about 1 μs; 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 [21] defines stream management interfaces, but not the data transmission. Only extensions to CORBA such as TAO's pluggable protocol framework [16] allow the efficient implementation of audio and video applications [20].

One approach for adding quality of service support to CORBA has been introduced by the QuO architecture [31]. 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 [25]. 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 [17] 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 a Infopipe Composition and Restructuring Microlanguage [24].

Open middleware platforms and communications frameworks such as OpenORB [3] and Bossa Nova [13] 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 [6, 10] 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 [30] and Da CaPo [23] 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 [9] associates processes with messages rather than protocols. In this way, messages can be sheperded through the entire protocols 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 function calls, while the developer may nevertheless choose to program in an active style if this simplifies the program structure.

As a more general abstraction, the Scout operating system [19] combines linear flows of data into *paths*. Paths expose invariants and allow components on the path to exploit this non-local information. This general principle is applicable to many aspects of information flows. For instance, in Scout, paths are the unit of scheduling similarly to sections of an Infopipe between two passive components, which are scheduled by one pump.

For constructing streaming applications from components, there are also free and commercial frameworks [18, 26, 28]. 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 implementations in push or active style,

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, however, is hardcoded into their server and may only be configured by adaptation rules.

6 Conclusions

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

The Infopipe middleware 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. Components can be implemenented as active objects, passive consumers, passive producers, or functions, whichever is most suitable for a given task and exisiting code can be reused regardless of its activity model. The Infopipe middleware 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.

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 and J. Walpole. Aspects of information flow. In *ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*, 2000.
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. 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.
6. G. Eisenhauer, F. Bustamante, and K. Schwan. Event services for high performance computing. In *International Conferemce on High Performance Distributed Computing (HPDC)*, August 2000.

7. 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.
8. J. Huang, A. P. Black, J. Walpole, and C. Pu. Aspects of information flow. In *ECOOP 2001 Workshop on Workshop on The Next 700 Distributed Object Systems*, June 2001.
9. 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.
10. C. Isert and K. Schwan. ACDS: Adapting computational data streams for high performance computing. In *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.
11. R. Koster and T. Kramp. A multithreading platform for multimedia applications. In *Proceedings of Multimedia Computing and Networking 2001*. SPIE, January 2001.
12. 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.
13. 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.
14. 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.
15. C. Krasic and J. Walpole. QoS scalability for streamed media delivery. Technical Report CSE-99-011, Oregon Graduate Institute, September 1999.
16. 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.
17. J. Magee, N. Dulay, and J. Kramer. Regis: A constructive development environment for distributed programs. *Distributed Systems Engineering Journal*, 1(5), September 1994.
18. Mircosoft. DirectX 8.0: DirectShow overview. `http://msdn.microsoft.com/library/psdk/directx/dx8_c/ds/0view/about_dshow.htm`, January 2001.
19. 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.
20. 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.
21. OMG. CORBA telecoms specification. `http://www.omg.org/corba/ctfull.html`, June 1998. formal/98-07-12.
22. 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`.
23. 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.
24. C. Pu, K. Schwan, and J. Walpole. Infosphere project: System support for information flow applications. *ACM SIGMOD Record*, 30(1), March 2001.

25. 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.
26. RealNetworks. Documentation of RealSystem G2 SDK, gold r4 release. `http://www.realnetworks.com/devzone/tools/index.html`, May 2000.
27. 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.
28. W. Taymans. GStreamer application development manual. `http://www.gstreamer.net/documentation.shtml`, January 2001.
29. 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.
30. 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.
31. 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.
32. 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.