Multimedia Systems 8: 406–419 (2002)
Digital Object Identifier (DOI) 10.1007/s00530-002-0062-3

Multimedia Systems
Springer-Verlag 2002

Infopipes: An abstraction for multimedia streaming

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

[1] Department of Computer Science and Engineering, OGI School of Science and Engineering, Oregon Health and Science University (e-mail: black@cse.ogi.edu)
[2] Fachbereich Informatik, University of Kaiserslautern
[3] School of Computing, Georgia Institute of Technology

Abstract. To simplify the task of building distributed streaming applications, we propose a new abstraction for information flow – Infopipes. Infopipes make information flow primary, not an auxiliary mechanism that is hidden away. Systems are built by connecting predefined component Infopipes such as sources, sinks, buffers, filters, broadcasting pipes, and multiplexing pipes. The goal of Infopipes is not to hide communication, like an RPC system, but to *reify* it: to represent communication explicitly as objects that the program can interrogate and manipulate. Moreover, these objects represent communication in application-level terms, not in terms of network or process implementation.

Key words: Quality of service – Streaming – Communication – Feedback – Real-rate systems

1 Introduction

Recent years have witnessed a revolution in the way people use computers. In today's Internet-dominated computing environment, information exchange has replaced computation as the primary activity of most computers. This revolution began with the use of the World Wide Web for accessing relatively static information. It has continued with the emergence of streaming applications, such as video and music on demand, IP-telephony, Internet radio, video conferencing and remote surveillance systems. Recent traffic studies (Chesire et al. 2001; Thompson et al. 1997) show that these applications are already major consumers of bandwidth on the Internet and are likely to become dominant in the near future. The advent of interconnected, embedded and sensor-based systems will accelerate the development and deployment of new streaming applications.

The salient characteristics of streaming applications are their extensive use of communication among distributed components and their real-time interaction with real-world processes. Consequently, developers of streaming applications spend much of their time reasoning about the communications and I/O behaviour of their systems. This change of emphasis from computation to communication is the motivation for our research.

This paper describes *Infopipes*, a new abstraction, together with associated middleware and tools, for simplifying the task of constructing streaming applications. The motivation for developing Infopipes as middleware is to provide a single set of abstractions with a uniform interface that can be made available on a diverse set of hosts and devices. Wide availability is important for streaming applications because, by their nature, they tend to span many potentially heterogeneous computers and networks, and interact with many different devices. The abstractions must also be appropriate to the problem domain: they should expose the primitives useful in that domain, and control and hide the unnecessary details.

The essence of streaming applications is creation and management of information flows via producer–consumer interactions among potentially distributed components. Hence, communication is a primary concern and should be *exposed*, not hidden. Moreover, it is application-level information that must be communicated, not low-level data, so exposing low-level network abstractions is inappropriate.

Exposing the basic communication elements, such as sources, sinks and routes, is inadequate: streaming applications are frequently also concerned with the quality of service (QoS) of that communication. For example, the correct execution of a streaming-media application is often critically dependent on the available bandwidth between the server and client. Adaptive applications may actively monitor this QoS aspect and adapt the media quality dynamically to match their bandwidth requirements to the available bandwidth (Jacobs and Eleftheriadis 1998; Karr et al. 2001; McCanne et al. 1997; Walpole et al. 1997). When constructing streaming applications, these timing and resource-management tasks tend to be the source of much complexity, since they touch on aspects of the environment that differ among applications, and even among different deployments of the same application. In contrast, the computation-intensive aspects of the application, such as media encoding and decoding, can often be addressed using standard components.

The desire to reify communication rather than hide it is in contrast to many distributed-systems middleware platforms that are based around remote procedure call (RPC) mechanisms (ISO 1998; OMG 1998b; OSF 1991; Sun 2002).

Of course, it is also undesirable and unmanageable to expose all of the underlying details of communication. In general, information-flow application developers will not want to reimplement low-level protocol functionality – such as marshalling, fragmentation, congestion control, ordered delivery and reliability – for every application they build. Thus, what we would like to do is to find a way to factor and prepackage such functionality so that it can be selected when needed to ensure a particular property. This emphasis on component-based composition and property composition is a central characteristic of the Infopipe approach.

This approach to dealing with the complexities of communication can be reapplied when dealing with the complexities of scheduling computation. Since timing is critical for many streaming applications, they need some way to control it. However, it is neither desirable nor necessary to expose application developers to all of the ugly details of thread management, scheduling, and synchronization. Instead we attempt to expose the QoS-related aspects of scheduling and hide the unnecessary details.

Thus, our primary goal for Infopipes is to select a suitable set of abstractions for the domain of streaming applications, make them available over a wide range of hardware and operating systems, and allow tight control over the properties that are important in this domain while hiding the unnecessary details.

A further goal, which we discovered to be important through our own experiences building real-time streaming applications, is the ability to monitor and control properties dynamically and in application-specific terms. This capability enables applications to degrade or upgrade their behaviour *gracefully* in the presence of fluctuations in available resource capacity. Since graceful adaptation is an application-defined concept, it cannot be achieved using a one-size-fits-all approach embedded in the underlying systems software. The alternate approach of exposing system-level resource-management information to application developers introduces unnecessary complexity into the task of building applications. Therefore, the goal for a middleware solution is to map system-level resource-management details into application-level concepts so that adaptive resource management can be performed by application components in application-specific terms.

A final goal for Infopipe middleware is to support tools that automatically check the properties of a composite system. For example, important correctness properties for a pipeline in a streaming application are that information be able to flow from the source to the sink, that latency bounds are not exceeded and that the quality of the information meets the requirements. Even though individual Infopipe components may exhibit the necessary properties in isolation, it is often non-trivial to derive the properties of a system that is composed from these components.

The remainder of this paper presents more detail about our ongoing research on Infopipes. Section 2 discusses the Infopipe model, loosely based on a plumbing analogy, and describes the behaviour of various basic Infopipe components. Section 3 discusses some of the properties that are important for composite Infopipes and introduces some preliminary tools we have developed. Section 4 discusses the implementation. Some example Infopipe applications are presented in Sect. 5. Section 6 discusses related work, and Sect. 7 concludes the paper.

2 The Infopipe model and component library

Infopipes are both a model for describing and reasoning about information-flow applications, and a realization of that model in terms of objects that we call Infopipe components. It is central to our approach that these components are real objects that can be created, named, configured, connected and interrogated at will; they exist at the same level of abstraction as the logic of the application, and this exposes the application-specific information flows to the application in its own terms. For example, an application object might send a method invocation to an Infopipe asking how many frames have passed through it in a given time interval, or it might invoke a method of an Infopipe that will connect it to a second Infopipe passed as an argument.

An analogy with plumbing captures our vision: just as a water-distribution system is built by connecting together pre-existing pipes, tees, valves and application-specific fixtures, so an information-flow system is built by connecting together predefined and application-specific Infopipes. Moreover, we see Infopipes as a useful tool for modelling not only the communication of information from place to place, but also the transformation and filtering of that information. The Infopipe component library therefore includes processing and control Infopipes as well as communication Infopipes. We can also compose more complex Infopipes with hybrid functionality from these basic components. Our goal is to provide a rich enough set of components that we can construct information-flow networks, which we call *Infopipelines*, for a wide variety of applications.

2.1 Anatomy of an Infopipe

Information flows into and out of an Infopipe through *ports*; *push* and *pull* operations on these ports constitute the Infopipe's data interface. An Infopipe also has a control interface that allows dynamic monitoring and control of its properties, and hence the properties of the information flowing through it. Infopipes also support connection interfaces that allow them to be composed, i.e., connected together at runtime, to form Infopipelines. The major interfaces required for an object to be an Infopipe are shown in Fig. 1.

It is central to our approach that Infopipes are *compositional*. By this we mean that the properties of a pipeline can be calculated from the properties of its individual Infopipe components. For example, if an Infopipe with a latency of 1 ms is connected in series with an Infopipe with a latency of 2 ms, the resulting pipeline should have a latency of 3 ms – not 3.5 ms or 10 ms.

Compositionality requires that connections between components are *seamless*: the cost of the connection itself must be insignificant. Pragmatically, we treat a single procedure call or method invocation as having insignificant cost. In contrast, a remote procedure call, or a method that might block the invoker, have potentially large costs: we do not allow such costs to be introduced automatically when two Infopipes are

Cloning	
clone	answers a disconnected copy of this Infopipe
Data	
pull	answers an item obtained from this Infopipe.
push: anItem	push an item into this pipe
Connection	
–>> aPortOrInfopipe	connect my Primary outport to aPortOrInfopipe
Port Access	
inPort	answers my Primary Inport
inPortAt: name	answers my named Inport
inPorts	answers a collection containg all of my inports
outPort	answers my primary Outport
outPortAt: name	answers my named Outport
outPorts	answers a collection containing all of my outports
nameOfInPort: anInPort	answers the name of anInPort
nameOfOutPort: anOutPort	answers the name of anOutPort
openInPorts	answers a collection containing all of my Inports that are not connected
openOutPorts	answers a collection containing all of my Outports that are not connected
Pipeline Access	
allConnectedInfoPipes	answers a collection containing all of the Infopipes in the same Infopipeline as myself.
inConnectedTo	answers a collection containing all of the Infopipes that are directly connected to my Inports
outConnectedTo	answers a collection containing all of the Infopipes that are directly connected to my Outports

Fig. 1. Principal interfaces of an Infopipe

connected. Instead, we encapsulate remote communication and flow synchronization as Infopipe components, and require that the client include these components explicitly in the Infopipeline. In this way, the costs that they represent can also be included explicitly.

Other properties may not compose so simply as latency. For example, CPU load may not be additive: memory locality effects can cause either positive or negative interference between two filters that massage the same data. While we do not yet have solutions to all of the problems of interference, we do feel strongly that addressing these problems requires us to be explicit about all of the stages in an information flow.

2.2 Control interfaces

The control interface of an Infopipe exposes and manages two sets of properties: the properties of the Infopipe itself, and the properties of the information flowing through it. To see the distinction, consider an Infopipe implemented over a dedicated network connection. The bandwidth of this Netpipe is a property of the underlying network connection. However, the actual data flow rate, although bounded by the bandwidth, may vary with the demands of the application.

We regard both pipe and flow properties as control properties because they are clearly related. Indeed, expressing pipe properties such as bandwidth in application-level terms (e.g., frames per second rather than bytes per second) requires information about the flow.

Different kinds of Infopipe provide different control interfaces. For example, we have fillLevel for buffers and slower and faster for pumps. We are investigating the properties and control information that should be maintained in Infopipes and in information flows to support comprehensive control interfaces.

2.3 Ports

To be useful as a component in an Infopipeline, an Infopipe must have at least one *port*. Ports are the means by which information flows from one Infopipe to another, and are categorized by the direction of information flow as either *Inports* (into which information flows, indicated by the symbol ▮◗) or *Outports* (from which information flows, indicated by the symbol ▮▶).

Each Infopipe has a set of named Inports and a set of named Outports; each port is owned by exactly one Infopipe. For straight-line pipes, both the Inport set and the Outport set have a single element, which is named Primary.

OutPorts have a method –>> anInPort that sets up a connection to anInPort. Infopipes also have a –>> method, which is defined as connecting the primary OutPort of the upstream pipe to the primary InPort of the downstream pipe.

Information can be passed from one Infopipe to another in two ways. In *push mode*, the Outport of the upstream component invokes the method push: anItem[1] on the Inport of the downstream component, as shown in Fig. 2a. In *pull mode*, shown in Fig. 2b, the situation is dual: the Inport of the downstream component invokes the pull method of the Outport of the upstream component, which replies with the information item. The ports P and S (shaded ▮ in the figure) invoke methods; we say that they are *positive*. Ports Q and R (shaded ▯) execute methods when invoked; we say that they are *negative*. In a well-formed pipeline, connected ports have opposite direction and opposite polarity. Any attempt to connect, for example, an Inport to another Inport, or a positive port to another positive port, should be rejected.

It is not obvious that Infopipes need the concept of port. Indeed, our first prototypes of straight-line Infopipes did not have ports: a pipe was connected directly to its upstream and downstream neighbours, and each pipe had two connection

[1] We follow the Smalltalk convention of using a colon (rather than parenthesis) to indicate where an argument is required. Often, as here, we will provide an example argument with a meaningful name.

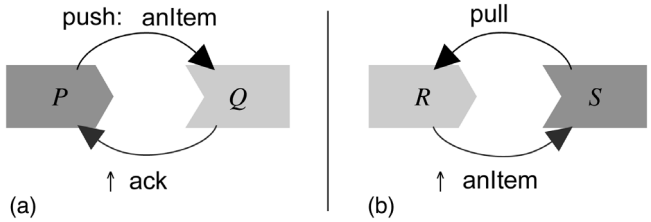


Fig. 2. (**a**) Push mode communication; (**b**) pull mode communication

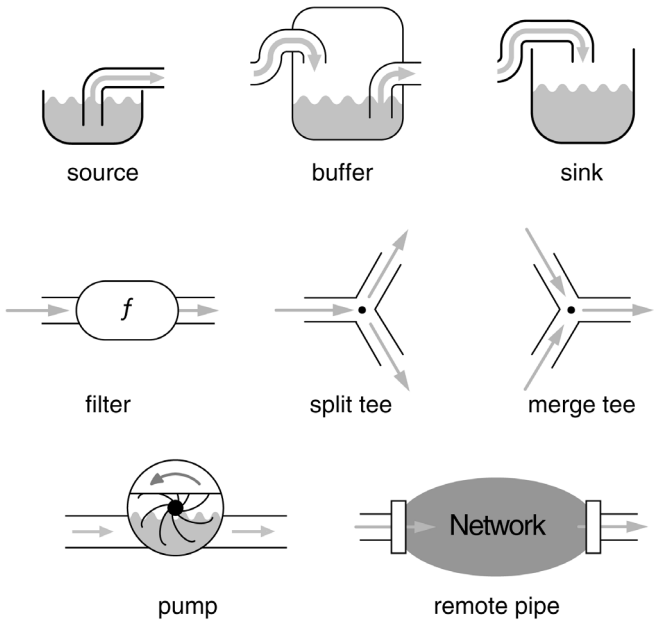


Fig. 3. Some Infopipe components

methods, input: and output:. However, the introduction of *Tees* – that is, pipes with multiple inputs and outputs – would have made the connection protocol more complex and less uniform. Ports avoid this complexity, and turn out to be useful in building RemotePipes and CompositePipes as well, as we shall explain later.

2.4 Common components

Figure 3 illustrates some Infopipe components. *Sources* are Infopipes in which the set of Inports is empty; *Sinks* have an empty set of Outports. *Tees* are Infopipes in which one or both of these sets have multiple members. These ports can be accessed by sending the Tee the messages inPortAt: aName and outPortAt: aName; the ports can then be connected as required. Figure 4 shows an example.

In addition, we can identify various other Infopipes.

- A *buffer* is an Infopipe with a negative Inport, a negative Outport, and some storage. The control interface of the buffer allows us to determine how much storage it should provide, and to ascertain what fraction is in use.
- A *pump* is an Infopipe with a positive Inport and a positive Outport. Its control interface lets us set the rate at which the pump should consume and emit information items.
- A *remote pipe* is an Infopipe that transports information from one address space to another. Although the Infopipe abstraction is at a higher level than that of address space, a middleware implementation must recognize that a host program executes in an address space that is likely to encompass only part of the Infopipeline. Remote pipes bridge this gap; the Inport and Outport of a remote pipe exist in different address spaces, and the remote pipe itself provides an information portal between those address spaces. Remote pipes can be constructed with different polarities, reflecting the different kinds of communication path. An IPCPipe between two address spaces on the same machine might provide reliable, low-latency, communication between a negative Inport and a positive Outport; such a pipe emits items as they arrive from the other address space. A Netpipe that connects two address spaces on different machines has two negative ports and provides buffering; items are kept until they are requested by the next connected Infopipe in the downstream address space.

An important aspect of component-based systems is the ability to create new components by aggregating old ones, and then to use the new components as if they were primitive. *Composite* pipes provide this functionality; any connected subnetwork of Infopipes can be converted into a CompositePipe, which clients can treat as a new primitive.

In order for clients to connect to a composite pipe in the same way as to a primitive Infopipe, without knowing anything about its internal structure, and indeed without knowing that it *is* a composite rather than a primitive, a composite pipe must have its own ports. We call these ports *ForwardedPorts*. The ForwardedPorts are in one-to-one correspondence with, but are distinct from, the open ports of the sub-components. We cannot use the same object for the ForwardedPort and the real port because the real port is owned by the sub-component while the ForwardedPort is owned by the CompositePipe itself. Figure 5 shows the internal structure of a composite pipe. From the outside, it is just an ordinary Infopipe with two Inports and two Outports. Open ports of different sub-components may have the same name, but their ForwardedPorts must have different names because the ports of an Infopipe must be distinguishable.

One inevitable difference between composite and primitive Infopipes is that the former need more complex initialisation: the internal structure of the Composite must be established before it can be used. It is therefore convenient to adopt a prototype-oriented style of programming, where a Composite is first constructed and then *cloned* to create as many instances as required. To support this style uniformly, all Infopipes (not just composite pipes) have a clone method, which makes a pipe-specific set of choices about what parameters to copy and what parameters to reinitialise. For example, when a pump is cloned, the pumping rate is copied from the prototype, but the ports of the clone are left open.

3 From pipes to pipelines: analysis and tools

3.1 Polarity checking and polymorphism

The concept of port polarity introduced in Sect. 2.3 is the basis for several useful checks that an Infopipeline is well-formed.

From the polarity of an Infopipe's ports, we can construct an expression that represents the polarity of the Infopipe itself.

```
"Create some Infopipes"
source    := SequentialSource new.
pump  := Pump new.
multicastTee      := MulticastTee new.
mixTee   := MixTee new.
sink    := Sink new.

"Connect them"
source –>> pump –>> multicastTee.
(multicastTee outPortAt: #Primary) –>> (mixTee inPortAt: #Primary).
(multicastTee outPortAt: #Secondary) –>> (mixTee inPortAt: #Secondary).
mixTee –>> sink.

"Make data items flow."
pump startPumping: 1000.

"result pipeline"
```

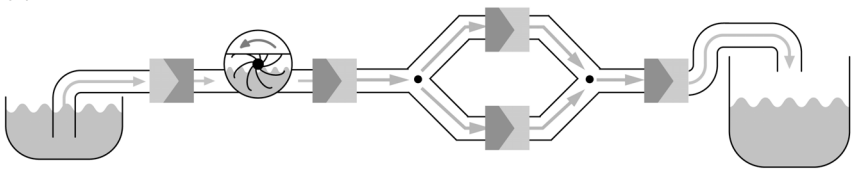


Fig. 4. Building a pipeline with Tees

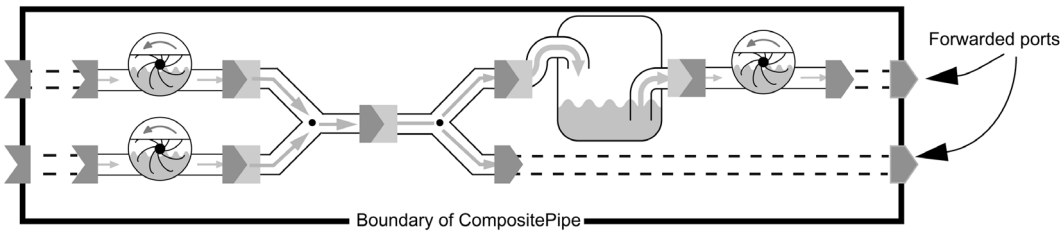


Fig. 5. Internal structure of a CompositePipe

We use a notation reminiscent of a functional type signature. Thus, a buffer, which has a negative Inport and a negative Outport, has a polarity signature $- \rightarrow -$, while a pump, which has two positive ports, has signature $+ \rightarrow +$.

Whereas buffers seem to be inherently negative and pumps inherently positive, some components can be modelled equally well with either polarity. For example, consider the function of a defragmenter that combines a pair of information items into a single item. Such functionality could be packaged as a $- \rightarrow +$ Infopipe, which accepts a sequence of two items pushed into its Inport and pushes a single item from its Outport. However, the same functionality could also be packaged as a $+ \rightarrow -$ Infopipe, which pulls two items into its Inport and replies to a pull request on its Outport with the combined item.

Rather than having two distinct Infopipes with the same functionality but opposite polarities, it is convenient to combine both into a single component, to which we assign the polarity signature $\alpha \rightarrow \bar{\alpha}$. This should be read like a type signature for a polymorphic function, with an implicit universal quantifier introducing the variable α. It means that the ports must have opposite polarities. For example, if a filter with signature $\alpha \rightarrow \bar{\alpha}$ is connected to the Outport of a pump with signature $+ \rightarrow +$, the α would be instantiated as $-$ and the $\bar{\alpha}$ as $+$, and hence the filter would acquire the induced polarity $- \rightarrow +$.

The polarity-checking algorithm that we have implemented is very similar to the usual polymorphic type checking and inference algorithm used for programming languages (Cardelli 1987). The main extension is the addition of a *negation* operation.

3.2 Ensuring information flow

Polarity correctness is a necessary condition for information to flow through a pipeline. For example, if two buffers (both with signature $- \rightarrow -$) were directly connected, it would never be possible for information to flow from the first to the second. The polarity check prohibits this. In contrast, a pipeline that contains a pump (with signature $+ \rightarrow +$) between the two buffers will pass the polarity check and will also permit information to pass from the first buffer to the second.

However, polarity correctness is not by itself sufficient to guarantee timely information flow. In studying these issues, it is useful to think of an Infopipeline as an energy-flow system. Initially, energy comes from pumps and other components with only positive ports, such as positive sources. Eventually, energy will be dissipated in buffers and sinks.

Components such as broadcast tees, which have signature $-\begin{array}{l}\nearrow+\\ \searrow+\end{array}$, can be thought of as amplifying the energy in the infor-

mation flow, since every information item pushed into the tee causes two items to be pushed out. However, a switching tee, which redirects its input to one or other of its two Outports, is not an amplifier, even though it has the same polarity signature. This can be seen by examining the flows quantitatively. If the input flow has bandwidth b items per second, aggregate output from the broadcast tee is $2b$ items per second, whereas from the switching tee it is b items per second. Similar arguments can be made for droppers and tees that aggregate information; they can be thought of as energy attenuators.

From these considerations we can see that the "energy" flow in a pipeline cannot be ascertained by inspection of the polarities of the components alone. It is also necessary to examine the quantitative properties of the flow through the pipeline, such as information flow rates.

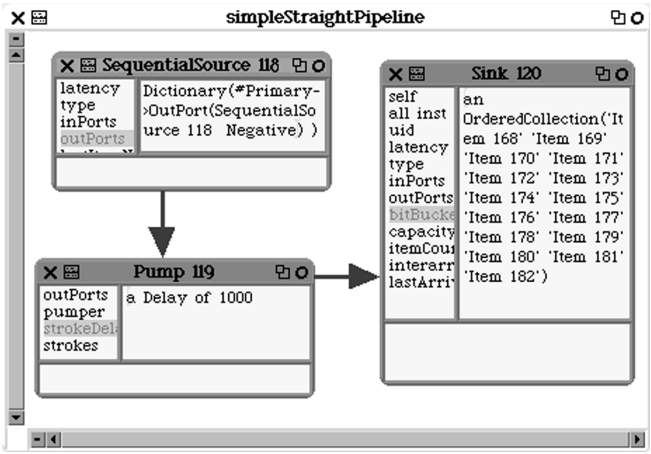


Fig. 6. Inspecting a simple straight pipeline

3.3 Buffering, capacity and cycles

So far, our discussions have focused on linear pipelines and branching pipelines without cycles. However, we do not wish to eliminate the possibility of cyclic pipelines, where outputs are "recycled" to become inputs. Examples in which cycles may be useful include implementation of chained block ciphers, samplers, and forward error correction.

It appears that a sufficient condition to avoid deadlock and infinite recursion in a cycle is to require that any cycle contains at least one buffer. This condition can easily be ensured by a configuration-checking tool. The polarity check will then also ensure that the cycle contains a pump. However, this rule may not be a necessary condition for all possible implementations of pipeline components, and it remains to be seen if it will disallow pipeline configurations that are useful and would in fact function correctly.

Three other properties that one might like to ensure in a pipeline are (1) that no information items are lost, unless explicitly dropped by a component, (2) that the flow of information does not block, and (3) that no component uses unbounded resources. However, although it may be possible to prove all of these properties for certain flows with known rate and bandwidth, in general it is impossible to maintain all three. This is because a source of unbounded bandwidth can overwhelm whatever Infopipes we assemble to deal with the flow – unless we allow them unbounded resources. We are investigating the use of queuing theory models to do quick checks on pipeline capacity.

3.4 The Infopipe configuration language

We have prototyped a textual pipeline configuration language by providing Infopipe components with appropriate operators in the Smalltalk implementation. This can be viewed as an implementation of a domain-specific language for pipeline construction by means of a shallow embedding in a host language.

The most important operator for pipeline construction is –»», which, as mentioned in Sect. 2.3, is understood by both Infopipes and ports. This enables simple straight-line Infopipes to be built with one line of text, such as SequentialSource new –» (p := Pump new) –» Sink new. The ability to name the Inports and Outports of an Infopipe explicitly permits us to construct arbitrary topologies with only slightly less convenience, as has already been illustrated in Fig. 4.

Using an existing programming language as a host provides us with a number of benefits, including the use of host language variables to refer to Infopipes, such as p in the above example. Because Smalltalk is interactive, the Infopipe programmer can not only start the pipeline (by issuing the control invocation p startPumping) but can also debug it using host language facilities. For example, p inspectPipleine will open a window (shown in Fig. 6) that allows the programmer to examine and change the state of any of the Infopipes in the pipeline.

4 Implementation issues

4.1 Threads and pipes

One of the trickiest issues in implementing Infopipes is the allocation of threads to a pipeline. Port polarity in the Infopipe abstraction has a relationship to threading, but the relationship is not as simple as it may at first appear.

A component that is implemented with a thread is said to be active. Clearly, a pump is active. In fact, any component that has only positive ports must be active, for there is no other way in which it can acquire a thread to make invocations on other objects.

A very straightforward way of implementing a pump with a frequency f Hz is to generate a new thread every $1/f$ seconds, and to have each such thread execute the code

```
outport push: (inport pull)
```

exactly once. The objection to this approach is that it may generate many threads unnecessarily, and thread creation is often an expensive activity. Moreover, because it is possible for many threads to be active simultaneously, every connected component must behave correctly in the presence of concurrency. In essence, this implementation gives each information item its own thread, and may thus have good cache locality.

An alternative approach is to give the pump a single thread, and to have that thread execute

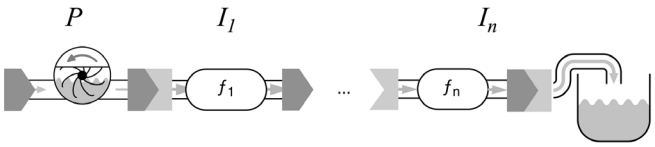


Fig. 7. A pump drives a series of transformation Infopipes

```
outport push: (inport pull).
strokeDelay wait
```

repeatedly, where strokeDelay wait suspends the caller for the appropriate inter-stroke interval. However, with the usual synchronous interpretation for method invocation, the pump has no idea how long it will take to execute outport push: or inport pull. Thus, it cannot know what delay is appropriate: the value of the delay is a property of the pipeline as a whole, not a local property of the pump.

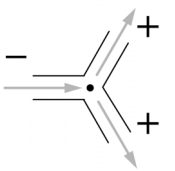
From the perspective of a particular component making a synchronous invocation, the time that elapses between invoking push: or pull on an adjacent component and the return of that invocation is an interval in which it has "loaned" its thread to others in the pipeline; we call this interval the *thread latency*. Note that thread latency, like the thread itself, is an implementation-level concept, and is quite distinct from information latency, the time taken for an information item to pass through a component. Thread latency can be reduced by adding additional threads, provided that the CPU scheduler is willing and able to make additional CPU time available. Information latency is harder to reduce!

Consider a number of passive components $I_1, I_2, \ldots I_n$, that are connected in series. Suppose that each I_i has polarity $- \rightarrow +$, and that it performs some transformation on the information that is pushed into it that takes time t_i. The transformed information item is then pushed into component I_{i+1}. If all of the push messages are synchronous, the time that elapses between invoking push: on I_1 and receiving the reply is given by

$$t_{\text{total}} = \sum_{i=1}^{n} t_i \ .$$

Now suppose that a pump P with frequency f is connected to the Inport of I_1, as shown in Fig. 7. If the required interval between strokes of the pump, $t_p = 1/f$, is less than t_{total}, then the single-threaded version of the pump will be unable to maintain the specified frequency. It will be necessary to use multiple threads in the pump, and other components in the pipeline will need to incorporate the appropriate synchronization code to deal correctly with this concurrency. (The pump may also need to use multiple CPUs; this depends on the proportion of t_{total} during which the processor is actually busy. If some of the I_n access external devices, t_n may be much greater than the CPU time used by I_n.)

Thread latency is an important parameter not only for pumps but also for other Infopipes. Consider a broadcast Tee that accepts an information item at its negative Inport and replicates it at two or more positive Outports. This can be implemented with two threads,



which will give the Tee's push: method the lowest thread latency, zero threads, in which case the pushes that the Tee performs on its downstream neighbours will be serialized, or one thread, which can be used either to provide concurrency at the Outports, or to reduce the Tee's thread latency at its Inport.

It should now be clear that allocating the right number of threads to a pipeline is not an easy problem. If there are too few, the pipeline may not satisfy its rate specification; if there are too many, we may squander resources in unnecessary bookkeeping and synchronization. Application programmers are relieved of the task of thread allocation by working with pumps and similar high-level abstractions and dealing instead with application domain concepts such as stroke frequency. But this leaves the Infopipe implementation the responsibility to perform thread allocation.

We have considered two approaches. The first, which we have prototyped, is entirely dynamic. Pump uses a timer to wake up after the desired stroke interval. It keeps a stack of spare threads; if a thread is available, it is used to execute the stroke. If no thread is available, a new thread is created. Once the thread has completed the stroke, it adds itself to the stack (or deletes itself if the stack is full).

The second approach, which we have not yet implemented, analyses the pipeline before information starts to flow. The components adjacent to the pump are asked for their thread latencies, the total thread latency for pull and push is computed. If this is less than t_p, we know that a single thread should be sufficient, and simpler single-threaded pipeline components can be utilized.

4.2 Creating polymorphic Infopipes

A polymorphic Infopipe must have methods for both pull and push:, and the behaviour of these methods should be coherent, in the sense that the transformation that the Infopipe performs on the information, if any, should be the same in each case.

Although polymorphic Infopipes are clearly more useful than their monomorphic instances, it is not in general a simple matter to create push: and pull methods with the required correspondence. Figure 8 shows sample code for a defragmenter. We assume that the component has a method assemble: i1 and: i2 that returns the composite item built from input fragments i1 and i2. The pull method, which implements the $+ \rightarrow -$ functionality, and the push: method, which implements the $- \rightarrow +$ functionality, both use the assemble:and: method, but, even so, it is not clear how to verify that pull and push: both do the same thing.

Indeed, a third implementation style is possible, providing the $+ \rightarrow +$ polarity; this is shown in Fig. 9. This defragmenter understands neither pull nor push:, but instead has an internal thread that repeatedly executes stroke.

It is clearly undesirable to have to write multiple forms of the same code, particularly when there must be semantic coherence between them. We can avoid this in various ways.

- Most simply, we can eliminate polymorphic pipes completely. In this situation, the defragmenter would be written with whatever polarity is simplest, probably $+ \rightarrow -$. If a different polarity is required, this would be constructed as a composition of more primitive Infopipes. For exam-

```
Defragmenter ›› pull
    | item1 item2 |
    item1 := inport pull.
    item2 := inport pull.
    ↑ self assemble: item1 and: item2.
```

```
Defragmenter ›› push: item
    isFirst
        ifTrue:  [buffer := item]
        ifFalse: [outport push: (self assemble: buffer and: item) ].
    isFirst := isFirst not.
```

Fig. 8. Methods of a polymorphic defragmenter

```
Defragmenter ›› stroke
    | item1 item2 |
    item1 := inport pull.
    item2 := inport pull.
    outport put: (self assemble: item1 and: item2)
```

```
Defragmenter ›› startPumping: period
    self strokeInterval: period.
    [[ self stroke
        strokeDelay wait] repeatForever] fork
```

Fig. 9. Defragmenter in the +→+ style

ple, a buffer, a $+ \rightarrow -$ defragmenter and a pump could be composed to create a $- \rightarrow +$ defragmenter.

- The second approach is to use a layer of middleware to "wrap" whichever method is most easily written by hand, in order to generate the other methods. This is possible because the hand-written methods do not send messages to the adjacent components directly, but instead use a level of indirection. For example, the defragmenter pull method sends pull to its own inport rather than pull to the upstream component. A clever implementation of inport pull can actually wait for the upstream component to send a push: message. We have explored this solution in some depth (Koster et al. 2001b); whenever adjacent Infopipes do not need to be scheduled independently of each other, they are run as coroutines in the same thread, thus avoiding scheduling overhead.
- A third possibility is to automatically transform the source code, so that one version would be written by hand and the others generated automatically. Even with the simple example shown in Fig. 8, this seems to be very hard; in the general case, we do not believe that it is feasible. The difficulty is that the transformation engine would need to "understand" all of the complexity of a general-purpose programming language like Smalltalk or C++.
- It is possible that this objection could be overcome by using a domain-specific source language, with higher-level semantics and more limited expressiveness. From a more abstract form of the method written in such a language, it might be possible to generate executable code in whatever form is required. This approach is currently under investigation.

4.3 Netpipes

Netpipes implement network information flows using whatever mechanisms are appropriate to the underlying medium and the application. For example, we have built a low-latency, unreliable Netpipe using UDP.

A Netpipe has the same polarity and data interface as a buffer; this models the existence of buffering in the network and in the receiving socket. However, the control interface of a Netpipe is different, since it reflects the properties of the underlying network. For example, the latency of a Netpipe depends on the latency of its network connection and the capacity of its buffer.

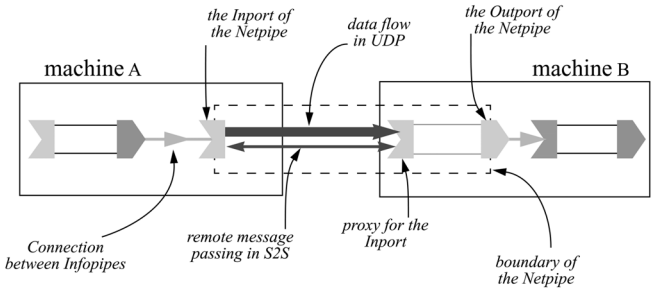


Fig. 10. Working with a Netpipe

The motivation for Netpipes is to allow Infopipe middleware components in two different address spaces to connect to each other. It is certainly true that an existing distributed computing platform, such as remote method invocation or remote procedure call, would allow such connections. However, if we used such a platform, we could be hiding the communication between the address spaces, and thus giving up any ability to control it – which was the reason that we originally created Infopipes. We would also be violating the seamless connection property described in Sect. 2.1.

However, it is not necessary to reimplement an entire distributed computing environment in order to retain control over the information flow in a Netpipe. Instead, we have bootstrapped the Netpipe implementation by using the features of an existing environment, such as naming and remote method invocation.

A Netpipe is an Infopipe with an Inport in one address space and an Outport in another, as shown in Fig. 10. This means that the Inport can be in the same address space as its upstream neighbour, and thus invocations of push: can use seamless local mechanisms for method invocation. Similarly, the Outport is in the same address space as its downstream neighbour, which can seamlessly invoke pull on the Netpipe. The Netpipe object itself, containing the buffering and the code, is co-located with the Outport.

Our prototyping environment, Squeak Smalltalk (Guzdial 2001; Squeak 2000), is equipped with a remote method invocation package called S2S, which stands for "Squeak to Squeak".

```
source := 's2s://MusicStore/source1' asRemoteObject.
pump:= 's2s://MusicStore/pump1' asRemoteObject.
netPipe := Netpipe from: 's2s://MusicStore/'.
sink := MIDIPlayer new.
source –>> pump –>> netPipe –>> sink.
monitor := Monitor monitored: netPipe controlled: pump.
pump startPumping: 100.
monitor startMonitoring: 1000.
sink startPlaying.
```

Fig. 11. Code for a streaming MIDI pipeline

S2S provides access transparency and location transparency in a similar way to CORBA and Java RMI. A local proxy can be created for a remote object; the proxy can then be invoked without callers needing to be aware that they are really using a remote object – except that the call is several orders of magnitude slower.

We take care that we use S2S only to configure and name Infopipe components, and *not* for transmitting information through the pipeline. For example, a Netpipe uses S2S to create an Inport on the remote machine, and an S2S proxy for this Inport is stored in the Netpipe. However, when the Inport needs to push information into the Netpipe, it uses a custom protocol implemented directly on UDP.

In this way, we arrange that Infopipes exhibit *access transparency*: the same protocol is used to establish local and remote Infopipe connections. However, we choose not to provide *location transparency*: connections between adjacent Infopipes must be local, and the –>> method checks explicitly that the ports that it is about to connect are co-located. Without this check, ports in different address spaces could be connected directly: information would still flow through the pipeline, but the push: or pull of each item would require a remote method invocation. As well as being very much less efficient, this would mean that the application would have no control over network communication.

Instead of signalling an error in the face of an attempt to connect non-co-located ports, an alternative solution would be to introduce a Netpipe automatically. We have not pursued this alternative, because in practice it is usually important for the programmer to be aware of the use of the network. For example, it may be necessary to include Infopipe components to monitor the available QoS and adapt the information flow over the Netpipe accordingly.

Figure 11 shows the code for setting up a MIDI pipeline using a Netpipe. The first two statements obtain S2S proxies for source and pump objects that already exist on a remote machine called MusicStore. We will refer to these remote objects as s and p, The third statement builds a Netpipe from MusicStore to the local machine. The fifth statement, source –>> pump –>> . . . , constructs the pipeline. It is interesting to see in detail how this is accomplished.

The invocation –>> is sent to source, which is a local proxy for remote object s. S2S translates this into a remote method invocation on the real object s on MusicStore. Moreover, because the argument, pump, is a proxy for p, and p is co-located with s, S2S will present p (rather than a proxy for pump) as the argument to the invocation. The method for –>> will then execute locally to both s and p, creating a connection with no residual dependencies on the machine that built the pipeline.

A similar thing happens with netPipe. Although netPipe itself is local, its Inport is on the host MusicStore. Thus, the connection between p and netPipe's Inport is also on MusicStore. Information transmitted between netPipe's Inport and OutPort does of course traverse the network, but it does *not* use S2S; it uses a customized transport that is fully encapsulated in and controlled by netPipe.

4.4 Smart proxies

When information flows from one address space to another, it is necessary not only to agree on the form that the information flow should take but also to install the Infopipe components necessary to construct that flow. For example, a monitoring application that produces a video stream from a camera should be able to access a logging service that records a video in a file as well as a surveillance service that scans a video stream for suspicious activity. In the case where the video is sent to a file, the file sink and the camera might be on the same machine, and the communication between them might be implemented by a shared-memory pipe. In the case of the surveillance application, the communication could involve a Netpipe with compression, encryption and feedback mechanisms over the Internet.

Notice that the Infopipe components that need to be co-located with the camera are different in these two cases. Since we wish to allow Infopipes to be dynamically established, we must address the problem of how such components are to be installed and configured.

Koster and Kramp proposed to solve this problem in a client–server environment by using dynamically loadable smart proxies (Koster and Kramp 2000). Their idea is that the server functionality is partly implemented on the client node; this enables the server to control the network part of the pipeline, as shown in Fig. 12. At connection setup, the server chooses a communication mechanism based on information about the available resources. A video server, for instance, could use shared memory if it happens to be on the same node as the client, a compression mechanism and UDP across the Internet, or raw Ethernet on a dedicated LAN. It then transmits the code for a smart proxy to the address space containing the client. In this way, application-specific remote communication can be used without making the network protocol the actual service interface; that would be undesirable because all client applications would have to implement all protocols used by any server to which they may ever connect. Smart proxies enable the service interface to be described at a high level using an IDL. Client applications can be programmed to this

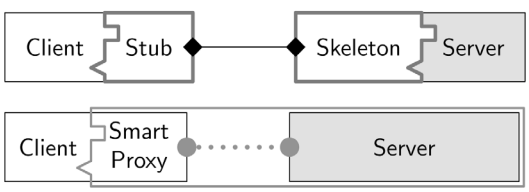


Fig. 12. Smart proxies

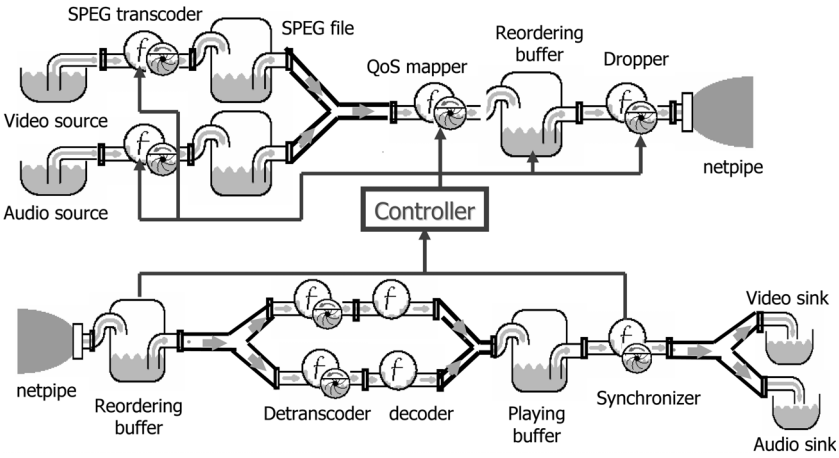


Fig. 13. The Quasar video pipeline

interface as if the server were local, although they actually communicate with the proxy.

The idea of smart proxies can be generalized and applied to Infopipes (Koster et al. 2001a). Since there are high-level interfaces between all elements of a pipeline, the granularity of composition can be finer. It is not necessary to send to the consumer address space a monolithic proxy implementing every transformation that needs to be performed on the information stream. Instead, it may be possible to compose the required transformation from standard pipeline elements that may already be available on the consumer side. Thus, it is sufficient to send a description or *blueprint* of the required proxy, and if necessary to send small specialized Infopipes that implement those pieces of the pipeline that are not already available.

5 Some example Infopipelines

5.1 The Quasar video pipeline

The Quasar video pipeline is a player for adaptive MPEG video streaming over TCP. It supports QoS adaptation in both temporal and spatial dimensions. MPEG-1 video is transcoded into SPEG (Krasic and Walpole 1999) to add spatial scalability through layered quantization of DCT data. To suit the features of TCP, MPEG video is delivered in a *priority-progress stream* (Krasic et al. 2001), which is a sequence of packets, each with a timestamp and a priority. The timestamps expose the timeliness requirements of the stream, and allow progress to be monitored; the priorities allow informed dropping in times of resource overload.

The Quasar video pipeline is shown as an Infopipeline in Fig. 13. At the producer side (the top part of the figure), the video frames first flow through an SPEG transcoding filter, and are buffered. The QoS mapper pulls them from the buffer and gives each packet a priority according to the importance of the packet and the preference of the user. For example, the user might be more concerned with spatial resolution than with frame rate, or vice versa. A group of prioritised packets are pushed in priority order into the reordering buffer. The dropper is a filter that discards stale packets, and low priority packets, when the network is unable to deliver them in time.

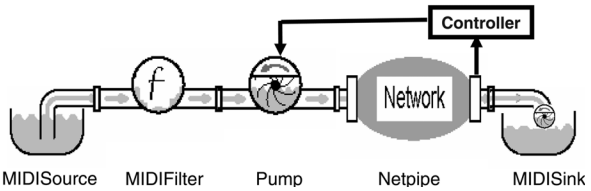


Fig. 14. The MIDI pipeline

The producer and consumer pipelines are connected by a TCP Netpipe. On the consumer side (at the bottom of the figure) the reordering buffer arranges packets in time order. The detranscoder and decoder are filters that convert the packets to MPEG and then to image format, after which they are pushed into the playing buffer. The synchronizer pulls them from that buffer at the time that they are required, and presents them to the video sink. The audio stream is handled in a similar way; the two streams are merged and split using Tees. The controller coordinates the rates of all the components through control interfaces.

5.2 The MIDI pipeline

The MIDI pipeline (see Fig. 14) was built using some existing libraries from the Squeak Smalltalk system. The Squeak MIDI player buffered an entire MIDI file before playing it. We adapted this player to deal with streaming data and wrapped it as three Infopipe components: the MIDISource, the MIDIFilter, and the MIDISink.

The MIDISource reads "note-on" and "note-off" commands from a MIDI file; the MIDIFilter combines a note-on command and its corresponding note-off command to generate a note event, which consists of a key and its duration. The MIDISink plays a stream of note events. To make the MIDI player stream over the Internet, we needed only to insert two pre-existing Infopipe components: a pump and a UDP Netpipe. To ensure that the MIDISink plays smoothly, we added a controller that monitors the fill level of the Netpipe and adjusts the pumping rate accordingly. In this prototype, the connections to the controller were not implemented using Infopipes; whether they should be is an open question. Instead,

we used direct method invocation, which means that invocations from the controller to the pump used S2S, which is very much slower than an Infopipe. We found that a buffer sufficient to hold 30 note events produced smooth playout while still minimizing the number of control messages.

6 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 (OMG 1998a) defines stream management interfaces, but not the data transmission. Only extensions to CORBA such as TAO's pluggable protocol framework allow the use of different transport protocols and, hence, the efficient implementation of audio and video applications (Mungee et al. 1999). Asynchronous messaging (OMG 2001a) and event channels (OMG 2001b) allow evading the synchronous RMI-based interaction and introduce the concurrency needed in an information pipeline. Finally, Real-time CORBA (OMG 2001a; Schmidt and Kuhns 2000), adds priority-based mechanisms to support predictable service quality end to end. As extensions of an RMI-based architecture, these mechanisms facilitate the integration of streams into a distributed object system. Infopipes, however, provide a high-level interface tailored to information flows and more flexibility in controlling concurrency and pipeline setup.

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 (see, for example, Lea 1997). Flow-based programming applies this concept to the development of business applications (Morrison 1994). 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.

QoSDREAM uses a two-layer representation to construct multimedia applications (Naguib and Coulouris 2001). On the model layer, the programmer builds the application by combining abstract components and specifying their QoS properties. The system then maps this description to the active layer consisting of the actual executable components. The setup procedure includes integrity checks and admission tests. The active-layer representation may be more fine-grained than the model specification, introducing additional components such as filters, if needed. In this way, the system supports partially automatic configuration. While the current Infopipe implementation provides less sophisticated QoS control, it provides a better model of flow properties by explicitly using pumps and buffers.

Blair and co-workers have proposed an open architecture for next-generation middleware (Blair et al. 1998; Eliassen et al. 1999). They present an elegant way to support open engineering and adaptation using reflection, a technique borrowed from the field of programming languages (Blair and Coulson 1998). In their multimedia middleware system, TOAST (Eliassen et al. 2000; Fitzpatrick et al. 2001), they reify communication through open bindings, which are similar to our remote pipes. The scope of this work is wider than that of Infopipes, which are specialized for streaming applications.

The MULTE middleware project also features open bindings (Eliassen et al. 2000; Plagemann et al. 2000) and supports flexible QoS (Kristensen and Plagemann 2000). It provides applications with several ways to specify QoS using a mapping or negotiation in advance to translate among different levels of QoS specification. In our approach, we typically use dynamic monitoring and adaptation of QoS at the application-level to implicitly manage resource-level QoS.

Ensemble (van Renesse et al. 1997) and Da CaPo (Vogt et al. 1993) are protocol frameworks that support the composition and reconfiguration of protocol stacks from modules. Both provide mechanisms to check the usability of configurations and automatically configure the stacks. Unlike these frameworks for local protocols, Infopipes use a uniform abstraction for handling information flows from source to sink, possibly across several network nodes; the Infopipe setup is controlled by the application. A similarity is that both allow for dynamic configuration: protocol frameworks dynamically (re)configure protocol stacks between a network interface and an application interface, while smart proxies dynamically construct part of an Infopipeline between a client-side service interface and a remote server, providing protocol-independent service access.

The Scout operating system (Mosberger and Peterson 1996) combines 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.

7 Summary and future work

Infopipes are a subject of continuing research; the work described here does not pretend to be complete, although early results have been encouraging. The applications that have driven the work described here have primarily been streaming video and audio. However, Infopipes also form part of the communications infrastructure of the Infosphere project (Liu et al. 2000; Pu et al. 2001), and we intend that Infopipes are also useful for applications such as environmental observation and forecasting (Steere et al. 2000) and continual queries (Liu et al. 1999).

We have been pursuing three threads of research simultaneously. The first, which pre-dates the development of Infopipes themselves, is the design and implementation of a series of video players that stream video over the Internet, adapting their behaviour to make the best possible use of the available bandwidth (Cen et al. 1995; Cowan et al. 1995; Inouye et al. 1997; Koster 1996; Krasic and Walpole 2001; Staehli et al. 1995). The second thread is related to the underlying technologies that support streaming media, in particular, adaptive

and rate-sensitive resource scheduling (Li et al. 2000; Steere et al. 1999a; Steere et al. 1999b) and congestion control (Cen et al. 1998; Li et al. 2001a; Li et al. 2001b). It is these technologies that enable us to design and build the Infopipes that are necessary for interesting applications.

The final thread is a prototyping effort that has explored possible interfaces for Infopipes in an object-oriented setting. We have used Squeak Smalltalk as a research vehicle; this has been a very productive choice, as it enabled us to quickly try out – and discard – many alternative interfaces for Infopipes before settling on those described here. The Squeak implementation is not real-time, but it is quite adequate for the streaming MIDI application (Sect. 5.2).

We are currently embarked on the next stage of this research, which involves weaving these threads together into a fabric that will provide a new set of abstractions for streaming applications. We are in the process of using the Infopipe abstractions described here to reimplement our video pipelines on a range of platforms including desktop, laptop and wireless handheld computers as well as a mobile robot. We are also exploring kernel-level support for Infopipes under Linux, with a view to providing more precise timing control and an application-friendly interface for timing-sensitive communication and device I/O.

Acknowledgements. This work was partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs, by NSF Grants CCR-9988440 and CCR-0219686, by the Murdock Trust, and by Intel. We thank Paul McKenney for useful discussions and Nathanael Schärli for help with the Squeak graphics code.

References

Blair GS, Coulson G (1998) The case for reflective middleware. Internal report MPG-98-38, Distributed Multimedia Research Group, Department of Computing, Lancaster University, Lancaster, UK

Blair GS, Coulson G, Robin P, Papathomas M (1998) An architecture for next generation middleware. In: Davies N, Raymond K, Seitz J (eds) IFIP international conference on distributed systems platforms and open distributed processing (Middleware '98), Lake District, UK. Springer, Berlin Heidelberg New York

Cardelli L (1987) Basic polymorphic typechecking. Sci Comput Program 8(2):147–172

Cen S, Pu C, Staehli R, Cowan C, Walpole J (1995) A distributed real-time MPEG video audio player. In: Fifth international workshop on network and operating system support of digital audio and video (NOSSDAV'95), Durham, N.H. Lecture notes in computer science, vol 1018. Springer, Berlin Heidelberg New York

Cen S, Pu C, Walpole J (1998) Flow and congestion control for Internet streaming applications. In: Multimedia computing and networking 1998, San Jose, Calif., 26–28 January 1998. SPIE, Bellingham, Wash.

Chesire M, Wolman A, Voelker GM, Levy HM (2001) Measurement and analysis of a streaming media workload. In: Proceedings of the 3rd USENIX symposium on internet technologies and systems (USITS'01), San Francisco, Calif. USENIX Association, Berkerley, Calif.

Cowan C, Cen S, Walpole J, Pu C (1995) Adaptive methods for distributed video presentation. ACM Comput Surv 27(4):580–583

Eliassen F, Andersen A, Blair GS, Costa F, Coulson G, Goebel V, Hansen Ø, Kristensen T, Plagemann T, Rafaelsen HO, Saikoski KB, Yu W (1999) Next generation middleware: requirements, architecture, and prototypes. In: Proceedings of the 7th workshop on future trends of distributed computing systems (FTDCS'99), Cape Town, South-Africa. IEEE Press, Los Alamitis, Calif.

Eliassen F, Kristensen T, Plagemann T, Raffaelsen HO (2000) MULTE-ORB: adaptive QoS aware binding. In: RM 2000, workshop on reflective middleware, New York, 7–8 April 2000

Fitzpatrick T, Gallop J, Blair G, Cooper C, Coulson G, Duce D, Johnson I (2001) Design and application of TOAST: an adaptive distributed multimedia middleware platform. In: Interactive distributed multimedia systems (IDMS 2001), Lancaster, UK. Lecture notes in computer science vol 2158. Springer, Berlin Heidelberg New York

Guzdial M (2001) Squeak: object-oriented design with multimedia applications. Prentice Hall, Upper Saddle River, N.J.

Inouye J, Cen S, Pu C, Walpole J (1997) System support for mobile multimedia applications. In: Proceedings of the 7th international workshop on network and operating systems support for digital audio and video, St. Louis, Mo. IEEE, Piscataway, N.J.

ISO (1998) Information technology: open distributed processing. ISO Standard ISO/IEC 10746, International Standards Organization

Jacobs S, Eleftheriadis A (1998) Streaming video using dynamic rate shaping and TCP flow control. J Vis Commun Image Represent 9(3):211–222

Karr DA, Rodrigues C, Loyall JP, Schantz RE, Krishnamurthy Y, Pyarali I, Schmidt DC (2001) Application of the QuO quality-of-service framework to a distributed video application. In: 3rd international symposium on distributed objects and applications, Rome. IEEE Press, Los Alamitos, Calif.

Koster R.(1996) Design of a multmedia player with advanced QoS control. MS thesis, Oregon Graduate Institute of Science and Technology, Beaverton, Ore.

Koster R, Black AP, Huang J, Walpole J, Pu C (2001a) Infopipes for composing distributed information flows. In: International workshop on multimedia middleware. ACM Press, New York

Koster R, Black AP, Huang J, Walpole J, Pu C (2001b) Thread transparency in information flow middleware. In: Guerraoui R (ed) Middleware 2001, IFIP/ACM international conference on distributed systems platforms, Heidelberg, Germany. Lecture notes in computer science, vol 2218. Springer, Berlin Heidelberg New York

Koster R, Kramp T (2000) Structuring QoS-supporting services with smart proxies. In: Second international conference on distributed systems platforms and open distributed processing (Middleware 2000). Lecture notes in computer science, vol 1795. Springer, Berlin Heidelberg New York

Krasic B, Walpole J (2001) Priority-progress streaming for quality-adaptive multimedia. ACM Multimedia Doctoral Symposium, Ottawa, Canada

Krasic C, Li K, Walpole J (2001) The case for streaming multimedia with TCP. In: Interactive distributed multimedia systems, 8th international workshop, IDMS 2001, Lancaster, UK, 2–7 Sept 2001. Lecture notes in computer science, vol 2158. Springer, Berlin Heidelberg New York

Krasic C, Walpole J (1999) QoS scalability for streamed media delivery. Technical report CSE-99-11, Department of Computer Science and Engineering, Oregon Graduate Institute, Beaverton, Ore.

Kristensen T, Plagemann T (2000) Enabling flexible QoS support in the object request broker COOL. IEEE ICDCS International Workshop on Distributed Real-Time Systems (IWDRS 2000), Taipei, 10–13 April 2000. IEEE Press, Los Alamitos, Calif.

Lea D (1997) Concurrent programming in Java. Addison-Wesley, Reading, Mass.

Li K, Krasic C, Walpole J, Shor M, Pu C (2001a) The minimal buffering requirements of congestion controlled interactive multimedia applications. In: Interactive distributed multimedia systems, 8th international workshop, IDMS 2001, Lancaster, UK, 2–7 Sept 2001. Lecture notes in computer science, vol 2158. Springer, Berlin Heidelberg New York

Li K, Shor M, Walpole J, Pu C, Steere D (2001b) Modeling the effect of short-term rate variations on TCP-friendly congestion control behavior. In: American Control Conference, Arlington, Va., 25–27 June 2001. American Automatic Control Council, New York

Li K, Walpole J, McNamee D, Pu C, Steere DC (2000) A rate-matching packet scheduler for real-rate applications. In: Multimedia computing and networking 2000, San Jose, Calif., 24–26 January 2000. SPIE, Bellingham, Wash.

Liu L, Pu C, Schwan K, Walpole J (2000). InfoFilter: supporting quality of service for fresh information delivery. New Generation Comput J 18(4):305–321

Liu L, Pu C, Tang W (1999). Continual queries for internet scale event-driven information delivery. IEEE Trans Knowl Data Eng 11(4):610–628

McCanne S, Vetterli M, Jacobson V (1997). Low-complexity video coding for receiver-driven layered multicast. IEEE J Sel Areas Commun 16(6):983–1001

Morrison JP (1994) Flow-based programming: a new approach to application development. Van Nostrand Reinhold, New York

Mosberger D, Peterson LL (1996) Making paths explicit in the Scout operating system. In: Petersen K, Zwaenepoel W (eds) Proceedings of the second USENIX symposium on operating systems design and implementation. ACM Press, New York

Mungee S, Surendran N, Krishnamurthy Y, Schmidt DC (1999) The design and performance of a CORBA audio/video streaming service. In: Thirty-second Hawaiian international conference on system sciences, Maui, Hawaii, 3–6 January 1999. IEEE Press, Los Alamitos, Calif.

Naguib H, Coulouris G (2001) Towards automatically configurable multimedia applications. In: International workshop on multimedia middleware, Ottawa, 5 October 2001. ACM Press, New York

OMG (1998a) CORBA telecoms specification. Object Management Group, Framingham, Mass. http://www.omg.org/cgi-bin/doc?formal/98-07-12. Cited 16 September 2002

OMG (1998b) CORBA/IIOP 2.3 Specification. Object Management Group, Framingham, Mass. http://www.omg.org/cgi-bib/formal/98-12-01. Cited 16 September 2002

OMG (2001a) The Common Object Request Broker: architecture and specification. Object Management Group, Framingham, Mass. http://www.omg.org/cgi-bin/doc?formal/01-09-34. Cited 16 September 2002

OMG (2001b) Event service specification. Object Management Group, Framingham, Mass. http://www.omg.org/cgi-bin/doc?formal/01-03-01. Cited 16 September 2002

OSF (1991) Remote procedure call in a distributed computing environment: a white paper. Open Software Foundation

Plagemann T, Eliassen F, Hafskjold B, Kristensen T, Macdonald RH, Rafaelsen HO (2000) Managing cross-cutting QoS issues in MULTE middleware. In: Elisa Bertino (ed) ECOOP 2000, object-oriented programming: Proceedings of the 14th European conference, Sophia Antipolis and Cannes, France, 12-16 June 2000. Springer, Berlin Heidelberg New York

Pu C, Schwan K, Walpole J (2001). Infosphere project: system support for information flow applications. ACM SIGMOD Rec 30(1):25–34

Schmidt DC, Kuhns F (2000). An overview of the real-time CORBA specification. IEEE Comput 33(6):56–63

Squeak (2000) Squeak. http://www.squeak.org/. Cited 15 September 2002

Staehli R, Walpole J, Maier D (1995). Quality of service specification for multimedia presentations. Multimedia Syst 3(5/6):251–263

Steere D, Baptista A, McNamee D, Pu C, Walpole J (2000) Research Challenges in Environmental Observation and Forecasting Systems. In: Mobicom 2000, proceedings of the sixth annual international conference on mobile computing and networking, 6–11 August, 2000, Boston, Mass. ACM Press, New York

Steere DC, Goel A, Gruenberg J, McNamee D, Pu C, Walpole J (1999a) A feedback-driven proportion allocator for real-rate scheduling. In: Hand, SM (ed) Proceedings of the third symposium on operating systems design and implementation. USENIX Association, Berkeley, Calif.

Steere DC, Walpole J, Pu C (1999b) Automating proportion/period scheduling. In: 20th IEEE real-time systems symposium, Phoenix, Ariz. IEEE Press, New York

Sun (2002) Java remote method invocation specification. Java™ 2 SDK v1.4, standard edition. Sun Microsystems Corp. http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC.html. Cited 15 September 2002

Thompson K, Miller GJ, Wilder R (1997). Wide-area internet traffic patterns and characteristics. IEEE Network Mag 11(6):10–23

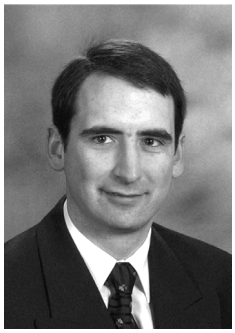
van Renesse R, Birman K, Hayden M, Vaysburd A, Karr D (1997) Building adaptive systems using Ensemble. Technical report TR97-1638, Computer Science Department, Cornell University

Vogt M, Plattner B, Plagemann T, Walter T (1993) A run-time environment for Da CaPo. In: Proceedings of INET'93, international networking conference. Internet Society, Reston, Va.

Walpole J, Koster R, Cen S, Cowan C, Maier D, McNamee D, Pu C, Steere D, Yu L (1997) A player for adaptive MPEG video streaming over the Internet. In: 26th Applied Imagery Pattern Recognition Workshop AIPR-97, Washington, D.C. SPIE, Bellingham, Wash.

Andrew P. Black holds a D.Phil in computation from the University of Oxford. At the University of Washington (1981–1986) he was part of a team that built two of the earliest distributed object-oriented systems. From 1987 until 1994 he was with the Distributed Systems Advanced Development group and the Cambridge Research Laboratory of Digital Equipment Corporation. Subsequently, he joined the faculty of the Oregon Graduate Institute as professor and head of the Computer Science Department. Since 2000 he has been pursuing his research interests in programming languages, programming methodology, and system support for distributed computing.

Jie Huang received a BS degree in computer and communications in 1992 and an MS degree in computer science in 1995, both from Beijing University of Posts and Telecommunications. She then held the post of assistant professor at the same school. Since September 1999 she has been a PhD student at the Oregon Graduate Institute, now the Oregon Health and Science University. Her interests are in software development methodology and programming languages, especially a domain-specific approach for building multimedia networking applications.



Rainer Koster received a MS in computer science and engineering in 1997 from the Oregon Graduate Institute of Science and Technology and a diploma in computer science in 1998 from the University of Kaiserslautern. He currently is a member of the Distributed Systems Group at the University of Kaiserslautern. His interests and research focus on quality-of-service support and distributed multimedia systems.

Jonathan Walpole received his PhD in computer science from Lancaster University, UK, in 1987. He worked for two years as a post-doctoral research fellow at Lancaster University before taking a faculty position at the Oregon Graduate Institute (OGI). He is now a full professor and director of the Systems Software Laboratory at the OGI School of Science and Engineering at Oregon Health and Science University. His research interests are in operating systems, distributed systems, multimedia computing, and environmental information technology.

Calton Pu received his PhD in computer science from University of Washington in 1986, and has served on the faculty of Columbia University and the Oregon Graduate Institute. He is currently a professor at the College of Computing, Georgia Institute of Technology where he occupies the John P. Imlay, Jr., Chair in Software, and is a co-director of the Center for Experimental Research in Computer Systems. Dr. Pu leads the Infosphere project, a collaboration between Georgia Tech. and OGI that is building systems support for information-driven distributed applications; his other research interests include operating systems, transaction processing and Internet data management.