10-2001

Reifying Communication at the Application Level

Andrew P. Black
Oregon Graduate Institute of Science & Technology

Jie Huang
Oregon Graduate Institute of Science & Technology

Jonathan Walpole
Oregon Graduate Institute of Science & Technology

Citation Details

"Reifying Communication at the Application Level," Andrew Black, Jie Huang and Jonathan Walpole, In Proceedings of the International Workshop on Multimedia Middleware, Ottawa, Canada, October 2001. Also availabile as OGI Technical Report CSE-01-006, June 2001.

Reifying Communication at the Application Level

Andrew P. Black *black@cse.ogi.edu*
Jie Huang *jiehuang@cse.ogi.edu*
Jonathan Walpole *walpole@cse.ogi.edu*

Department of Computer Science & Engineering
Oregon Graduate Institute of Science & Technology

A Position Paper submitted to the International Workshop on Multimedia Middleware, Ottawa, October 2001

Introduction

Middleware, from the earliest RPC systems to recent Object-Oriented Remote Message Sending (RMS) systems such as Java RMI and CORBA, claims *transparency* as one of its main attributes. Coulouris *et al.* [1] define transparency as "the concealment from the … application programmer of the separation of components in a distributed system." They go on to identify eight different kinds of transparency,

We considered titling this paper "Transparency Considered Harmful", but that title is misleading because it implies that all kinds of transparency are bad. This is not our view. Rather, we believe that the choice of *which* transparencies should be offered by a middleware platform is critically dependent on the use to be made of that platform. Specifically, we argue that network transparency and concurrency transparency are inappropriate for middleware that is designed to support multimedia applications. This is because a network that is "transparent" is a network that is hidden, and thus one whose Quality of Service aspects are also hidden.

The ability to hide communication is a great strength of RPC and RMS. Hiding the network frees the programmer to focus on the real purpose of the program, that is, executing complex application logic. But hiding can also be a great weakness: for many multimedia systems, the real purpose of the program is to *control communication*, and this requires that communication be exposed, not hidden.

Consider a multimedia player streaming live video and audio over the internet. The complexity of this application does not lie in the display of video on the screen or the playing of audio through the speaker; these problems have largely been solved and library code is available for reuse. Instead, the complexity comes from *controlling the communication* between the source and the sink, that is, in deciding what to do if packets are lost, if bandwidth is suddenly restricted, or if latency becomes highly variable. How can we address these problems if our middleware hides communication?

But exposing communication does not mean revealing all of the details of its implementation in terms of byte streams or shared memory buffers. The exposure should be at a level of abstraction appropriate to the application. For example, a video player is concerned with video frames, not streams of bytes, and so its communications operations should be in terms of frames.

This work was partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs, by NSF Grant CCR-9988440 and by Intel.

Application Level Abstractions

Our view of QoS specification and control is primarily an adaptive one, based on dynamic observation and control, as opposed to static specification of requirements and reservation of resources. This contrasts with conventional approaches to QoS in networks, such as RSVP and the IETF's Integrated Services Architecture. We believe that our abstractions can support both approaches, but in our experience building multimedia applications over the Internet, we have found that the former approach is more useful than the latter. In this adaptive view, QoS specification and enforcement are done *by the application*: applications continually monitor their own performance and make appropriate adjustments. For this to be natural, both the monitoring and the control of quality must be in application-level terms.

In essence, our approach argues for an appropriate division of concerns between application, middleware and system layers. We believe that system-level or resource-level details, such as the available (or desired) bandwidth measured in bits or packets per second, should be hidden from applications, and *not* reified as in current network QoS approaches. Instead, we reify quality in application-level terms, and provide interfaces for controlling it, also in application-level terms. These control interfaces, together with appropriate feedback control elements (*e.g.*, from the SWiFT Toolkit [2]), provide the basic abstractions and building blocks for this approach. Specifically, monitoring is done by connecting the control interfaces (which are defined in application-level terms) to an appropriate feedback controller.

InfoPipes

InfoPipes are the name we give to these application level abstractions for information flow. InfoPipes are a middleware framework in which communication is *reified* rather than hidden.

The verb reify refers to the process of taking something abstract and making it real. InfoPipes are "real" objects: they can be created, named, and manipulated at will. They can be sent messages, and will respond with

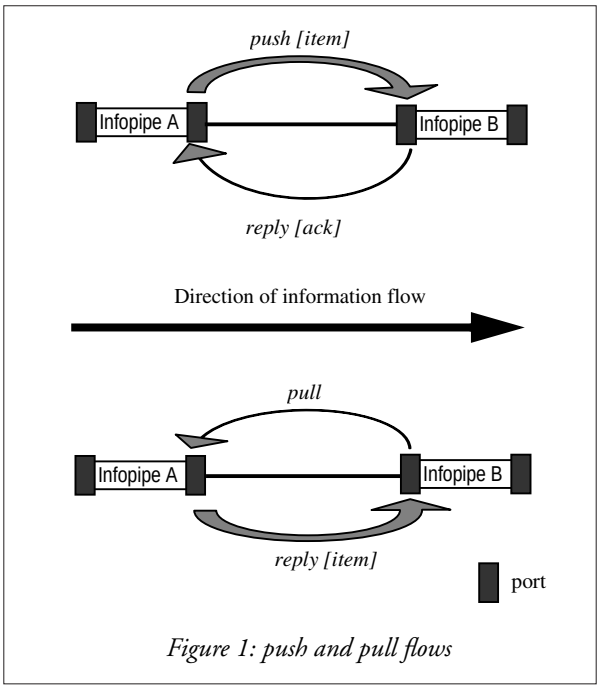

Figure 1: push and pull flows

answers. For example, we might send a message asking an InfoPipe how many frames have passed through it in a given time interval, or we might connect one InfoPipe to another by sending one the –>> message with the other as argument. InfoPipes are part of the communications infrastructure of the Infosphere project [6].

Data Interfaces

InfoPipes have been described elsewhere [3, 4], so we will limit ourselves here to a quick overview. An analogy with plumbing conveys the big picture: just as a water distribution system is built by connecting together pre-existing pipes, tees, valves and application-specific fixtures, so we envisage an information flow system being built by connecting together pre-defined and application-specific InfoPipes.

When two InfoPipes are connected, either of them may initiate the process of information transfer. So the data interface of an InfoPipe has two operations: pull and push: anItem[†]. As shown in Figure 1, an InfoPipe may send the push: message to its downstream InfoPipe, thus transmitting an information item, or may send the pull message to its upstream InfoPipe to request an information item. Our initial plan was to use a *polarity check* to ensure that only compatible components could be connected [3]. We have subsequently realized that, in many cases, given a component of one polarity, we can automatically generate a component with the same functionality but with the opposite or different polarity [5].

We intend to support application level streaming and to maintain application level QoS-properties in information flows. For example, an MPEG player can view the information flow as video frames rather than UDP packets. We can timestamp video frames so that we can calculate the current frame rate. We can label video frames with priorities so that the less important frames will be the ones that will be dropped during congestion. These functions can be accomplished by various kinds of filters.

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.

Some InfoPipe Interfaces

To make the above a little more concrete, we now describe the interfaces of some significant InfoPipe components in enough detail for the reader to begin to see how they might be used to build an application.

Ports

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 understood two connection messages, 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, and turn out to simplify NetPipes and CompositePipes as well, as we shall explain.

Ports are either InPorts or OutPorts; both have exactly one owner: the InfoPipe component in which they are embedded. OutPorts understand the message –>> anInPort, which sets up a connection to anInPort.

Each InfoPipe has a set of named InPorts and a set of named OutPorts. For straight-line pipes, each of these sets has a single element, named Primary. InfoPipes also understand the –>> message, which is defined as connecting the Primary OutPort of the upstream pipe to the Primary InPort of the downstream pipe.

Sources and *Sinks* are InfoPipes in which one or other of these sets of ports is empty. *Tees* are InfoPipes in which one or both of the sets of ports have multiple members. These ports can be accessed by sending the Tee the messages inPortAt: aName and outPortAt:

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

```
"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"
```

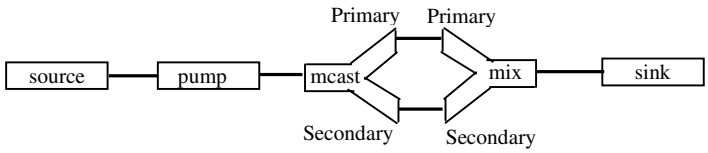

Figure 2: Building a pipeline with Tees

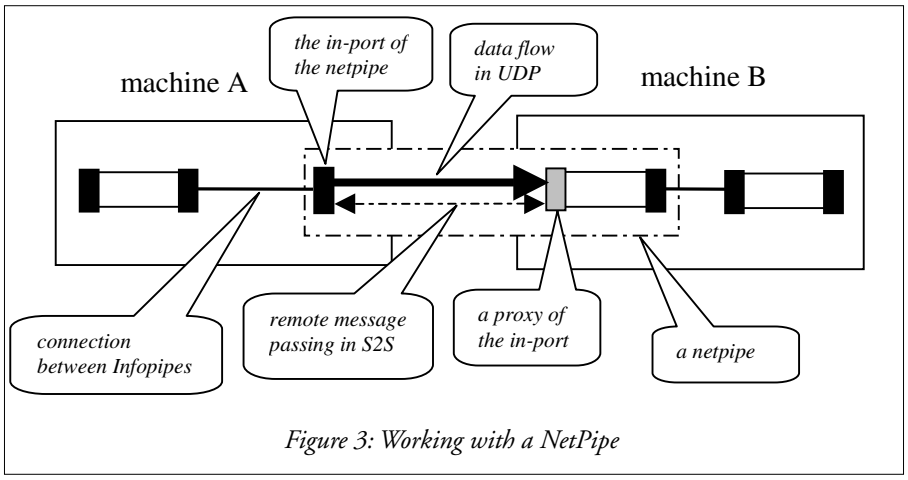

Figure 3: Working with a NetPipe

aName; the ports can then be connected as required. Figure 2 shows an example.

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. How can such an InfoPipe interact with other InfoPipe components?

A NetPipe contains a buffer at the output end, so we can monitor it and manipulate the data items in it; this would not be possible if we did not reify the existence of buffering in the network. Hence, a NetPipe provides the same data interface as a Buffer. InfoPipes with data connections to a NetPipe may not know that it is a NetPipe. Naturally, the control interface of a NetPipe is different from that of a Buffer; 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.

We use a remote messaging package called S2S to implement NetPipe. (S2S is like Java RMI, but for Squeak Smalltalk). As shown in Figure 3, a NetPipe's InPort is remote. The netpipe and its InPort can communicate through S2S remote messages easily, but relatively slowly. S2S is used to send the connection messages while the pipeline is being built. After the connection is set up, the NetPipe uses UDP for information transfer.

Using S2S provides us with access transparency: the same connection establishment protocol is used for local and remote connections. However, we do not 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, data would still flow through the pipeline, but the push or pull of each data 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 of network communication.

Figure 4 shows the code for setting up a MIDI pipeline using a NetPipe. The first two statements obtain s2s proxies for source and pump objects, lets call them s and p, that already exist on a remote machine called MusicStore. The third statement builds a NetPipe from MusicStore to the local machine. The fifth statement,

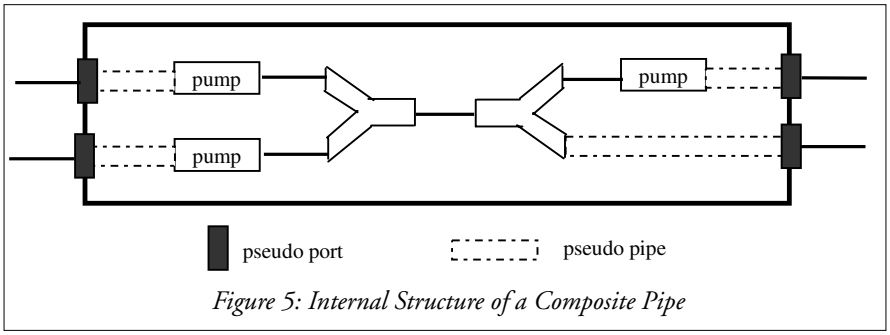

Figure 5: Internal Structure of a Composite Pipe

```
source ← 's2s://MusicStore/source1'
     asRemoteOjbect.
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.
```

Figure 4: Code for a streaming MIDI pipeline

source —>> pump —>> …, constructs the pipeline. It is interesting to see in detail how this is accomplished.

The message –>> is sent to source, which is a local proxy for remote object s. S2S translates this into a message send to 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 as the argument. The method for –>> will then execute locally to both the s and p, creating a connection with no residual dependencies on the machine that built the pipeline.

A similar thing happens with the netPipe. Although the netPipe itself is local, its InPort is on MusicStore. Thus, the connection between p and netPipe's InPort is also on MusicStore. Data transmission 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.

Composite Pipes

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. CompositePipes provide this functionality: if aPipelineElement is one element of an interconnected network of InfoPipes, we can create a new component that encapsulates that network by writing CompositePipe from: aPipelineElement.

In order to connect to a composite pipe in the same way as a primitive Infopipe, without clients knowing its internal structure, a composite pipe must have its own ports. We call these ports PseudoPorts. The PseudoPorts 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 PseudoPort and the real port, because, for example, the real ports are owned by the sub-components while the PseudoPorts are owned by the CompositePipe itself. We use a special type of pipe—a PseudoPipe—to connect PseudoPorts and their corresponding real ports. Figure 5 shows the internal structure of a composite pipe. From the outside, it is just an ordinary Infopipe with multiple InPorts and multiple OutPorts. Open ports of different sub-components may have the same name, but their PseudoPorts must have different names because the ports of an Infopipe must be distinguishable.

Summary

We believe that middleware for adaptive streaming applications should reify communication at the application level, rather than hiding it like RPC, or exposing it in implementation terms. We have designed and prototyped InfoPipes to capture this belief, and are experimenting using InfoPipes to program multimedia application. The development of InfoPipes continues, particularly the refinement of the control interfaces.

References

[1] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 3rd edn: Addison-Wesley, 2001.

[2] A. Goel, D. Steere, C. Pu, and J. Walpole, "SwiFT: A Feedback Control and Dynamic Reconfiguration Toolkit," Oregon Graduate Institute, Department of Computer Science & Engineering, Beaverton, OR, Technical Report CSE-98-009, June 1998.

[3] J. Huang, A. P. Black, J. Walpole, and C. Pu, "InfoPipes—an Abstraction for Information Flow," to be presented at Workshop on the Next 700 Distributed Object Systems, Budapest, Hungary, 2001.

[4] R. Koster, A. P. Black, J. Huang, J. Walpole, and C. Pu, "Infopipes for Composing Distributed Information Flows," Oregon Graduate Institute, Department of Computer Science, Beaverton, OR, Technical Report CSE-01-005, May 2001. Submitted as a position paper to this workshop.

[5] R. Koster, A. P. Black, J. Huang, J. Walpole, and C. Pu, "Thread Transparency in Information Flow Middleware," submitted to IFIP/ACM International Conference on Distributed Systems Platforms—Middleware 2001, Heidelberg, Germany, 2001. Available as OGI TR CSE-01-004.

[6] C. Pu, K. Schwan, and J. Walpole, "Infosphere Project: System Support for Information Flow Applications," *ACM SIGMOD Record*, **30**(*1*), 2001.