Adaptive Pipeline
an Object Structural Pattern for Adaptive Applications*

Edward J. Posnak, **R. Greg Lavender**, and **Harrick M. Vin**

Distributed Multimedia Computing Laboratory
Department of Computer Sciences, University of Texas at Austin
Taylor Hall 2.124, Austin, Texas 78712-1188
E-mail: {ejp,lavender,vin}@cs.utexas.edu, Telephone: (512) 471-9732, Fax: (512) 471-8885
URL: http://www.cs.utexas.edu/users/dmcl

1 Intent

The Adaptive Pipeline pattern (1) decouples the compositional structure of filters, which perform transformations on a data stream, from the algorithms used to implement these filters, and (2) enables both to be changed dynamically and independently, to allow an implementation to adapt itself to the run-time environment. This pattern distinguishes itself from other pipeline patterns by addressing quality of service (QoS) issues, which are relevant to a large class of applications.

2 Motivation

A variety of software systems, ranging from communications protocols to image processing tools, apply a series of transformations to a data stream. Such software systems are generally structured as a set of modular components, each implementing one transformation, that communicate via a common interface. This recurring design pattern, known as Pipes and Filters, allows arbitrary processing tasks to be constructed from modular building blocks, and benefits from the reusability that arises due to the functional overlap in these tasks [7, 9]. Software libraries that are built using this architecture can be easily extended to support new processing requirements as new content types, communication protocols, and technologies emerge. However, since the applicability of the Pipes and Filters pattern is limited to systems where the pipeline is fixed throughout its execution, it does not address the following important design considerations:

- *Performance Sensitivity:* Many important software systems are resource intensive and/or sensitive to variations in resource availability. The perceptible quality of distributed client server systems (e.g. the world wide web) is heavily dependent on the available CPU cycles and network bandwidth.
- *Computer and Network Heterogeneity:* For software to be widely useful, it must operate in computing environments, ranging from hand-held devices to powerful workstations, and in communications environments, ranging from telephone lines to high speed and wireless networks. Whereas the computational resources available to an application may vary by factors of ten, the available network resources may vary by orders of magnitude.
- *Variation in Resource Availability:* Changes in system load will cause the availability of resources to change significantly over time. This often results in annoying stalls, poor presentation quality, and/or eventual failure.

*This research was supported in part by IBM, Intel, the National Science Foundation (Research Initiation Award CCR-9409666), NASA, Mitsubishi Electric Research Laboratories (MERL), Sun Microsystems Inc., and the University of Texas at Austin.

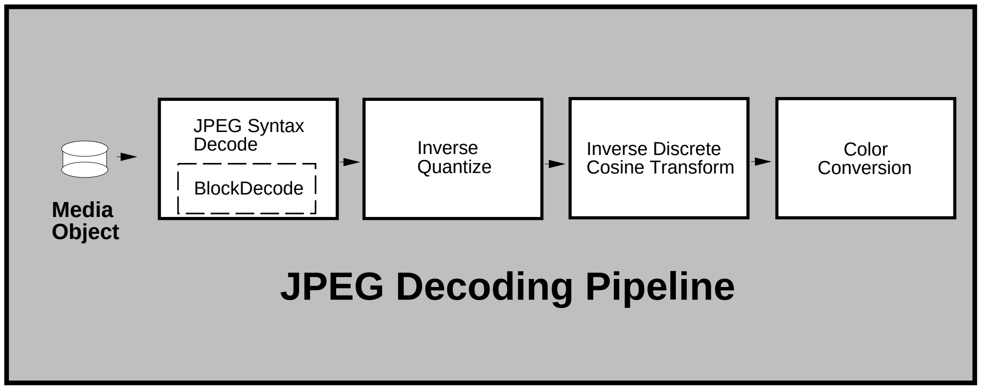

Figure 1 : Filter Composition for a JPEG Decoder.

To address these concerns, performance-sensitive software systems must adapt themselves to widely heterogeneous and dynamically changing environments. Applications can do this by dynamically adjusting their resource consumption to maximize the perceived quality under the current conditions. To be useful, such adaptive applications must be as efficient as their static counterparts.

The *Adaptive Pipeline* is an object structural pattern for developing efficient, adaptive pipeline systems. Specifically, it enables the development of adaptive systems by allowing a pipeline's structure, as well as the implementation of its filters, to be changed independently during the execution of a program. This dynamic composition of filters allows an application to adjust the resource consumption of the pipeline at runtime, to adapt to different computing and communications environments as well as changes in resource availability. The Adaptive Pipeline pattern enables the development of performance-sensitive systems by allowing filters to tightly integrate their implementations using type parameterization as an implementation technique. Significant performance gains can be achieved by allowing a filter's implementation to be parameterized by other filters, since the compiler can make use of inline methods to minimize the overhead of crossing multiple abstraction boundaries. This type of composition effectively collapses a sequence of filters into statically bound, but highly efficient code. A carefully engineered balance between the use of static and dynamic binding of filters can allow for efficient implementation while maintaining a modular, configurable architecture.

To illustrate how the Adaptive Pipeline pattern facilitates the development of adaptive, performance-sensitive applications, consider the design of a multimedia toolkit that allows applications to process digital audio, video, and images. The toolkit must perform the tasks of accessing, decoding, and manipulating objects that may be encoded in a variety of compressed formats, (e.g. MPEG, JPEG, H.261, etc.). To accomplish these tasks the toolkit must implement a number of transformations, such as primitive decompression and image processing operations, that may be successively applied to a media stream. Since many compression algorithms employ the same set of transformations (e.g. Huffman coding, discrete cosine transform, etc.), and many applications perform different permutations of the same image processing operations (e.g. scale, clip, rotate, etc.), it is desirable to implement these operations as composable filters in pipeline architecture. If the Adaptive Pipeline pattern is used, then different implementations of a filter can be used interchangeably to adjust the resource cost and quality characteristics of the pipeline at run time. This provides an application using the toolkit with a powerful mechanism for adapting the quality of presentation to the available resources.

Such adaptation is possible, for example, in JPEG video decoding. Figure 1 shows a pipeline of filters that collectively perform the the sequence of transformations required by the JPEG standard. In this pipeline, the compressed input is decoded into 8x8 blocks of frequency coefficients, which are then inverse quantized, converted into a pixel representation using the inverse discrete cosine transform, and finally converted into a displayable color space. The filters shown having a solid border and connected by arrows are dynamically configurable. Thus different implementation of these modules can be used interchangeably at runtime to adapt various aspects of the presentation quality (e.g. frame rate, fidelity, etc.) to match the available resources. The Block Decode filter is shown with a dashed border to indicate that it has been statically bound to the JPEG Syntax Decode filter using type parameterization. Since this filter is invoked with high frequency and has little need for dynamic reconfigurability, binding it statically will significantly improve the pipeline's runtime performance, without sacrificing flexibility. Since the multimedia toolkit is designed using the adaptive pipeline pattern, other decoding and presentation processing tasks can be similarly composed to realize adaptive, highly efficient implementations.

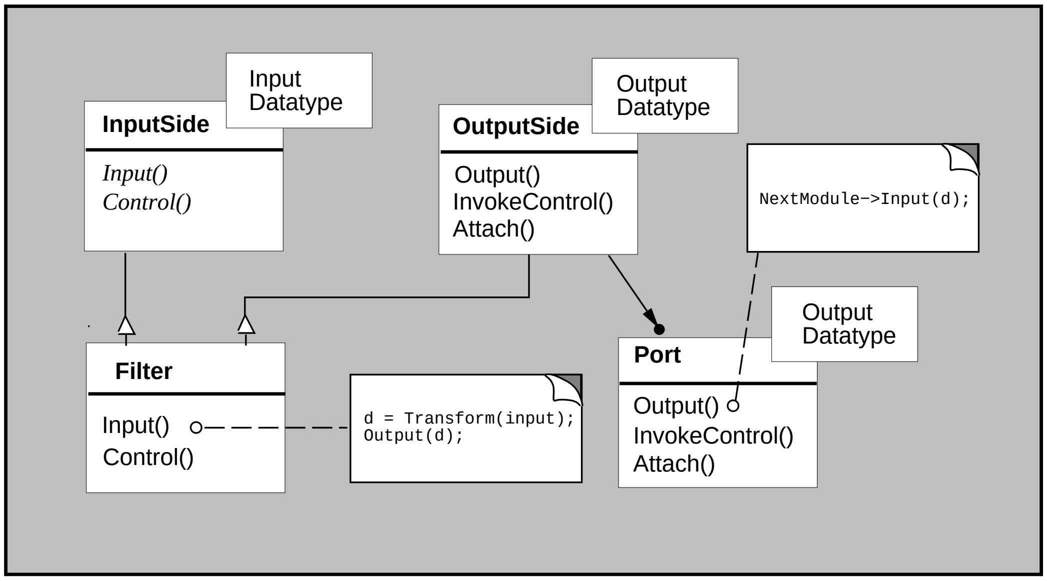

Figure 2 : Structure of the Adaptive Pipeline Pattern

3 Applicability

The Adaptive Pipeline pattern should be applied to design the following types of systems:

- Resource intensive applications that must run in environments that are highly heterogeneous with respect to bandwidth and computational capacity.
- Communication protocols, which normally contain several abstraction layers with a number of functional components residing in each layer.
- Other software systems in which a stream of data passes through a series of transformations. Specific types of applications include encoders and decoders for data streams that are tagged, compressed, encrypted, or encoded (e.g. HTML pages, MPEG streams, encrypted email, RPC arguments/results, etc.) for transfer across networks.

  Pipeline architectures in general are applicable to the development of a variety of software systems. However, experience in building many of these systems has shown that their usefulness depends a great deal on efficient implementation and/or the ability to dynamically adapt to the runtime environment [1, 2, 4, 8, 11, 14, 15]. Whereas the need for adaptation motivates dynamic configurability of filters, the need for efficiency motivates the selective use of type parameterization for performance-critical components.

4 Structure and Participants

The structure of the Adaptive Pipeline pattern is illustrated in the Booch class diagram shown in Figure 2.

The key participants in the Adaptive Pipeline pattern include the following classes:

- **InputSide** defines the interface for inputting data or control information to a filter. This also defines the filter's type for the purposes of composition.
- **OutputSide** provides a reusable implementation for attaching a component's output to another component's input.
- **Port** maintains the state of some attachment between components. The OutputSide uses ports to attach and output to multiple filters.

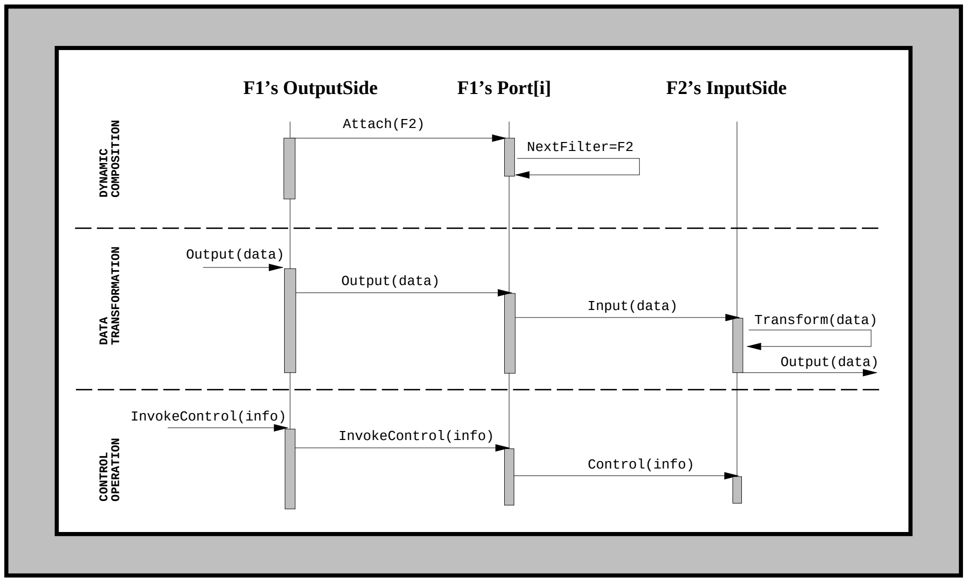

Figure 3 : Collaborations in the Adaptive Pipeline Pattern

- **Filter** implements the transformation of input data (i.e., it defines the Transform() method). A Filter inherits from InputSide, which defines its type and input interface, and OutputSide, which provides the implementation to attach to other components.

5 Collaborations

Figure 3 illustrates the three phases of collaborations in the Adaptive Pipeline pattern. F1 is a filter whose output will be the input for filter F2.

1. *Dynamic Composition* : In this phase, an output port of filter F1 is attached to filter F2. The OutputSide implementation inherited by filter F1 invokes the `Attach()` method on one of its `Port` objects, which saves a reference to filter F2, so that subsequent `Output()` and `Control()` operations on that port will be forwarded to filter F2.
2. *Data Transformation* : In this phase, input data that has been transformed by F1 will now be passed to F2 so that the F2's transformation can be performed. F1 invokes its `OutputSide::Output()` method, which invokes the `Output()` method of the appropriate Port object. The port object passes the data to F2 by invoking F2's `Input()` method. After F2 has transformed the input data, it can then forward the data on to the next filter in the same fashion.
3. *Control Operation* : In this phase control information is passed from F1 to F2. To do so, F1 invokes its `Output-Side::InvokeControl()` method, which calls the `InvokeControl()` method on the appropriate `Port` object. The port object passes the control information to F2 by invoking F2's `Control()` method. The call returns after F2 has appropriately updated its state.

As a concrete example of these collaborations, recall the JPEG decoder pipeline in Figure 1. Dynamic Composition occurs when the Inverse Quantize filter attaches itself to an Inverse Discrete Cosine Transform filter. Data Transformation occurs when the Inverse Quantize filter de-quantizes the data and passes it to the Inverse Discrete Cosine Transform filter. Finally, the Inverse Quantize filter provides a control operation to allow other modules to change its internal quantization parameters.

```
template <class I>                            // I is the input DataType
class InputSide {
public:
       virtual void Input(I& data, int port = 0) = 0;
       virtual bool Control(ControlType& op) { return false; };
};


template <class O>                            // O is the output DataType
class OutputSide {
public:
         virtual bool Attach(InputSide<O>& m, int iport=0, int oport=0);
         virtual bool Detach(int oport=0);
protected:                                    // these should only be invoked internally
         inline void Output(O& data, int oport = 0) { outport[oport].Output(data); };
         bool InvokeControl(ControlType& op, int oport=0);
         Port<O> outport[MAX_PORTS];
};
```

Figure 4 : InputSide and OutputSide classes.

6 Consequences

Pipeline architectures in general provide the following benefits:

- They allow filters to be reused to implement new protocols.
- They help create extensible applications by allowing filters to be easily added or removed according to a well-defined set of interfaces.

The Adaptive Pipeline pattern has the following additional benefits:

- It allows the implementation to be configured at run time to adapt to heterogeneous environments and changing resource availability.
- It addresses the tension between modularity and performance by allowing performance-critical filters to be efficiently composed using type parameterization.

Just as all pipeline architectures, the Adaptive Pipeline has the following drawbacks:

- Black box abstractions can sometimes be inefficient in cases where filters would achieve significant performance benefits by more tightly coupling their implementations.

7 Implementation

The following issues should be considered when implementing the Adaptive Pipeline pattern.

1. *Type-checked composability.*

```
class A : public InputSide<X>, public OutputSide<Y> { ...  };
class B : public InputSide<Y>, public OutputSide<Z> { ...  };

// dynamically bind filter a to filter b
A a;
B b;
a.Attach(b);                                // a.Output() -> b.Input()
...
a.Input(data);                              // will transform data and pass to b.Input()
```

Figure 5 : Dynamic composition.

The interfaces between filters should be uniform to allow arbitrary composition of filters, yet they should be strongly typed to prevent misconfiguration. Although uniform input and output interfaces can be inherited from the `InputSide` and `OutputSide` base classes, respectively, the composition of a `OutputSide` of data type X with a `InputSide` of data type Y must be prevented. This can be effectively accomplished in C++ using templates to parameterize the `InputSide` and `OutputSide` base classes by the data type produced or consumed. This results in an extensible set of similar, yet distinguishable interfaces that allows the compiler to reject the composition of filters whose respective input and output data types do not match. To see this, observe that the signatures for the `OutputSide::Attach()`, `OutputSide::Output()`, and `InputSide::Input()` methods shown in Figure 4 all incorporate the template parameter. Thus, if a filter that inherits from `OutputSide<X>` attempts to attach to a filter that inherits from `InputSide<Y>` a compilation error will occur since the former filter's `Attach()` method requires a `InputSide<X>` as its first parameter. Similarly, type checking also guarantees that the former filter can only output objects of type X and the latter's `Input()` method will only be passed objects of type X.

2. *Input and Control operations.*

The specific function of a filter is defined by its implementation of the `Transform()` and `Control()` methods. The `Input()` method implementation normally invokes `Transform()` to execute the algorithm for the specific transformation that the filter performs, and then invokes `Output()` to pass the transformed data on to the downstream filter(s). The signature for `Input()` (see the code in Figure 4) includes the input data and an optional input port, which may be used to determine how the data is to be processed. For example, a color conversion filter that combines blocks from different color planes may accept red blocks on port 0, green blocks on port 1, etc.

The `Control()` method allows a filter to pass, to another filter, information that will influence the manner in which the latter processes its input data. For example, passing a new set of quality of service parameters via the `Control()` interface may cause a filter to modify its transformation algorithm or reconfigure its internal filters. It is desirable to make the `Control()` interface uniform, while allowing different filters to process different types of control operations. This can be accomplished by defining the type of the argument to `Control()` as a reference to the abstract class `ControlType` and instantiating all control operations as concrete subclasses of `ControlType`. Each `Filter` class then overrides the `Control()` method with an implementation that selectively processes the particular control operations that the filter is interested in, and invokes the default implementation, `InputSide::Control()`, for all others.

3. *Dynamic composition.*

Figure 5 shows how a filter of type `A` is dynamically bound to filter of type `B`.

Implementing dynamically composable filters requires that each `Filter` object maintain a modifiable set of references to the other filters to which it is attached. The `Attach()` and `Detach()` methods are responsible for updating this set during program execution as filter composition changes. The `Output()` and `Control()` methods will reference members from this set to pass data and control operations to the appropriate downstream filters. Management and use of this set is simplified by the use of a `Port` class, which encapsulates the state of an attachment. The port set, shown as a simple array in the `OutputSide` class definition above, allows attached filters to be referenced by numeric handles, i.e. port numbers,

```
template <class O>                            // O is the output DataType
class Port {
public:
       inline void Output(O& data, int oport = 0) { InputSide->Input(data); };
protected:
       InputSide<O> *InputSide;               // the filter attached to this port
       int inputPort;                         // input port attached to this port
       bool attached;                         // true if a filter is attached
};
```

Figure 6 : Port class.

rather than explicit pointers. The benefit of this approach is that the code for maintaining and using references to other filters has been factored out into a reusable implementation class. The implementation of the `Port` class is shown in Figure 6.

4. *Static versus dynamic binding.*

A general problem with highly modular and dynamic software architectures is that excessive layering often leads to inefficient implementations. Runtime costs are paid each time a program's control flow crosses an abstraction boundary, which has been enforced for the sake of information hiding and a high degree of modularity. [1, 2, 3, 14, 15].

In implementing the Adaptive Pipeline pattern, the performance penalty of a dynamically dispatched procedure call is incurred for each `Input()` operation between dynamically bound filters. The higher the frequency of invoking such dynamically bound methods, the greater the performance penalty.

Static type parameterization of filters can significantly reduce the overhead of crossing filter boundaries by allowing efficient choices (such as inlined method calls) to be made during code generation. Type parameterization of filters can be implemented as shown in the example in Figure 7. Here the `BlockFilter` filter provides an inline method that enables it to be tightly integrated with other filters, such as the `Decoder`.

When implemented as above, static composition provides the compiler with enough information to effectively collapse a sequence of filters into highly efficient code, while maintaining a level of abstraction that is purely syntactic (i.e., has no run-time cost). Whereas inlining in general does not always result in better performance, empirical evidence suggests that careful selection of methods to be inlined, as well as the register allocation strategies of an optimizing compiler, can have a positive effect on performance [5]. The tradeoff of static binding is that once filters are bound in this fashion, they cannot be dynamically reconfigured.

A carefully engineered balance between the use of static and dynamic binding of filters can allow for efficient implementation while maintaining a modular, configurable architecture. Whereas static binding should be used to improve performance for filters that have minimal reconfigurability demands and whose input methods are invoked with high frequency, dynamic binding is best suited for filters that have high reconfigurability demands and low invocation frequency. A useful adaptive composition framework will provide mechanisms that allow filters to be composed both statically and dynamically. This allows implementers to make the same kind of abstraction tradeoffs that are evident in the ANSI C++ Standard Template Library (STL) [10, 13], which is a model for preserving syntactic abstractions while balancing the tension between static and dynamic binding to incur minimal run-time overhead.

8 Sample Code

In this section, we sketch an implementation for the JPEG decoder discussed earlier, and show how dynamic reconfigurability is used to adapt the quality of presentation to suit different environments. Recall from Figure 1(a) that a JPEG decoder can be composed of the following filters: a syntax decoder, a block decoder, an inverse quantizer, an inverse discrete cosine

```
class BlockFilter :  public InputSide<Block>, public OutputSide<Block> {
public:
       virtual void Input(Block& data, int port = 0) {...};
       virtual bool Control(ControlType& op) {...};
       inline void Transform(Block& data, int port = 0) {...};
};

template <class F>                            // F is the type of filter to bind to
class Decoder :  public InputSide<Stream>, public OutputSide<Block> {
public:
       Decoder(F& filter) :  f(filter) { ...  };
       virtual void Input(Stream& data, int port = 0) {
                   ...                        // create Block b
                   f.Transform(b);            // inline method call
       };
private:
       F f;
};

BlockFilter f;                                // instantiate filter
Decoder<BlockFilter> df(f);                   // statically bind Decoder d to Filter f
```

Figure 7 : Static composition via type parameterization.

```
BlockDec                    bd;                 // construct a block decoder
JPEGSyntaxDec<BlockDec>     syntax(bd);         // parameterize into JPEG syntax decoder
IQDec                       iq;                 // construct inverse quantizer
IDCTDec                     idct;               // construct inverse DCT
ColorConversion             cc;                 // construct color converter


// construct the pipeline by attaching the modules
syntax.Attach(iq);
iq.Attach(idct);
idct.Attach(cc);
```

Figure 8 : Dynamic composition.

```
// detach the old color conversion filter
idct.Detach();

// instantiate new faster color conversion filter
FastColorConversion fc;

// attach three pipelines to new color converter
idct.Attach(fc);
```

Figure 9 : Dynamic reconfiguration.

```
// One of JPEGSyntaxDec’s internal methods contains the following code:

// instantiate the control operation (qp is the new set of quantization parameters)
SetQParams op(qp);

// cause op to be passed to IQ’s Control() method
if (InvokeControl(op) == false)
           HandleError();
```

Figure 10 : Control operation.

transform, and a color converter. Since the block decoder is invoked with high frequency, it is parameterized into the JPEG syntax decoder to reduce the invocation overhead. The remaining filters are less performance-sensitive, and are configured dynamically. The code sample in Figure 8 shows declarations of the filters and how they are dynamically composed to generate a JPEG decoder.

Dynamic configurability allows the JPEG decoder to adapt the quality of presentation to the environment, for example, by selecting a color conversion algorithm that provides the appropriate quality versus speed tradeoff. If it becomes necessary to use fewer CPU cycles to maintain a certain frame rate, a faster, lower quality color conversion algorithm can be dynamically configured as illustrated in Figure 9.

As stated previously, filters can pass control information via the `Control()` interface. Figure 10 shows how the syntax decoder can request the inverse quantizer to change its quantization parameters.

9 Known Uses

Pipeline architectures have been used to develop many different types of software systems in which a sequence of transformations on a data stream are performed. For example, the UNIX operating system provides users with a number of reusable programs (e.g. sed, grep, awk) that can be pipelined together in an arbitrary fashion to implement more complex data transformations. The same type of flexible module composition is used by the STREAMS subsystem to implement sequences of transformations (e.g. network protocol stacks) on input/output streams[12]. Since UNIX and STREAMS both pass data between filters via repositories (a buffer of ASCII characters and a queue of STREAMS messages, respectively), they are pure occurrences of the Pipes and Filters architecture [7, 9].

The x-kernel is another compositional pipeline subsystem that performs a function similar to STREAMS, but also supports finer-grained composition of filters. Rather than implementing protocols as separate filters, the x-kernel factors out common functions such as message demultiplexing, selective retransmission, etc., into a large number of composable *micro-protocol* filters. Since the x-kernel does not make use of type parameterization, increased granularity of decomposition decreases the efficiency of the implementation. To improve performance, the x-kernel eliminates the need for data repositories (and the associated overhead of copying) by pushing data through its filters. Thus the x-kernel is more like an Adaptive Pipeline architecture than UNIX or STREAMS.

The Adaptive Pipeline pattern has also been used to implement a toolkit for implementing Presentation Processing Engines that support multimedia applications [11]. By allowing fine-grained composition of compression and image processing filters, the toolkit facilitates the development of extensible presentation processing engines that can be dynamically configured to adapt to changes in resource availability and user preferences. By incorporating static binding the toolkit permits the efficient implementation of performance-sensitive multimedia applications.

10 Related Patterns

The Streams pattern [6] describes a software architecture where stream objects, representing queues of data elements, are composed in a fashion that models data flow. Control flow is not explicitly represented at the architectural level, but is instead captured as an internal feature of each stream object. This type of architecture is more suitable for systems in which control flow is relatively static.

The Pipes and Filters pattern describes an architecture consisting of components that transform data (filters) and connections that transmit data between these components (pipes) [7, 9]. This architecture effectively promotes reusability, maintainability, and testability when the subtasks of a system can be easily identified and broken into independent but cooperative components.

The Adaptive Pipeline pattern is, in principle, similar to the Pipes and Filters architecture, but has the following fundamental differences:

- The applicability of the Pipes and Filters pattern is limited to systems where the order in which filters are applied to a data stream is strongly determined. The Adaptive Pipeline pattern extends the applicability of this architecture to adaptive systems through the use of dynamic (i.e. run-time) binding of filters.
- A basic property of the Pipes and Filters architecture is that data is not shared between filters. Although the independence of filters is fundamental to the Adaptive Pipeline pattern, sharing of information between filters is not prohibited. Sharing a certain amount of information is necessary in many practical implementations, and is useful in improving the efficiency of implementation.
- Whereas the Pipes and Filters pattern describes both push and pull data flow models, which may use data repositories in between filters, the Adaptive Pipeline pattern defines only a push model that does not include pipes. Implementation experience has shown that buffering and copying significantly degrade performance; the use of these mechanisms should be demotivated at the design stage, and only implemented (within the filters) when absolutely necessary. Whereas the use of pipes allows the filters in a pipeline to operate in parallel, in the absence of pipes, the delineation of application-specific data units allows multiple pipelines to operate in parallel.

11 Acknowledgements

The authors would like to acknowledge Sriram Rao, Prashant Shenoy, Scott Page, and Pawan Goyal for their feedback on early versions of this manuscript. We would also like to thank Frank Buschmann, Peter Sommerlad, and Steve Berczuk for their constructive comments at PLoP '96, which significantly improved the presentation of the ideas in paper.

REFERENCES

[1] T. Braun and C. Diot. Protocol Implementation Using Integrated Layer Processing. In *Proceedings of ACM SIGCOMM'95*, pages 151–161, Boston, Massachusetts, September 1995. ACM. *Computer Communication Review*, Volume 22, Number 4.

[2] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of ACM SIGCOMM'90*, pages 200–208, Philadelphia, Pennsylvania, September 1990. IEEE.

[3] D.D. Clark. Modularity and Efficiency in Protocol Implementation NIC-RFC 817. In *DDN Protocol Handbook*, pages 3.63–3.88. U.S. Department of Defense, July 1982.

[4] C. L. Compton and D. L. Tennenhouse. Collaborative Load-Shedding for Media-Based Applications. In *International Conference on Multimedia Computing and Systems*, volume 1, pages 496–501, Boston, Massachusetts, May 1994.

[5] J.W. Davidson and A.M. Holler. Subprogram Inlining: A Study of its Effects on Program Execution Time. *IEEE Transactions on Software Engineering*, SE-18(2):89–102, February 1992.

[6] S.H. Edwards. Streams: A Pattern for "Pull-Driven" Processing. In J.O. Coplien and D.C. Schmidt, editors, *Pattern Languages of Program Design*, chapter 21, pages 417–426. Addison-Wesley, 1995.

[7] D. Garlan and M. Shaw. *An Introduction to Software Architecture*, volume I of *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, 1993.

[8] N. Hutchinson and L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, January 1991.

[9] R. Meunier. The Pipes and Filters Architecture. In J.O. Coplien and D.C. Schmidt, editors, *Pattern Languages of Program Design*, chapter 22, pages 427–440. Addison-Wesley, 1995.

[10] D.R. Musser and A.A. Stepanov. Algorithm-Oriented Generic Libraries. *Software Practice and Experience*, 24(7), July 1994.

[11] E. J. Posnak, H. M. Vin, and R. G. Lavender. Presentation Processing Mechanisms for Adaptive Applications. In *Proceedings of Multimedia Computing and Networking, San Jose, CA*, February 1996.

[12] D.M. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63(8):311–324, October 1984.

[13] A. Stepanov and M. Lee. The Standard Template Library. Technical report, Hewlett-Packard Laboratories, July 1995.

[14] D.L. Tennenhouse. Layered Multiplexing Considered Harmful. In Harry Rudin and Robin Williamson, editors, *Proc. IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks*, Zurich, Switzerland, May 1989.

[15] Ian Wakeman, Jon Crowcroft, Zheng Wang, and Dejan Sirovica. Layering considered harmful. *IEEE Network*, January 1991.