A Scalable Approach to Continuous-Media Processing

Dragos-Anton Manolescu Klara Nahrstedt
Department of Computer Science
1304 W. Springfield Ave., Urbana, IL 61801
{manolesc,klara}@cs.uiuc.edu

Abstract

Techniques that emphasize software reuse and scalability are becoming more important than ever. In this paper we present a component-based model for continuous-media applications. Components encapsulate expert knowledge and facilitate reuse. They provide a toolkit that is used to create a wide range of continuous-media applications. Our model is scalable in several dimensions: media transformations, number of processors, number of configurations, media types, and processing and communication requirements. The paper is organized as a catalog of four software patterns. It is our belief that developers and researchers working on continuous-media applications can benefit from and apply software patterns as well.

Introduction

Current multimedia systems are complex. They span many areas (e.g., music, graphics, real-time processing) and have to scale in several different directions (e.g., number of processing nodes in a distributed environment, number of continuous-media streams). Consequently, various techniques that *emphasize reuse* (e.g., application frameworks [1, 5, 9], design patterns [4, 2], software components [8]) and *provide scalable software solutions* are attracting the interest of the industry.

In this paper we present a component-based model applicable to continuous-media processing. The model is based on four software patterns that express expert knowledge about software construction. Systems employing these patterns scale well in several dimensions: (1) media transformations, (2) number of processors, (3) number of data streams, (4) number of possible configurations, (5) media types, and (6) processing and communication requirements. We begin with the **media flow architecture** which is extremely suitable for continuous-media processing [15, 17]. In the **payloads** pattern, we describe a mechanism for encapsulating different types of information—e.g., quality of service (QoS) parameters, continuous-media data, event notifications—within messages. The abstraction increases flexibility and ensures scalability with the number of streams and media types. In the **payload passing protocol** we discuss several protocols that are available for inter-module communication. Finally, in **control and processing partitions** we organize applications according to the spatial and temporal requirements of continuous-media.

To the best of our knowledge, this is the first attempt to create a scalable, reusable architecture for continuous-media processing based on a catalog of software patterns. We intend to extend the catalog, hoping that it will prove a valuable resource for developers and researchers.

1 Media flow architecture

The media flow architecture organizes applications as a *network of processing modules*[1] that apply a series of transformations to one or several streams of continuous-media data. Each processing module takes its input from an upstream module, performs a simple, generic transformation (e.g., Huffman coding, arithmetic coding, quantization, DCT) and passes the results to a downstream module—Figure 1. Enforcing strict, simple inter-module interfaces yields a large number of possible combinations that provide solutions to many problems. For example, JPEG, MPEG–1 and MPEG–2 decoders can be constructed by parameterizing the architecture with specific algorithms [16].

❖**Benefits** The media flow architecture ensures *scalability* in several dimensions: number of processing modules, processors, data streams, possible configurations and media types; and processing requirements. It facilitates the rise of end-user programming, automation and software components and emphasizes reuse at the module level. Users that have knowledge only about the application domain create new applications by simply connecting modules, without performing any programming [10, 17]. Sometimes visual

[1] In this context, "module" is any processing unit within the application domain.

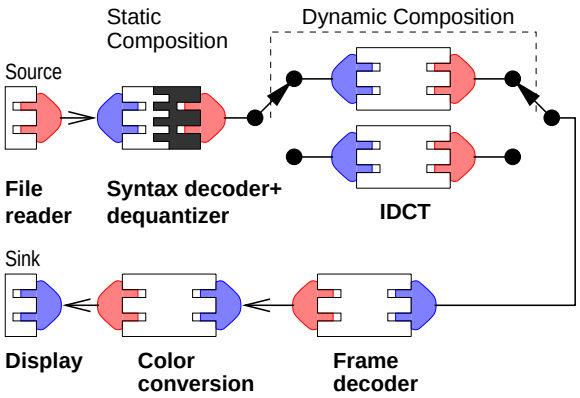

Figure 1. A JPEG player following the media flow architecture.

programming tools or scripting languages [7] assist the creation of module networks. Because the interaction mechanism between modules is fairly simple, media flow is suitable for multimedia frameworks [13].

❖**Liabilities** Media flow is *not a good choice* for applications with dynamic control flow or many feedback loops. For instances where the overhead of inter-module communication is too high, a different architectural choice might be a better solution.

Processing modules do not make any assumptions about their context and communicate only with other media-processing modules. Consequently, signaling errors that occur at the module level is cumbersome and difficult. An error message is meaningful only when combined with some knowledge about the network topology—we present one possible solution in Section 4.

❖**Discussion** *Modules* play a key role in media flow architectures. Applications that follow this pattern manifest an increased degree of modularity. This makes it easy to distribute the development effort among different groups (e.g., compression modules could be developed by the data compression group). Depending on the number of ports and their types, there are three *module types*: **sources** interface with an input device (e.g., digital video camera) and have one or more output ports; **sinks** interface with an output device (e.g., display) and have one or several input ports; and **filters** have both input and output ports—not necessarily only one in each direction—and perform processing on the information fed into the input port and write it to the output port.

Inter-module communication is performed by passing messages through unidirectional input and output ports, thereby replacing direct calls. Unidirectional ports are not a limitation. Rather, they increase a component's autonomy, such that—provided that there are no feed-back loops—processing is unaffected by the presence or absence of output connections. Therefore, it is possible to change the output connections at runtime.

To enable interconnectivity between two modules, the output port of the upstream module and the input port of the downstream module (shown as sockets in Figure 1) have to be *plug-compatible*. Having several port (plug) types limits the interconnectivity and requires adapters to connect incompatible ports.

Specialized filters are more efficient than implementing their functionality with several generic filters, because of the reduced communication overhead. However, specialization also limits the number of possible configurations that a filter can be part of. For example, a JPEG decoder module offers better performance than obtaining the same functionality by connecting a syntax decoder, dequantizer, IDCT and color converter. However, in the latter case, the filters are reusable and can be used for other applications (e.g., an MPEG decoder).

Good processing modules comply with the following *design guidelines*. Filters (1) are designed independently of their use, and (2) cooperate only by using the output of one as the input to another. Different applications are implemented by interconnecting basic filters—DCT, IDCT, quantizers, dequantizers, etc. Decoupling filters from a particular problem increases their reusability and allows applications to use them as "black boxes." However, in many circumstances the *black-box approach* has performance penalties because it does not take into account context-specific factors. Good filters balance these two forces.

In the context of continuous-media applications, some filters maintain *logical state*—e.g., decoding B-frames from an MPEG sequence requires the I-frames and P-frames. Filters without internal state can be replaced while the system is running. Different implementations of any stateless filter can be exchanged at runtime to adjust the resource cost and quality characteristics such that the QoS requirements are maintained [15]. Consequently, the **dynamic composition** of filters allows an application to (1) adjust the resource consumption of the system at runtime; and (2) adapt to different computing and communications environments as well as changes in resource availability.

Within a network, adjacent *performance-critical modules* can be regarded as a composite [4] filter and replaced with an optimized version that trades flexibility for performance. **Static composition** provides the underlying application (e.g., compiler) with enough information to collapse a sequence of modules into a functionally equivalent primitive module, reducing the overhead of inter-module communication [15]. A good balance between the use of static and dynamic binding of modules allows for *efficient implementations* while maintaining a modular, configurable architecture.

❖**Examples** The media-flow architecture is used by

the Presentation Processing Engine (PPE) framework. Posnak et al [17] have used the PPE to build JPEG and MPEG players. They have reported both design and code reuse. When using the PPE framework, a media player plug-in for Netscape required a few lines of Tcl and little domain expertise. However, reusing the Berkeley MPEG source code was more difficult, since it required a good understanding of the existing code and a significantly larger programming effort and expertise. Their implementations are also competitive with commercial products. The decoding time for a JPEG frame is within 5% of the time required by the IJG decoder. Similarly, the MPEG frame rate is at most 10% lower than the Berkeley player. This analysis shows that reusable, high-performance solutions are possible with today's technology. Other examples include ActiveMovie [12] and the Berkeley Continuous-Media Toolkit [7].

❖**Implementation notes** This section follows the guidelines from [15]. Processing modules inherit an input interface from `PushInput` and an output interface from `PushOutput`—the `Push` prefix is explained in Section 3. `PushInput`—Figure 2—is an abstract class that determines the input data type. Subclasses implement the `Put()` method to process the input data and pass it downstream. `PushOutput` determines the output data type and defines methods for dynamic composition—`Attach()` and `Detach()`. Attachment between two modules is abstracted by the `PushPort` class. This facilitates establishing one-to-many connections, in which case `PushOutput` has to be modified accordingly. C++ templates parameterize the classes from Figure 2 by the types of data that enters and leaves the processing module. The compiler's type checking system prevents connecting two modules which are not plug-compatible. Type parameterization—templates—and function inlining allow developers to implement static composition.

Based on these classes, implementing processing modules is straightforward. Figure 3 shows the skeleton of an IDCT filter.

```
template<class DataType> class PushInput {
public:
  virtual ~PushInput() { };
  virtual void Put(DataType &data) =0;
};

template<class DataType> class PushOutput {
public:
  virtual ~PushOutput() { };
  virtual void Attach(PushInput<DataType>
                      *next)
   {
     _port=new PushPort<DataType>(next); };
  virtual void Detach()
   { delete _port; };
protected:
 inline void Output(DataType &data) {
   _port->Output(data); };
private:
 PushPort<DataType> *_port;
};

template<class DataType> class PushPort {
public:
  PushPort(PushInput<DataType> *module)
    : _module(module) { };
  ~PushPort() { };
  inline void Output(DataType &data)
    { _module->Put(data); };
protected:
  PushInput<DataType> *_module;
};
```

Figure 2. The `PushInput`, `PushOutput` and `PushPort` classes.

2 Payloads

Payloads are *messages* that are exchanged during inter-module communication. They encapsulate various types of information that is sent from one module to another and ensure a decreased coupling between the message structure and the communicating entities. If inter-module communication is restricted to message passing, in addition to continuous-media data (e.g., audio samples, video frames) payloads also accommodate all types of control information (e.g., QoS parameters, notification messages).

❖**Benefits** Payloads increase the *overall flexibility* and permit the addition of *new attributes* (e.g., priority levels, support for asynchronous notifications, new QoS parameters) with minimal changes. They enable applications to *scale* with the number of media streams and data types. The decreased coupling between processing modules and payload contents facilitates the optimization of the message passing mechanism. Transfer mechanisms that require additional information about messages extract it directly through a well-defined protocol.

Adding *new message types* does not require changing the existing software components which are not interested in them. For example, in a media flow architecture, filters pass downstream the messages they do not understand, without performing any processing—also known as "tunneling."

❖**Liabilities** The major liability of this mechanism is its *inefficiency* when compared with direct calls. One way to improve performance is to pack multiple messages into one container message such that all of them are transferred in one step. Although this technique has been successfully applied in operating systems [3], it is not viable for continuous-media applications, where typical payloads contain time-sensitive data and have large memory footprints. Optimized copying techniques—see below—provide better solutions.

❖**Discussion** Payloads are composed of two *compo-*

```
class IDCT: public PushInput<InBlock>,
            public PushOutput<OutBlock> {
public:
  IDCT() { };
  ~IDCT() { };
  void Put(InBlock &iBlock)
  {
    OutBlock oBlock;
    // compute the IDCT
    Output(oBlock);
  };
};
```

Figure 3. An `IDCT` filter.

nents, a descriptor component (header) and a data component. **Descriptors** contain general information about the payload (e.g., asynchronous notification, priority level), as well as type-specific parameters (e.g., image size, sample rate). For continuous-media applications the **data component** is much larger than the descriptor and, if present, the media type is determined from the descriptor. Usually, payloads corresponding to control messages consist of descriptors only.

In the simplest form, the descriptor component contains just a tag that identifies the *payload type*. Another important component is the *length of contents*. For payloads corresponding to closed Logical Data Units (LDUs), the length is known in advance and can be encoded within the descriptor. When payloads correspond to open LDUs, the end of the content has to be signaled explicitly (i.e., with a special control message). In a media flow architecture, when a source reaches the end of the input stream, it signals this condition to all filters which are connected to its output(s). The mechanism is propagated down the filter network until it reaches the sink.

A distributed application following the media flow architecture can have its processing modules running on *different processors* that communicate over a high speed serial link. In this case, payloads need to provide serialization and deserialization methods to encode them into a reliable byte stream at the transmitter side and decode them from a byte stream at the receiver side. Therefore, the payloads pattern enables media flow architectures to *scale* with the number of processors. It also minimizes the effort to distribute an existing application on several processing nodes.

Continuous-media processing requires transferring payloads between the processing modules of a media-flow architecture. Because of the large memory footprints of multimedia data, *payload copying* is an expensive operation. Consequently, efficient transfer mechanisms minimize payload copying. Whenever it is not possible to avoid copying, an *optimized technique* is employed. **Shallow copies** have individual descriptors and share the data components. However, entities are not allowed to modify the copy. **Deep copies** have individual descriptors and data components. To increase efficiency, they can be implemented as copy-on-write. The change of ownership is more difficult if the communicating modules reside across hardware boundaries.

❖**Examples** Microsoft ActiveMovie [12], one of the most recent media flow architectures, allows users to play digital movies and sound encoded in various formats. In ActiveMovie, payloads are either media samples or QoS data. Media data originates at the source and is passed downstream. QoS data provides a means to gracefully adapt to load differences in the media stream. It is used to send notification messages from a renderer (sink). All components of the architecture recognize a special asynchronous event which requires graceful flushing of old data, followed by resynchronization. This pattern is also employed by the Berkeley continuous-media toolkit [7] and the VuSystem [10].

❖**Implementation notes** A flexible implementation solution is to regard the payload as a composite message. Concrete classes corresponding to various message types are derived from the abstract class `Message`—Figure 4—and provide implementations for its interface. The `Payload` class is a composite `Message` that extends the interface with several descriptor-specific methods. In this example, each payload has associated with it a priority level and an asynchronous flag. Clients can subclass `Payload` to extend the descriptor-specific interface in a transparent way.

3 Payload passing protocol

Payload passing protocol describes the relationship between control flow and media flow. This pattern establishes 3 different protocols for inter-module communication. In the *push* protocol, both control flow and media flow originate at the source. Likewise, in the *pull* protocol, control flow originates at the sink. The *indirect* protocol employs a shared repository (mailbox) accessible to both parties.

❖**Benefits** This pattern enables applications to *scale* across different protocols and delivery mechanisms. Each protocol is applicable for different communication requirements. For example, the first can accommodate *asynchronous notifications* (e.g., resynchronization messages) generated by the upstream modules. The second is best suited for *slow sinks* (e.g., disks) which cannot process the continuous-media as fast as the rest of the system. The third is very efficient for instances where shared memory is available and allows for processing at *different rates*.

The protocols avoid the need for dynamic synchronization policies by laying out structural rules about how different processing modules may communicate, therefore decreasing *complexity*. Adopting only one protocol ensures

```
class Message {
public:
  Message();
  virtual ~Message();
  virtual Serialize() =0;
  virtual Deserialize() =0;
  /*
  Other methods for messages
  */
};

class Payload: public Message {
public:
  Payload();
  virtual ~Payload();
  // Message interface
  virtual Serialize();
  virtual Deserialize();
  /*
  Other methods for messages
  */
  // Payload-specific interface
  int PriorityLevel() const;
  bool IsAsynchronous() const;
  /*
    Other descriptor-specific methods
  */
  // Composite interface
  virtual void AddMessage(Message *);
  virtual void RemoveMessage(Message *);
private:
  int _priorityLevel;
  bool _isAsynchronous;
  /*
    Other descriptor-specific data
  */
  vector<Message *> _components;
};
```

Figure 4. `Message` **and** `Payload` **classes.**

no deadlock, since no two modules can simultaneously send messages to each other.

❖**Liabilities** None of the 3 protocols is perfect. *Mixed solutions* that combine the advantages of more than one protocol are sometimes viable, creating a new protocol. For example, the upstream module can use the push protocol until the downstream module blocks communication. Then the pull protocol can be used, until the downstream module is ready to accept other payloads from the receiver.

Having more than one input port complicates *control flow* and requires *additional policies*. Two possible scenarios have been identified in the context of hardware data flow architectures. In the static model, a filter recomputes its output value each time a new payload is available at an input port. The dynamic model tags the payloads with context descriptors. A new output value is computed only when payloads with identical tags (context descriptors) are present at the input ports. The choice between the two models depends on the application's requirements. However, the overhead of payload matching sometimes makes the dynamic model infeasible.

Some protocols require additional *data repositories* (buffers). These have to be able to recognize high-priority payloads or asynchronous notifications. Section 2 describes a solution that enables payload identification while maintaining a decreased coupling between their contents and the buffers.

❖**Discussion** For the **push protocol**, the upstream module issues a message whenever new values are available. This mechanism may be *implemented* as procedure calls containing new data as arguments, as non-returning point-to-point messages or broadcasts, as prioritized interrupts, or as continuation-style program jumps.

The protocol is *applicable* in event-driven contexts, where the computation—flow—is initiated by an external event to the source or by a continuous loop in the source itself.

The upstream module does not know if the other end is ready to receive or not. To prevent data loss, the downstream module employs buffers to queue the incoming payloads. However, buffers (1) require additional scheduling policies, (2) introduce unpredictable delays and (3) are not viable if the payloads have large memory footprints.

The push protocol can also use buffers for *flow control* [18]. A high water mark limits the amount of payloads that can be stored in the buffer; the upstream module does not place data in the queue above this limit. When the queue exceeds its high water mark, the filter sets a flag and the upstream module stops sending data. When it notices this flag set and the queue drops below a low water mark, the filter reactivates the upstream module.

For the **pull protocol**, the downstream module requests information from the upstream module with a procedure/method call that returns the values as results. This mechanism can be *implemented* via a sequential protocol, may be multi-threaded with other requests on either side, and may perform in-place updates rather than returning results.

Payload transfer is initiated by the downstream module. Consequently, this protocol is *applicable* in demand-driven contexts and for instances where the upstream modules operate faster than the receiver.

Because there is no provision for the upstream modules to trigger a data transfer, pull protocols can not deal with *asynchronous notifications* or *prioritized payloads*.

The **indirect protocol** requires the availability of a shared repository that is accessible to the communicating parties. It can be *implemented* via transfers to shared memory which occur at fixed rates, or via polling. Whenever the upstream module is ready to pass a payload downstream, it writes it in the shared repository. When ready to process another input, the downstream module gets a payload from the repository.

The indirect model is *applicable* when the communicating modules process payloads at different rates. Assuming that not all payloads are required by the downstream module, the upstream module overwrites the contents of the shared repository before the downstream module reads it.

The performance depends on the nature of the repository. For systems that use *shared memory*, this protocol is quite efficient. However, the shared resource requires the additional overhead typically associated with synchronization problems—managing the critical sections. Shared memory is not necessarily available if the modules are located across hardware boundaries.

❖**Examples** The payload passing protocol in the VuSystem [10] has the following requirements: (1) reduced latency, which is equivalent to no buffering; (2) feed-back to upstream modules; and (3) no multi-threading (the VuSystem runs as a single-threaded process). Payloads are passed with one function call and the timing constraints are propagated through back-pressure. By temporarily refusing a payload, a downstream module slows down upstream processing. The mechanism is simple and does not require multi-threading. Other examples include the Presentation Processing Engine [17] and the Java Media Framework.

❖**Implementation notes** Figure 2 shows a push mechanism in the context of media flow architectures. Computation is triggered by calling the `Put()` method of the source.

The implementation for a pull mechanism is symmetric. `PullOutput`—Figure 5—is an abstract class which determines the output data type. Subclasses implement the `Get()` method to get the input data from the upstream module, process it and return the transformed value. `PullInput` determines the input data type, provides the module interconnection mechanism and maintains a pointer to the previous (upstream) module. This time, the `PullPort` class encapsulates the attachment between an input port and an output port.

Figure 6 shows the implementation of the IDCT module from Figure 3 following the pull model. This time, computation is triggered by invoking the `Get()` method of the sink.

4 Processing and control partitions

Multimedia applications have a dual functionality with divergent requirements. Continuous-media processing has to take place in a timely manner, according to its soft real-time deadlines. Control tasks like QoS monitoring, filter network management and user interaction have more relaxed requirements. They emphasize flexibility and ease of programming, rather than performance. Organizing the application according to these requirements yields the processing and control partitions.

```
template<class DataType> class PullOutput {
public:
  virtual ~PullOutput() { };
  virtual DataType Get() =0;
};

template<class DataType> class PullInput {
public:
  virtual ~PullInput() { };
  virtual void Attach(PullOutput<DataType>
                      *previous)
    {
      _port=new PullPort<DataType>(previous);
    };
  virtual void Detach()
    {
      delete _port;
    }
protected:
  inline DataType Input() {
    return _port->Input(); };
private:
  PullPort<DataType> *_port;
};

template<class DataType> class PullPort {
public:
  PullPort(PullOutput<DataType> *module)
    : _module(module) { };
  ~PullPort() { };
  inline DataType Input() {
    return _module->Get(); };
protected:
  PullOutput<DataType> *_module;
};
```

Figure 5. The `PullOutput`, `PullInput` and `PullPort` classes.

❖**Benefits** Partitioning the application increases *modularity*. The processing and control partitions are part of the same application and the inherent coupling between them cannot be overlooked. However, they can have different architectures and designs. As long as the overhead associated with inter-partition communication is small, they can even be implemented in different programming languages [10].

Removing QoS monitoring and control from the processing partition and localizing it within the control partition increases *flexibility* and improves *cohesion* within the control partition. The separation of processing and control also enables experimenting with and accommodating *new QoS parameters* without disrupting media processing.

The control partition orchestrates the *dynamic composition* inside the processing partition. This enables applications to adapt to various environments and *scale* with resource availability.

Another benefit of application partitioning is that *error handlers* can be placed in the control partition. In a media flow architecture, processing code is organized as filters which are loosely coupled with other parts of the system.

```
class IDCT: public PushInput<InBlock>,
            public PullOutput<OutBlock> {
public:
  UDCT() { };
  ~IDCT() { };
  OutBlock Get()
    {
      OutBlock ob;
      // compute the IDCT
      return ob;
    };
};
```

Figure 6. The `IDCT` processing module following the pull model.

The control partition has additional knowledge about the application and can interpret the notifications emitted by the processing code in the global context.

The control partition can be extended with user interfaces to obtain *visual programming tools*. These assist end-users in creating applications by establishing connections in the processing partition and do not require any programming expertise.

❖**Liabilities** The main problem with this organization is the *overhead* associated with the bidirectional inter-partition communication. However, the typical amounts of information exchanged between the partitions are small—the control partition sends control data, while the processing partition posts event notifications—and the communication cost is reasonable.

❖**Discussion** The **control partition** corresponds to timing control, QoS monitoring and management and user interface code. *Timing control* is employed by a Logical Time System (LTS) [7]. This is a user configurable clock that abstracts the time and encapsulates a mapping from real time to application time. The LTS schedules the continuous-media processing within the processing partition. *QoS monitoring and management* is performed by the QoS broker [14]. Factoring out the QoS code and localizing it within this partition facilitates *scaling* with the number of QoS parameters. *User interface code* can be generated automatically by software tools and has to cover a large number of possible actions which cannot be determined in advance [10].

Because it has to handle relatively infrequent events generated by the processing partition or corresponding to user actions, performance is not the main issue. Rather, *ease of programming* and *extensibility* are essential requirements. In media flow architectures, the control partition also arranges filters to operate cooperatively, without imposing centralized synchronization control.

The **processing partition** contains the code that performs media data processing according to the typical requirements for continuous-media applications [19]. This partition does not take into account any aspects of user interaction and the focus is only on *performance* (e.g., maintaining a set of QoS parameters, which are reported to the control partition). The code is usually subject to *instrumentation* and developers fine-tune the critical parts. Although the size of the processing partition can be smaller than the control partition (50%-80% of an interactive application is devoted to user interface aspects [9]), the code executes many times a second (e.g., every 33ms for a rate of 30 frames/second) and most of the running time of an application is spent here.

The control partition has knowledge of (some of) the *internal representations* used by the processing partition and, when required, translates them into a format understood by the user (e.g., a graphical representation). Likewise, by translating in the other direction, users can interact with the processing partition as well. For example, the control partition reads a color value from the processing partition (values denoting RGB or HSV components), converts it to coordinates and displays a cursor on a 2D color map. When the user makes a selection, it does the reverse conversion and writes the selected value back to the processing partition.

In multi-threaded systems, the two partitions can be implemented as *separate threads* (preferably lightweight, to ensure fast interpartition communication), with different priorities. For instances where the processing partition performs heavyweight, time-consuming computations, the application arranges to use the control *wait states* to perform computations for the processing partition.

❖**Examples** The Berkeley parallel MPEG–1 encoder [6] is partitioned in a similar way. The control partition is I/O intensive and consists of "master server" (which schedules slave processes), "decode server" (which directs the transfer of encoded frames between slave processes if decoded frames are used as reference frames) and "combine server" (which concatenates the encoded frames to create the output bit-stream). The processing partition is CPU intensive and consists of "slave processes" which perform frame encoding. Other examples include ActiveMovie [12], VuSystem [10] and the prototype from [11].

❖**Implementation notes** Inter-partition communication is bidirectional, and the exchange of information across the partition boundary is small compared to the traffic within the processing partition.

The processing partition posts event notifications and resumes execution as soon as the control partition receives them. This arrangement minimizes the interference between the two partitions. Figure 7 sketches the implementation of a simple file reader module. Each time the downstream module invokes the `Get()` method, it returns a character from the input file stream. When the end of file is reached, it also sends the `EofMsg` to the control partition.

```
class FileReader: public PullOutput<char> {
public:
  Random(const char *fName) {
    _inputFileStream=new ifstream(fName); };
  ~Random() { delete _inputFileStream; };
  char Get()
    {
      char c=_inputFileStream->get();
      if (c==EOF)
        Post(EofMsg::Instance());
      return c;
    };
  void SetFile(const char *newFName)
    {
      delete _inputFileStream;
      _inputFileStream=new ifstream(newFName);
    };
private:
  ifstream *_inputFileStream;
};
```

Figure 7. Inter-partition communication.

The `Post()` method—implementation not shown—delivers the message and returns immediately.

The control partition manages and reconfigures the processing partition. This is employed by sending control information to the appropriate processing modules. In the code example from Figure 7, the control partition invokes the `SetFile()` method of the file reader module. Generally, processing modules have to implement accessors and mutators for the information that needs to be read or modified.

Summary

We have presented 4 software patterns for continuous-media processing. Applications adopting these patterns *reuse* expert knowledge and *scale* well on several dimensions: (1) new **media transformations** are added by creating filters that implement them; (2) applications following the media flow architecture can be **distributed** on several processors with minimal modifications; (3) multiple **media streams** are supported by establishing connections from sources to sinks and encapsulating them in payloads; (4) new **applications** are obtained by connecting the existing modules in different ways; (5) adding a new **data type** requires implementing specific filters or reusing existing filters that operate on similar data; and (6) dynamic composition permits **adaptation** to different environments and resource availability.

These patterns are gradually being embraced by the industry. Their presence within the forthcoming Java Media Framework demonstrates their validity and confirms them as recurring solutions that have passed the test of time.

References

[1] P. Ackermann. Design and implementation of on object-oriented media composition framework. In *Proc. International Computer Music Conference*, Aarhus, September 1994.

[2] F. Buschman, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. John Wiley & Sons, July 1996. ISBN 0-47195-869-7.

[3] R. H. Campbell, V. Russo, and G. Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proc. USENIX C++ Workshop*, pages 109–123, Santa Fe, NM, November 1987.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[5] S. J. Gibbs and D. C. Tsichritzis. *Multimedia Programming—Objects, Environments and Frameworks*. Addison-Wesley, 1995. ISBN 0-201-42282-4.

[6] K. L. Gong and L. A. Rowe. Parallel MPEG–1 video encoding. In *Proc. Picture Coding Symposium*, Sacramento, CA, September 1994.

[7] M. H. Jackson, J. E. Baldeschwieler, and L. A. Rowe. Berkeley Continuous Media Toolkit API. Submitted for publication, September 1996.

[8] R. E. Johnson. Frameworks=(Components + Patterns). *Communications of the ACM*, 40(10), October 1997.

[9] T. Lewis, editor. *Object-Oriented Application Frameworks*. Manning, 1995. ISBN 1-884777-06-6.

[10] C. J. Lindblad. A programming system for the dynamic manipulation of temporally sensitive data. Technical Report 637, Massachutes Institute of Technology, August 1994. Laboratory for Computer Science.

[11] D.-A. Manolescu. Algebraic model and object-oriented architecture for hyper-media documents. Master's thesis, University of Illinois at Urbana-Champaign, 1997.

[12] Microsoft Corporation, Seattle, WA. *ActiveMovie Software Development Kit version 1.0*. `http://www.microsoft.com/devonly/tech/amov1doc/`.

[13] M. Mühlhäuser and J. Gecsey. Services, frameworks, and paradigms for distributed multimedia applications. *IEEE Multimedia*, 3(3), Fall 1996.

[14] K. Nahrstedt and J. M. Smith. The QoS broker. *IEEE Multimedia*, 2(1), Spring 1995.

[15] E. J. Posnak, R. G. Lavender, and H. M. Vin. Adaptive pipeline: an object structural pattern for adaptive applications. In *The 3rd Pattern Languages of Programming conference*, Monticello, IL, September 1996.

[16] E. J. Posnak, R. G. Lavender, and H. M. Vin. Presentation processing mechanisms for adaptive applications. In *Proc. Multimedia Computing and Networking*, San Jose, CA, February 1996.

[17] E. J. Posnak, R. G. Lavender, and H. M. Vin. An adaptive framework for developing multimedia software components. *Communications of the ACM*, 40(10), October 1997.

[18] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[19] R. Steinmetz and K. Nahrstedt. *Multimedia: Computing, Communications & Applications*. Prentice Hall, 1995. ISBN 0-13-324435-0.