A Data Flow Pattern Language

Dragos-Anton Manolescu*

1 Introduction

There have been several attempts to write patterns about the data flow paradigm [Edw95, Meu95, Sha96, BMR$^+$96], including variants specialized for particular domains [PLV96]. Based on the insights provided by these studies and on a different set of examples, in this paper I revisit the subject and define a pattern language. The major contributions of this approach are the following:

1. Within data flow architectures, I refine the granularity and identify 3 other patterns (Payloads, Payload passing protocol and Processing and control partitions) which are not restricted to data flow systems.
2. Although some of the issues associated with these patterns are also considered by other studies, discussing them outside the context of data flow systems provides more thorough coverage, offers new examples and encourages their use as stand-alone patterns.

Payloads is applicable to any instance where different entities communicate by exchanging messages. It facilitates decreased coupling between messages and the entities that operate with them. The Payload passing protocol offers several solutions for the inter-module message passing mechanism and could be employed by any modular application. Processing and control partitions organizes an application such that different architectural, design and implementation solutions for contradictory requirements can be married. Finally, some of the issues discussed in Data flow architecture and covered in more depth here are: (1) processing with two priority levels at the filter level, (2) changing the behavior at runtime, (3) dynamically adapting to different requirements, (4) improving performance with static composition and (5) facilitating visual programming tools.

The presentation follows the guidelines from [MD96]. Each pattern is illustrated with a fairly large number of examples from different areas, such as avionics systems, digital video, hardware, music composition and improvisation, multimedia, operating systems, parallel computers and scientific visualization. The domain diversity confirms the applicability of each pattern and helps readers with different backgrounds achieve a better understanding. Figure 1 illustrates the pattern language applied to the same application.

I believe that the pattern language offers a broader perspective over the various issues related to these patterns. For each of them, I have also tried to synthesize insights and conclusions from other authors. This approach also improves the clarity of the presentation, allowing the reader to focus on one problem at a time.

*National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, 152 Computing Applications Building, 605 East Springfield Avenue, Champaign IL 61820–5518, email: daman@ncsa.uiuc.edu.

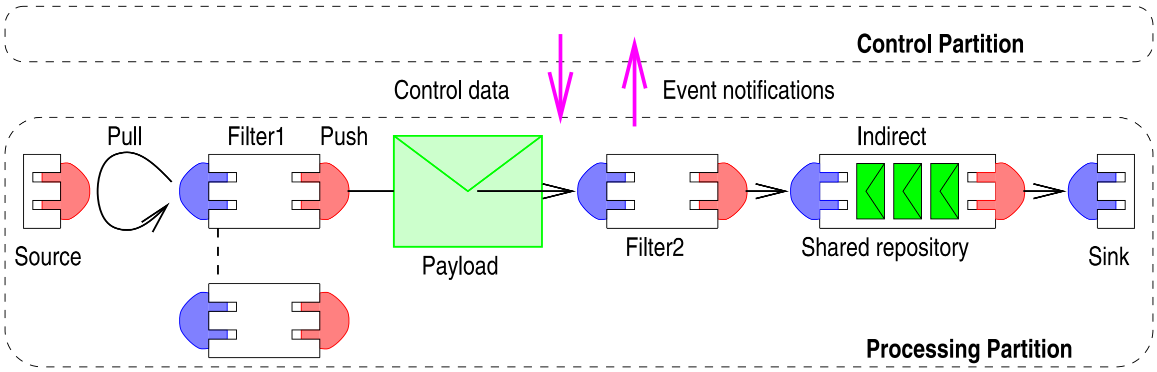

Figure 1: A snapshot of the four data flow patterns.

2 Pattern: Data flow architecture

Context

A variety of applications from different domains (scientific visualization, media processing, encoders and decoders, to name just a few—an exhaustive categorization is provided in [Lea96]) apply a series of transformations to a data stream. Their architectures emphasize data flow and consist of a set of interconnected processing modules (networks of modules) where control flow is not represented explicitly. The modules are self-contained entities that perform generic operations and can be used in a variety of application contexts. The application's functionality is determined by the following:

- types of modules within the network; and
- interconnections between the modules.

Therefore, processing modules are the computational units, while the network represents the operational unit.

Generally, the developers cannot anticipate all the possible ways the application will be used, or all the different environments in which it will run (e.g., cluster of workstations or palmtop computer). The resulting application could also be required to dynamically adapt to different requirements, while maintaining the same transfer function.

Problem

Some applications perform (in the same order) a number of operations on similar data elements. Sometimes they could also be required to dynamically change their functionality (e.g., apply different operations) and/or adapt to different requirements, without compromising performance. What architecture can accommodate these requirements?

Forces

- In some circumstances, a high-performance toolkit that is applicable for a wide range of problems from a specific domain is required;
- The application's behavior might need to change dynamically or adapt to different requirements at runtime;

- For complex applications, it is not possible to construct a set of components that cover all potential combinations;
- The loose coupling associated with the black-box paradigm usually has performance penalties. Generally, it is difficult to decouple the algorithmic properties of software modules from the entities using them, without decreasing performance (generic, context-free efficient algorithms are difficult to obtain);
- Software modules could have different, incompatible interfaces; share states; or need global variables;
- Specialized modules decrease the overhead of enforcing strict interfaces but also reduce the number of possible configurations.

Solution

Applications following the data flow architecture are organized as a network of processing modules[1] that apply a series of transformations to one or several data streams. Each module takes its input from an upstream module, performs a simple, generic transformation and passes the results to a downstream module. Enforcing strict and simple inter-module interfaces yields a large number of possible of combinations that provide solutions to many problems from a particular domain. This can also be regarded as an incarnation of the "divide and conquer" principle: instead of attempting to solve a complex problem all at once, split it into sub-parts and deal with each separately.

Inter-module communication is done by passing messages (sometimes called tokens) through unidirectional input and output ports, thereby replacing direct calls. Depending on the number of ports and their types, modules could be classified as follows (a comprehensive classification according to various criteria is available in [Lea96]):

Sources Usually they interface with an input device and have one or several output ports.

Sinks Interface with an output device and have one or several input ports.

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.

Unidirectional input and output 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 connections at the output port(s). Therefore, because any component depends only on the upstream modules, it is possible to change its output connections at runtime.

Two modules can be connected only if the output of the upstream module is plug-compatible with the input of the downstream module. Having more than one plug type—shown as a socket in Figure 1—decreases the number of possible configurations, and therefore the number of solvable problems.

Filter modules should take into account the following design guidelines:

- The transfer function should not model the domain, depend on a particular solution or have side effects.
- Each filter should be designed independently of its use.

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

- Filters should cooperate only by using the output of one as the input to another.

Decoupling filters from a particular problem increases their reusability and allows applications to use them as black-boxes. However, the black-box approach has performance penalties because it does not take into account context-specific factors. Good filters offer an even balance between these two forces.

Two priority levels are available for processing at the filter level. High priority processing can be done when the filter receives a message from the upstream module. Once acquired, the data can be processed at a later time, at lower priority. However, in both cases scheduling (i.e., "when" the actual processing takes place) is non-deterministic.

Applications following the data flow architecture can adapt to heterogeneous and dynamically changing environments. Different implementations for stateless filter modules are exchanged at runtime to adjust the resource cost and quality characteristics within some specified limits. 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 could be regarded as a composite [GHJV95] filter and replaced with an optimized version which trades flexibility for performance [PLV96]. Static composition provides the underlying application (e.g., compiler) with enough information to collapse a sequence of modules into a functionally equivalent primitive filter, reducing the overhead of inter-module communication.

Figure 2 illustrates an example of a data flow architecture with 2 sources, 1 sink and 6 filters. Filters 5 and 6 can be statically composed into filter 56, to improve performance. Static binding should be used to improve performance for filters that have low reconfigurability demands and are called with high frequency. Dynamic binding is best suited for filters that have high reconfigurability demands and low invocation frequency. A good balance between the use of static and dynamic composition allows for efficient implementation while maintaining a flexible, configurable architecture. For example, the framework from [PLV97] describes a JPEG decoder consisting of several basic filters—block decoder, inverse quantifier, IDCT, frame decoder and color converter. The block decoder is called thousands of times per image and its components are statically bound. The other components are called with lower frequencies and the overhead of dynamically dispatched method calls is tolerable. Ex?

Modules play a key role in data flow architectures. Applications that follow this pattern manifest an increased degree of modularity, which makes it very easy to distribute the development effort among different groups (e.g., visualization modules could be developed by the visualization group, etc.).

Users that have knowledge only about the application domain (e.g., scientific visualization) create new applications by simply connecting modules, without performing any other programming. As mentioned in [Foo88], this is usually done with visual programming tools [AEW96] which assist the creation of module networks.

The network normally triggers recomputations whenever a filter's output value changes. While manipulated interactively, it should be possible to temporarily disable the network, such that no output is computed until a valid, stable state is reached. Disabling computations is useful if several parameters need to be modified without having the filters recompute every time a change is made, or if processing takes a long time.

A data flow architecture is applicable only for data-driven applications, where the output is obtained by performing various sequential transformations to the input. The transfer function is

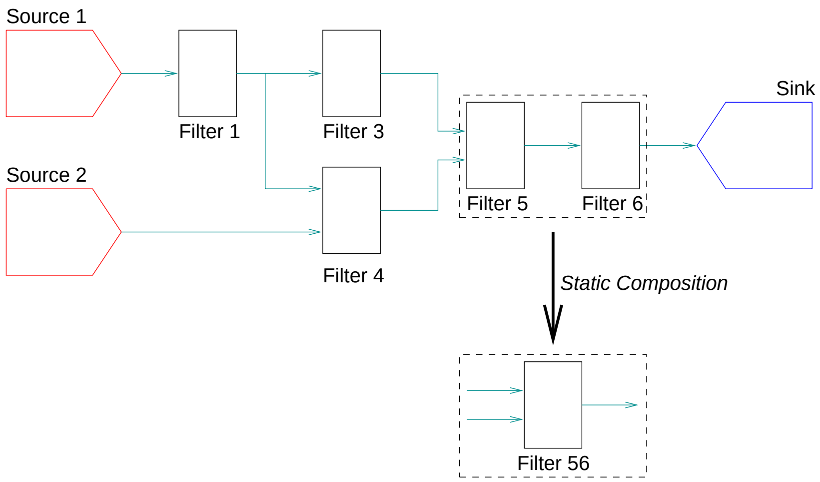

Figure 2: Data flow architecture.

determined by a combination of the individual transfer functions of each module. Control flow is distributed across the participating entities and is not explicitly represented at the architectural level.

The knowledge of domain experts is encapsulated within filters and could be reused in many different applications. Adopting this pattern facilitates the rise of end-user programming, automation and software components and emphasizes reuse at the filter level. Because the interaction mechanism between modules is fairly simple, the architecture is suitable for domain-specific frameworks and often is the subject of visual programming tools.

Data flow applications also manifest a high degree of scalability in several dimensions—number of transformations, processors, data streams (shown by the dotted line in Figure 1); and requirements.

The fine granularity decomposition into filters provides access to the data stream at various processing stages—e.g., between Filter4 and Filter5 in Figure 2. For multimedia or visualization applications it enables the use of signal processing techniques (e.g., translation, interpolation) which have been shown to improve performance [PLV97].

It is not a good choice for applications with dynamic control flow or feedback loops. For instances where the overhead of enforcing the inter-module interface is too high, a different architectural choice might be a better solution.

Signaling errors that occur inside filter modules is cumbersome and difficult [BMR+96]. Modules do not make any assumptions about their context and communicate only with other data-processing modules. Only when combined with some knowledge about the network topology and the application's domain could an error message generated by a module be meaningful. The Processing and control partitions provide one possible solution.

Resulting context

Applications following the data flow architecture can run on heterogeneous distributed systems. Because control flow is not explicitly represented, this change is transparent for sources, filters and sinks.

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

Figure 3: The `PushInput` class.

```
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;
};
```

Figure 4: The `PushOutput` class.

Implementation notes

This section follows the guidelines from [PLV96]. Processing modules inherit an input interface from `PushInput` and an output interface from `PushOutput`. `PushInput`—Figure 3—is an abstract class that determines the input data type. Subclasses implement the `Input()` method to process the input data and pass it downstream. `PushOutput`—Figure 4—determines the output data type and defines methods for dynamic composition—`Attach()` and `Detach()`. Attachment between two modules are abstracted by the `PushPort` class—Figure 5. This facilitates establishing one-to-many connections, in which case `PushOutput` has to be modified accordingly.

C++ templates parameterize the classes presented in Figures 3–5 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.

Based on these classes, implementing processing modules is straightforward. Figure 6 shows the code corresponding to a module that takes one `integer` input and outputs its square root. However, this simple implementation has several deficiencies with respect to how processing is initiated and how the data is passed between modules. Read on to find out more!

Type parameterization—templates—and function inlining allow to implement static composition. Figure 7 shows the code corresponding to a filter that increments its `integer` input. Processing—incrementing—is performed by the `Transform()` inline method. The `FastInc` class can

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

Figure 5: The `PushPort` class.

```
class Sqrt : public PushInput<int>, public PushOutput<double> {
public:
  Sqrt() { };
  ~Sqrt() { };
  void Input(int &i)
  {
    double d=sqrt(i);
    Output(d);
  };
};
```

Figure 6: An `Sqrt` processing module.

.

```
class FastInc : public PushInput<int>, public PushOutput<int> {
public:
  FastInc() { };
  ~FastInc() { };
  virtual void Input(int &i)
    {
      int j=i;
      Transform(j);
      Output(j);
    };
  inline void Transform(int &i) { i++; };
};
```

Figure 7: A filter for static composition. Only the inlined method is used by the composite filter.

parameterize the filter from Figure 8. Because `FastFilter` invokes only the inlined `Transform()` method, `FastInc`'s input and output ports are bypassed and the implementation is efficient. Figure 9 shows how to instantiate the static composition of `FastInc` with `FastFilter`. The resulting filter first increments the input and then outputs its square root.

Examples

1. The Application Visualization System (AVS) [AVS93] is a powerful visualization package that uses the data flow architecture. Different modules (file readers, data processing, etc.) are organized into a network with the aid of a visual network editor. AVS modules have the following characteristics:

   - Not all AVS modules are plug-compatible. There are several types of input and output ports, and the system does not allow two modules to be connected if their corresponding ports are not compatible.
   - Besides input and output ports, AVS modules also have a control port which generally allows parameterizing of some internal variable by the output of some other module. For example, the color of a plot could be determined by a module which compares the plotted value with a reference.

   Figure 10 shows an example of an AVS network.

2. Iris Explorer [EXP93] is a visualization program originally developed by Silicon Graphics. It allows users to visualize data and create visualization applications by connecting modules into a map. Explorer modules are small software tools, which could have several input ports, several output ports, and a module control panel which provides access to the module's functions. Whenever a module is fired, it processes the data it has received and passes it to the next module downstream. Firing can occur at several different stages of the map-building process. A module fires when:

   - it is launched in the Map Editor, provided it has all the data it needs on its input ports;
   - an upstream module fires, sending data downstream;

```
template<class T>
class FastFilter : public PushInput<int>, public PushOutput<double> {
public:
  FastFilter(T &component)
    : _component(component) { };
  ~FastFilter() { };
  virtual void Input(int &i)
    {
      int j=i;
      _component.Transform(j);
      double d=sqrt(j);
      Output(d);
    };
private:
  T _component;
};
```

Figure 8: A composite filter.

```
FastInc fi;
FastFilter<FastInc> ff(fi);
```

Figure 9: Instantiating the composite filter.

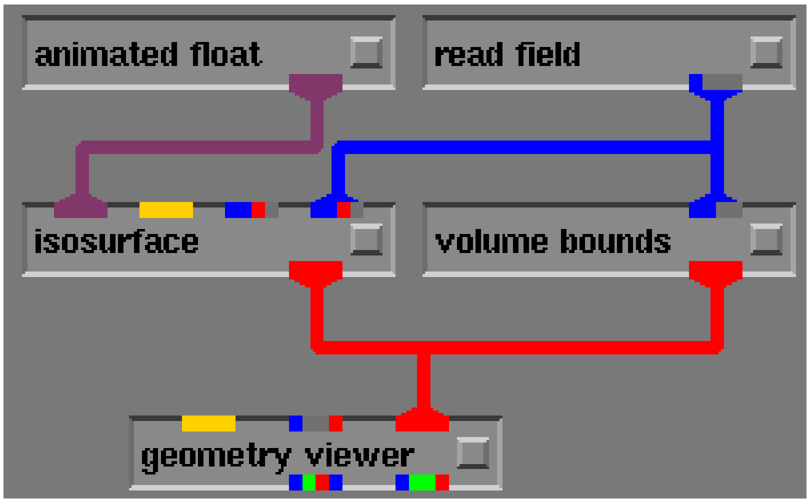

Figure 10: AVS Network. Top side ports are inputs, while bottom side ports are outputs. The port types are color-coded and the network editor does not permit connecting incompatible ports.

- one of its parameters is changed;
- it is fired manually;
- it is connected to an upstream module which has already fired and has data on its output port;
- it is triggered by an external event.

Module ports accept or output only one data type—lattice, pyramid, geometry, parameter or pick. The Map Editor permits temporary disabling of a module or a group of modules.

Another system component is the Module Builder, which is used to create modules without doing any programming beyond that needed to implement the module's function.

3. Data flow architectures have been investigated from a hardware perspective as well [Gur85, Pap91]. The execution model offers attractive properties for parallel processing—implicit synchronization of parallel activities and self schedulability. Unlike the von Neumann model which explicitly states the sequence of instructions, in the data flow model the execution of any instruction is driven by the operand availability. This emphasizes a high degree of parallelism at the instruction level. The Monsoon Project [Pap91] developed by MIT and Motorola produced a data-flow multiprocessor targeted to large-scale scientific and symbolic computation. Its success motivated much of the work on similar projects [NPA92] and contributed to spread the interest in data flow and parallel programming.

4. Avionics Control Systems (ACS) also use this pattern [Lea94]. ACS are complex systems and constructing a set of components that merge all possible combinations is not feasible. Rather, a means for combining different types of components (filters) together to serve a particular purpose is required—the data flow architecture. The algorithmic properties of most filter modules are independent of the entities using them; their functions are not designed in ways that are intrinsically coupled to any particular source or sink.

5. One of the most recent data flow architectures is Microsoft's ActiveMovie [AMS], which allows users to play digital movies and sound encoded in various formats. It consists of sources, filters and renderers connected in a filter graph. All graph components are implemented as Component Object Model (COM) objects. Filters have pins which are connected with streams, but other communication channels for specialized communication (e.g., error notifications) are available.

6. MET++ [Ack94] is a framework for multimedia applications. It supports the development of multimedia applications by providing reusable objects for 2–D graphics, user interface components, 3–D graphics, video, audio, and music. The audio resources are modeled as modules of a source-filter-sink architecture.

Variants[2]

1. The data flow architecture is used in the VuSystem [Lin94], with the same taxonomy as in "Solution." Because the *data flow in the network is continuous media* and has soft real-time requirements, Chris Lindblad calls it "*media flow architecture.*" VuSystem applications[3] illustrate very well the flexibility of this architecture. In the "room monitor," the output of a

[2] The differences between the data flow architecture and the variants discussed here are shown in italics.

[3] Some of these applications are available on the Web at `http://www.tns.lcs.mit.edu/vs/vusystem.html`.

surveillance camera is analyzed such that just the frames which contain motion are recorded. The "joke browser" extracts only selected parts (e.g., jokes) from the complete recording of a late night show. Although these two applications are very different (the former processes real-time video while the latter provides content-based access), both of them have been built with the same tool just by connecting existing modules.

2. The multimedia framework described in [GT95] also employs the media flow architecture. The framework uses a `Component` abstract class that represents media data processing modules. There are two kinds of components:

   Passive Passive components are characterized by *state* and *behavior*. Media processing is performed in response to messages sent to the object.

   Active Active components also have *state* and *behavior*. However, *processing is performed spontaneously*, even when no messages have been sent to the object.

   Each component has one or two ports, which allow several types of information to flow in the network. Like AVS, two modules could be connected only if the corresponding ports are plug-compatible. Connectors are stand-alone objects which each connect 2 modules.

3. Bill Walker's Ph.D. thesis [Wal94] describes a framework that uses this pattern in the context of computer-assisted music. Unlike typical data flow architectures, his system has *feed-back loops* and *shared state*. The modules are called `Composers` and `Transformers`, and global information independent of the components is encapsulated within a `PolicyDictionary` object.

4. In [Rit84], Dennis Ritchie describes a variant of this pattern and how it is implemented in the UNIX stream system. Data flow is *bidirectional* and modules have queues for each direction. The queues are also employed for flow control—Section 4.

Related Patterns

- Inter-module communication is done through messages, which could contain either data or control information. Payloads describes how messages encapsulate different types of information.

- Data flow architectures require modules to pass information to each other. Payload passing protocol provides several options for this mechanism.

- A separate part of a data flow architecture system is in charge of coordinating the modules (for dynamic composition) and handling error messages. Processing and control partitions describes how to organize such applications.

- Applications following the Layers architectural pattern [BMR+96] also manifest modules which exchange data between them. However, in this case each module corresponds to a different abstraction level and the data has different semantics on each layer—of course, the Data flow architecture could be used within individual layers.

- Static composition is an instance of the Composite pattern [GHJV95], which ensures a consistent interface for primitive (e.g., statically composed filter) and container (e.g., sequence of filters) objects.

- Streams [Edw95], Pipes and filters [Meu95, BMR+96] and Pipeline [Sha96] provide similar solutions but are subsets of the pattern described here.

3 Pattern: Payloads

Context

Sometimes separate entities[4] need to exchange information with each other. This could be done either by sending messages (payloads) through a communication channel which exists between them or with direct calls. If communication is restricted to message passing (as in Data flow architectures), then payloads encapsulate *all* types of information—data, control, etc. The communicating entities need a mechanism to distinguish what type of information corresponds to each payload, as well as other other message attributes like priority level, asynchronous event, etc.

It is also possible that the communication channel does not directly support payload transfer—for instance, the communication is done through a serial connection. These sorts of details should be transparent for the entities carrying on a dialogue.

Some overhead is associated with each message transfer. Depending on the communications profile (frequency and amounts of data exchanged), unless the payload transfer mechanism is optimized, it could become prohibitively expensive.

Problem

Entities send and receive various types of information. How is it possible to ensure a low coupling between them and the different message types, while allowing communication over different channels and reducing the message passing overhead?

Forces

- Communication is restricted to messages;
- The coupling between an entity and the data that is passed to it has to be reduced;
- Messages contain different types of information (e.g., control, data);
- An entity might give special treatment to certain messages (e.g., asynchronous events, high-priority messages);
- The communication channel might not directly support the format of the messages;
- Copying blocks of data is an expensive operation.

Solution

Direct calls are not feasible for communication between entities located across logical boundaries or that need to exchange complex data. For these instances, all the information is packaged in <u>payloads</u> (messages) which are logically passed between the communicating parties—shown as envelopes in Figure 1.

[4] In this context, "entity" is used in a general way, to denote any software component that exchanges data with another—for example, modules within the same application, or peer layers in a layered architecture.

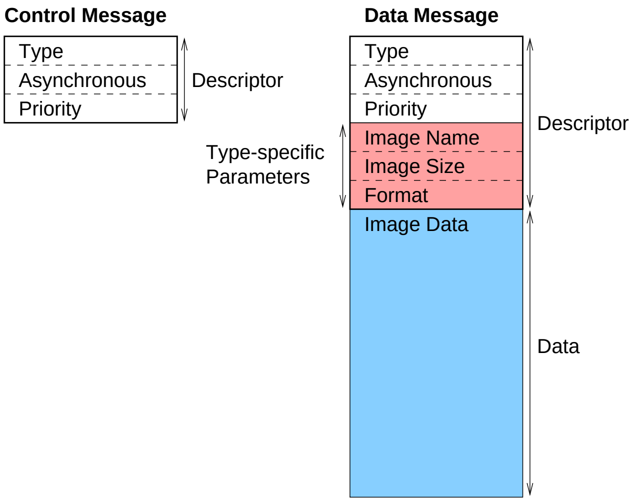

Figure 11: Different payloads and their components.

Payloads encapsulate various types of information sent from one module to another. They provide an abstract model that allows to identify the type of information contained within each message, as well as its attributes. A decreased coupling between entities and messages facilitates the optimization of the message passing mechanism. Transfer mechanisms that require additional information about messages (Section 4) are able to obtain it directly from the message, through a well-defined protocol.

Payloads are composed of two components, a descriptor component and a data component:

Descriptor component Descriptors contain general information about the payload—type,[5] asynchronous event, priority level—as well as type-specific parameters—image size, if the data component corresponds to an image.

Data component Some payloads have a data component, and the data type corresponding to this component is determined from the descriptor. Control payloads do not have a separate data component. Generally, the data component is much larger than the descriptor.

Figure 11 shows the components of a control message and a data message corresponding to an image. In the simplest form, the descriptor component contains just a tag that identifies the payload's type. If the length of the content is not known[6] and can not be encoded within the descriptor, the end of contents has to be signaled explicitly.

For example, in a data 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.

If the two entities communicate over a high speed serial channel (e.g., Ethernet), payloads need to provide serialization and deserialization methods. These encode them into a reliable byte stream at the transmitter side and decode them from a byte stream at the receiver side. Besides allowing distribution of the architecture over several networked computers (which need not be homogeneous), these methods also facilitate the checkpointing of the entire system.

[5]If an object-oriented language is used for implementation, the payload's type can be determined from its class and there is no need to encode it separately.

[6]Also known as "Open LDU" (logical data unit), as opposed to "Closed LDU," for which the length is known.

Payload copying is expensive because of its high overhead. For example, instead of copying the data, the originating entity could pass to the receiver a reference (or pointer). If the fan-out is larger than 1, the payload has to be cloned such that each receiver gets its own copy. But cloning wastes space and might not even be viable if the payloads have large memory footprints. Under these circumstances, the receiver entities could share the same payload. Should a receiver modify the incoming payload, it has to obtain its own private copy.

Whenever it is not possible to avoid copying, then an optimized technique could be employed:

Shallow copy Copy just the descriptor and share data components. However, entities are not allowed to modify the copy.

Deep copy Copy the data components as well. To increase efficiency, they could be implemented as copy-on-write.

If the entities that exchange the data reside across hardware boundaries, the change of ownership requires the entire payload to be transferred.

The main liability of this pattern is its inefficiency when compared with direct calls. This is due to the high overhead associated with the message passing mechanism. One possible way to improve performance is to pack multiple messages into one container message such that all of them are transferred in one step—the Composite message pattern [SC95] uses the same principle; however, rather than improving performance, its main objective is to abstract the communication structure. This approach is well suited to high message rates or bursty traffic, such that the sender does not have to hold a message too long before passing it further. It has been successfully applied in operating systems which consist of several interacting components [TW97, CRJ87].

The Payloads pattern allows to modify the message contents at runtime. Modules could insert or remove message components before passing them further. Consequently, the traffic could be reduced by piggybacking information on passing messages. Another benefit is that adding new message types does not require changing the existing entities which are not interested in them. For example, in a Data flow architecture (Section 2), filters pass downstream the messages they do not understand, without performing any processing—also known as "tunelling." Payloads increase the overall flexibility and permit the addition of new functionality and/or features (e.g., priority levels, support for asynchronous events, etc.) with minimal changes.

Implementation notes

A flexible implementation solution is to regard the payload as a composite message [SC95]. Concrete classes corresponding to various message types are derived from the abstract class `Message`—Figure 12—and provide implementations for its interface. The `Payload` class—Figure 13—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.

Examples

1. VuSystem's modules [Lin94] communicate by exchanging payloads. Examples of payloads are `VideoFrame` for single uncompressed video; `AudioFragment` for a sequence of audio samples; and `Caption` for close-captioned text. Descriptor members common to all payloads are:

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

Figure 12: The `Message` abstract base class.

```
class Payload : public Message {
public:
  Payload();
  virtual ~Payload();
  // Message interface
  virtual Serialize() { /* */ };
  virtual Deserialize() { /* */ };
  /*
  Other methods for messages
  */
  // Payload-specific interface
  int PriorityLevel() const { return _priorityLevel};
  bool IsAsynchronous() const { return _isAsynchronous};
  /*
    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 13: The `Payload` class.

Channel Channel number, which is used whenever payloads corresponding to different sources are multiplexed on the same channel.

StartingTime Indicates the time when a payload is valid. It is initialized when the payload is recorded, and then modified by `VsRetime` filters.

Duration Indicates for how long the payload is valid. The `VsRetime` filter modifies this value at runtime to allow presentations at a different rate than the one used for capture.

The data component could be image data or the compressed image data for a video frame, a sequence of audio samples, or text corresponding to close-captions.

Data components are located in shared memory segments. This facilitates inter-process communication and improves performance. (The VuSystem uses the MIT-SHM X extension.) However, whenever payloads need to be copied, one of the optimized techniques mentioned in "Solution" is employed.

The payloads also provide marshalling/unmarshalling methods which allow VuSystem modules to reside on different computers.

2. Abstract Syntax Notation One (ASN.1) [X2088a] is a method of specifying abstract objects in the Open Systems Interconnection (OSI) architecture. The Basic Encoding Rules [X2088b] describe how to encode values of each ASN.1 type as a string of bytes. BER encodes each part of a value as a *(identifier,length,contents)* tuple. The identifier and the length correspond to the descriptor component, while the contents correspond to the data component. Although ASN.1 and BER have been designed for the application-presentation interface within the OSI architecture, they have also been applied for distributed systems.

3. In ActiveMovie [AMS], payloads are either media samples or quality control data. Media data originates at the source and is passed only downstream. Quality control data provides a means to gracefully adapt to load differences in the media stream. It is used to send notification messages from a renderer either upstream, or directly to a designated location. All the elements of the architecture architecture recognize an asynchronous event which requires graceful flushing of old data, followed by global resynchronization.

4. In UNIX streams [Rit84], the objects that flow in the network are message blocks. A header specifies their type (data or control), as well as other attributes (e.g., asynchronous event).

5. The Message Passing Interface (MPI) [SOHL+96] is a standardized and portable message passing mechanism which runs on a wide variety of parallel computers. MPI messages call the descriptor component "message envelope."

Related Patterns

- Many applications that adopt the Data flow architecture use payloads to encapsulate the data that flows through the network.

- Marshalling and unmarshalling methods require payloads to reconstruct themselves from a byte stream. A Factory method [GHJV95] could be employed at the receiver's end to reconstruct the appropriate type of payload.

- The Composite message pattern [SC95] facilitates the packaging of several messages into a composite, thus improving performance by reducing the message passing overhead.

- Payload copying can be optimized using the Proxy [GHJV95, BMR+96] pattern. A proxy payload has the same interface but postpones the data transfer until its data or descriptor components are needed.

4 Pattern: Payload passing protocol

Context

Collaborating logical modules communicate by passing Payloads between them. Depending on the application's domain, the requirements could be very different. Some applications process asynchronous events (e.g., user input) or assign different priority levels to various requests. Other applications manipulate large amounts of data (e.g., images) or time-sensitive information (e.g., audio or video).

Sometimes the receiving module operates slower than the modules which send data to it. For instance, the speed of a module that writes to disk is determined by the hardware, which usually is one of the slowest subsystems in a computer. To avoid data loss, the slow module needs to be able to throttle down the other modules, thus determining flow control.

Therefore, the inter-module data transfer mechanism is subject to the application's requirements and has to take place in a transparent manner.

Problem

Applications have domain-specific requirements. How can payloads be transferred between modules in a way that is compatible with the application's requirements?

Forces

- Large payloads render buffering inviable;
- Payloads that contain time-sensitive data have to be delivered in a timely manner, such that no deadlines are violated;
- Asynchronous or prioritized events are sent from one module to the other;
- Shared resources for inter-module communication (e.g., shared memory) might not be available, or the overhead associated with synchronization is too expensive;
- Flow-control has to be determined by the receiving module.

Solution

The payload passing protocol describes several <u>ways to assign control flow</u> among modules that communicate through unidirectional channels. They avoid the need for dynamic synchronization policies by laying out structural rules about how different modules communicate, thus decreasing complexity.

Three different <u>mechanisms</u> to assign control flow among modules that exchange Payloads are the following:

Pull (functional) 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.

In the pull model, payload transfer is initiated by the receiving module. Therefore, it is applicable in demand-driven contexts and instances where the senders operate faster than the receiver.

Because there is no provision for the sending module(s) to trigger a data transfer, this mechanism can not deal with asynchronous or prioritized events.

Figure 1 shows the pull model in a data flow application—between the Source and Filter1. This is also illustrated separately in Figure 14.

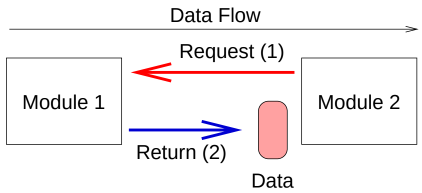

Figure 14: The pull model for inter-module communication.

Push (event driven) 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 push model 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 sending module does not know if the other end is ready to receive or not. To prevent data loss, the receiving module uses a data repository to queue the upcoming messages. If the system processes asynchronous events or high priority messages, the data repository must identify and pass them to the downstream module ahead of low priority messages. While a simple FIFO queue is suitable for the former situation, the latter requires a priority queue. Payloads provides one possible solution for message identification. However, data repositories require additional scheduling policies, introduce unpredictable delays and are not viable if the payloads have large memory footprints.

A push mechanism can also use the data repositories for flow control [Rit84]. A high water mark limits the amount of payloads that can be stored in the repository; the sending module does not place data in the queue above this limit. When the queue exceeds its high water mark, it sets a flag and the upstream module stops sending data. When the downstream module notices this flag set and the queue drops below a low water mark, it reactivates the sender.

The data flow application from Figure 1 employs the push model between Filter1 and Filter2. The transfer protocol is also illustrated in Figure 15.

Indirect (shared resources) This model requires the availability of a shared repository (sometimes called mailbox, pipe [BMR$^+$96] or put/take buffer stage [Lea96]) that is accessible to

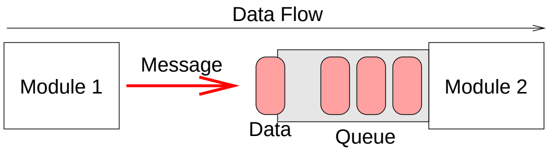

Figure 15: The push model for inter-module communication.

both modules. This mechanism can be implemented via transfers to shared memory which occur at fixed rates, or via polling. Whenever the sender module is ready to pass a payload to the receiver, it writes it in the shared repository. When ready to process another input, the receiver module gets a payload from the repository.

The indirect model is applicable when the sender and the receiver process payloads at different rates. Assuming that not all the payloads are required by the receiver, the upstream module overwrites the contents of the shared repository before the downstream module reads it. This is particularly important for instances where the payloads contain time-sensitive information. For example, assume a digital video camera which captures frames at a high rate and makes them available in a shared buffer. If the frame rate is not critical for the rest of the system, then the next module reads frames from the shared buffer at a different, lower rate. Ex?

The performance depends on the nature of the repository. For systems which use shared memory it is quite efficient. However, the shared resource requires the additional overhead typically associated with synchronization problems [SPG91]—managing the critical sections. Shared memory is not necessarily available if the modules are located across hardware boundaries.

The indirect model is illustrated in Figure 1 between Filter2 and Sink, and separately in Figure 16. The square waves show how the two modules access the shared repository at different rates (the writes and reads are shown as arrows).

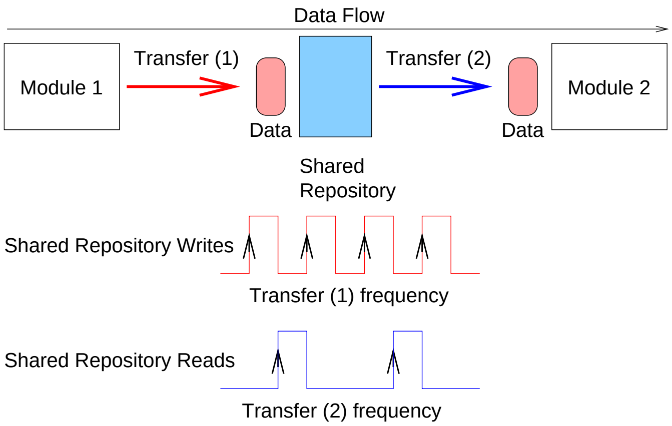

Figure 16: The indirect model for inter-module communication.

In the context of concurrent systems [Lea96] identifies two basic forms of flow policies (pull-based flow for demand-driven applications and push-based flow for event-driven contexts), as well as a mixed strategy (exchange). The rest of this section provides some additional details with general applicability and shows the relationships with other patterns.

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

Figure 17: The `PullOutput` class.

Depending on how flow control is implemented, data repositories introduce an additional problem: if there is no message that marks the end of stream or the data repository cannot detect it, the last messages need to be automatically flushed by a timeout mechanism. The Payloads pattern facilitates the detection of this situation while maintaining a loose coupling between messages and data repositories.

Each of the three models described below has its own problems. Mixed mechanisms that combine the advantages of more than one method are sometimes viable. For example, the upstream module can use the push model until the downstream module blocks communication. Then the pull model can be used, until the downstream module is ready to accept other messages from the receiver.

Having more than one input port complicates flow control and requires additional policies. Two possible scenarios have been identified in the context of hardware data flow architectures [Den80, Pap91]. 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 associative memory required by the latter and the dynamic token matching overhead sometimes make this scenario not feasible.

Implementation notes

Figures 3–5 show a push mechanism in the context of data flow architectures. The network is constructed starting from the source—the `Attach()` method establishes a connection from an output port to an input port. Computation is triggered by calling the `Input()` method of the source.

The implementation for a pull mechanism is symmetric. `PullOutput`—Figure 17—is an abstract class which determines the output data type. Instead of taking an argument, now the `Output()` method has a return value. Subclasses implement it to get the input data from the upstream module, process it and return the transformed value. `PullInput`—Figure 18—determines the input data type, provides the module interconnection mechanism and maintains a pointer to the previous (upstream) module. This time, the `PullPort` class—Figure 19—encapsulates the attachment between an input port and an output port.

Figure 20 shows the implementation of the processing module from Figure 6 following the pull model. This time, computation is triggered by invoking the `Output()` method of the sink.

Implementing the indirect model goes along the same lines, but adds the shared repository (e.g., a circular buffer) between the input port and the output port.

For simplicity's sake, several important features have been removed from the classes shown in Figures 3–5 and 17–19. First, none of them employs buffering. An easy way to add buffers is to use one of the container classes (e.g., `Stack`) provided by the Standard Template Library [MS96]. Second, the payloads are primitive types. They are not shareable (for the push protocol) and are

```
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;
};
```

Figure 18: The `PullInput` class.

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

Figure 19: The `PullPort` class.

```
class Sqrt : public PullInput<int>, public PullOutput<double> {
public:
  Sqrt() { };
  ~Sqrt() { };
  double Output()
    {
      return sqrt(Input());
    };
};
```

Figure 20: The `Sqrt` processing module following the pull model.

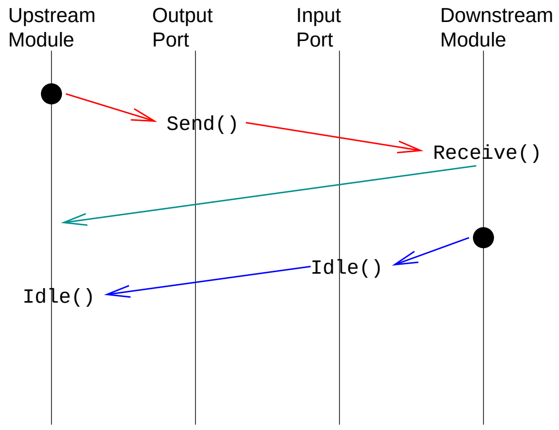

Figure 21: Inter-module communication protocol in the VuSystem (Example 1).

copied each time they cross the inter-module boundary (for the pull protocol). Payloads discusses how several optimized techniques could be employed to address these problems.

Examples

1. The VuSystem [Lin94] module data protocol has the following requirements:

   - Reduced latency, which is equivalent to no buffering.
   - Feed-back to upstream modules.
   - No multi-threading (the VuSystem runs like a single-threaded process).

   To pass a payload, the upstream module calls the `Send()` method on its output port, which calls the `Receive()` method of the downstream module. If the downstream module accepts the payload, the call to `Receive()` succeeds (returns `True`). The downstream module could indicate that it is not ready for new data by returning `False` from `Receive()`. Upon receiving `False` from `Send()`, the upstream module stops trying to send data.

   Whenever the downstream module becomes ready for more data, it calls the `Idle()` method on its input port. This translates into a call to the `Idle()` method on the upstream module, that sends any waiting data to the downstream module.

   Figure 21 illustrates the methods that are called for inter-module communication. Therefore, payloads are efficiently passed with one function call. 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. If a module has little work to do, it can perform everything within the `Receive()` method; otherwise, it can schedule processing later—see the discussion about the 2 priority levels in Data flow architecture.

2. Data flow hardware also employs push and pull mechanisms. For example, consider the arithmetic expression $e = ((a + 1) \times (b - 6))$. A reduction machine takes a bottom-up approach, computing first the subexpressions $a + 1$ and $b - 6$, and finally the multiplication which yields the result. This eager evaluation corresponds to the push model. A demand-driven computation takes a top-down approach. It begins by requesting the value of e, which

triggers the evaluation of the product. This in turn triggers the addition and the subtraction. The lazy evaluation corresponds to the pull model.

3. In ActiveMovie [AMS], the pins are responsible for providing interfaces to connect with other pins and for transporting the data. These interfaces transfer time-stamped data, negotiate data formats and maximize throughput by selecting an optimal buffer management and allocation scheme. Separate protocols are implemented for media samples, quality management, end-of-stream and flushing.

4. The UNIX stream input-output system [Rit84] uses a message queue and water marks for flow control.

Related Patterns

- Data flow architecture uses this pattern to transfer messages between modules.

- Data repositories need various types of information about the messages they store (e.g., priority level, etc.). Payloads ensure uniform access to this information, while maintaining a loose coupling between repositories and messages.

- Adjacent layers within the Layers [BMR+96] pattern communicate with each other. The Module data protocol could be used to exchange data across layer boundaries.

- To ensure a loose coupling between communicating modules, the notification mechanism could be implemented with the Observer [GHJV95] pattern.

5 Pattern: Processing and control partitions

Context

An interactive application has a dual functionality:

1. On one hand, the application *interfaces with the user*. This handles the event-driven programming associated with user interfaces. Usually it is built on top of a toolkit that offers an application programming interface (API) to the underlying graphics subsystem, as well as a set of "standard" widgets (e.g., menus, buttons). To satisfy the user's visual perception, the response times need to be on the order of hundreds of milliseconds.

2. On the other hand, the application *handles the data processing* according to its specification. Different domains have different requirements. For example, numerical applications emphasize correctness and computational speed; likewise, multimedia applications work with time-sensitive data and have to ensure that processing takes place in a timely manner. Domain-specific libraries and toolkits are employed for various tasks, including the use of available hardware resources (e.g., math co-processors, graphics accelerators).

The problems addressed by the same application belong to different domains and therefore have different requirements. Optimal architectural, design and implementation solutions for these requirements are disparate and sometimes contradictory. Depending on the overall requirements, the final solution is usually best suited for one of the two aspects, compromising the other.

Problem

The same application has different functionalities with divergent requirements. How can the conflicting architectural, design and implementation solutions be combined without compromising any of them?

Forces

- User actions and their sequence are non-deterministic; therefore, user interface code has to cover a large number of possibilities;
- Data processing has strict requirements and the sequence of operations (the algorithm) is known in advance;
- Human users require response times on the order of hundreds of milliseconds, and user-interface code usually has to wait for user intervention;
- Applications emphasize performance (e.g., correctness, accuracy, etc.) that is irrelevant for the user interface (e.g., multimedia applications display new frames every 33 milliseconds);
- Generally, a large fraction of the running time is spent waiting for user input;
- The user interface code and data processing code are part of the same application and they collaborate with each other.

Solution

Organize the application into two different partitions, according to their requirements:

Control partition The control partition is closer to the end-user. The rest of this section assumes that the control partition handles user interaction.[7]

Commonly, 50%-80% of an application is devoted to user interface aspects [Lew95] and has to cover a large number of possible actions which can not be determined in advance.

Usually, the control partition corresponds to the event-driven code that handles various types of system messages. (User actions like key presses, mouse movements or mouse button clicks are transformed into system messages as well.) Consequently, the code executes only when performing user interaction. Generally, this accounts for a fraction of the total execution time.

Because it has to handle relatively infrequent events and most of the time it waits for user input, performance is not the main issue. Rather, ease of programming and powerful visual expression are essential requirements. Sometimes parts of this code are automatically generated by software tools.

Processing partition This partition is closer to the machine. It contains the code that performs data processing according to the application's requirements.

[7] The partitioning process is not limited to user interfaces. Rather, *it is suitable for any application that has a dual functionality with divergent requirements.* Quality of service (QoS) monitoring in multimedia applications is another example. The unique requirements associated with user interface code make it an excellent example for describing the pattern.

Unlike in the control partition, control flow is determined in advance. The code is implemented according to well-defined requirements which completely describe its functionality.

Although the size of the code is small compared to the control partition, most of the running time of an application is spent here.

This partition does not take into account any aspects of the user interaction, and the focus is only on performance, as defined by the application's domain (e.g., processing speed, numerical accuracy, real-time deadlines). Usually it is subject to instrumentation, which allows the developers to examine the execution profile and eventually fine-tune the critical parts.

The processing and control partitions are part of the same application and the inherent coupling between them can not be overlooked. However, they could have different architectures, designs, and could even be implemented in different programming languages, as long as the overhead associated with the inter-partition communication does not affect their performance or increase complexity too much.

The control partition has knowledge of (some of) the internal representations used by the processing partition and, if required, could translate these into a format understood by the user (e.g., a graphical representation). It could also allow a user to interact with the in-band partition, translating in the other direction as well. For example, most interactive drawing programs allow Ex? the selection of a color by positioning a cursor over a color map. When the color selection tool is brought up, the control partition reads the current color value from the processing partition (values denoting RGB or HSV components), converts it to color map coordinates and displays the cursor. When the user makes a selection, it does the reverse conversion and writes the selected value back to the processing partition.

Partitioning the application increases flexibility and modularity. In multi-threaded systems, the two partitions could be implemented as separate threads (probably lightweight, to ensure fast interpartition communication), with different priorities. For instances where the processing partition performs heavyweight, time-consuming computations, the application could arrange to use the control partition's wait states to perform computations for the processing partition.

Usually different development teams work on each partition. For example, user interaction could benefit from expertise in human-computer interaction, while the data processing is best understood by domain experts. The partitions help to distribute the development effort. They also facilitate the generation (from the same sources) of non-interactive applications suitable for batch execution with job queuing systems, or porting the application to another system with a different user interface toolkit.

The control partition also plays an important role in the context of visual programming. In such systems [AVS93, EXP93], it assists users in assembling, controlling and manipulating the processing partition without requiring programming expertise.

Another benefit of application partitioning is that some error handlers can be placed in the processing partition. It is realistic to assume that the processing code is made up of several reusable software components, which are loosely coupled with other parts of the system. The control code has additional knowledge about the application and can be regarded as a Mediator [GHJV95] that coordinates various components and interprets error messages in the global context. This aspect is particularly useful for applications following the data flow architecture.

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 notifictions (See Figure 1)—and the communication cost is reasonable.

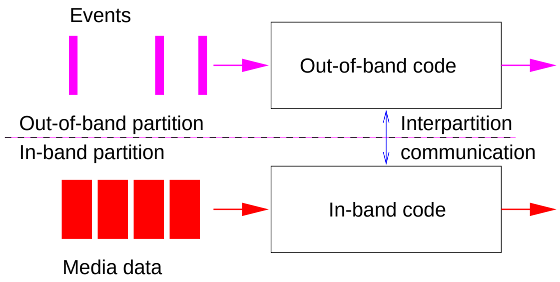

Figure 22: Control and processing partitions within an application.

Figure 22 illustrates the two partitions within an application. The rate and size of the data blocks which are fed to each partition show their different requirements.

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 code within the processing partition posts event notifications that allow to resume execution as soon as the control partition receives them. This arrangement minimizes the interference between the two partitions. Figure 23 sketches the implementation of a simple file reader module. Each time the the downstream module invokes the `Output()` method, this returns a character from the input file stream. When the end of file is reached, it also sends the `EofMsg` to the control partition. The `Post()` method (its implementation is not shown) only 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 23, 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 need to be read or modified.

Examples

1. The VuSystem [Lin94] defines in-band and out-of-band partitions. They have different architectures, designs, and are also implemented in different programming languages. Their characteristics are summarized Table 1.

2. In ActiveMovie [AMS] the graph manager (control partition) connects filters and controls the media stream. It also has the ability to search for a configuration of filters that will render a particular media type and automatically build the corresponding graph. The stream control architecture allows the graph manager to communicate with individual filters and to control the movement of data through the filter graph. An error detection and reporting protocol defines how errors are handled by filters and propagated to the graph manager. Because ActiveMovie is also a software development kit, the graph manager provides a set of COM interfaces to allow communication between the filter graph and other applications.

3. AVS [AVS93] and Iris Explorer [EXP93] are interactive visual programming tools which rely heavily on the control partition. They offer full access over the processing partition, which can be built, configured and operated with from the user interface.

```
/* This class corresponds to EOF messages posted by the
processing partition and implements the Singleton pattern.
*/
class EofMsg;

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

Figure 23: Inter-partition communication. The processing partition posts notification messages, while the control partition sends control information.

Characteristic	Out-of-band	In-band
Objective	Handle user interface and event-driven program functions.	Low-level media processing.
Emphasis on	Ease of programming.	High-performance.
When?	Awaits user input or other events and performs actions based on events.	Repeated actions that occur many times a second.
How?	Much of the code rarely executes in a given session, because it is built to handle many possible situations.	Small part of the application code which executes most of the running time.
Tool	Tcl/TclXt/TclAw	C++

Table 1: Partitions within the VuSystem.

Variants

1. This pattern is also present in [Wal94]. `Improvisor` objects (control) coordinate different `Composer` and `Transformer` objects (processing). In this case, the out-of-band partition *does not handle user interfaces.*

2. The Berkeley parallel MPEG–1 encoder [GR94] 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.

Related Patterns

- Data flow architectures consist of a set of interconnected modules which could be regarded as the processing partition. The control partition orchestrates the dynamic composition within the processing partition. This enables applications to adapt to various environments and scale according to resource availability.

- The Layers architectural pattern [BMR+96] also partitions an application such that each division corresponds to a different level of abstraction. Each layer defines interfaces for the one above it and uses the services provided by the one below. Unlike Control and processing partitions, the objective is to *ensure a loose coupling between the divisions* such that one could be changed (eventually at runtime) without affecting the others.

References

[Ack94] Philipp Ackermann. Design and implementation of on object-oriented media composition framework. In *Proc. International Computer Music Conference*, Aarhus, September 1994.

[AEW96] Philipp Ackermann, Dominik Eichelberg, and Bernhard Wagner. Visual programming in an object-oriented framework. In *Proc. Swiss Computer Science Conference*, Zurich, Switzerland, October 1996.

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

[AVS93] CONVEX Computer Corporation, Richardson, TX. *ConvexAVS Developer's Guide*, first edition, December 1993.

[BMR+96] Frank Buschman, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. John Wiley & Sons, July 1996. ISBN 0-47195-869-7.

[CRJ87] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proc. USENIX C++ Workshop*, pages 109–123, Santa Fe, NM, November 1987.

[Den80] J. B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11), 1980.

[Edw95] Stephen Edwards. *Streams: A Pattern for "Pull-Driven" Processing*, volume 1 of *"Pattern Languages of Program Design"*, chapter 21. Addison-Wesley, 1995. Edited by James O. Coplien and Douglas C. Schmidt. ISBN 0-201-60734-4.

[EXP93] Silicon Graphics, Inc., Mountain View, CA. *IRIS Explorer User's Guide*, 1993.

[Foo88] Brian Foote. Designing to facilitate change with object-oriented frameworks. Master's thesis, University of Illinois at Urbana-Champaign, 1988.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995. ISBN 0-201-63361-2.

[GR94] Kevin L. Gong and Lawrence A. Rowe. Parallel MPEG–1 video encoding. In *Proc. Picture Coding Symposium*, Sacramento, CA, September 1994.

[GT95] Simon J. Gibbs and Dionysios C. Tsichritzis. *Multimedia Programming—Objects, Environments and Frameworks.* Addison-Wesley, 1995. ISBN 0-201-42282-4.

[Gur85] J. R. Gurd. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1), January 1985.

[Lea94] Doug Lea. Design patterns for avionics control systems, November 1994. DSSA Adage Project ADAGE-OSW-94-01.

[Lea96] Doug Lea. *Concurrent Programming in Java.* The Java Series. Addison-Wesley, 1996. ISBN 0-201-69581-2.

[Lew95] Ted Lewis, editor. *Object-Oriented Application Frameworks.* Manning, 1995. ISBN 1-884777-06-6.

[Lin94] Christopher 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.

[MD96] Gerard Meszaros and Jim Doble. Metapatterns: A pattern language for pattern writing. The 3rd Pattern Languages of Programming conference, Monticello, Illinois, September 1996.

[Meu95] Regine Meunier. *The Pipes and Filters Architecture*, volume 1 of *"Pattern Languages of Program Design"*, chapter 22. Addison-Wesley, 1995. Edited by James O. Coplien and Douglas C. Schmidt. ISBN 0-201-60734-4.

[MS96] David Musser and Atul Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 1996. ISBN 0-201-63398-1.

[NPA92] R. S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proc. 19th Annual International Symposium on Computer Architecture*, pages 156–167. ACM, 1992.

[Pap91] Gregory M. Papadopoulos. *Implementation of a general-purpose dataflow multiprocessor.* MIT Press, 1991.

[PLV96] Edward J. Posnak, R. Greg Lavender, and Harrick M. Vin. Adaptive pipeline: an object structural pattern for adaptive applications. In *The 3rd Pattern Languages of Programming conference*, Monticello, IL, September 1996.

[PLV97] Edward J. Posnak, R. Greg Lavender, and Harrick M. Vin. An adaptive framework for developing multimedia software components. *Communications of the ACM*, 40(10), October 1997.

[Rit84] Dennis M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[SC95] Aamod Sane and Roy Campbell. Composite Messages: A Structural Pattern for Communication Between Processes. In *Proc. OOPSLA Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems*, October 1995.

[Sha96] Mary Shaw. *Some Patterns for Software Architecture*, volume 2 of *"Pattern Languages of Program Design"*, chapter 16. Addison-Wesley, 1996. Edited by John M. Vlissides, James O. Coplien and Norman L. Kerth. ISBN 0-201-89527-7.

[SOHL+96] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI—The Complete Reference.* The MIT Press, 1996. ISBN 0-262-69184-1.

[SPG91] Abraham Silberschatz, James L. Peterson, and Peter B. Galvin. *Operating System Concepts.* Addison-Wesley, third edition, 1991. ISBN 0-201-51379-X.

[TW97] Andrew S. Tanenbaum and Albert S. Woodhul. *Operating Systems—Design and Implementation.* Prentice-Hall, second edition, 1997. ISBN 0-13-638677-6.

[Wal94] William F. Walker. *A Conversation-Based Framework For Musical Improvisation.* PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[X2088a] CCITT Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1), 1988.

[X2088b] CCITT Recommendation X.209: Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), 1988.