3 Pattern: Payload passing protocols

Problem

Applications consist of several collaborating components. How to distribute control flow among these components in a way that is compatible with the application's requirements?

Forces

- Interaction between software components can be demand-driven, event-driven or asynchronous;
- The available resources (e.g., memory) are limited;
- Disconnected operation is required.

Subpatterns

The following subpatterns describe several ways to assign control flow among collaborating software components. Each subpattern lays out structural rules about how different components communicate. They avoid the need for dynamic synchronization policies [Lea96]. Therefore, the payload passing protocols enable applications to scale across different protocols and communication mechanisms.

3.1 Pull

Context

Not all subsystems of a computer operate at the same speed. The hard disk, for instance, is one of the slowest subsystems. On average, current drives need time in the order of milliseconds to satisfy a read or write request.

Consider an application that reads the output of some sensor and records this value on a hard drive. The application consists of two components. The producer interfaces with the sensor and can obtain a new value each 500μs. The consumer interfaces with the hard drive and for each write it needs on average 8ms. The data flow is from the producer (upstream) to the consumer (downstream). Which component should trigger the readings?

Solution

Use the **pull** protocol. The downstream component requests information from the upstream component with a procedure/method call that returns the values as results.

How? 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 [Lea94].

When? In the pull model, data flow is initiated by the downstream component. Consequently, it is applicable in demand-driven contexts and instances where the senders operate faster than the receiver. Why not However, because there is no provision for the upstream component(s) to trigger the data flow, this protocol cannot deal with asynchronous or prioritized events.

Other Applications employing this protocol do not need buffers. The downstream component requests a new data item only when it is ready to process it. Buffers require additional memory or permanent storage. They also can grow dynamically, thus expanding an application's memory image. Consequently, the pull protocol is an attractive solution for instances where these resources are limited. However, the lockstep execution requires a live connection between components. Therefore, it is not a good choice for disconnected operation.

The pull protocol and the direction of control and data flow is illustrated in Figure 13.

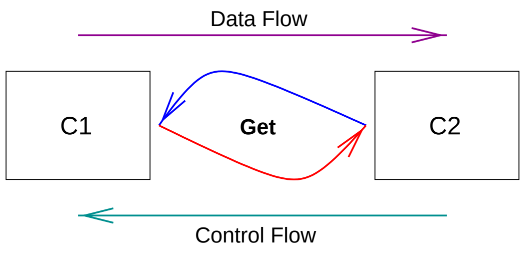

Figure 13: The pull protocol.

In summary, the pull protocol has the following **benefits** and **liabilities**:

✓ It is best suited for demand-driven applications. Typically, the slowest component determines flow control and the application delivers best effort performance while performing only the necessary work.

✓ Does not require buffers. It is suitable when resources are limited or the application is restricted to a fixed memory footprint.

✗ It cannot handle asynchronous or prioritized events. All messages are held by the upstream component until a new one is required by the downstream component.

✗ Disconnected operation is not possible. Both parties must be present and linked through a live connection.

✗ Not feasible for long end-to-end delays. The round-trip required to bring each data item takes twice the end-to-end delay.

3.2 Push

Context

There are instances when applications have to deal with unexpected events. A user wants to be able to interrupt at any moment a running program that takes too long to complete.

Consider an application following the model-view-controller (MVC) architecture [BMR$^+$96]. The user interacts with the controller by manipulating various graphical user interface widgets. User actions are converted into events. The model and view change state based on these events. Data flow corresponding to user's actions is from the controller (upstream) to the model and view (downstream). Which component generates the transfer of events?

Solution

Use the **push** protocol. The upstream component sends the new data items downstream whenever these are available.

How?

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 [Lea94].

When?

The push model is applicable in event-driven contexts, where the computation—data flow—is initiated by an external event or by a continuous loop in the upstream component.

Why not

However, the upstream component does not know if the other end is ready to receive or not. To prevent data loss, the downstream component uses a buffer to queue the upcoming data. If the system processes asynchronous events or high priority messages, the buffer must be able to identify and pass them downstream ahead of low priority messages. While a simple FIFO queue is suitable for the former situation, the latter requires a priority queue. The payloads pattern provides one possible solution for message identification. However, buffers require additional scheduling policies, introduce unpredictable delays and are not viable if the payloads have large memory footprints.

Other

The push protocol can use buffers for flow control [Rit84]. A high water mark limits the amount of payloads that can be stored in the repository; the upstream component 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 component stops sending data. When the downstream component notices this flag set and the queue drops below a low water mark, it reactivates the sender.

The push model and the direction of cotrol and data flow is illustrated in Figure 14.

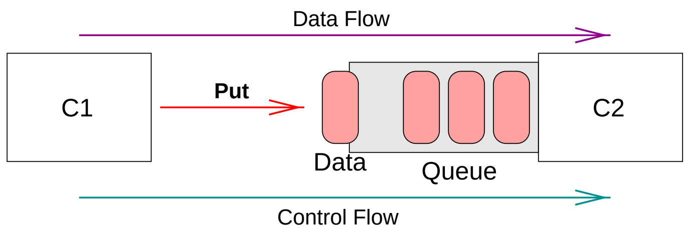

Figure 14: The push protocol.

This protocol has the following **benefits** and **liabilities**:

✓ It is best suited for event-driven applications. Whenever an event is generated or received from an external source, it is processed right away.

✓ Supports asynchronous notifications and prioritized messages. Since the upstream component triggers the data flow, these sort of messages can be passed downstream as they arrive.

✓ Supports disconnected operation. Once all data items have been queued, the upstream component and the connection are not required anylonger.

✗ Requires buffers between communicating components. Additional logic is needed to manage the data items within buffers.

✗ Needs upper bounds for buffers. Real applications do not use infinite buffers; developers have to estimate buffer upper bounds and handle in a reasonable way any attempt to pass them.

✗ Increases resource consumption. Buffers use memory or persistent storage which otherwise can be used for different purposes.

3.3 Indirect

Context

Sometimes the operation of cooperating components is asynchronous with respect to each other. This is typical whenever their communication is crossing domain boundaries.

In [RW94], the authors describe a software simulator for disk drives. The simulator is based on AT&T's C++ task library and consists of two tasks. One task models the disk mechanism. The second task models the SCSI bus interface and its transfer engine. Since the disk mechanism and the SCSI subsystem are drived by different clocks, their interaction is asynchronous. Data flow changes direction according to the request type—read or write. How to assign control flow in this situation?

Solution

Use the **indirect** protocol. This requires the availability of a shared repository (sometimes called mailbox, pipe [BMR^{+}96] or put/take buffer stage [Lea96]) that is accessible to both modules.

How?

This mechanism can be implemented via transfers to shared memory which occur at fixed rates, or via polling [Lea94]. Whenever the sender component (producer) is ready to pass data to the receiver, it *puts* it in the shared repository. When ready to process another input, the receiver component (consumer) *gets* a data item from the repository.

When?

The indirect model is applicable when the sender and the receiver process payloads asynchronously, eventually at different rates. Whenever not all data is required by the receiver, the sender can overwrite the contents of the shared repository. 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 receiver component reads frames from the shared buffer at a different, lower rate. However, the performance depends on the nature of the repository. For systems with shared memory the procotol is efficient. But 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.

Why not

The indirect model is illustrated in Figure 15. The square waves show how the two components access the shared repository at different rates (the writes and reads are shown as arrows).

In summary, the indirect protocol has the following **benefits** and **liabilities**:

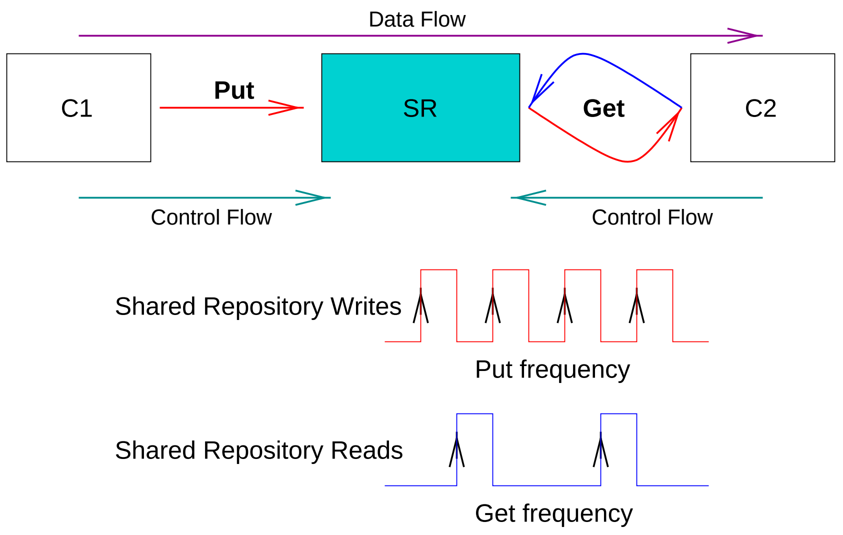

Figure 15: The indirect protocol.

✓ It is best suited whenever the interaction between components is asynchronous. This is typical for components located across domain boundaries.

✓ The sender and receiver can access the shared repository at different rates. Their processing rates are independent and can change dynamically.

✓ It is efficient. Transfers through shared memory are fast.

✓ Low coupling between the two parties. The sender and the receiver interact only with the shared repository. They do not need to know about each other.

✗ Increases the overhead. Managing the critical sections is tedious and time-consuming.

✗ Requires a shared repository. Shared memory may not be available or viable.

Additional observations

This section provides some additional details with general applicability and shows the relationships with other patterns.

End-of-stream

Depending on how control flow 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.

Mix

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

Workflow

Workflow systems use pull and push protocols for scheduling through worklist handling [Moh97]. A worklist contains all activities which are ready to be performed by the worker (either an user or an application). The workflow engine updates this list each time a new activity is ready. In pull systems, the worker inspects the worklist and selects (pulls) the next activity to be performed. This protocol is suitable for self-schedulable workers. In push systems, the worker is not given a choice. The workflow engine selects and passes (pushes) the next activity to the worker. The protocol is employed whenever the system controls work scheduling.

>1 inputs

Multiple input ports complicate control flow and require 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 2–4 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 16—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 component, process it and return the transformed value. `PullInput`—Figure 17—determines the input data type, provides the interconnection mechanism and maintains a pointer to the previous (upstream) component. This time, the `PullPort` class—Figure 18—encapsulates the attachment between an input port and an output port.

Figure 19 shows the implementation of the processing module from Figure 5

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

Figure 16: The `PullOutput` class.

```
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 17: 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 18: The `PullPort` class.

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

Figure 19: The `Sqrt` processing module following the pull model.

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 2–4 and 16–18. 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 copied each time they cross the inter-module boundary (for the pull protocol). The Payloads pattern discusses several optimized techniques that address these problems.

Examples

1. **Multimedia** applications like VuSystem [Lin94] and ActiveMovie [AMS] employ one or several payload passing protocols. VuSystem does not use buffers. Payloads are passed with one function call and timing constraints are propagated through back-pressure. By temporarily refusing a payload, a downstream component slows down upstream processing. The mechanism is simple and does not require multi-threading. In ActiveMovie, pins are responsible for providing interfaces to connect with other pins and for transporting data. These interfaces transfer time-stamped data, negotiate data formats and maximize throughput by selecting an optimal protocol—push or pull.

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. **Operating systems** use these protocols as well. For example, the UNIX stream input-output system [Rit84] employs a message queue and water marks for flow control.

4. Payload passing protocols are also present in **distributed object systems**. The CORBA event service [OHE97] allows objects to register their interest in specific events. The event channel supports both the push and pull protocols for event notification.

Related Patterns

- Applications following the data flow architecture replace direct calls with a message passing mechanism. Usually this is one of the above protocols.

- Buffers need various types of information about the messages they store (e.g., priority level, etc.). The payloads pattern ensures uniform access to this information, while maintaining a loose coupling between buffers and messages.

- Adjacent layers within a layered architecture [BMR$^+$96] communicate with each other. The forementioned protocols solve the inter-layer communication problem whenever direct calls are not feasible.

- Both parties engaged in a pull protocol are aware of each other. The use of the Observer [GHJV95] pattern decreases this coupling.