A Dataflow Pattern Catalog for Sound and Music Computing

Pau Arumí
Music Technology Group
Universitat Pompeu Fabra
Barcelona, Spain
parumi@iua.upf.edu

David García
Music Technology Group
Universitat Pompeu Fabra
Barcelona, Spain
dgarcia@iua.upf.edu

Xavier Amatriain
CREATE
Univ. of California Santa Barbara
Santa Barbara, CA, USA
xavier@create.ucsb.edu

ABSTRACT

This article describes a set of patterns the authors have seen emerging during years of experience developing assorted applications in the sound and music domain and receiving influences from theoretical models, existing systems, and colleagues.

Those patterns aim at offering a generative pattern language that falls within a generic data flow architecture. Regardless the audio domain origins of the patterns, we foresee that they have applicability in other domains.

Contributions of this paper are: General Dataflow Patterns, that address problems about how to organize high-level aspects of the dataflow architecture, by having different types of modules connections; Flow Implementation Patterns, that address how to physically transfer tokens from one module to another, according to the types of flow defined by the "general dataflow patterns". Tokens life-cycle, ownership and memory management are recurrent issues in those patterns; and finally, Network Usability Patterns, that address how humans can interact with dataflow networks.

1. INTRODUCTION

The Sound and Music Computing field has a rich history of frameworks that to offer a more or less complete set of tools for application builders [7, 13, 5, 6, 19]. Most of these frameworks end up offering a variation of Dataflow Networks [14].

There have been previous efforts in building pattern languages for the dataflow paradigm, most noticeably the one by Manolescu [15]. The present article takes our experience building generic audio frameworks and models [1, 27] and maps them to traditional Graphical Models of Computation (mostly Dataflows but also Petri Nets and Process Networks, [14, 17, 18]) in order to offer an initial pattern catalog —aiming to grow into a pattern language— for the audio domain[1].

All the patterns presented in this catalog fit within the generic architectural pattern defined by Manolescu as the Data Flow Architecture pattern.

The Data Flow Architecture pattern solves the problem by designing a system that performs some number of sorted operations on similar data elements (that we will call *tokens*) in a flexible way so that the overall functionality can be dynamically changed without compromising performance. The solution is an architecture composed of a *network* of *modules* with strict interfaces at the module boundary, allowing a large number of possible combinations.

Modules read incoming tokens through their in-ports and write them through their out-ports. Module connections are done by connecting out-ports to in-ports, forming a network. Tokens flow through modules in two different fashions: at a regular (or almost regular) rate, which is known as a *stream* flow, and without any regularity, which is known as *event* flow. For example, the flow of data from an audio card is a stream flow, while the flow of note-on and note-off messages from a MIDI keyboard is an event flow.

Each module in a network is periodically *executed*, which means a call to the module's *execution method* (also known as *module's algorithm*).

The Data Flow Architecture pattern does not impose any restrictions on message passing protocol, module execution scheduling, or data token implementation. All these aspects imply different problems that can be addressed in other fine-grained patterns, like the ones in the present pattern language. This pattern granularity [25] proved very useful because we have been able to incorporate orthogonal patterns that work synergistically among them and with the existing ones from Manolescu.

Taking into account the previously introduced background, the main contributions of this paper are:

- **General Dataflow Patterns:** Address problems about how to organize high-level aspects of the dataflow architecture, by having different types of modules connections. Belonging to this category: Semantic Ports address distinct management of tokens by semantic; Driver Ports address how to make modules executions independent of the availability of certain kind of tokens. Stream and Event Ports address how to synchronize different streams and events arriving to a module; and, finally, Typed Connections address how to deal

[1]The only previous efforts in the audio domain that we are aware of are: several Music Information Retrieval pattens [4] and a catalog with 6 real-time audio patterns presented in a Workshop at the ICMC 2005 —though we are not aware of its publication

PLoP '06, October 21¿23, 2006, Portland, OR, USA.
ISBN: 978-1-50448-372-3

with typed tokens while allowing the network connection maker to ignore the concrete types.

- **Flow Implementation Patterns:** Address how to physically transfer tokens from one module to another, according to the types of flow defined by the *general dataflow patterns*. Tokens life-cycle, ownership and memory management are recurrent issues in those patterns. Cascading Event Ports address the problem of having a high-priority event-driven flow able to propagate through the network. Multi-rate Stream Ports address how stream ports can consume and produce at different rates; Multiple Window Circular Buffer address how a writer and multiple readers can share the same tokens buffer. and Phantom Buffer address how to design a data structure both with the benefits of a circular buffer and the guarantee of window contiguity.
- **Network Usability Patterns:** Address how humans can interact with dataflow networks. Recursive Networks makes feasible for humans to deal with the definition of big complex networks; and Port Monitor address how to monitor a flow from a different thread, without compromising the network processing efficiency.

The proposed patterns are inspired by our experience in the audio domain. And some patterns are clearly motivated by the requirements of the spectral processing. A use case that exemplifies its complexity is the analysis-synthesis using sinusoids plus residual (see figure 1), where different Fast Fourier Transforms are done consuming different number of tokens (audio samples) in parallel. Nevertheless, we believe that those have an immediate applicability in other domains.

All patterns in the "general dataflow patterns" category can be used on any other dataflow domain. Typed Connections, Multiple Window Circular Buffer and Phantom Buffer have applicability beyond dataflow systems. And, regarding the Port Monitor pattern, though its description is coupled with the dataflow architecture, it can be extrapolated to other environments where a normal priority thread is monitoring changing data on a real-time one.

Most of the patterns in this paper can be found implemented in many audio systems. However, examples of a few others (namely Multi-rate Stream Ports, Multiple Window Circular Buffer and Phantom Buffer) are hard to find, so they should be considered innovative patterns (or proto-patterns). But all the patterns in this catalog can be found in the CLAM framework. And since it is an open-source software, an example of each pattern in C++ can be easily obtained there.

Finally, it also has to be taken into consideration that the CLAM framework has demonstrated its adaptability —with many specific applications— to several scenarios within the sound and music domain [2] such as real-time processing and synthesis and off-line audio analysis.

2. SEMANTIC PORTS

Context

Applications with a dataflow architecture consist on a directed graph of modules, like shown in figure 2. Is a very common case that a module receives tokens with different semantics. For example, a module that mixes n audio channels will receive tokens of audio data corresponding to each channel. Identifying which token corresponds to each channel —the token semantics— is fundamental to produce output tokens containing the audio mix. The Payloads pattern described by Manolescu provides a solution to this problem consisting on adding a descriptor component into each token which provides the semantic information about the token, as well as type-specific parameters. The implication of applying Payloads is that incoming tokens needs to be dispatched according to its descriptor component, before doing any processing.

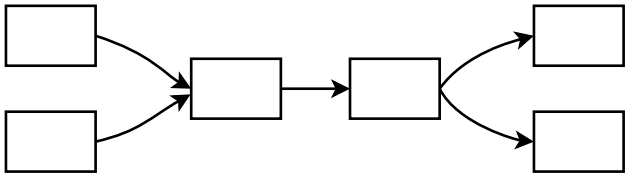

Figure 2: A directed graph of components forming a network

Tokens produced by a module may also have different semantics. One might want to send to a connected module only tokens with a given semantics and not all the produced tokens.

Problem

How can a module manage tokens according to their semantics in order to deal with the incoming ones in different ways and send the produced ones to different destinations?

Forces

- Module implementation should be as simple as possible, because modules are developed by different authors while general infrastructure is just implemented once by experienced programmers.
- Dispatching tokens adds complexity to module programming
- Module execution should be efficient in time, often real-time constrains are imposed.
- Dispatching tokens adds a run-time overhead.
- Token semantics fields on tokens add overhead
- Token semantics should be given by the module and they should not be restricted.
- Incoming might also have different priorities, and modules should consume the tokens with greatest priority first.

Solution

Use different ports for every different token semantics in each module. Modules should have as many in-ports and out-ports as different input and output semantics are needed. Instead of connecting modules directly, connect modules by pairing out-ports with in-ports, as shown in figure 3. Module's execution method knows the semantics associated to each port, thus, it can obtain tokens of specific semantics just by picking the proper in-port. Because connections are done among ports instead of modules, a processed tokens will target the proper destination just by sending tokens through the proper out-port.

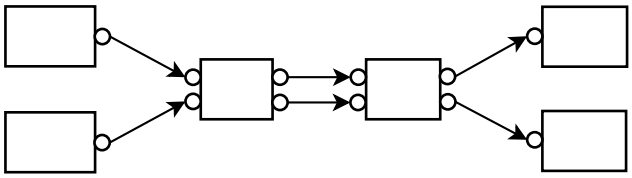

Figure 3: A network of components with multiple ports

Consequences

Tokens do not need to incorporate a description component. Module implementation is simplified because programming a token dispatcher regarding its semantics is not needed. Also, run-time penalty associated with dispatching is avoided. The pattern solution implies that token semantics is not defined inside the tokens with a description component, but semantics is something intrinsic to the ports.

Retaking the audio mixer example; instead of having a "channel" field on each token arriving to the mixer, using the Semantic Ports pattern, we would have a mixer with n different in-ports, each one receiving tokens of a single channel.

Tokens with different priorities should be routed to different in-ports. The module knows the priority of each in-port and so is able —in its execution method— to consume tokens in the right order.

Related Patterns

Most patterns in this collection build on Semantic Ports: Driver Ports, Stream and Event Ports, and Cascading Event Ports are clear examples of separation of ports regarding its semantics.

Semantic Ports also relates to Payloads in the sense that the problems they solve are similar but, since they have different forces, they end up with different solutions.

Semantic Ports can handle different token types by using the Typed Connection pattern.

Examples

CLAM uses Semantic Ports to separate different flows. Visual environments like Pure-Data (PD) [22] or MAX/MSP [21] also do. They ports separate both audio ("tilde") streams lower rate streams on their semantic. We find another good examples in Open Sound World (OSW) [5] and the JACK sound server [9].

Anti-examples —systems that do not use *Semantic Ports* because they use other approaches— are also interesting to see for this pattern: Marsyas [24] and SndObj [13], they do not use separated ports for its network connections but they do it at module level. SndObj modules, for instance, keeps a pointer to their connected producers and reads its output signal doing a direct call.

3. DRIVER PORTS

Context

Module execution on dataflow system is driven by the availability of flowing tokens. But not all token flows drive the execution.

Imagine a module which receives an audio signal and performs a low-pass filter with a given cutoff frequency. The audio signal is fed into the module with a constant rate but the cutoff values are seldom fed into the module. These cutoff values typically come from a sequencer module or a knob in the user interface. Each execution of the module must wait for the availability of new audio signal data. But there is not such dependency on the seldom received cutoff values, it just uses the last value. To summarize, whereas audio stream tokens drive the modules execution, the frequency event tokens does not.

Problem

How can we make module execution depend on the availability of tokens on certain in-ports and not on others?

Forces

- Concrete module implementation should be simple.
- Visual programming tools should be able to distinguish the flow that drives the module execution from the one that does not.

Solution

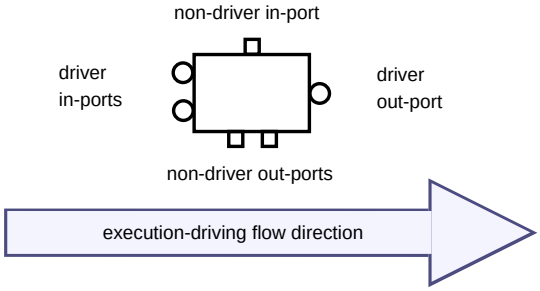

Figure 4: A representation of a module with different types of in-ports and out-ports

Allow the concrete module developer to define which are the driver in-ports and which are not. Give the modules a common interface from which external entities can know which are the drivers and which are not. The module execution will be enabled by the availability of enough tokens on the driver in-ports. Note that enabling is not the same as triggering. The network scheduling policy determines if a module will be executed as soon as it is able —in a *pull* strategy— or if it will be postponed until other module executions end.

This solution is rather general and it can be implemented with different strategies. A concrete design is shown in figure 5. This class collaboration separates the general infrastructure in base classes making the concrete classes simpler to implement —this is actually an example of *white box reuse* in frameworks. Some module services are implemented in the base class, usually delegating to its ports, and freeing the concrete module writer from this responsibility.

Is important to note that although concrete modules create its port objects, the module base class aggregates them to provide a generic interface to all the ports. Examples of

such operations are *ableToExecute* which can be useful to a firing manager (or scheduler); and *driverPorts*/*nonDriverPorts* which give the lists of driver and non-driver ports to, say, a GUI client.

Other patterns like Stream and Event Ports and Typed Connections also benefit from using this class structure. However, each pattern enriches the *Port* and *Module* base class interfaces to fit its needs.

Consequences

Separating driver and non-driver ports makes it possible to check whether a module is ready to be executed or not without relying on the concrete module implementation which gets simpler and safer to programming errors.

Visual builder tools can distinguish driver and non-driver flows by identifying driver and non-driver ports and displaying them differently.

As mentioned in [11] module networks are often built with visual programming tools. Such tools should give the user a clear separation between stream ports and event ports, else, event connections might hide the main dataflow —the stream flow that drives the modules execution.

For example, CLAM's visual builder called Network Editor (see figure 6) uses horizontal connections (left to right) for driver flow, and vertical (top-down) connections for the non-driver flow.

Other visual builders takes different approaches. Open Sound World (OSW), for instance, paints the driver ports in green while the non-driver ports are gray. This can be appreciated —though if the copy is not colored it can be hard— in figure 7.

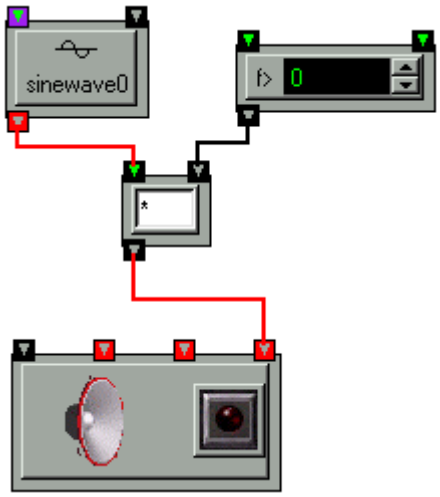

Figure 7: Screenshot of Open Sound World visual builder

Related Patterns

Driver Ports is strongly related to Stream and Event Ports. Driver ports tend to be stream ports. However, they are better off being separate patterns because they solve orthogonal problems. Moreover, examples exist where driver ports and stream ports are totally independent.

Token availability conditions are complex when connected stream ports produce and consume tokens at different rates. Multi-rate Stream Ports pattern solves this.

Examples

Pure Data (PD) [22] and MAX/MSP [21] are graphical programming environments for real-time musical applications that are widely used by composers. Its ports are called *inlets* and *outlets* and they are visually arranged horizontally. With few exceptions (notably the "timer"), objects treat their leftmost inlet as "hot" in the sense that messages to left inlets can result in output messages. Other inlets are "cold", they only store the received message and do not trigger any execution. Thus, the "hot" or leftmost inlets are the driver ports. However, since modules have only one driver port and modules are executed at the time a token arrives at the driver port, the following problematic situation may occur when two modules are connected by more than one connection: the module might be triggered before receiving all its data because the "hot" inlet was not the last to receive the data. In order to avoid this output messages are —by convention— written from right to left and modules connections should be (visually) done without any crossing lines. Finally, PD and MAX/MSP is important example where driver ports do not coincide with stream ports.

Open Sound World (OSW) [5] have a similar approach to PD and MAX/MSP but it does not limit the number of driver ports. In the JACK audio server [9] all ports are drivers. CLAM also uses *Driver Ports* and restricts its drivers to be constant-rate stream ports.

4. STREAM AND EVENT PORTS

Context

A module may receive tokens of both kinds —stream and event— coming from different sources. Moreover, streams may arrive at different rates. For example, a module may receive two audio samples streams one at 44100 Hz and the other at 22050 Hz. Figure 8 shows another example: a module is receiving two streams at different (though constant) rates and an irregularly distributed flow of events. Its output stream has the same rate as the second input stream.

Such a module consumes its incoming tokens and then calls its execution method which will take the consumed tokens as its input. When receiving tokens at different rates, the module needs to synchronize all the incoming tokens prior to its processing. This synchronization can also be seen as a time-alignment of incoming tokens, and it implies knowing the time associated to each token. Here, is important to differentiate the time associated to the tokens, with the "real" time where the module is executed. The two kind of times might be totally different. Figure 9 illustrates the alignment of tokes of different nature.

While incoming stream tokens always needs to be accurately aligned, this is not always true for incoming event tokens. Some applications requires a precise alignment of event tokens, while others admit a loose time alignment.

An obvious approach is to use the Payloads pattern, adding a precise time information —*time-stamp*— to each token. In real-time systems, this time-stamp relates to the time when the token is introduced into the system. In non real-time systems it relates to a virtual time. Transformations on a token should preserve the original time-stamp. However,

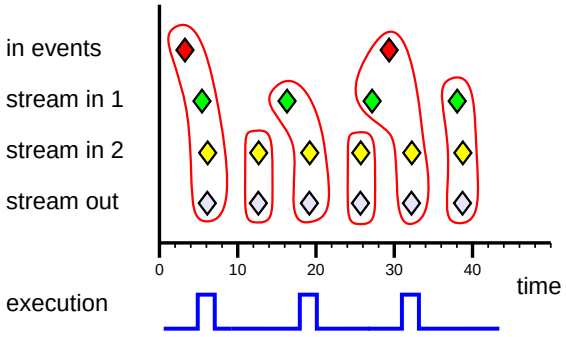

Figure 9: Alignment of incoming tokens in each execution. Note that time corresponds to token's time-information and does not relate to the module execution time (though they are equally spaced).

this Payloads approach can be overkill when stream tokens flow at a high rate, as it happens with audio samples.

Problem

Syncronizing incoming tokens requires time information. How can we get the time information of incoming tokens?

Forces

- Time-stamp is an overhead when the data token is relatively small.
- Propagating timestamps from the consumed tokens to the produced tokens is a run-time overhead and makes concrete module implementation more complex.
- Concrete module implementation should be simple.
- All stream sources must share the same hardware clock, so that their (token) rates can not vary among different streams.

Solution

Calculate time information of incoming stream tokens instead of using time-stamps. If the application needs accurate timing for events use time-stamps —only for event tokens—, else do not.

Separate the stream and event flow in different kinds of ports: stream and event ports. Place the stream timing responsibility into the stream in-port class. Stream in-ports are initially configured with a "token-rate" and "first-token-time" values, and they also keep the sequence —with a counter, for example— of consumed tokens. When a module execution method asks the stream in-port for new tokens to consume, the in-port provides the time information along with the tokens itself.

Ports parameters ("token-rate" and "first-token-time") configuration is a key issue to solve. Two main approaches exist: ports handshaking and centralized management.

Ports handshaking consists in propagating parameters downstream. In-ports receives parameters from their connected out-ports, and modules propagate them from in-ports to out-ports. In most cases, modules only need to copy them from the in-ports to out-ports. However, in some cases, the module processing may introduce delay and may change the token-rate; thus, this must be reflected in the out-port settings. In consequence, modules do not impose port parameters, they receives it and propagate them. Of course, source modules [2] are the exception to that rule. They must set the out-port parameters, because they are the *source* of the stream.

The second approach —centralized management— consist in incorporating an entity that orchestrates the configuration —and maybe the modules execution— of the whole network. This configuration manager is responsible for configuring all the stream ports in the network.

Alignment of event tokens with stream tokens is done in slightly different ways depending on whether the application needs accurate event timing or not —that is, whether they incorporate time-stamps or not. Note that on each execution, the module may consume not only one stream token but a bunch of them. In some cases, like with audio samples, even a large number of them like, say, 1000. If incoming events are time-stamped, the module knows the time information for all the incoming tokens, thus the module can align each event token with the stream tokens precisely. If events are not time-stamped, the module should align all consumed events with the first consumed stream token of each in-port.

Consequences

Making the stream tokens time implicit avoids space overhead. Event tokens may have time-stamps, but it is not required. Time-stamps do not have as much overhead in event tokens as in stream tokens, since they flow non continuously, in much lower frequency than streams.

Events Jitter:.

Having a big number of stream tokens to be consumed on each execution and having non time-stamped event tokens at the same time is a common cause for jitter. That is, unsteadiness or irregular variation on the time the system respond to incoming events. The amount of jitter is bounded by the time interval between executions, which is proportional to the number of stream tokens consumed on each execution. Thus, making modules consume fewer stream tokens each time, reduces jitter. Of course, when events comes in with time-stamps, jitter can be eliminated completely. The consequences of having jitter varies enormously depending on the concrete application. In most cases jitter can be neglected, in other cases, however reducing it is paramount.

Ports connectivity:.

The solution forbids time-stamps in the stream tokens and this restricts how stream ports can be connected. Because time must be inferred from the incoming stream sequence, in-ports must receive well formed sequences —without gaps, etc.— of an individual stream. Therefore, in general, N-to-1 connections of stream ports are to be forbidden by the system. However, an exception to this rule exists when the in-port is able to implicitly perform a combining operation prior to keeping track of the incoming sequence. For example, imagine an in-port of audio samples fed by multiple different streams. Parallel samples are added, forming an

[2] Source modules are that ones that do have stream out-ports but do not have any stream in-ports

audio mix. Because the mix is a single token, the in-port assigns time the same way as if the source were unique. Apart from addition, packing —that is, create a new composite token— is another common combining operation.

In general, multiple stream combinations is better handled explicitly in specific modules. It gives the system designer flexibility to choose his or her combining operation. Moreover, a system can have token types without any valid combining operation, thus making implicit combinations impossible.

We have seen that, in general, N-to-1 connections of stream ports are to be forbidden by the system. On the other hand, N-to-1 connections of event ports are perfectly fine, since there is no need to infer the token time information from the order of arrival. Time information is either read from the time-stamp or simply ignored.

Splitting one stream to multiple streams is a different story: 1-to-N connections of both stream and event ports are allowed. The consideration that has to be done here is how to duplicate outgoing tokens flowing to multiple destinations. Two strategies exist: one is making a copy of each outgoing token to every in-port, and the other is passing a managed reference to each in-port. In the later case, the in-ports will have to enforce read-only semantics.

Related Patterns

Stream and Event Ports uses to go together with Driver Ports and, in most of the cases, stream ports are also driver ports.

Systems that use Stream and Event Ports may also use Multi-rate Stream Ports for designing the stream ports —allowing stream ports to consume and produce at different cadences—, and may use Cascading Event Ports for its event ports —allowing event ports to propagate events immediately.

Stream ports designed with the Multi-rate Stream Ports pattern defines the number of released tokens for each stream port on each execution. This numbers influence how the ports "token-rate" settings are propagated —in this case, being re-calculated— from in-ports to out-ports.

Stream and Event Ports can handle different token types by using the Typed Connection pattern.

Examples

SuperCollider3 [16] and CSL [19] use the pattern but events can not arrive at any time: they have a dual rate system, control rate and audio rate, with the control rate being a divisor of the audio block rate.

Marsyas [24] uses this pattern though its event ports do not follow the dataflow architecture because connections are not done explicitly. It implements the token packing technique (from multiple sources) as mentioned in the Ports Connectivity section.

CLAM and OSW [5] clearly use this pattern, separating ports for streams and for events.

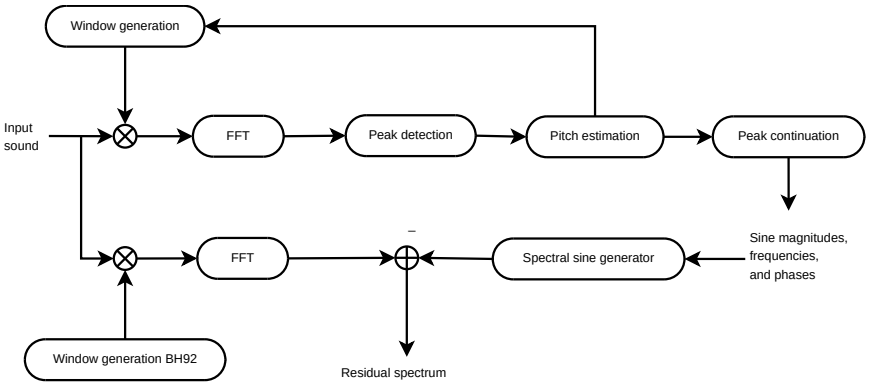

Figure 1: A use case for audio dataflow: the Spectral Modeling Synthesis.

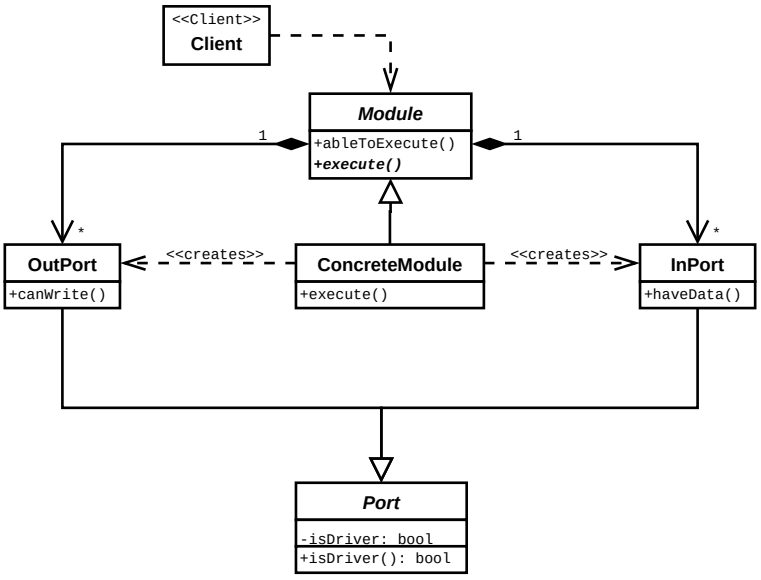

Figure 5: Separated Module and ConcreteModule classes, to reuse behaviour among modules

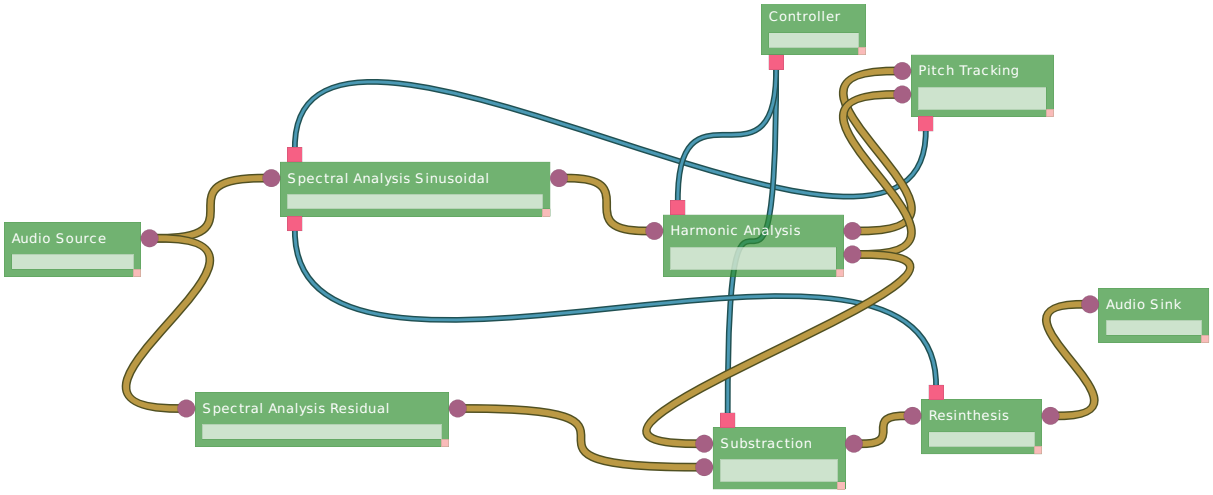

Figure 6: Screenshot of CLAM visual builder (NetworkEditor) doing SMS analysis-synthesis

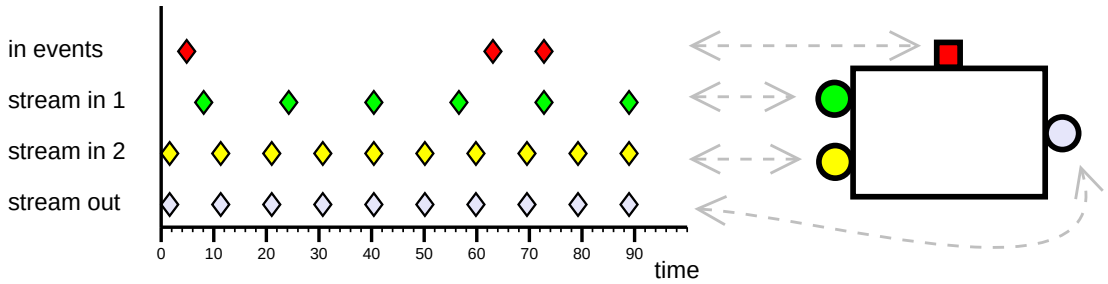

Figure 8: Chronogram of the arrival (and departure) time of stream and event tokens

5. TYPED CONNECTIONS

Context

Most simple audio applications have a single type of token: the sample or the sample buffer. But more elaborated processing applications must manage some other kinds of tokens such as spectra, spectral peaks, MFCC's, MIDI... You may not even want to limit the supported types. The same applies to events channels, we could limit them to floating point types but we may use structured events controls like the ones OSC [26] allows.

Heterogeneous data could be handled in a generic way (common abstract class, void pointers...) but this adds a dynamic type handling overhead to modules. Module programmers should have to deal with this complexity and this is not desirable. It is better to directly provide them the proper token type. Besides that, coupling the communication channel between modules with the actual token type is good because this eases the channel internal buffers management.

But using typed connections may imply that the entity that handles the connections should deal with all the possible types. This could imply, at least, that the connection entity would have a maintainability problem. And it could even be unfeasible to manage when the set of those token types is not known at compilation time, but at run-time, for example, when we use plugins.

Problem

Connectable entities communicate typed tokens but there is an unlimited number of types of tokens. Thus, how can a connection maker do typed connections without knowing the types?

Forces

- Process needs to be very efficient and avoid dynamic type checking and handling.
- Connections are done in run-time by the user, so they can mismatch the token type.
- Dynamic type handling is a complex and error prone programming task, thus, placing it on the connection infrastructure is preferable than placing it on concrete modules implementation.
- Token buffering among modules can be implemented in a wiser way by knowing the concrete token type rather than just knowing an abstract base class.
- The set of token types evolves and grows.
- A connection maker coupled to the evolving set of types is a maintenance workhorse.
- A type could be added in run time.

Solution

Split complementary ports interfaces into an abstract level, which is independent of the token-type, and a derived level that is coupled to the token-type. The class diagram of this solution is shown in figure 10.

Let the connection maker set the connections through the generic interface, while the actual bind ins done in the subclasses to be type-safe. Also, the connected entities use the token-type coupled interface to communicate with each other efficiently.

```
#include <typeinfo>

class AbstractFemale
{
public:
    void bind( AbstractMale& male ) {
        if ( isCompatible(male) )
              doTypedBinding(male);
        else
            throw InvalidTypes();
    }
    virtual void isCompatible(
                const AbstractMale& male
                ) = 0;

protected:
    virtual void doTypedBinding(
                AbstractMale& male
                ) = 0;
}

template<class Token> class Female :
                      public AbstractFemale
{
public:
    bool isCompatible( const AbstractMale& male) {
        return typeid(Token) == male.tokenType();
    }
    void doTypedBinding( AbstractMale& male ) {
        _male = &dynamic_cast< Male<Token>& >(male);
    }
private:
    Male<Token>* _male;
};

class AbstractMale
{
public:
    const std::type_info& tokenType() = 0;
};

template<class Token> class Male :
                      public AbstractMale
{
public:
    const std::type_info& tokenType() {
        return typeid(Token);
    }
}
```

Listing 1: Sample C++ code for TypedConnections

Use run-time type checks when modules get connected (*binding time*) to get sure that connected ports types are compatible, and, once they are correctly connected (*processing time*), rely just on compile-time type checks.

To do that, the generic connection method on the abstract interface (`bind`) should delegate the dynamic type checking to the concrete (token-type coupled) classes using the abstract methods `isCompatible`, `tokenType` and `doTypedBinding`. The implementation in C++ of listing 1 shows two classes that gets connected by a pointer in the concrete female pointing to the concrete male. Depending on the situation, a pointer is not enough and a (token-type coupled) "connection" entity is needed.

Consequences

By applying the solution, the connection maker is not coupled to token types. Just concrete modules are coupled to the token types they use.

Type safety is assured by checking the dynamic type on

binding time and relying on compile time type checks during processing time. So this is both efficient and safe.

Because both sides on the connection know the token type, buffering structures can deal with tokens efficiently during allocation, initialization, copy, etc.

Concrete modules just access to the static typed tokens. So, no dynamic type handling is needed.

Besides the static type, connection checking gives the ability to do extra checks on the connecting entities such as semantic type information. For example, implementations of the bind method could check that the size and scale of audio spectra match.

Related Patterns

This pattern enriches Multi-rate Stream Ports and Event Ports, and can be also useful for the binding of the visualization and the Port Monitor.

The proposed implementation of Typed Connections uses the Template Method [12] to call the concrete binding method from the generic interface.

Examples

OSW [5] uses Typed Connections to allow incorporating custom data types.

The CLAM framework uses this pattern notably on several pluggable pairs such as in and out ports and in and out controls, which are, in addition, examples of the Multi-rate Stream Ports and Event Ports patterns.

But the Typed connection pattern in CLAM is not limited to port like pairs. For example, CLAM implements sound descriptors extractor modules which have ports directly connected to a descriptor container which stores them. The extractor and the container are type coupled but the connections are done as described in a configuration file, so handling generic typed connections is needed.

The Music Annotator [3] is a recent application which provides another example of non-port-like use of Typed Connections. Most of its views are type coupled and they are mostly plugins. Data to be visualized is read from an storage like the one before. A design based on the Typed Connection pattern is used in order to know which data on the schema is available to be viewed with each vista so that users can attach any view to any type compatible attribute on the storage.

6. CASCADING EVENT PORTS

Context

Music and audio systems for real-time usage offer users interfaces, such as GUI slider or MIDI interfaces, to alter the processing while playing. Users should get fast feedback on their actions, so elapsed time between the event and its effect on the processing should be as short as possible.

Some modules transform such events and propagate them in a proper way to other modules. Imagine, for instance, a MIDI mapper module that maps MIDI events to a floating point frequency value, more suitable to control a frequency filter module. To get a responsive behaviour, modules should be aware of this frequency change as soon as possible.

One simple design consist on propagating incoming events during module execution. However modules that receive the event may be executed before the module that is going to propagate the event.

Problem

How can we send event tokens and run associated actions on the receiver, including propagation, so that they get to the destination before other modules are executed?

Forces

- Module execution should be able to send events.
- Event token reception may imply changing the module state.
- Event token reception may imply sending new event tokens to other modules.
- Event token propagation should not be costly in respect to module execution in order not to break execution cadence and real-time restrictions.
- Coarse event tokens are hard to propagate by copy.
- Feedback loops on event ports should be allowed.
- 1 to N event ports connections should be allowed.
- N to 1 event ports connections should be allowed.
- All modules are executed from a single process.

Solution

Provide the concrete module implementers a way to bind a event in-port with a callback method to be called on event reception. This callback might change the module state, or propagate other events through the module event out-ports. Sending an event through an event out-port imply the immediate cascade execution of callback methods associated with every connected event in-port.

If event propagation by copy is too expensive, propagate event tokens using references instead of copies and make them read-only for the receiving modules. Limit the life of event tokens sent by reference to the cascade propagation and, forbid the receiving modules to keep references further than the callback execution.

Consequences

Event in-port callback implementers should be careful not to do too much things on them. Events may be sent in bursts; thus, expensive callbacks could break the real-time restrictions.

Propagation of coarse events is something that could add penalty to in-port callbacks, but, by using references, this is avoided. Sending references could be dangerous when considering 1 to N connections, as one of the receiving modules may modify the event token. This is solved by making them read-only for the receiving modules.

Another danger associated to sending references is that modules might keep references to such tokens. Because of this, keeping references is forbidden, but we could loose this restriction by using reference counting on event tokens. The use of garbage collectors is not a good solution due to real-time restrictions.

The solution allows setting up loops on the event ports connection graph. Those loops might be harmless but they

might be pernicious because the cascade callback calling enters in a non-ending loop. Harmful loops happen whenever the call sequence reaches a port that was already involved on the cascade.

Static analysis of the network topology to warn the user about harmful loops is useless: Not every event reception implies propagation on the event out-ports of the module; it depends on the callbacks methods. Because the sending of events is a synchronous call, one simple solution is to block sending tokens through a port which is already sending one. This is implemented just by adding a "sending-on-progress" flag in each port.

Related Patterns

Event tokens could be restricted to a given set of types, but we could also use the Typed Connections pattern to a more flexible solution.

Cascading Event Ports provides a flexible way for communicating the two partitions of the Out-of-band and in-band-partitions pattern [15]. The user interface partition communicates with the processing partition via connected event ports. Since both partitions are in different threads, a safe thread boundary must be established. Using, for example, the Message Queuing pattern [10].

This pattern could be seen as a concrete adaptation of Observer [12] to a dataflow domain where modules can act both as *observers* and *subjects*, in a way that they can be chained.

Examples

CLAM implements controls as cascading event ports. Multiple control inputs and outputs are supported. By default, events are copied as part of the module state but you can add a callback method to process each control in a special way. In its current version (0.91) event tokens are limited to floating point numbers.

PD [22], MAX/MSP [21] they all use cascading event ports for their non-audio-related modules "hot inlets".

7. MULTI-RATE STREAM PORTS

Context

Many applications in the audio and music domain need to process chunks of consecutive audio samples in a single step. A common example is an FFT transformation which consumes N audio sample tokens and produces a single spectrum token. Therefore, the rate of spectrum tokens is $\frac{1}{N}$ the sample rate. The FFT transformation may also need to process overlapping sample windows. That is, the FFT module reads N samples through an in-port and, after the execution, the window slides a step of M samples, where M and N are different. In this case, the rate of spectrum tokens is $\frac{M}{N}$ the rate of samples.

This example shows two different —though related— problems:

- Streams can flow at different rates. Like, for example, sample and spectrum streams do.
- Modules may need to process different numbers of tokens on each execution, regardless of the rate of its incoming streams. For example, an FFT module may require 512 samples while another FFT module may require 1024.

That means that the number of tokens a module consume and produce should be flexible, allowing modules to operate with different consuming and producing rates.

How to approach this problem is not obvious. Some real-life systems[3] perform multi-rate processing inside their modules while restricting inter-module communication to a single rate. Since the number of tokens that a module's algorithm needs is not the same as the number of tokens consumed on each execution, input and output buffering is needed inside the module. A weird effect of this approach is that the module execution not always implies its algorithm execution; when not enough tokens are ready for the algorithm, the module execution just adds incoming tokens to the internal buffers. Of course, this solution yields complex code in every concrete module.

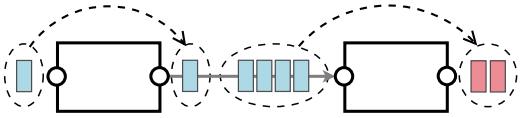

Figure 12: Two modules consuming and producing different numbers of tokens

Problem

How to allow modules accessing a different number of consecutive tokens on every stream port?

Forces

- The number of accessed tokens and the number of released tokens is independent for each port.
- For a given port, the number of accessed tokens and the number of released tokens are unrelated.
- Modules have to process a sorted sequence of stream data tokens (usually a time sorted sequence)
- All the stream tokens have the same priority.
- An out-port could be connected to multiple in-ports so that the tokens produced might be consumed by different modules.
- A module may need access to a number of consecutive tokens for each incoming stream to be able to execute.
- A module execution may produce a number of consecutive tokens for each output stream.
- Arbitrary consuming and producing rates in a network renders static scheduling of executions impossible.
- Feedback loops should be allowed
- Copy of coarse tokens may be an important overhead to avoid.
- Concrete module implementation should be simple.
- All modules are executed from a single process.

[3]One example is the JACK [9] audio server, with the Jamin mastering tool.

Solution

Design stream ports so that they support consuming and producing at different rates. This way they can adapt to the rate the module's algorithm needs, while keeping the buffering details outside the module. The in-port and out-ports should give access to N tokens from a queue[4] and should release M tokens on every module execution. Let the module developer define N and M for each port.

Give the ports an interface for accessing tokens —but only a window of N tokens at the head of the queue— and for releasing them. Releasing tokens means that M tokens will be dequeued and put away from the module reach. "Access" and "release" are operations implemented as port methods and they are to be called consecutively by the module execution method. Seen as a single operation, "access-and-release" is equivalent to "consume", when called on an in port, and "produce", when called on an out-port.

Make the ports own the tokens flowing between two modules, and make them responsible for all necessary buffering between out-ports and in-ports.

Buffers can be either associated with in-ports or with out-ports. In 1-to-N connections —that is, a single out-port connected to N in-ports— this decision makes the difference between heaving N different buffers (at the in-ports) or having a single buffer (at the out-port).

In the following paragraphs we discuss the implications of having buffers at the in-ports and at the our-ports.

Buffers at the in-ports:.

This is the simplest solution to implement. Give to each in-port an associated buffer (figure 13). Tokens being produced from an out-port are then passed to the connected in-port buffers. Tokens can be either passed by reference or by copy. Passing tokens by copy is easy to implement since each in-port is the owner of its tokens. Passing references, on the other hand, is more efficient because copies are avoided. This efficiency gain can be very important when tokens are to be passed to many in-ports or when tokens are coarse objects.

Of course, the efficiency gain associated with passing references instead of copying comes with a price: it is more difficult to implement. Given that multiple modules receive a reference to the same token, aliasing problems have to be avoided. In-ports have to be designed in a way that guarantees read-only semantics —or copy-on-write semantics— on the incoming tokens. The other aspect to be addressed here is the tokens life-cycle. Since token memory can not be freed —or recycled— while references to it exist, we need a reference-counting mechanism.[5]

However, passing references to the in-ports is not always feasible. Some applications may require their modules to operate on tokens placed on contiguous memory —examples of this are very common in the audio domain— as a consequence, such modules need the actual tokens data (not references) placed together in circular buffers at the in-ports.

This shortcoming can be overcome placing the buffers at the out-ports, which allows having both reference passing and contiguity. Again, we will see that efficiency comes with a price.

9[4]By queue we mean the abstract data type with its generic operations without making assumptions on it implementation.

9[5]Like C++ smart pointers

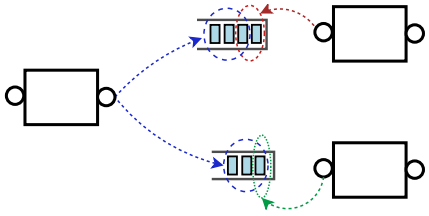

Figure 13: Each in-port having its own buffer.

Buffers at the out-ports:.

Having a single buffer for a 1-to-N connection —thus, associated to the out-port— allows benefiting from passing tokens by reference while achieving data contiguity (figure 14).

For that, the buffer must be implemented with a circular buffer. Not only the out-port is accessing the buffer but also the N connected in-ports.

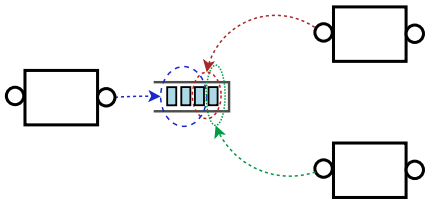

Figure 14: A buffer at the out-port is shared with two in-ports.

Allowing a buffer to be written by a producer and read by N different consumers, while allowing each one to produce or consume at a different cadence needs to done carefully. The following two basic restrictions must be enforced by design:

- The out-port can not over-write tokens that still have to be red/consumed by some in-port.
- The in-ports can not read/consume tokens that still have to be written/produced by the out-port.

Though complex to implement, this approach avoids the need for unnecessary copies of tokens and allows contiguous memory access to all involved modules.

Summing up:.

We have seen different strategies for implementing ports buffering that present a trade-off between simplicity and efficiency. Placing buffers at the in-ports and passing tokens by copy is the most simple approach. If copies are to be avoided, token references can be passed, but they have to be managed. Sometimes this is not enough. Apart from avoiding copies we need memory contiguity. Then, a circular buffer must be placed at the out-port and some restrictions must be enforced.

Consequences

Since all buffers adaptation is done at the ports level by the general infrastructure, concrete module implementors do not have to deal with buffers adaptation. That results in simpler, less error-prone code in every module.

Modules with different production and consumption rates can be connected together, as drawn in figure 12. As a result, this increases the number of the possible networks that can be built out of a set of modules.

The number of tokens stored in each port connection depends on two factors: In one hand, the requirements given by each port regarding the number of accessed and released tokens and, on the other hand, the scheduling policy in use.

This solution implies that, in general, the network will need a dynamic scheduler of module executions. To facilitate the task of such scheduler, modules may provide an interface to inform whether they are ready to execute or not. Such module method could be easily implemented in the module base class by delegating the question into every driver port, and returning the *and* combination if its responses.

Related Patterns

Multi-rate Stream Ports is applied in the context of systems that uses Stream and Event Ports, it addresses how stream ports can be designed so that they offer a flexible behavior.

Multiple Window Circular Buffer pattern addresses the low level implementation of the more complex variant of Multi-rate Stream Ports, that is *buffers at the out-ports*.

The design and implementation of *buffer at the out-ports* is a clear example of the Multiple Window Circular Buffer pattern.

Examples

In most of the systems reviewed, ports of the same type have all the same window size, and thus do not need to use this pattern. This is, for example, the case of the CSL [19] and OSW [5] frameworks and the visual programming tool MAX [23].

On the other hand the Marsyas [24], SuperCollider3 [16] and CLAM frameworks allow different window sizes, but they follow different approaches.

SuperCollider3 [16] features variable block calculation and single sample calculation. For example, modules corresponding to different voices of a synthesizer may consume and produce different block sizes. The SuperCollider3 framework permits embedded graphs that have a block size witch is an integer multiple or division of the parent. This allows parts of a graph which may require large or single sample buffer sizes to be segregated allowing the rest of the graph to be performed more efficiently.

Marsyas allows buffer size adaptation using special modules. CLAM —probably for its bias towards the spectral domain— is the most flexible, allowing any port connection regardless of its window size. CLAM sets up a buffer at each out-port.

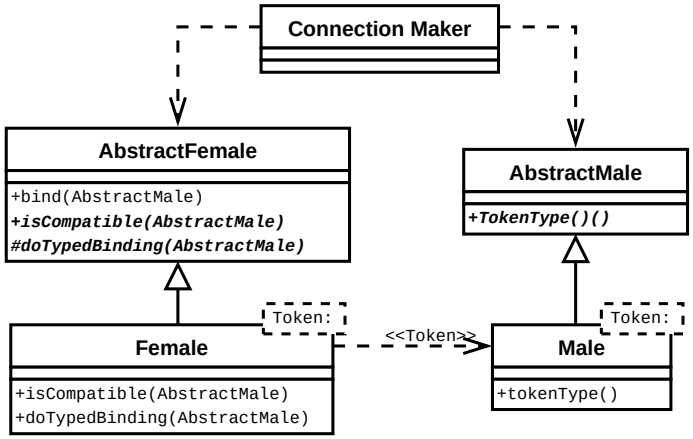

Figure 10: Class diagram of a cannonical solution of Typed Connections

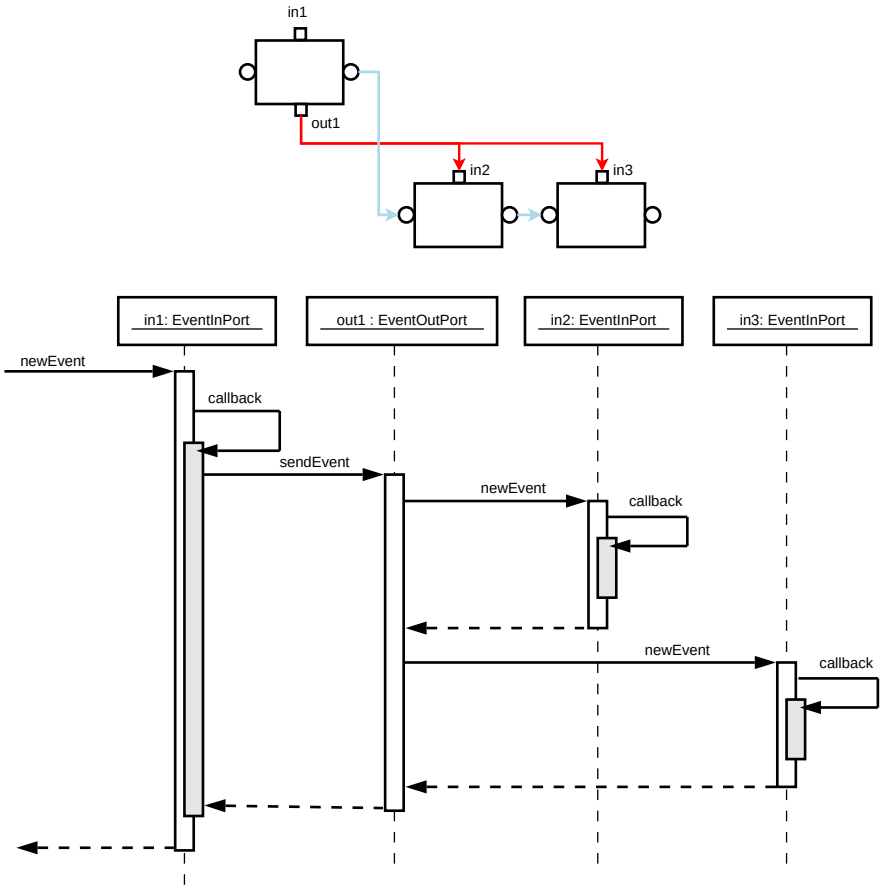

Figure 11: A scenario with cascading event ports and its sequence diagram.

8. MULTIPLE WINDOW CIRCULAR BUFFER

Context

As a result of incorporating the Multi-rate Stream Ports pattern, the ports-connection queues needs a complex behaviour, in order to access and release different number of tokens.

If it was not enough, such systems often have real-time requirements and some optimization factors must be taken into account: avoiding unnecessary copies, totally avoiding allocations and being able to work with contiguous tokens. Take for example (again) modules that performs the FFT transformation —delegating to some external library— upon a chunk of audio sample tokens; input samples must be provided to the library as an array. In the audio domain, not only FFTs need to operate with arrays, temporal domain processing is typically done that way too.

A simple implementation of Multi-rate Stream Ports consists in having a buffer associated to each in-port. But, unfortunately, this means copying tokens. The copy-saving implementation of Multi-rate Stream Ports requires a single buffer to be shared by an out-port and many in-ports. A design for this is not obvious at all. So, this is what this pattern addresses.

Finally, note that though a normal circular buffer is not suited for accommodating the given requirements, what we are seeking may be seen as a "generalized" circular buffer. Moreover, this can be useful in scenarios other than dataflow architectures.

Problem

What design supports a single source of tokens with one writer and multiple readers, giving each one access to a subsequence of tokens?

Forces

- Each port must give access to a subsequence of N tokens (the window).
- The subsequence of tokens should be in contiguous memory, since many algorithms or domain tool-kits and libraries works on contiguous memory.
- Windows sizes and steps should all be independent.
- Reading windows can only map tokens that have been already produced through the writing window.
- Allocation during processing time should be avoided, since (normal) dynamic memory allocation breaks the real-time requirements.
- All buffer clients executes in the same process.

Solution

Have a contiguous circular buffer with windows that maps (contiguous) portions of the buffer. There will be as many reading windows as needed but only one writing window. Associate the reading windows with the writing window because, as we will see they will need to calculate their relative distances. Also, provide them means for sliding along the circular buffer.

The modules (the buffer clients) executions must be done in the same thread. Its scheduling can be done either statically —fixed from the beginning— if all ports consuming and producing rates are known; or dynamically, which is much simpler to implement.

Windows clients need to follow the following protocol in order to avoid data inconsistencies:

- The access to windows mapped elements and the subsequent slide of the window must be done atomically in respect of other window operations. So, these operations might be regarded as a single read-and-slice (or write-and-slice) operation. Only when a window has finished the sliding, other clients can access their own window.
- A reading window can only start a read-and-slice (also known as *consume*) operation when it is not overlapping the writing window (overlapping other reading windows is perfectly fine). This reader-overlapping-writer problem indicates that the client is reading too fast. This problem should be detected and, as a response, the reading module should not be executed till more data has been written into the buffer.
- The writing window can only start a write-and-slice (or *produce*) operation when it is not overlapping the furthest reading region. Such overlapping is possible since regions are circulating over the underlying circular buffer. This writer-overlapping-reading problem indicate either that a client is reading too slow or that the buffer size is not large enough. When this is detected, the writing module should not be executed and should wait for the readers to advance.

The solution design uses the Layered pattern [10] for arranging different semantic concepts at different layers. Concretely, we distinguish three levels of abstraction (figure 15). Starting from the layer that gives direct service to the clients:

Windows Layer.

This is the upper or more abstract layer, which gives the clients a view of the windows advancing on an infinite buffer. It offers, at least, the following interface :

- Accessing the N contiguous elements mapped by the window.
- Advancing a window its slicing step (not necessarily N) as if the windows where on an infinite buffer.
- Checking if a window is ready to be accessed-and-slided.

The state of this layer keeps the relative distances between each reading and the writing window. This layer is in charge of detecting when a reading window overlaps with the writing window, and delegates other checks to its underlying layer.

Circular Windows Layer.

The state of this layer keeps the physical pointers (to a circular buffer) for each window and also provides physical pointers to the upper layer. This layer is in charge of detecting —and preventing— circular overlapping with a reading window. That is, the case when the writer is about to write on a still not read element.

Phantom Buffer Layer.

This is the lower layer, which knows nothing about writing and reading and writing windows and is solely dedicated to provide chunks of contiguous elements for each window. Therefore, this layer's goal is to to provide contiguous elements subsequences of size N or smaller. Where N is the size of the biggest window.

The main problem this layer has to solve is the discontinuity problem associated to circular buffers —the next element in a logical sequence of the last physical element is the first physical element. To idea behind the solution is to replicate the first N elements at the end of the buffer. This can be implemented using a data structure that we call "Phantom Buffer" and is presented in this catalog as the Phantom Buffer pattern.

Consequences

The non-overlapping restrictions might suggest that there always exists a distance between writing and reading windows and, thus, causing the introduction of certain latency. But this is not the case, because the non-overlapping restrictions only apply, at the time of an access-and-slide operation. After a window have been slided, it is perfectly legal to be in an overlapping state. This allows the reading windows to consume the same tokens that the writing window has just produced.

The reader-too-slow and writer-too-slow problems can be handled in the context of a dynamic scheduler. Before doing any access-and-slide operation, a *canProduce()* or *canConsume()* check is done, so that the operation can be safely aborted.

The consequence of the layered approach is a flexible design that allows changing the underlying data structure easily, without affecting the windows layer and its client. It also eases the implementation task since the overall complexity is split in well balanced layers which can be implemented and tested separately. On the other hand, those many levels of indirections might carry a performance penalty. However, it should be note that the implementation does not require polymorphism at all. Thus, when implemented in C++, with a modern compiler, most of the indirections should be converted to in-line code by the compiler, reducing the function-calls overhead.

In general, setting the window parameters can be done at configuration time; that is, before the processing or module executions starts.

Related Patterns

This pattern solves the *buffer at the out-port* approach of the Multi-rate Stream Ports pattern, which was the optimal one. Multiple Window Circular Buffer uses the Layered pattern [10].

Examples

This pattern is maybe a *proto-pattern* [28] as the authors only know their own implementation in the CLAM framework [27]. Nevertheless, CLAM is a general purpose framework and several applications with different requirements have proven the value of the pattern.

9. PHANTOM BUFFER

Context

The goal of Multiple Window Circular Buffer is to design a generalized circular buffer where, instead of having a writing and a reading pointer, we deal with a writing window and multiple reading windows. The difference between a window and a plain pointer is that a window gives access to multiple elements which, moreover, need to be arranged in contiguous memory. Multiple Window Circular Buffer relies, for its elements storage, upon some data structure with the following functionalities: One, to be able to store a sequence of any size ranging from 0 to MAX elements; and two, to be able to store each subsequence up to N elements in contiguous memory.

A normal circular buffer efficiently implements a queue within a fixed block of memory. But in a normal circular buffer the contiguity guarantee does not hold: given an arbitrary element in the buffer, chances are that its next element in the (logical) sequence will be physically stored on the other extreme of the buffer.

Note that, here, window management is not relevant at all because it is a responsibility of upper layers. Thus, the only concern of this pattern is how the low-level memory storage is organized.

Problem

Which data structure holds the benefits of circular buffer while guarantees that each subsequence of N elements sits in contiguous memory?

Forces

- Element copies is an overhead to avoid.
- Buffer reallocations are to be avoided.
- It should be possible for clients to read and write a subsequence of elements using a pointer to the first element. The rational is that modules might want to use existing libraries that, typically, use pointers as its input and output data interface.
- All buffer clients executes in the same process.

Solution

The *buffer with phantom zone —phantom buffer* for short— is a simple data structure built on top of an array of $MAX + N$ elements. Its main particularity is that the last N elements are a replication of the first N elements. This guarantees that starting at any physical position from 0 to $MAX - 1$, exists a contiguous subsequence of size up to N elements. In effect, this is clear considering the worst case scenario: take the element at position $MAX - 1$; let it be the first one in a subsequence; since it is a circular buffer of MAX elements, the next element is in the position 0, but positions from 0 to $N - 1$ are also replicated at the end (starting at position MAX); thus the contiguity condition is guaranteed.

Interface of a PhantomBuffer class should include two methods: one for accessing a given window of elements, and the other, for synchronizing a given window of elements. For example, in C++:

A client that wants to read a window, should call the *access* method, and read elements starting from the returned pointer. If the client wants to write, the sequence is a little different; first if should call *access*, write elements and,

```
template<class T> class PhantomBuffer
{
public:
    T* access(unsigned pos, unsigned size);
    void synchronize(unsigned pos, unsigned size);
    ...
};
```

Listing 2: PhanomBuffer class definition in C++

finally, call *synchronize*. This method synchronizes, when needed, a portion of the phantom zone with its counterpart in the buffer beginning. To be accurate, a copy of elements will only be necessary when the window passed as argument to *synchronize* have intersection with the phantom or the initial zone.

Summing up, a phantom buffer offers a contiguous array where the last N elements are a replication of the first N. Each write on the first or last N element is automatically synchronized in its dual zone. Thus, the client of a phantom buffer will always have access to chunks of up to N contiguous elements,

Consequences

As a result of this design, clients must be well behaved. This includes two aspects: The first is that clients that receive a pointer for a given window should not access elements beyond that window; the second is that, after a write, a client must call the *synchronize* method. Failing to do any of this might result in a serious run-time failure.

Certainly, this results in a lack of robustness. But this is the price to pay for the requirement of providing plain pointers to the window, and avoiding unnecessary copies and reallocations. However, the phantom buffer interface should not be directly exposed to the concrete module implementation. The port classes presents a higher level interface to the module while hiding details such as window parameters and synchronizations.

The circular buffer allocation should be done at configuration time and the phantom size depends (must be greater) on the maximum window size.

Related Patterns

This pattern can be regarded as a part of a more extensive pattern that provides a generalized circular buffer with many readers. In this context, the windows management issues are addressed in the more general Multiple Window Circular Buffer pattern. Phantom Buffer provides a refinement of the lower-level layer drawn in the general pattern. Therefore, these two patterns collaborate together to give a complete solution for a generalized circular buffer.

Examples

This pattern can be found implemented in the CLAM framework. Specifically in the *PhantomBuffer* class.

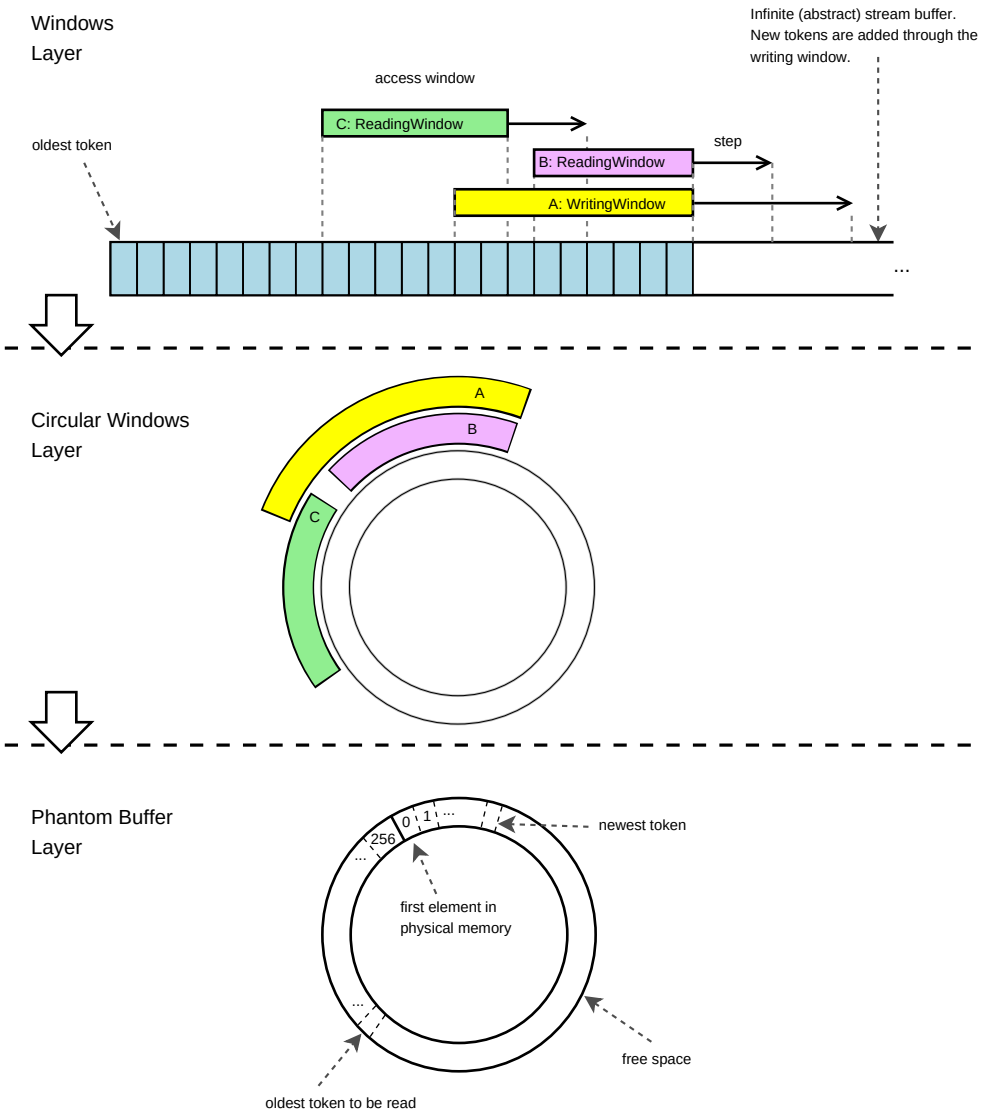

Figure 15: Layered design of port windows.

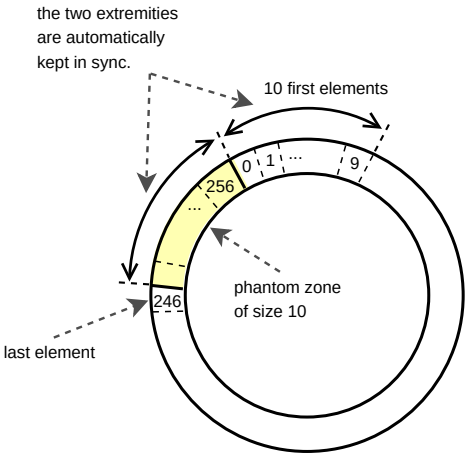

Figure 16: A phantom buffer of (logical) size 246, with 256 allocated elements and phantom zone of size 10.

10. RECURSIVE NETWORKS

Context

The potential of the interconnected modules model is virtually infinite. You can connect more and more modules to get larger and more complex systems. But module networks are normally defined by humans and humans have limitations on the complexity they can handle. So, big networks with a lot of connections are difficult to handle by the user, and this fact limits the potential of the model.

One of the reasons why audio systems become larger is duplication. Duplication happens, for example, whenever two audio channels have to be processed the same way, This duplication is very hard to maintain, because it implies having to apply repeated changes, and this is a very tedious and error prone process.

Duplication may happen also outside the system boundaries. The same set of interconnected modules may be present on several systems. Fixes on one of those systems do not apply to the other one so we have to apply it repeatedly and this is even more tedious and error prone.

Problem

How to reduce the complexity the user has to handle in order to define large and complex networks of interconnected modules?

Forces

- User defining big networks maybe too complex
- Human complexity handling is limited on the number of elements and relations
- Divide and conquer techniques help humans to handle complexity by focussing on smaller problems instead of the whole problem
- Duplications of sets of modules and connections is hard to maintain
- Reuse of previously designed networks helps on productivity
- Encapsulation hides details that can be useful on tracing the behavior of the system

Solution

By applying the 'divide & conquer' idea, we allow the user to define an abstraction of a set of interconnected modules as a single module that can be used in any other network. Some of the stream and event ports of the internal modules may be externalized as the stream and event ports of the container module.

Several internal *stream in-ports* may be merged as a single external one. So that, incoming stream tokens are read by all the internal stream in-ports. The same happens with in and out *event ports*. But it doesn't happen with the *stream out-ports*. The same reasons that forbid to *stream out-ports* feed a single *stream in-port* apply here.

If the system forbids merging ports on externalization, the externalized ports may be the internal ones. But when port merging is permitted, the user needs an abstraction on connecting a single port. This abstraction is given by a Proxy [12] port.

Depending on the implementation, the *proxy port* may act as a proxy on connection time or additionally on process time.

A *connect time port proxy* is a proxy port that delegates binding calls to the proxied ports. This way, during processing time the communication is done directly at non-proxy port level.

A *processing time port proxy* is a proxy port that acts as a the complementary in/out port for the internal ports. For example, an in proxy port is seen as out port for the internal ports connected to it. This is similar to have an identity module that just pipes tokens. The *processing time port proxy* adds overhead but it is useful when we need a clear boundary between inwards and outwards.

Also several approaches can be used for the flow control to handle *recursive networks*. One approach is to make the inner modules visible to the outer flow control, so that once all the modules are accessible by the flow control, all happens the same way it would happen if the recursive network was not there.

A second approach is to hide the inner modules to the flow control. This can be done by providing an inner flow control to the subnetwork. The subnetwork execution as module triggers the inner flow control. This approach is useful when a special flow control is needed. Also when we want to keep control on the proxied modules while processing.

Related Patterns

This pattern is a direct use of the Composite and Proxy [12].

The flow control approach that hides inner modules to the outer flow control by providing a inner one, is a Hierarchical Control [10].

Adjacent performance critical modules can be replaced by an optimized version as an static composition, trading flexibility by performance, using Adaptive Pipeline [20].

Examples

Most audio domain frameworks implement Recursive Networks. For example MAX/MSP [21], CSL [19], OSW [5], Aura [8], Marsyas [24] and CLAM.

CLAM provides examples of most of the variants explained before. CLAM *Processing Composites* are compiled networks that provide their own flow control and they are seen for the flow control as a single module. *Processing Composites*'s ports are connection proxies so, external modules are actually connected on processing time to the inner ports. On the other side, CLAM also provides dynamic assembled networks, In this case, dummy modules which pipes directly event and stream tokens, are used as process time port proxies.

11. PORT MONITORS

Context

Some audio applications need to show a graphical representation of tokens that are being produced by some module out-port. While the visualization needs just to be fluid, the processing has real-time requirements. This normally requires splitting visualization and processing into different threads, where the processing thread has real-time requirements and is a high priority scheduled thread. But because the non real-time monitoring should access to the processing

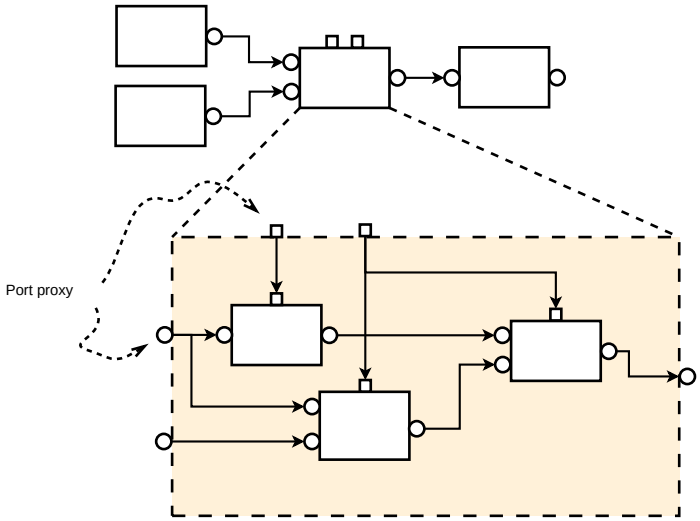

Figure 17: A network acting as a module.

thread tokens some concurrency handling is needed and this often implies locking.

Problem

We need to graphically monitor tokens being processed. How to do it without locking the real-time processing while keeping the visualization fluid?

Forces

- The processing has real-time requirements (ie. audio)
- Visualizations must be fluid; that means that it should visualize on time and often but it may skip tokens
- Just the processing is not filling all the computation time

Solution

The solution is to encapsulate concurrency in a special kind of process module, the *Port monitor*, that is connected to the monitored out-port. *Port monitors* offers the visualization thread an special interface to access tokens in a thread safe way.

In order to manage the concurrency avoiding the processing to stall, the *Port monitor* uses two alternated buffers to copy tokens. In a given time, one of them is the writing one and the other is the reading one. The *Port monitor* state includes a flag that indicates which buffer is the writing one. The *Port monitor* execution starts by switching the writing buffer and copying the current token there. Any access from the visualization thread locks the buffer switching flag. Port execution uses a *try lock* to switch the buffer, so, the process thread is not being blocked, it is just writing on the same buffer while the visualization holds the lock.

Consequences

Applying this pattern we minimize the blocking effect of concurrent access on two fronts. On one side, the processing thread never blocks. On the other, the blocking time of the visualization thread is very reduced, due that it only lasts a single flag switching.

Any way, the visualization thread may suffer starvation risk. Not because the visualization thread will be blocked but because it may be reading always the same buffer. That may happen when every time the processing thread tries to switch the buffers, the visualization is blocking. This effect is not that critical and can be avoided by minimizing the time the visualization thread is accessing tokens, for example, by copying them and release.

When this effect is too notorious, a solution might be to use three buffers. This way, even when the visualization is blocking the buffer, the processing thread may alternate on the other buffers. The constraints that should apply are:

- Two buffer marks are always kept the *reading* buffer and the *last written* buffer.
- Three mutually exclusive operations may happen:
  - The processing thread should choose to write on any buffer that has none of those marks.
  - When the processing thread ends writing it updates the *last written* buffer.
  - When the visualization thread access, it moves the *reading* mark to the current *last written* mark.

This solution is not that good for real-time as the one based on just two buffers. The former may block the processing thread, while the latter never blocks it.

Another issue with this pattern is how to monitor not a single token but a window of tokens. For example, if we want to visualize a sonogram (a color map representing spectra along the time) where each token is a single spectrum. The simplest solution, without any modification on the previous monitor is to do the buffering on the visualizer and pick samples at monitoring time. This implies that some tokens will be skipped on the visualization, but, for some uses, this is a valid solution.

The number of skipped tokens is not fixed, thus, this solution may show time stretching like artifacts that may not be acceptable for some application. Double/triple buffering on the port monitor the full window of tokens solves that.

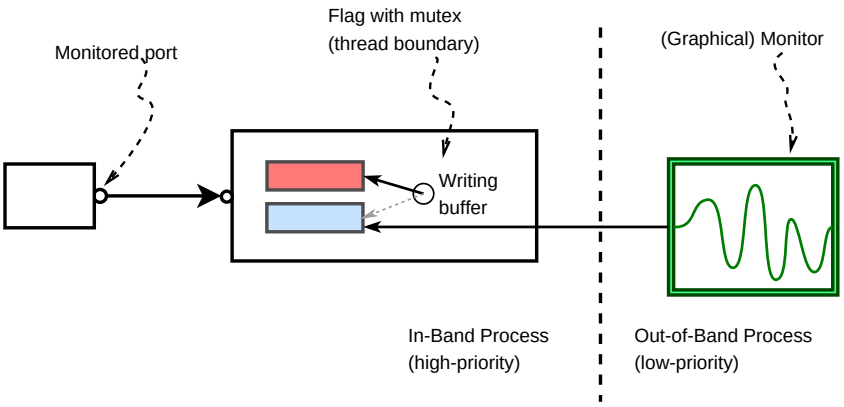

Figure 18: A port monitor with its switching two buffers

It is reliable but it affects the performance of the processing thread.

Related Patterns

Port Monitor is a refinement of Out-of-band and In-band Partition pattern [15]. Data flowing out of a port belongs to the In-band partition, while the monitoring entity (for example a graphical widget) is located in the out-of-band partition.

It is very similar to the Ordered Locking real-time pattern [10]. Ordered Locking ensures that deadlock can not occur, preventing circular waiting. The main difference is in their purpose: *Port Monitor* allows communicate two band partitions with different requirements.

Examples

The CLAM Network Editor [2] is a visual builder for CLAM that uses Port Monitor to visualize stream data in patch boxes. The same approach is used for the companion utility, the Prototyper, which dynamically binds defined networks with a QT designer interface.

The Music Annotator also uses the concurrency handling aspect of Port Monitor although it is not based on modules and ports but in sliding window storage.

12. ORGANIZING THE CATALOG

The 10 patterns presented in this catalog have different scope. Some are very high-level, like Semantic Ports and Driver Ports, while other are much focused on implementation issues, like Phantom Buffer). However, they act as a *pattern language* in the sense that each pattern references higher-level patterns describing the context in which it can be applied, and lower-level patterns that could be used after the current one, to further refine the solution. These relations form a hierarchical structure drawn in figure 19. The arcs between patterns mean "enables" relations: introducing a pattern in the system enables other patterns to be used.

This pattern catalog shows how to approach the development of a complete dataflow system for sound and music computing, in an evolutionary fashion without needing a *big up-front design*. The patterns at the top of the hierarchy suggest that you start with high level decisions driven by questions like: "do all ports drive the module execution or not?" and "do you have to deal only with stream flow or also with event flow?" It might also happen that at some point you will need different token types. Then you'll have to decide "does ports need to be strongly typed while connectable by the user?". Each of these questions is addressed by one *General Dataflow Pattern*.

Having reached this point, it is feasible to get a relatively simple but useful dataflow system allowing many module connections. However, it will have some limitations. For example, limited ports connectivity and fixed block size. You might find yourself in the need of implementing more flexible flows for stream and events. Then you'll want to look at the *Flow Implementation Patterns*. They show how to implement efficient stream flows with flexible connectivity (i.e. consuming/producing different number of tokens) and event controls that propagates immediately.

Humans might needs to interact with your system. Possible interaction includes building (complex) networks and monitoring the flowing data. This is what the *Human Usability Patterns* do. They can be introduced in the first stages of the system evolution or later on.

13. ACKNOWLEDGEMENTS

Authors of this paper would like to thank Dietmar Schuetz for being an excellent mentor in our first time writing patterns encouraging us to improve the patterns again and again.

We are deeply grateful to Ralph Johnson who provided enormous amounts of insightful feedback during the PLoP 2006 workshops. We couldn't have even imagined all the real-world experience that Ralph accumulates on the topic of dataflow systems! He also confirmed our impression that this area exposes many new patterns to be researched and documented.

Josep Blat, from the UPF, have provided great support for the CLAM project. This work has been funded by UPF scholarships and by a grant from the STSI division of the Catalan Government. We are also indebted with the Software Pattern community and the Linux Audio Community for giving us the chance to build upon their work.

14. REFERENCES

[1] X. Amatriain. *An Object-Oriented Metamodel for Digital Signal Processing*. PhD thesis, Universitat Pompeu Fabra, 2004.

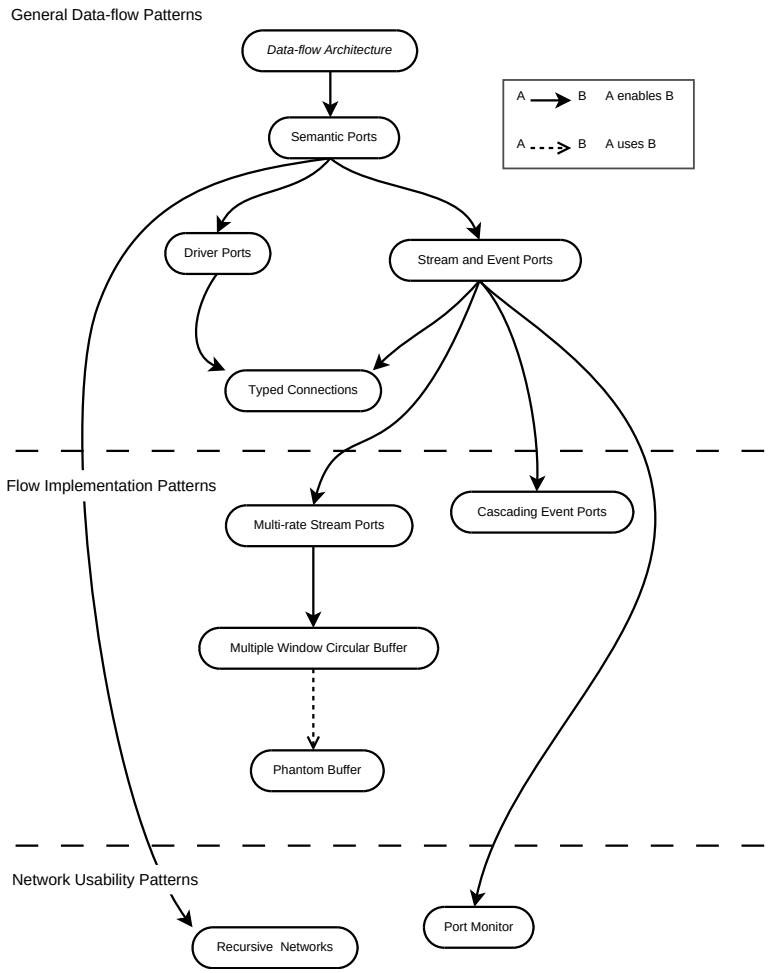

Figure 19: Introducing some patterns enables other patterns to be used.

[2] X. Amatriain and P. Arumí. Developing cross-platform audio and music applications with the clam framework. In *Proceedings of the 2005 International Computer Music Conferenc (ICMC'05)*, 2005. in press.

[3] X. Amatriain, J. Massaguer, D. Garcia, and I. Mosquera. The clam annotator: A cross-platform audio descriptors editing tool. In *Proceedings of 6th International Conference on Music Information Retrieval*, London, UK, 2005.

[4] J. Aucouturier. *Ten Experiments on the Modelling of Polyphonic Timbre*. PhD thesis, University of Paris 6/Sony CSL Paris, 2006.

[5] A. Chaudhary, A. Freed, and M. Wright. An Open Architecture for Real-Time Audio Processing Software. In *Proceedings of the Audio Engineering Society 107th Convention*, 1999.

[6] P. Cook and G. Scavone. The Synthesis Toolkik (STK). In *Proceedings of the 1999 International Computer Music Conference (ICMC99)*, Beijing, China, 1999. Computer Music Association.

[7] R. B. Dannenberg and E. Brandt. A Flexible Real-Time Software Synthesis System. In *Proceedings of the 1996 International Computer Music Conference (ICMC96)*, pages 270–273, 1996.

[8] R. B. Dannenberg and E. Brandt. A Portable, High-Performance System for Interactive Audio Processing. In *Proceedings of the 1996 International Computer Music Conference (ICMC96)*, pages 270–273. International Computer Music Association, 1996.

[9] P. Davis, S. Letz, F. D., and Y. Orlarey. Jack Audio Server: MacOSX port and multi-processor version. In *Proceedings of the first Sound and Music Computing conference - SMC04*, pages 177–183, 2004.

[10] B. P. Doublass. *Real-Time Design Patterns*. Addison-Wesley, 2003.

[11] B. Foote. Designing to Facilitate Change With Object Oriented Frameworks. Master's thesis, University of Illinois at Urbana Champaign, 1988.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] V. Lazzarini. Sound Processing with the SndObj Library: An Overview. In *Proceedings of the 4th*

International Conference on Digital Audio Effects (DAFX '01), 2001.

[14] E. Lee and T. Park. Dataflow Process Networks. In *Proceedings of the IEEE*, volume 83, pages 773–799. 1995.

[15] D. A. Manolescu. A Dataflow Pattern Language. In *Proceedings of the 4th Pattern Languages of Programming Conference*, 1997.

[16] J. McCartney. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4):61–68, 2002.

[17] T. Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77. April 1989.

[18] T. M. Parks. *Bounded Schedule of Process Networks*. PhD thesis, University of California at Berkeley, 1995.

[19] S. T. Pope and C. Ramakrishnan. The Create Signal Library ("Sizzle"): Design, Issues and Applications. In *Proceedings of the 2003 International Computer Music Conference (ICMC '03)*, 2003.

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

[21] M. Puckette. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, 1991.

[22] M. Puckette. Pure Data. In *Proceedings of the 1997 International Music Conference (ICMC '97)*, pages 224–227. Computer Music Association, 1997.

[23] M. Puckette. Max at Seventeen. *Computer Music Journal*, 26(4):31–43, 2002.

[24] G. Tzanetakis and P. Cook. *Audio Information Retrieval using Marsyas*. Kluewe Academic Publisher, 2002.

[25] J. Vlissides. *Pattern Hatching, Desing Patterns Applied*. Addison-Wesley, 1998.

[26] M. Wright. Implementation and Performance Issues with Open Sound Control. In *Proceedings of the 1998 International Computer Music Conference (ICMC '98)*. Computer Music Association, 1998.

[27] CLAM website: http://www.iua.upf.es/mtg/clam.

[28] Pattern and Software: Essential Concepts and Terminology, http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html.