FlowVR :
A Middleware for High Performance Interactive Applications

http://flowvr.sf.net

INRIA, LIG
Grenoble, France

LIFO/Université d'Orléans
Orléans, France

Main Contributors:
Jeremie Allard, Jean-Denis Lesage, Antoine Vanel,
Valerie Gouranton, Sebastien Limet, Emmanuel Melin,
Matthijs Douze, Bruno Raffin, Sophie Robert,
Matthieu Dreher, Jeremy Jaussaud, Xavier Martin

March 23, 2020

Contents

Documentation Organisation

The document you are reading intend to give a good understanding of FlowVR from the general concepts down to the details of application programming and the various associated utility tools. You should read it if you are starting with FlowVR. You can also use it as a reference document even if you already have some exprience with FlowVR.
Refer to the FlowVR doc repository for the extensive FlowVR related documentation.

Part I

Getting Started

Table of Contents

Chapter 1

Setting up your environment

Before diving into this manual, you should check out the "Getting Started with FlowVR" guide on FlowVR's wiki. It will help you through installing FlowVR and making sure your environment is properly set.
The next recommended read is the FlowVR-Appy tutorial, through which you'll master the basics of FlowVR application development, generation and execution.
In this manual's "Getting Started" section, we quickly review the process of generating and running a FlowVR application. If you feel at ease with these concepts, you may jump to Part II, "Overview".

Chapter 2

Using an existing application

Now that you have installed FlowVR, ran `flowvr-demo-tictac.sh` and `flowvr-demo-fluid.sh`, let us have a closer look at one of the examples : Primes.
All the examples are installed in share/flowvr/examples and share a similar structure.

2.1 Compiling and installing

Go into the primes directory:

```
cd [your installation path]
cd share/flowvr/examples/primes
```

Launch the following script that calls cmake to compile and install the application locally:

```
./make-app.sh
```

Once compiled, source the configuration script to set your environment variables :

```
source bin/primes-config.sh
```

The binaries you have just compiled will be added to your current shell environment.
Launching a FlowVR application is a two step process:

- Instantiating the application's network. This step is performed by a Python script that calls the `flowvr-appy` library. It produces intermediate files containing the sequence of commands required to launch the application.
- Actual application launching. This is controlled by the `flowvr` (see section 5.6) command. It reads through the intermediate files and sends commands to the FlowVR daemons.

2.2 Generating the application network

The primes application is an example with 3 communicating modules. There are several communication patterns that work with it. The `example` variable in `primes.py` switches between different configurations.
The following command instanciates `primes` based on the `primes.py` in the current directory and the default parameter value `example=4`

```
python primes.py
```

To change the parameter value for this particular example, pass it on the command line :

```
python primes.py 4
```

`flowvr` creates the following intermediate files:

- `primes.net.xml` : An xml file describing the network of the application : the modules, filters, synchronizers, and how they are connected together. You can use the flowvr-glgraph (see section 16.1) utility to view this file as a graph.
- `primes.cmd.xml` : An xml file containing internal FlowVR commands used when launching the application.
- `primes.run.xml` : An xml file containing the launching command for each module. Usually it relies on the `flowvr-run-ssh` (see section 16.3) launcher that uses an ssh connection to launch distant and local modules.

```
flowvr-glgraph primes.net.xml
```

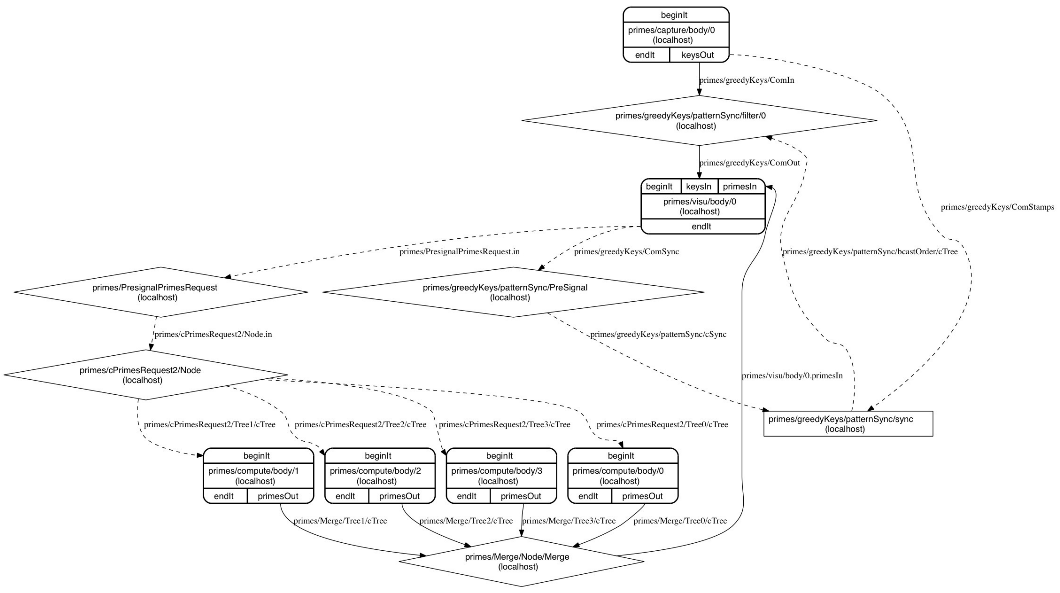

beginIt
primes/capture/body/0 (localhost)
endIt
keysOut
primes/greedyKeys/ComIn
primes/greedyKeys/patternSync/filter/0 (localhost)
primes/greedyKeys/ComOut
beginIt
keysIn
primesIn
primes/visu/body/0 (localhost)
endIt
primes/greedyKeys/ComStamps
primes/PresignalPrimesRequest.in
primes/greedyKeys/ComSync
primes/greedyKeys/patternSync/bcastOrder/cTree
primes/PresignalPrimesRequest (localhost)
primes/greedyKeys/patternSync/PreSignal (localhost)
primes/cPrimesRequest2/Node.in
primes/greedyKeys/patternSync/cSync
primes/cPrimesRequest2/Node (localhost)
primes/visu/body/0.primesIn
primes/greedyKeys/patternSync/sync (localhost)
primes/cPrimesRequest2/Tree1/cTree
primes/cPrimesRequest2/Tree2/cTree
primes/cPrimesRequest2/Tree3/cTree
primes/cPrimesRequest2/Tree0/cTree
beginIt
primes/compute/body/1 (localhost)
endIt
primesOut
beginIt
primes/compute/body/2 (localhost)
endIt
primesOut
beginIt
primes/compute/body/3 (localhost)
endIt
primesOut
beginIt
primes/compute/body/0 (localhost)
endIt
primesOut
primes/Merge/Tree1/cTree
primes/Merge/Tree2/cTree
primes/Merge/Tree3/cTree
primes/Merge/Tree0/cTree
primes/Merge/Node/Merge (localhost)

Figure 2.1: Application network of the Primes application (default example 4).

2.3 Local execution

FlowVR is daemon based. Before you start an application, you need to have a daemon running on each of the involved nodes. In this first case, the only machine is your local machine. In a separate terminal, start a daemon using the `flowvrd` command (see subsection 9.1.1). The daemon started with the `-top` option will provide various data related to the different components it is in charge of, such as their execution frequency.

```
flowvrd --top
```

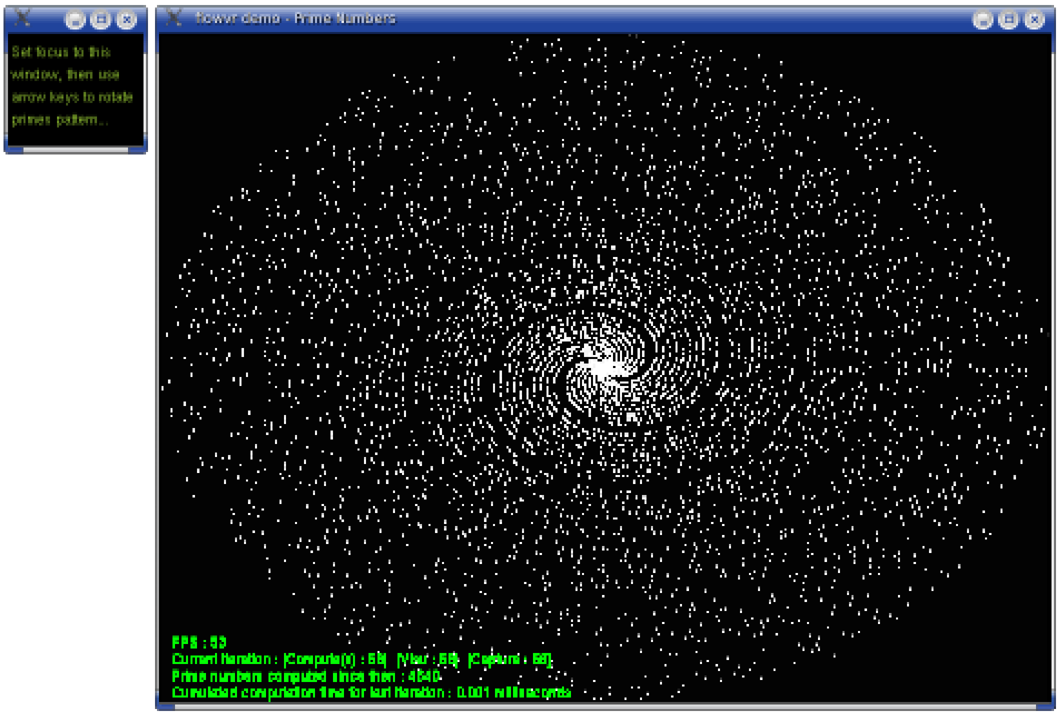
You can show a list of the available options with

```
flowvrd --help
```

The `--help` option should work with most flowvr-related executables.
The `flowvr` (see section 5.6) command takes as argument the prefix of the intermediate files that are generated by the application instantiation, `primes` in this case. The components are then loaded after parsing the `primes.cmd.xml` file obtained with `python primes.py`.

```
flowvr primes
```

The bigger window (see subsection 2.6.2) displays a list of prime numbers, moving at each computation ; the small one is dedicated to simple interactions with the rendering windows (focus on that small window and use arrow keys to rotate the view).



Set focus to this window, then use arrow keys to rotate primes pattern...

```
FPS : 53
Current Iteration : [Compute(0) : 68] [Visu : 68] [Capture : 68]
Prime numbers computed since then : 4640
Cumulated computation time for last iteration : 0.001 milliseconds
```

Figure 2.2: When running, the Primes application opens a visualization window and a small window for capturing keyboard events

When you launch an application, a minimalist shell is started in the same terminal. To stop the application (not the daemon), simply type :

```
stop
```

and press [Enter]. If you lose control of the application, you can kill the zombie processes by executing, in another terminal, the following command :

```
flowvr-kill
```

In extreme cases, restart the daemon before starting anew.

2.4 Distributed Execution

FlowVR allows to execute an application in a distributed context. In the following part, we will show you how to execute the primes application on different machines.
First you have to identify at least two machines. One of them could be your local machine.
Before launching, make sure that:

1. All machines involved must have access to FlowVR libraries and binaries. The easiest way is to have a shared file system, such as nfs, that gives all hosts access to your installation directory for FlowVR and Primes.

2. Each machine access paths should include FlowVR libraries and binaries. The easiest way is to add the `flowvr-suite-config.sh` to the `.bashrc` and `.bash_profile` on each machine involved.

3. FlowVR relies on ssh to access the distant machines. Make sure you allowed distant connections via ssh without password authentification on all hosts (install ssh keys).

4. Allow FlowVR to open windows on the different machines if distant modules need to open windows on their local machine.

5. Update the host list in `primes.py`. NB that if you involve distant machines, mapping components on the local host must use the actual name of the local machine and *not* `localhost`.

   For instance a possible hostlist for Primes is:

   ```
   host_list = ['pc1', 'pc1', 'pc2', 'pc2']
   ```

6. Start `flowvrd` on *each* involved machine.

7. Start the application:

   ```
   python primes.py 8 && flowvr primes
   ```

   If you run into a "`Long flush time`" warning, it might come from the fact that the machine from which you launch the app is not recognized on the network. You can solve this problem by using the `-b` option:

   ```
   flowvr primes -b xx.xx.xx.xx
   ```

   where xx.xx.xx.xx is the ip address of your machine.

   You can check whether the various components are distributed as expected through the output of the `flowvrd` instances.

2.5 Distributed Execution Over High-Performance Networks

The method described earlier covers the steps to run your application over a TCP/IP network, which is the default daemon behavior. If you want to run FlowVR over a high-performance network such as Infiniband, all you have to do is pick a different daemon 'Net' plugin and start your daemons with a slightly different command. Have a look at Section 9.3, 'Using MPI as a network layer'.

2.6 Insitu

2.6.1 Example

The fluid_insitu example show the some of the core principles applied to an *insitu* scenario. We want to instrument an MPI simulation to perform insitu analysis or online visualisation. We also want to do so using whist alterer as less as possible the simulation code.
We're going to use a fluid simulation and simplify the data on the fly on an helper core. Then, we'll perform some analysis and visualisation on a dedicated node.

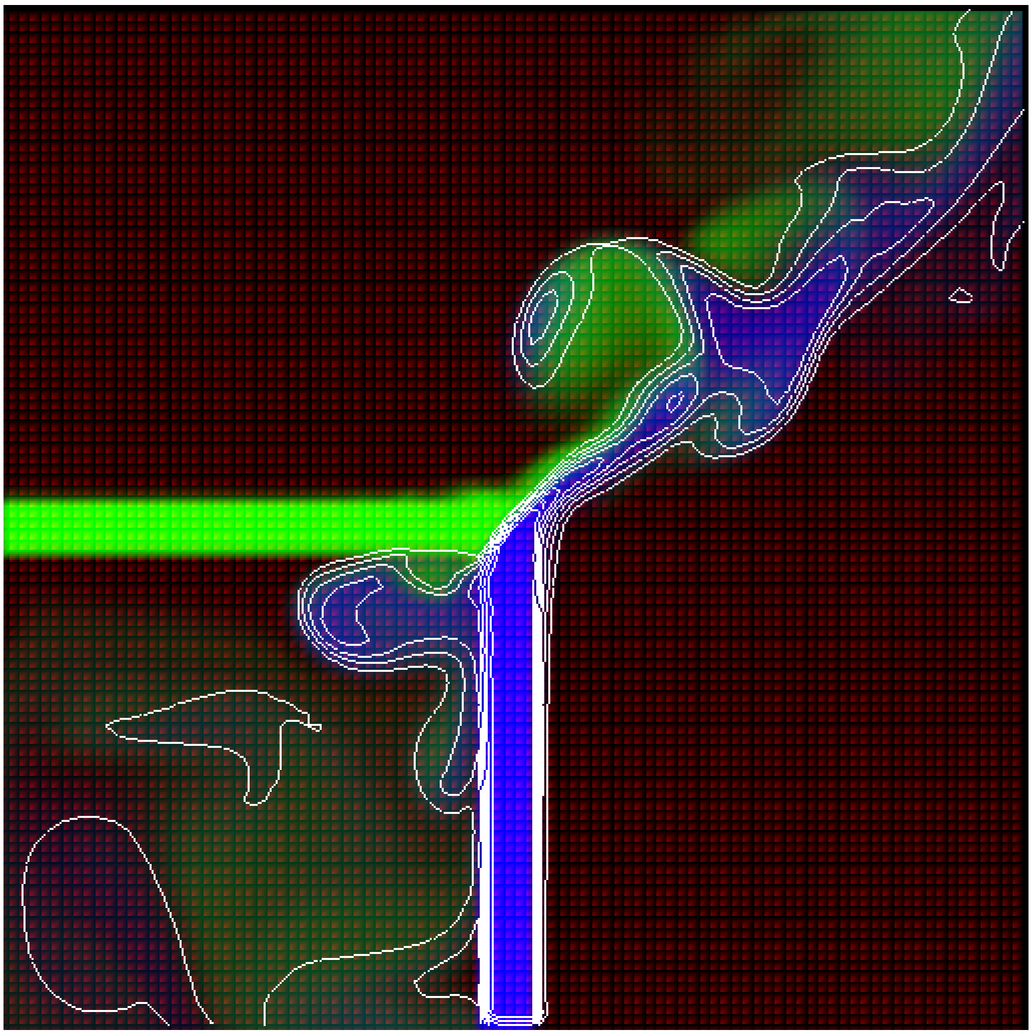


Figure 2.3: Screenshot from the fluid_insitu example, showing the rendering of the data simplified insitu. It also displays isolines computed on the rendering node.

2.6.2 Synchronisation of the insitu modules

Let's begin with the insitu modules. They are going to simplify the simulation data to reduce both data transferts and post processing costs. They will also discard frames depending on the frequency at which the user wants the insitu processsing to be done. Wether we decide to discard or process a frame, we need all those modules to take the same decision for each frame. Hence we are going to keep all those modules synchronised.
The user chosen frequency is to be broadcasted to all the insitu modules. We do so using a blocking port so that every module process each frame with the same frequency message, thus ensuring the coherency.
The frequency also goes through a GreedySynchronizor (the `sync` port of which is the `endIt` of one of the synchronized modules) so that you always use the last value given by the user.

2.6.3 Visualisation on the dedicated core

The simplified data is gathered to a dedicated node using a *tree merge* construct. It is then forwarded an analysis module computing isolines which are, along the simplified data, send to the visualisation module.

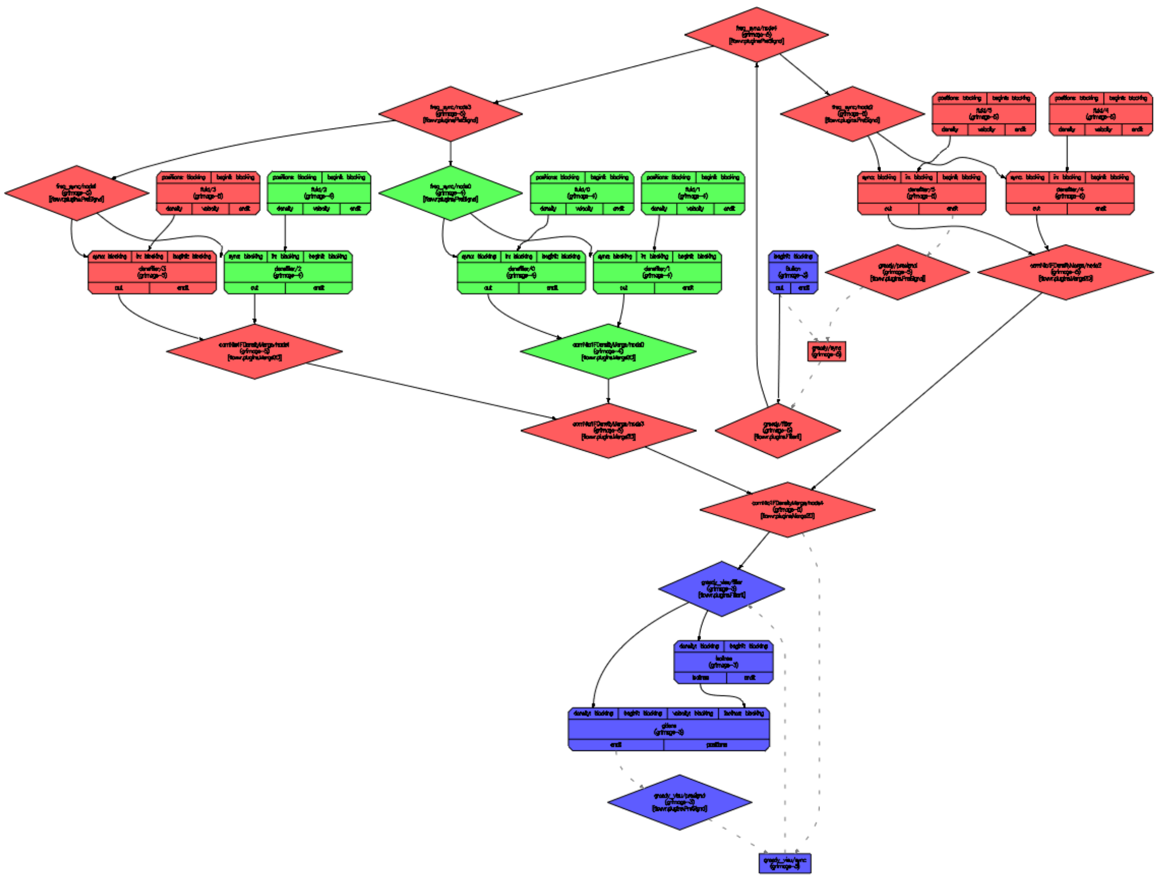


Figure 2.4: Exemple of a graph generated by the fluid_insitu example on three machines. Each color denotes a machine.

We added a GreedySynchronisor after the *tree merge* so that you always visualize the very last recieved frame.

Part II

Overview

Table of Contents

Chapter 3

Application Model

The FlowVR library provides a programming and execution environment for interactive applications. Its goal is to enforce application modularity for leveraging software engineering issues while enabling high performance executions on parallel and distributed architectures. The target architectures are multi-processor machines, PC clusters and grids.
A FlowVR application is a set of distributed iterative tasks. These tasks, also called components or modules, ignore networking issues. They simply get messages from input ports, process the received data and provide results on output ports. They are interconnected using FIFO data communication channels, forming a network. At execution, these tasks are distributed on the target machines and the run-time environment (flowvrd) takes care of moving data efficiently between tasks.
FlowVR relies on a data-flow model, often used in scientific visualization. This model is well adapted to interactive applications and it enables to extract parallelism for efficient parallel execution without requesting the application developper to be a parallel programmer. This is where object oriented approaches like scene-graphs usually fail.

3.1 Module

The base component of FlowVR is the **module (see section 11.5)**. A module has input and output ports. It executes an endless loop and its application programming interface (API) is built around three basic instructions:

- **wait**: lock the module as long as no new message is available on each of its **connected input ports**, except for a special type of non blocking input ports called event ports (see 6.2.3.4).
- **get**: get a pointer on the new message received on a given port.
- **put**: publish a new message on a given output port.

The rest of the module is up to you as long as these semantics are respected. It can be a process, a thread or a group of collaborating threads. The API is minimal and makes it easy to turn existing code, be it multi-threaded, parallel or even GPU kernel, into modules.
Every module has:

- an output port **endIt** where a message is sent every time the module enters the **wait** instruction,
- an input port **beginIt** that, when active (as in linked to another module's output port), locks the module as long as no message is received (the message is used only as an event, the content is ignored).

Similarly to classical parallel programming environments, the FlowVR API provides low level message handling functions. For instance, FlowVR does not provide data structure serialization. A message is simply a sequence of bytes a module has full access to. This enables a fine-grain control on data copies that can induce significant costs for large messages. Higher level message handling methods can be implemented using this API.

3.2 Filter

A **filter (see chapter 8)** has input and output ports. As opposed to modules that can only access the last message received on each input port, a filter has access to the full buffer of incoming messages stored locally. It can modify or discard any of these messages. A filter can for example re-sample messages, by discarding all incoming messages except the last one then forwarding it on its output port.
The API to develop modules is intentionally simple yet constraining (see the **wait** for instance). The goal is to keep module development simple. On the other hand, filters offer more freedom in particular regarding buffer access. Usually an application developer does not have to develop new filters. Filters provided with FlowVR perform generic message handling tasks, making them easy to reuse in multiple applications. Combining filters and modules makes it possible to implement complex behaviors, as we will see in the following section. You can also make your own custom filters.

3.3 Connection

A **connection (see section 3.3)** is a simple FIFO channel connecting an input port to an output port. Several connections can share a single output port as their source. In this case each message available on the output port is broadcasted on each connection. An input port can only have one incoming connection. If you need to connect multiple output ports to a single input port, what you are looking for is a filter.
The simplest application a user can write involves two modules connected through a connection. The size of the buffer associated to a connection is only limited by the amount of available memory. If the receiver is slower than the sender, an overflow will occur once the memory is saturated. We will see how to avoid that situation later.
Each message sent on the FlowVR network has its payload as well as a list of stamps. Stamps are lightweight and identify the message. Some stamps are automatically set by FlowVR. The user can also define new stamps if required. A stamp can be a simple ordering number, the id of the source that generated the message or more advanced data like a 3D bounding volume. To some extent, stamps enable to perform computations on messages without having to read their content. The stamp list can be sent on the network without the message payload if the destination does not require it. It may improve performance by avoiding useless data transfers. Such a message is called a **STAMP** message in opposite to a **FULL** message (stamp list and payload). A full connection carry FULL messages, while a stamp connection carry only STAMP messages. If a STAMP connection is connected to a FULL output port, it will automatically extract the stamp list from the emitted messages and route to the destination this STAMP message.

Chapter 4

Typical uses

4.1 Usual synchronization policies

We introduce a simple application that gives a glimpse on the flexibility FlowVR can offer. This application connects one data producer, the module *compute*, and the *visu* consumer module (see Figure 20.4). This could be a 3D mesh generator linked to an OpenGL rendering process. We show how we can change the way data are exchanged between both modules simply by changing the network. FlowVR enables to separate task implementation (see section 11.5) from their assembly (see Part III). For all these examples, the modules do not have to be recompiled. Only the network specification changes. It enables to easily test various network options.

4.1.1 Data-Driven Policy

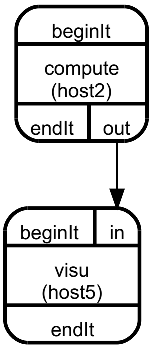


Figure 4.1: A simple FlowVR application with only one connection between two modules.

The simplest way to link the two components is to use a single FIFO connection (see Figure 20.4). In this case the consumer frequency reaches at most the one of the producer. If the consumer is slower than the producer, the number of messages sent will grow because the consumer will not be able to process them. In the model, the buffers associated to FIFO connections are unlimited, but practically they are limited by the amount of memory available. So a memory overflow can occur.

4.1.2 Demand-Driven Policy

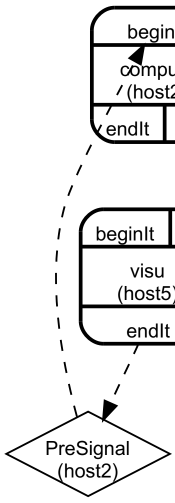


Figure 4.2: The *visu* module pulls on demand messages from *compute*.

A simple approach to avoid such overflow is to switch from a push to a pull paradigm (see Figure 4.2). Because FlowVR allows cycles in the network graph, it is possible to have the*visu* module controlling the frequency of *compute*. Each time*visu* ends an iteration, it sends a request message on its*endIt* port. This message is forwarded through a*PreSignal* filter to the*beginIt* port of *compute*. This message will unlock *compute*, that will proceed to produce a new data. The *PreSignal* filter is required to avoid the deadlock caused by the cycle. This filter sends a first message before forwarding incoming messages : it puts an initial token into the cycle to unlock it.

4.1.3 Data-Driven Policy with Frequency Constraint

To control the frequency of a module, another approach is to use a *MaxFrequency* filter (see Figure 4.3). Each time the module ends an iteration, it sends a request message to this filter (*endIt* port). From these messages, the filter is able to compute the frequency of the*compute* module. The filter can control the module frequency by sending messages to the *beginIt* port up to a limit custom maximum frequency. Notice that here we have a cycle too. In this case the*MaxFrequency* filter is in charge of sending the unlocking first message. As this filter is always used in cycles, it is convenient to give it this responsibility.

4.1.4 Asynchronism Based on Resampling

For large interactive applications, having all modules running at the frequency of the slowest one can severely affect the reactivity of computations. A possible approach is to consider that a data stream is a sampled signal that can be resampled. The consumer only gets the data it can process, discarding the other ones.

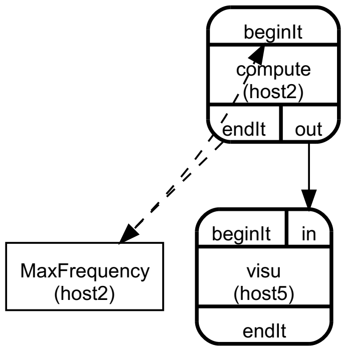


Figure 4.3: The max frequency filter bounds the frequency of the *compute* module.

We present the greedy pattern (see Figure 4.4), a basic resampling pattern favoring the reactivity by providing the consumer with the most recent data available. We use filters that have access to the full buffer of messages stored locally to implement this implement sampling policy. This pattern is organized around a special synchronization filter, called a synchronizer, and 2 filters.
Each time the module*visu* ends an iteration, the synchronizer *sync* receives one message. A *PreSignal* filter is set along the message path between *visu* and *sync* to emit at starting time a first message to unlock the cycle created by synchronizer. *sync* also receives a stamp message for each message sent by the producer. When the synchronizer receives a request from*visu*, it forwards to the*filterIt* filter the stamp of the last stamp message received from*PreSignal*. *filterIt* waits to receive the full message having this same stamp, discards all older messages locally stored and forwards this message to *visu*. If no message is available when the synchronizer receives the request, it tells the module to reuse the last message already received, sending it a special empty message.

4.1.5 Gathering Data from Multiple Producers

A VOIR *should be moved to the "Component assembly principles"->"Filters" sub-section* FIN
Assume now that *compute* is a parallel application starting four modules *compute/O*,... *compute/3*. We could modify *visu* to have four input ports to receive the data part produce by each process. Though possible, this is an approach we usually avoid with FlowVR. It makes the *visu* module code dependent of its external environment. To enforce the modularity of the application, *visu* is not modified and the reduction of the partial results to one message is performed by an extranal filter (see Figure 4.5). Performance is affected if this involves data copies that would not perform a modified *visu* module. The reduction can also be implemented along a binary tree merging pattern (see Figure 4.6) for performance reasons. Switching between these different patterns is external to the module code. It involves neither code modification nor module recompilation.

4.1.6 Component Mapping on Hosts

In the previous figures you can see that a host name, *host1*, *host2*, etc. is associated with each component. This is the host machine that executes this component. Before to start an application the user has to specify the mapping of modules (see section 5.7), while filters are usually mapped automatically according to module mapping.

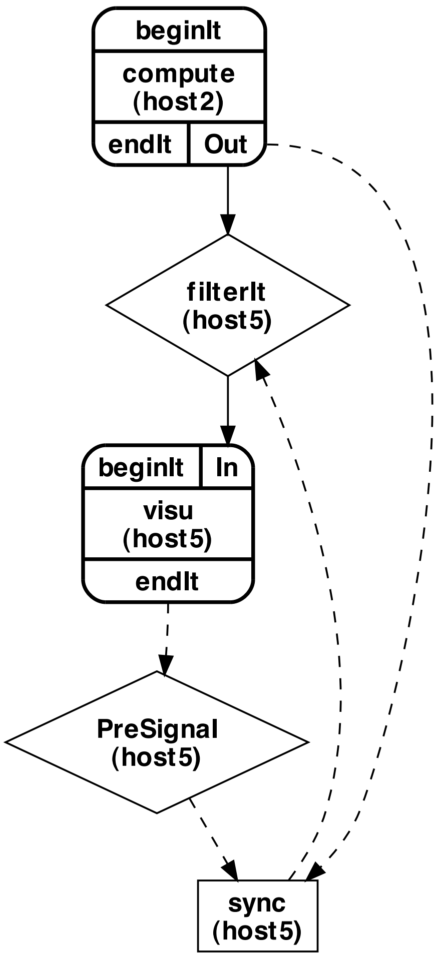


Figure 4.4: The greedy pattern enables *compute* and *visu* to run at independent frequencies. Dashed arrows represent STAMP connections where only the message's stamps are transferred.

4.2 Component assembly principles

A VOIR *A revoir en details* FIN
Designing a complete application can be difficult for the FlowVR beginner. The use of network components and their placement in an application is not always easy. The goal of this section is to give some recipes about application network design.

4.2.1 Module-to-Module Connection

A connection between two modules links an output port to an input port. It is the responsibility of the FlowVR user to ensure that the type of data sent by the output port is compatible with the receiving module (see Figure 4.7). In this case the connection between the modules is FIFO: the receiver module *visu* gets the message data in the order they were put by the emitter module *compute*. Each new call to a `wait` performed by *visu* blocks until a message is available on the port.

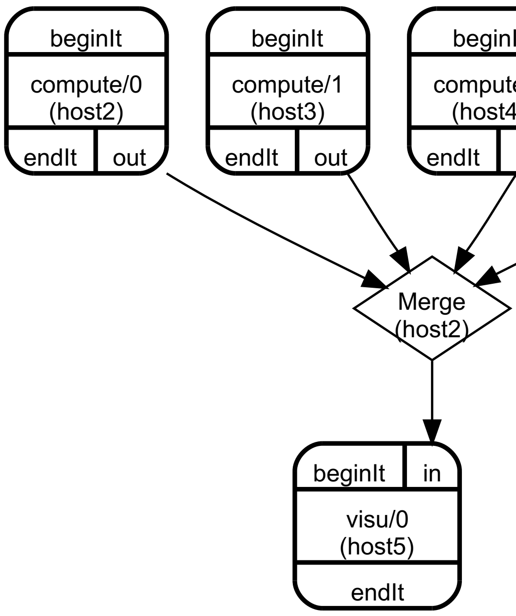


Figure 4.5: Merging messages with one filter.

4.2.2 Connection Cycles

A VOIR *putting a default value on an output port before the first call to the* `wait` *method is undefined behavior.* FIN
Cycles in connections are possible. In the Primes example, the *compute* module send calculated prime numbers inside small packets, in order to let the *visu* module build the spiral pattern progressively. To avoid the latter to be flooded with messages, a possibility demonstrated in the *1_fifo_one_to_one* and *2_fifo_two_computes Primes* test cases is to close the communications in a cycle. A connection relying on `endIt`/`beginIt` predefined ports is then added to let *visu* warn *compute* module is ready to receive new data. This creates a cycle (see Figure 4.8). If the modules have been properly programmed, i.e putting a default value on each output port before the first call to the `wait` method (see 6.2.1.3), the modules will not deadlock when calling the `wait`.

4.2.3 Connection Fan-out

Messages can easily be duplicated and sent along several connections (fan-out) using a routing node. For instance it is very useful for parallel computations where several routing nodes are used to build a broadcast along a binary tree (see Figure 4.9). Each Message put by `visu` will be received by all `compute` modules in a FIFO order.

4.2.4 Filters

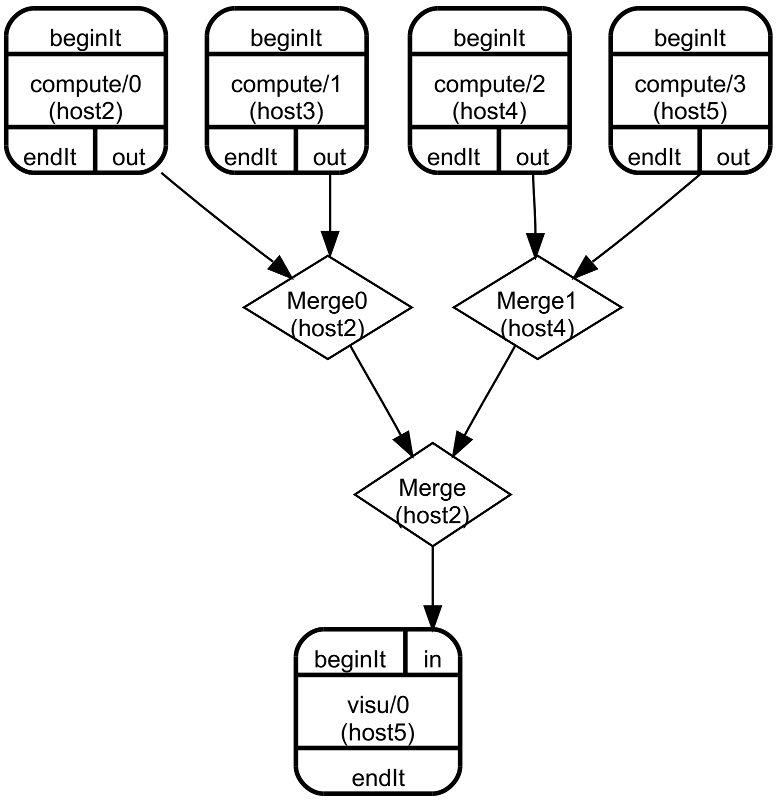


Figure 4.6: Merging messages along a binary tree of filters.

Filters are usually used to control the dataflow between modules. A filter can be used to send/discard chosen messages along a connection (keeping only the messages verifying some preconditions, like an iteration number, for instance), to gather data from different incoming connections, to create a new message, etc.

4.2.4.1 Filters versus Modules

Often the beginner uses a (meta)module where a filter would have been more appropriate. Here are the key arguments to help you make the right choice:

- Filters are plugins loaded in the FlowVR daemon while modules are external processes. A higher performance is generally obtained using filters.
- Filters have access to all incoming messages while modules just receive the last message available. Thus, filters have more freedom to handle messages.

4.2.4.2 Example of Filter Uses

Amongst filters provided with FlowVR (see section 8.4), the merge and scatter filters are good examples of what is possible to do with filters.

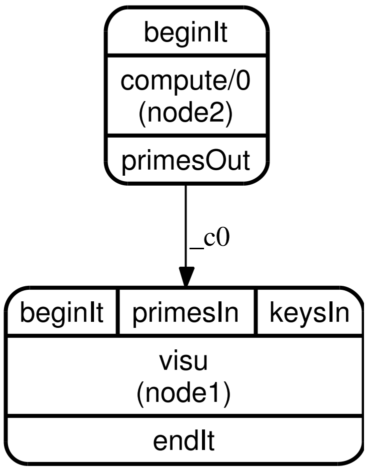


Figure 4.7: Two modules connected together. Message exchange follows a FIFO mode.

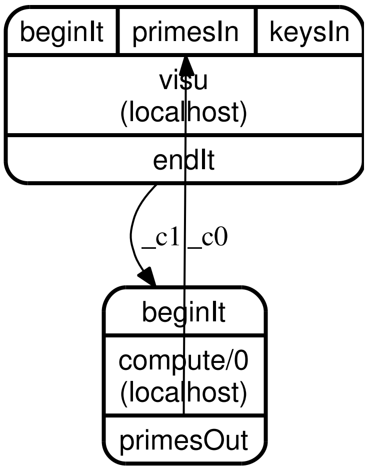


Figure 4.8: Cycle between two modules.

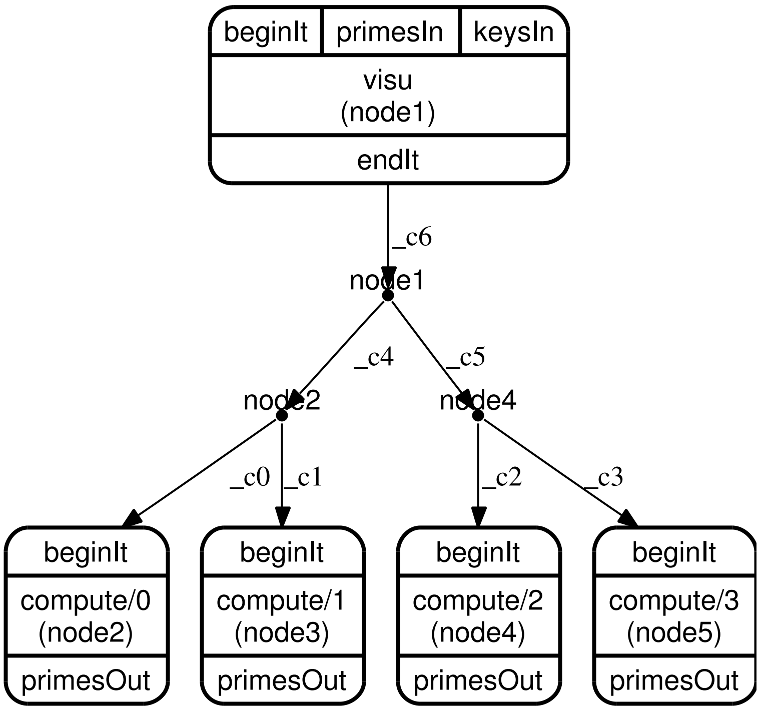


Figure 4.9: Broadcast along a binary tree using routing nodes.

The scatter filter enable to split a message in N parts, each part being sent on a different output. The scatter is often necessary when using parallel modules. Scattering allows to reduce the bandwidth by sending only data used by the receiving modules.
Similarly, merging messages with a merge filter is sometimes required to built a message containing the complete result of a parallel metamodule where each module produces only a part of the result. The first example (see Figure 4.5) is a merge to gather the results of the `compute` metamodule of the prime example, while the second example (see Figure 4.6) uses a tree of merge filters to built the density grid computed for the fluid example.

4.2.5 Dataflow Synchronization Modes

The various test cases of the *Primes* example aim at demonstrating how network design takes a crucial part in the final behavior of a distributed application. We encourage you to run the different tests in order to *feel* the impact of each modification. Please refer to the share/flowvr/examples/primes/README for the launching commands.

4.2.5.1 FIFO mode

This is the default mode when no synchronization object controls a connection. In this case, all messages produced by the emitter will be consumed in the same order by the receiver. No data will be lost or inverted in FIFO mode. Consequently all modules of the application will work at the speed of the slowest module. Moreover, if a module like a tracker works a lot quicker than his receiver, the size of the buffer holding the data between the modules will increase continuously, conducting to an overflow (crash) during execution.

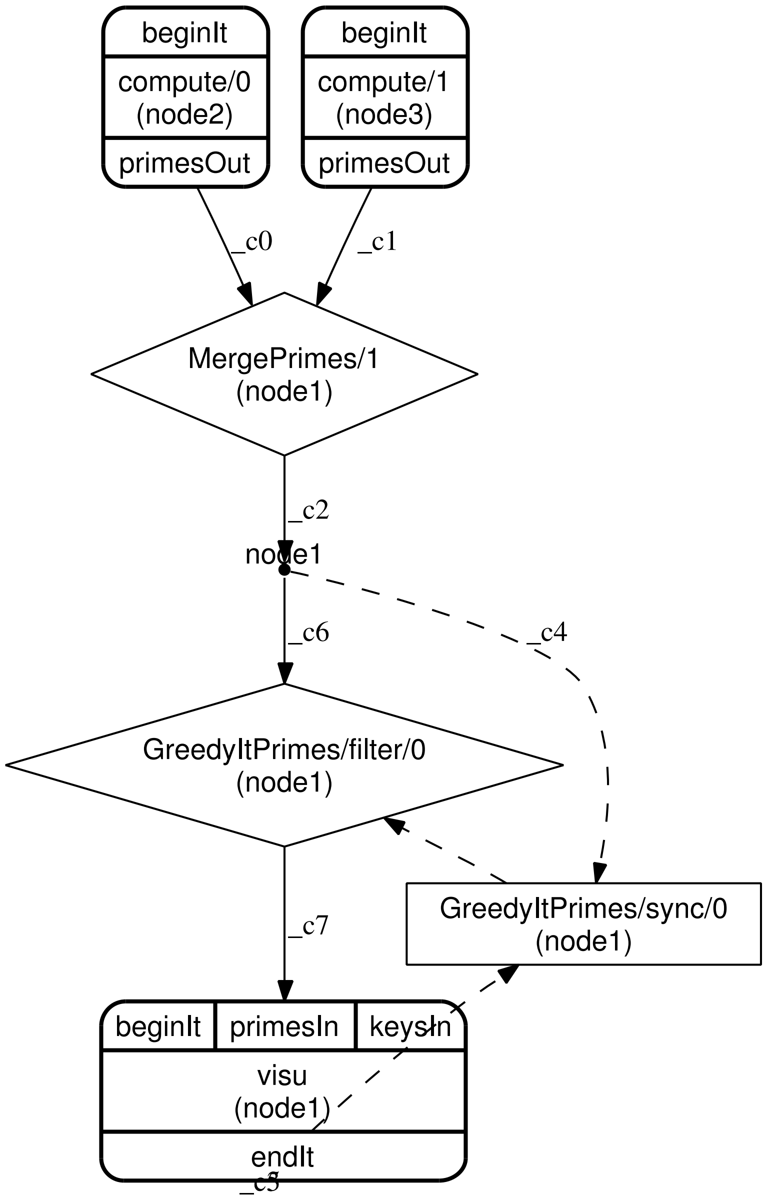


Figure 4.10: The Primes application with 2 computing node and 1 rendering node. The connection is filtered using 2 forms of merge filters whose role is to accumulate prime numbers packets in a single message, delivered to the visu module at the end of the chain.

In the *Primes* example, the tests *1_fifo_one_to_one* and *2_fifo_two_computes* uses a FIFO mode (see Figure 4.11) that makes the *visu* module work at the same speed as the slowest connected module, either *compute* or *capture*. To prevent messages overflow, the communication scheme is organized in a cycle (see subsection 4.2.2).

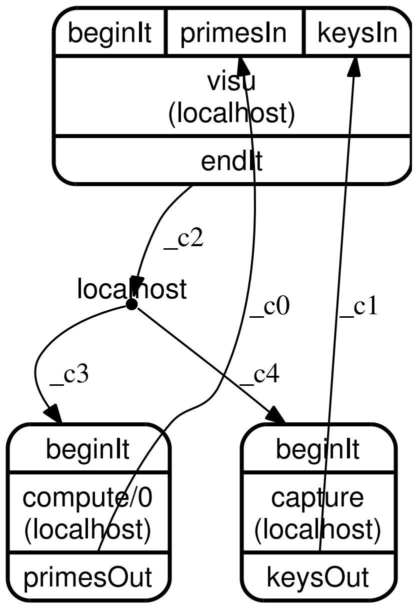


Figure 4.11: Primes modules organized in a FIFO cycle.

4.2.5.2 Sampling (or Greedy) Mode

The sampling (or greedy) mode refers to sampling systems where components periodically read data whose values are independently (asynchronously) updated (digital temperature sensor for instance).
This mode is designed for low latency and real time applications. This synchronization method allows modules to work at their maximum speed. At each new iteration, the module uses the latest data available, ignoring all precedent data. By this way, the latency can be improved but some data produced are lost. This mode can be implemented between two modules using a `GreedySynchronizor` synchronizer and a `FilterIt` filter.
The synchronizer has two input ports: `stamps` and `endIt`. `stamps` should be connected to the source of the data while `endIt` should be connected to the `endIt` activation port of the module receiving the data. With this information, the synchronizer will know when the module has finished an iteration and thus needs a new data. It will read the stamps received on the `stamps` port and find the most recent message. The chosen stamps will be forwarded to the `order` output port. This port should then be connected to the `FilterIt` filter on its `order` input port. The filter has another input port named `in` which should be connected to the same source of data that the synchronizer is. Using these informations, the filter will forward to its `out` output port the messages corresponding to the received orders. This output should then be connected to the input port of the destination module which will so receive the sampled data.
The *Primes* example introduces this technique in the test *4_capture_greedy*, between the *capture* and *visu* modules (see Figure 4.12).

The receiving module *visu* has its `endIt` port connected to the synchronizer `GreedyKeys/sync/0`. It enables this module to ask a new message to the synchronizer each time it completes an iteration. The synchronizer has a second input port to receive the stamps of each message sent on the output port of the source module *capture*. When the synchronizer receives a message from *visu*, it selects amongst the available stamps the most recent one and sends this stamp to the filter `GreedyKeys/filter/0`. If no new data has been received since the last iteration, the synchronizer replays the same stamp. The filter waits on its input port a message with a matching stamp and forward this message on its output port. This is the message that will receive the module *visu* for the pending iteration. Notice that thanks to the greedy, the initial cycle introduced with FIFO connections (see Figure 4.11) is suppressed.

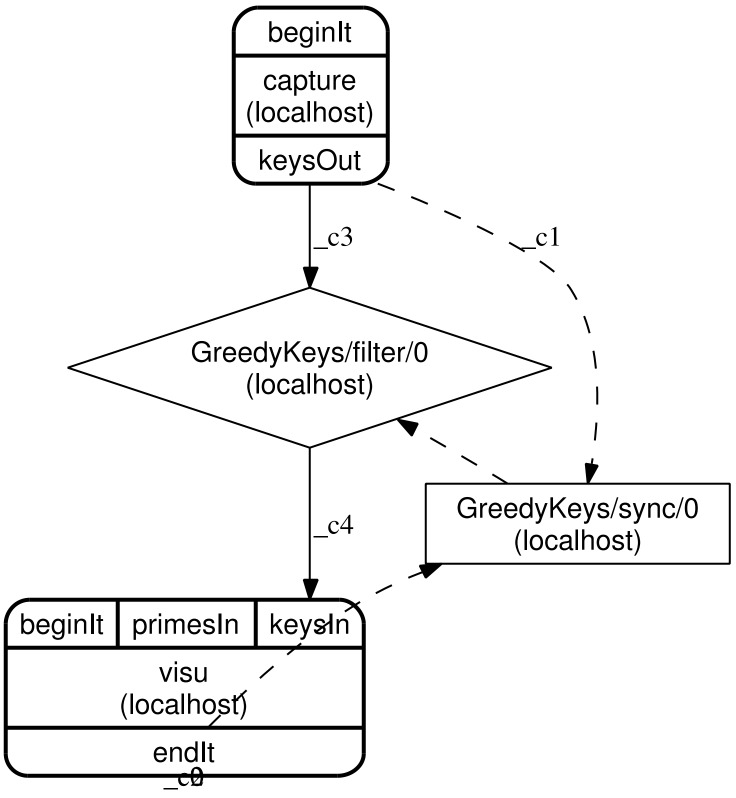


Figure 4.12: A greedy connection between *capture* and *visu* modules.

4.2.5.3 Frequency Synchronizer

The iteration frequency of a module can be bounded using a a `MaxFrequencySynchronizor` synchronizer (see section 8.5). The synchronizer is simply connected to the `beginIt` input port of a module and set the synchronizer parameter the maximum authorized frequency.
In the *Primes* example, the *visu* module is constrained to a movie-like framerate of 25 Hz (see Figure 4.13) using this synchronizer.

4.2.5.4 Other Synchronization Modes

By implementing other synchronizers and filters, it is possible to express other synchronization modes. In the same application, several synchronization modes upon different connections can be present.

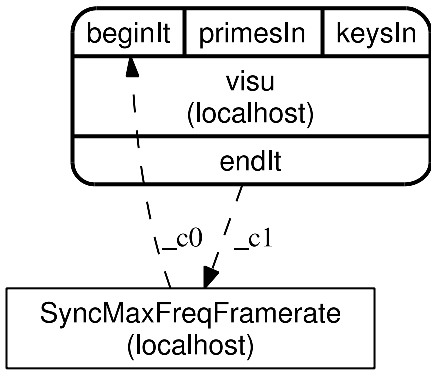


Figure 4.13: The *visu* module execution driven by a *MaxFrequency* synchronizer.

In particular a "synchronized sampling" can be implemented by having a synchronizer controlling one or several filters. In addition to the traditional greedy, the *Primes* example exposes another synchronizer-filter pair in the *3_compute_mergeIt* test. Instead of a `FilterIt` filter (see subsubsection 4.2.5.2), the greedy variation assigned for a proper desynchronisation of the *compute* module is built with a `MergeIt` filter. This component has the ability to accumulate messages iteration by iteration, which is more suitable for messages delivering computed data that should not be lost. When the *visu* module request the information, it receives an all-in-one message. In the case *visu* is faster than *compute* module and no more new messages have arrived since the past request, an empty message is delivered. Thus the *visu* module runs at full speed. Refer to the `GreedyItPrimes` network component (see Figure 4.10) for an illustration of a `FilterIt` filter paired with a greedy synchronizer. With both `FilterIt` and `MergeIt` synchronized connection for *capture* and *compute* modules, the *Primes* example as in the test *4_capture_greedy* is fully greedy. The computation module is usually the slowest of the application. The greedy synchronization between computation and visualization allows the latter to display at a better frame rate independently of the speed of the computation. The second greedy connection is for minimizing the latency of the input device. For a better interaction, it is important for the application to use the last input status when starting a new iteration. It is important to keep in mind that if one of these connections are not greedy but FIFO, implicit synchronizations appear, and the greedy connection becomes useless. This is why it is generally recommended to have only FIFO or greedy synchronization on a specified FlowVR application.

4.2.6 Host Assignment for Network Objects

Like modules, filters and synchronizers need to be mapped on a host. If this choice is not determinant to make the application work, it can have an incidence on the performance. Generally a filter or a synchronizer is placed either on the host running the source or destination module it is related to. The choice between these two solutions depends on whether the bandwidth or the latency should be favored.
To minimize the network traffic, it is better to put the filters and synchronizers on the host running the source module of the connection. It will avoid sending over the physical network messages that are not required by the destination module. To favor latency, it is better to put the filters and synchronizers on the host running the destination module of the connection. The interactions between the destination module and the filters and synchronizers will be more reactive as messages exchange will be just a pointer exchange through the shared memory and not an effective data transfer over the physical network.

Other intermediate solutions can be appropriate to take advantage of the specific network architecture of the cluster.

Part III

User Manual

Table of Contents

Chapter 5

Flowvr-appy

We detail in this section how to develop and assemble the components required to build an application. This "application" part of FlowVR is called flowvr-appy.
flowvr-appy is called from a Python script that instanciates a set of components (Section 5.2). flowvr-appy generates commands that `flowvr` uses to start up the application (Section 5.6).

5.1 Step-by-step tutorial

A step-by-step tutorial on flowvr-appy is also available. You can read it alongside this section.

5.2 Hierarchical Components

Because FlowVR is designed for large applications, the application network, also called the dataflow graph, can be complex. We provide the user tools to face this complexity and avoid him the burden of explicitly describing such a graph. We rely on the composite design pattern to support hierarchies of components. It enables to encapsulate in one component a complex pattern recursively built from simpler ones.
A component defines input and output ports. We distinguish two kinds of components:

Primitive components. A primitive component is a base component that does not contain an other component. Primitive components are modules, filters and synchronizers. They have an `addPort` method used to declare the ports of the primitive.

Composite components. A composite component contains other components (composite or primitive). Composites have a `getPort` method that can be used to expose the ports of the enclosed primitives. They do not have an `addPort`, though: ports can only be created on primitives.

5.3 Component objects

Component objects are in the following Python class hierachy:

- `Component`. A Component has a list of ports. Ports can be accessed with `getPort`.
  - `Primitive`. A basic component that procudes and/or consumes messages. It has a name, a host, and a run object that contains information on how to start it up.
    - `Module`. A primitive run as an executable.

- `FlowvrdPrimitive`. A primitive run by the FlowVR daemon.
- `Filter`
- `Synchronizer`
- `Composite`. A component that conceptually groups a set of components. It “publishes” some or all of the ports of the primitives it contains.

5.4 Component Programming

The goal of this section is to give the basics and more important concepts about component programming. They are implemented in flowvrapp.py.

5.4.1 Component Naming Convention

Primitives have a unique name (or id).
It is conventional to see composites like directories in a file hierachy. Therefore, primitives in a composite are of the form `compositeName/primitiveName` (the slash indicates the structure). Composites can be nested also.

5.4.2 Primitive constructor

To be instanciated, a primitive must have the following information:

- its name (or id)
- a way to start it up. For modules, this is Run instance. For filters, it is a string that defines the class that flowvrd must instanciate.
- a set of ports. These are added to the primitive with the `addPort()` method.
- filters and synchronizers are in addition passed a set of parameters, that are used by the filter code. They are set directly by accessing a dictionary: `self.parameters["nb"] = 18`.

5.4.3 Composite constructor

flowvr-appy ignores composites, since they do not correspond to ports or links that are actually instanciated by FlowVR.
For readability, composite constructors should take a `prefix` argument, which is prefixed to the names of all components contained in the composite.
The constructor of the composite instanciates all the components it contains, unless there is missing information at the call of the constructor. For example, when the standard composite `GreedyMultiple` is instanciated, it does not know how many inputs it will have to synchronize.
Ports are added to the composite by directly assigning to the ports dictionary:

```
class MyComposite(Composite):
  def __init__(self, prefix):
    Composite.__init__(self)
    child_module = MyModule(prefix + "/child")
    self.ports["out"] = child_module.getPort("out")
```

5.4.4 addPort

The `addPort` method of primitives has up to 4 arguments:

- the name of the port. The port name of a primitive must be unique. This is the only mandatory argument.
- `direction = "in"`. By default the ports are output ports.
- `messagetype = "stamps"`. By default messages are full.
- `blockstate = "nonblocking"`. By default the port is blocking.

The order of the optional arguments is unspecified, so the parameter name should be specified, as in:

```
class MyModule(Module):
 def __init__(self, name):
   Module.__init__(name)
   self.addPort("theoutport", messagetype = "stamps", direction = "out")
```

5.4.5 link

A link connects two ports from different components. It is used like

```
outputport.link(inputport)
```

5.4.6 `FlowvrApp` object

The `FlowvrApp` object `app` implicitly records all primitive instances. The XML files required by `flowvr` are generated with

```
app.generate_xml("appname")
```

where appname is the prefix of the XML files.

5.5 Application Compilation

The XML files are generated by running the script containing the `app.generate_xml()` call.

5.6 Application Processing: `flowvr`

The application is processed using the `flowvr` command. Its main options are:

- `-l`: request to store all outputs of the application when running. These logs are stored in files prefixed with `log-`.
- `-v`: repeat this option up to 4 times to increase the verbosity level of `flowvr`. Very useful for debugging.
- `-h`: provide the full list of available options.

For instance the primes application is started on the local host using (need a FlowVR deamon running (see section 9.1))

```
python primes.py 2 && flowvr primes
```

`flowvr` takes several input files (name matching the application name):

- `.net.xml`: file containing the primitive components mapped on the targets hosts and connected using connection representing the dataflow between components. This file is used to produce images of the graph (see chapter 13). It can be sometimes useful to open this file when debuging an application.
- `.cmd.xml`: list of low level commands sent to the FlowVR deamons to set-up the application network
- `.run.xml`: list of the command that will be executed to launch the metamodules. Can be interesting to inspect in case of issues when starting the modules. Be aware that some environement variables that are set when executing these commands are not visible in this file. It is not possible to copy on of these command and execute it in a shell unless you provide by hand the correct environement variables.

5.7 Launching commands

The launching command of a primitive contains the information necessary to `flowvr` to start the primitive up.

5.7.1 Modules

In the simplest case, the command line of the module is specified as the `cmdline` parameter in the Module constructor in flowvr-appy.
The application launching can be customized via the run object passed in to the `Module` constructor. It can be:

- `FlowvrRunSSH`: this is simplest case. There is one `FlowvrRunSSH` per module. It is built implicitly if `cmdline` is specified.
- a single `FlowvrRunSSH` can be used for several modules. In this case, the launching command is expected to start all modules (eg. by starting several threads).
- `FlowvrRunSSHMultiple` is used for modules designed to be run in several instances. They are each assigned a *rank*. The

you can request the module to be bound to one/several cores using the `cores` parameter. This parameter follow the syntax of the `corelist` option of `flowvr-run-ssh` (20.8.2).

5.7.2 MPIModules

You can use a different object to run MPI modules more easily, yet you still have to pick the one corresponding to your MPI implementation. They all use `mpirun` instead of `flowvr-run-ssh` to start the modules.

- `FlowvrRunMVAPICH`: uses `mpirun` instead of `flowvr-run-ssh` to start the modules.
- `FlowvrRunMPICH`: uses `mpirun` instead of `flowvr-run-ssh` to start the modules.
- `FlowvrRunOpenMPI`: uses `mpirun` instead of `flowvr-run-ssh` to start the modules.

They all have a "core" and a "bindobject" parameter to ...

5.7.3 Filters and synchronizers

In this case, the run object is a string specifying what class should be instanciated by `flowvrd`.

5.7.4 Core execution preference

(Linux) You can pin a module to run exclusively on one of your CPU's cores, allowing you to shape how the work is shared. FlowVR example "corepref" has multiple "compute" nodes running on a single core.

5.8 Standard filters and modules

There are a few standard modules of general interest that come with FlowVR:

- `FWrite` writes all messages it gets on its input port to a file.
- `FRead` reads a file dumpled by fwrite, and replays the messages.
- `SpyModule` connects to a port, and reports some stats about the messages it receives, on an xterm window that it opens. It requires the Python module interface to be compiled.
- `DefaultLogger` is dedicated to traces : it collects traces from the daemon. Create a raw message that can be writen in a file with a FWrite.

Chapter 6

Modules

In this chapter we present how to program a module, the base component of any FlowVR application. A Module is based on 2 pieces of code:

- The module task, i.e. the code that will be executed by the application on the target machine.
- The module application (or app) that defines the interface of the module (module id and port names) and the command line used to start it.

A **Module** is a **single thread of execution** running a potentially infinite loop. At each iteration a module gets data from input ports and sends results on output ports. The module API has 5 main instructions: `init()`, `wait()`, `get()`, `put()`, `close()`. The module API is not thread-safe, i.e. if several execution threads share the same module, the programmer must ensure a proper thread synchronization. Also note that several modules can be interlaced into a single thread of execution.
A **Launching command** is associated to to a set of modules. In the simplest case, a single command launches a single module. A parallel or multi-threaded code (a MPI parallel code or C++ code using POSIX threads for instance) can run several modules on one or more machines. Such codes are generally launched from a single command (`mpirun` for instance). In this case we usually associate them to the same metamodule.
Currenty, modules can be implemented either in C++, Python or C. You merely use bindings to the underlying C++ API.

6.1 Launching Commands

See the Flowvrapp documentation (chapter III) for information on how to describe the application's structure.
The application is launched by `flowvr`, which starts the command on the specified host, in the directory where `flowvr` was called. Therefore, the command line can be an executable in the `PATH`, an absolute file name, or a relative file name. Note that when the module is started up via ssh, the `PATH` and `LD_LIBRARY_PATH` must be set in the shell's startup scripts (`.bashrc` or similar).
The script `flowvr-run-ssh` is in charge of starting the modules. It sets several environment variables that are used by the API to contact the local `flowvrd` daemon:

- `FLOWVR_MODNAME`: the modules's name inside the application graph.
- `FLOWVR_RANK` and `FLOWVR_NBPROC`: rank is used to identify multiple instances of the process running in parallel.
- `FLOWVR_PARENT`: the PID of the process to attach to. (i.e. the daemon)

One goal when designing the module API was to be as little intrusive as possible to easily turn any piece of code into a FlowVR module.
The main way to transmit data to a module is through the argument of its command line or through environment variables.

6.2 Module Programming

This section describes the programming of a module in C++, using the native FlowVR module API. You can program a module in any language for which there exists a binding, such as Python (see chapter 10).
For C++ modules, the related files are:

- include/flowvr/module.h : The header file to include within each module code.
- include/flowvr/moduleapi.h : The main module API header file.
- include/flowvr/parallel.h: The parallel interface API.
- share/flowvr/examples/tictac/src/put.cpp : The `get` module code from *Tictac*.
- share/flowvr/examples/primes/src/compute.cpp : The `compute` module code from *Primes*.
- share/flowvr/examples/fluid/modules/src/fluid.cpp). An example of MPI module.

6.2.1 Interface

6.2.1.1 `initModule`

A module must first be registered and initialized to connect to the flowvr daemon.

```
Moduleapi* initModule(std::vector<Port*>& ports, const std::string &
    instancename = std::string(""), const std::string &modulename = std::
    string(""))
```

This method takes as argument:

- `ports`: a vector of user defined input and output ports (Section 6.2.2).
- `instancename` and `modulename`: these optional arguments are usually omitted. If set they overwrite the module name automatically set by FlowVR using `modulename/instancename` as new name.

6.2.1.2 `wait`

```
int wait()
```

A FlowVR module should be a loop. The `wait` is a blocking function call that delimits the beginning of the next iteration. Before to proceed to the next iteration, a module waits until a new message is available on each of its connected input ports (input ports not connected to anything are automatically disabled), except for a special type of non blocking input ports called event ports (see 6.2.3.4). The value returned is the current status of the module. Zero is returned if an error occurred or if the

application received the order to stop. In this case the programmer should ensure the module execution ends properly. The stamp list specification (see subsection 7.4.2) for each input port is received at the first call to the `wait`.
A classical module loop looks like this:

```
 while (pFlowVRModule->wait())
  {
    // work to be done during an iteration
  }
// wait returns zero: end the module execution now
  pFlowVRModule->close();
// more things to do specific to this module to properly stop it.
```

6.2.1.3 `get`

```
int get(InputPort* port, Message& message)
```

Get the current message on a given input port (defined by `port`). The new message is stored in the `message` variable. This method should not be called before the first call to `wait`. This is a non-blocking call, it simply provides the reference to the message registered during the last `wait`.
The message content is available as long as there is an active reference on it. The returned `message` is such a reference. If `message` is reused later to receive an other message this reference is lost.
During an iteration (i.e. between two `wait()`), dupplicate calls to `get()` on any given port will return invalid references.
After a call to this function, the user can test whether the data of an incoming message is *unique* or not. If it is, then it is possible to safely modify this data after casting away the constness of the buffer:

```
BufferWrite data;
// get message
module->get( port, message );
// check if only you can access the data
if ( message.data.unique( Buffer::ALLSEGMENTS ) ) {
   // perform a shallow copy
   data.Buffer::operator = ( message.data ); // const_cast
} else {
   // perform a deep copy
   const size_t size = message.data.getSize( Buffer::ALLSEGMENTS );
   data = module->alloc( size );
   message.data.copyTo( data.writeaccess() );
}
// in-place processing of data
// ...
```

6.2.1.4 `put`

```
int put(OutputPort* port, MessageWrite& message)
```

Send a new message to an output port. Only one message can be sent on a given port at each iteration (i.e. between two calls to `wait`). This method should not be called before the first call to `wait`. The message should not be modified after calling this method. It can be read, but it is strongly adviced to erase the reference to this message as soon as possible (destroy the `message` or call the `clear()` method). As long as the message is referenced it is kept in memory.

```
int put(OutputPort* port, const StampList * )
```

This other prototype allows stamplist forwarding. It can only be used at the very first iteration, before putting the first message (see 7.4.4).

6.2.1.5 `getStatus`

```
int getStatus()
```

This method returns the status of the module, zero if an error occurred or if the module needs to be stopped. All other methods also return the status after completion.

6.2.1.6 `close`

```
int ModuleAPI::close()
```

This method should be called to properly exit the module.

6.2.1.7 `alloc`

```
BufferWrite ModuleAPI::alloc(int size)
```

Allocate a new buffer (see subsection 7.2.1) of a given size to store the content of a message before to `put` it. This call returns a read/write BufferWrite (see section 7.2) object stored in the shared memory segment controlled by FlowVR.
For performance-critical modules, it is advised to directly work within this type of FlowVR buffer. They are allocated in the shared memory segment. If the module allocates its own memory space for messages (not in the shared memory segment), an extra copy to the FlowVR handled buffer will be required, incurring some performance penalties.
Note that, as all other `ModuleAPI` methods, this method should not be called before `init()`. This is an important requirement as sometimes allocations can be used to construct some data structures. They might be incorrectly initialized if this is done before the call to the `init` method.
Dynamic allocations can be costly and memory fragmentation issues can appear if too many allocations are requested. When a module often allocates buffers of the same size, it is more efficient to take advantage of the `BufferPool` mechanism (see subsection 7.2.3) to reduce the required dynamic allocations.

6.2.1.8 `abort`

```
int ModuleAPI::abort()
```

This method may be called by any module. This method sends a message to the launcher and triggers the `stop` command as if it was typed in its console. Redundant calls are ignored by the launcher.

6.2.2 Predefined Input and Output Ports

Each module has two predefined ports:

- `endIt` output port: each time a module calls the `wait` method it automatically sends a message on the `endIt` output port. This port is a signal port. It enables a module to signal that it ended the current iteration and that it is ready to perform a new one.
- `beginIt` input port: when enabled (i.e. connected) , this port forces the `wait` method (see 6.2.1.1) to block until a new entry on this port is available. This port is an activation port, allowing to synchronize the module activity on an external signal (to control its frequency for instance).

These ports do not have to be declared when programming a module.
The *Primes* example heavily uses `endIt` and `beginIt` ports for synchronization purpose (see subsection 4.2.5).

6.2.3 User Defined Ports

The information related to each port is:

- its direction (input or output)
- whether it's a full or stamps port
- its blocking state (blocking or non-blocking)
- for output ports, the type of the stamps it plans to send out.

6.2.3.1 Port Vector

Ports need to be declared before the module initialisation. All user defined ports need to be stored in a vector passed as argument of initModule (see subsection 6.2.1).

6.2.3.2 Output Port

```
ModuleAPI::OutputPort(const std::string& myname, StampList* mystamps=NULL,
    bool bOwn = false);
```

Output port constructor:

- `myname`: port name
- `mystamps`: a stamp list specification (see subsection 7.4.2) if the messages sent on that port will carry user defined stamps. This argument is optional and the stamp list specification can be specified latter.
- `bOwn`: only used when passing a stamp list specification in `mystamps`. If set to `true` the stamplist is deleted upon destruction of the port, not otherwise.

6.2.3.3 Input Port

```
ModuleAPI::InputPort(const std::string& myname,
                          StampList* mystamps=NULL,
                         bool bOwnStampList = false,
                     bool bIsNonBlockingPort = false);
```

Input port constructor:

- `myname`: port name
- `mystamps`: The stamp list specification usually does not need to be specified here (leave the default NULL value) as it will be set by the system at initialisation according to the stamp list specification of the incoming messages.
- `bOwnStampList`: only used when passing a stamp list specification in `mystamps`. If set to `true` the stamplist is deleted upon destruction of the port, not otherwise.
- a stamp list specification (see subsection 7.4.2) if the messages sent on that port will carry user defined stamps. This argument is optional and the stamp list specification can be specified latter.

6.2.3.4 Example

In this example we create an extra input and output port. We also add stamps to the output port:

```
// declare an input and output port
flowvr::InputPort *in = new flowvr::InputPort("in");
flowvr::OutputPort *out = new flowvr::InputPort("out");

// add an int stamp
flowvr::StampInfo *siMycounter = new flowvr::StampInfo("mycounter", flowvr::
    TypeInt::create());
out->stamps->add(pStampComputeTime);

// prepare the ports vector
std::vector <flowvr::Port*> ports;
ports.push_back(in);
ports.push_back(out);
```

6.2.4 Event Ports

It is possible to have non blocking input ports called **event ports**. In opposite to classical input ports, the `wait` does not wait until a new message is available on event ports. Thus, if no message was available when `wait` was called, `get` returns an invalid message (make sure to test it).
Event ports can be convenient for grabbing messages that are sent episodically, i.e. at a rate significantly lower than the expected module iteration rate. Typically event ports can be used to receive control messages from a GUI.
If a module has only event ports and that the `beginIt` port is not connected, it will never block at the `wait` call. This may lead to a free running module that consumes much more CPU resources than expected.
Messages can be accumulated on input ports, for instance if many messages are sent between two iterations of the event port holder, possibility leading to buffer overflows. Inserting a filter to erase or merge accumulated messages can be required to avoid such pitfall.

The same behavior can be obtain by putting an adequate filter before a classical input port. This approach has the advantage of changing the module behavior for a given application without having to change the module code. Generally we advice not to abuse of event ports that tend to affect the genericity of modules.
Being an event port or not need to be set before calling the `initModule` method (see subsection 6.2.1) either:

- At port creation (see subsubsection 6.2.3.3)
- Setting to true the port non blocking flag (call after `initModule` has no effect):

```
InputPort::setNonBlockingFlag(bool bBlock)
```

A VOIR *Et l'initialisation des modules paralléle ?* FIN

6.2.5 Probing Ports State

- `bool Port::isConnected()`: return `true` if the port is connected. Call valid only after the first `wait`.
- `bool Port::isInput()` and `bool isOuput()`: return `true` if match the expected port type.
- `bool InputPort::isNonBlockingPort()`: return `true` if event port (see 6.2.3.4).

Chapter 7

Messages, Stamps and Data Buffers

Related files:

- *include/flowvr/message.h : The message header file.*
- *include/flowvr/buffer.h : The buffer API header file.*
- *share/flowvr/examples/tictac/src/put.cpp : The* `get` *module code from Tictac.*
- *share/flowvr/examples/primes/src/compute.cpp : The* `compute` *module code from Primes.*

Modules and network components send and receive messages. This section describes how to handle these messages. This is relevant for module programming (see section 11.5) as well as for filter and synchronizer programming (see chapter 8).
This section is mainly devoted to the C++ version. The Python API is described in section 10.1.6.

7.1 Messages

A message is composed of two parts, the stamp list and the data buffer (payload). The message API is defined in include/flowvr/message.h.
It defines 3 classes:

- `Message()`: Read-only stamp list and data buffer. Message returned by a `get` (see 6.2.1.2).
- `MessageWrite()`: Read/write stamp list and data buffer. Message that can be sent by a `put` (see 6.2.1.3).
- `MessagePut()`: Read/write stamp list but read-only data. Message that can be sent by a `put` (see 6.2.1.3). Useful if you need to resend a data buffer extracted from a received message (in this case the data buffer is read only).

For these three classes the user can have direct access to:

- `Stamps stamps`: the stamp list (`stamps`)
- `Buffer data`: the data buffer

These classes also defines some utility methods including:

- message comparison operators (== and `!=`) ,

- `void clear()`: free the message content (stamps and data)
- `Type getType()` return the message type (STAMP or FULL)
- `bool valid()`: proble the message validity

7.1.1 Message: FULL, STAMP, Null, Empty, Valid ?

A VOIR *Ingo please review this part. Not sure what to do with the Null messages (*`Message::Null`*). Text was (not sure correct): Message validity should also be tested calling the* `bool Message::valid()` *method (probe the message structure).* FIN
Some methods, like `frontMsg()` on input message queues, can return invalid messages. This is in particular the case when event ports (see 6.2.3.4) have no message available:

```
if( msg.valid() == false ) { ... // no message available on this event port }
```

Also as the `wait` is a blocking instruction for modules, it can be necessary to generate empty messages if no new content is available simply to unlock the wait. In this case the module should proble the message content and behave accordingly.
A message is `FULL` if it has a non null data buffer, even if the buffer data is of size 0. Otherwise this is a `STAMP` Message. Use the method `bool Buffer::empty()` to test if a data buffer is empty, i.e. of size 0.
You can find a code line like the following one in some filters:

```
    m.data = alloc(0);
```

It produces messages where only the stamp list is meaningful. But, as a zero size buffer is allocated, it is considered a `FULL` message. This make the filter more generic as this output port is `FULL`. A `FULL` port can always emit `STAMP` messages if needed (just need to be connected to a `ConnectionStamps`
A VOIR *add link to connection stamp section when written* FIN
The PreSignal filter generates messages with `it=-1`. These special messages are used to unlock cycles and are sent when the filter starts. Other components can thus receive such messages. Test for such messages and consider them as activation messages triggering a first action (include/flowvr/plugins/flowvr.plugins.PreSignal.cpp). Not doing any thing when receiving such message may lead to a deadlock as the cycle this signal was supposed to unlock stay in a deadlock state.

7.2 Data Buffers

A VOIR *voir les liesn vers doxygen flowvr::Buffer flowvr::Buffer flowvr::BufferWrite All the shared memory management is deferred to an abstract flowvr::BufferImp class.*
Buffer Buffer to the constructor. When a Buffer or BufferWrite object is destroyed the reference counter is decremented, and the associated buffer is automatically freed if no other reference exists.
FIN

7.2.1 Buffer

The data of a message are necessarily stored in a unique continuous memory space A VOIR *This is not true anymore* FIN , a buffer.
FlowVR defines (include/flowvr/buffer.h):

- `BufferWrite`: a read/write buffer used to prepare the data before to send it.

- `Buffer`: a read-only buffer the user has access to at message reception.

These buffers are all allocated in the shared memory segment.
To allocate a `BufferWrite`:

- for a module use the `alloc` (see 6.2.1.6) method from the module API.
- for a filter use `BufferWrite buf= alloc(size).`

Allocating a size zero buffer is possible and can be useful if your protocol needs an empty message.
$\boxed{\text{A VOIR}}$ *somewhere document the -1 stamp message in corespondance with the -1 buffer* $\boxed{\text{FIN}}$
To write into the buffer:

- `BufferWrite::writeAccess()`: return a pointer at the beginning of the buffer.
- `BufferWrite::getWrite<Type>(offset)` : return a pointer of the template type `Type` to the specified offset in the buffer (measured in bytes).

To read the buffer content, typically after a get (see 6.2.1.2):

- `Buffer::readAccess()`: return a read-only pointer at the beginning of the buffer.
- `Buffer::getRead<Type>(offset)`: return a read-only pointer of type `Type` to the specified offset in the buffer.

It is possible to make a new buffer by copying (fully or partially) the references of an existing buffer (data are not duplicated):

- `BufferWrite` or `Buffer` copy constructors for a full or partial (specifying an offset and size as parameters) copy.
- `operator=` for a full copyof the reference. Again, data won't be duplicated.

Use with care for read/write buffers as two copies have write access to the same memory space. This can be useful for instance if some data need to be extracted from an existing buffer to make a new message sent on a different port.
To resize an existing buffer:

- `resize`: Change the buffer size. If necessary allocate a new buffer and copy existing data into this new buffer. If the requested size is larger, this methods fails.
- `expand`: Change the buffer size. If a non null buffer already exists and the requested size is larger, this methods fails (safeguard for performance purpose).

FlowVR manages reference counters on buffers. When such counter reaches zero the memory associated to the buffer is deallocated. The same buffer may be accessed (read only) by several modules (same message sent to various modules running on the same host). Giving a module direct control to free memory would be unsafe (one module deallocating a buffer that an other still need to access). However this is important the user signals FlowVR when a buffer is not in used anymore by destroying the buffer or calling the `clear()` method (can also be done at the message level (see section 7.1)).

7.2.2 Example

The following code allocates and fills a buffer to send a set of identifiers indicating currently pressed keyboard keys:

```
flowvr::MessageWrite msgWrite; // MessageWrite object
unsigned char *pMsgData = 0; // Pointer to allocated memory
unsigned int keyPressedCount = ...; // Hold the number of keys currently
    pressed

// Request a FlowVR buffer large enough to store pressed keys identifiers :
msgWrite.data = pFlowVRModule->alloc(keyPressedCount*sizeof(unsigned char))
    ;

// Get writing reference :
pMsgData = (unsigned char *)msgWrite.data.getWrite<unsigned char>();

// Fill buffer with the identifiers of the keys which are pressed :
for (int i=0; i<MAX_KEYS; i++)
  if (tKeysState[i])
  {
    *pMsgData = (unsigned char)i;
    pMsgData++;
  }
  // Put message on a given port (stamp list automatically filled)
  pFlowVRModule->put(pPortPrimesOut, msgWrite);
```

And below is the code to read keys state information from received message:

```
flowvr::Message msgRead;      // Message object
unsigned int keysPressedReceviedCount; // Number of keys identifiers to
    read
unsigned char* pKeysPressed = 0; // Application buffer to store keys
    pressed

// Get the message from the input port
pFlowVRModule->get(pPortPrimesIn, msgRead);

// Count the number of pressed keys from message size (returned in bytes) :
keysPressedReceviedCount = msgRead.data.getSize() / sizeof(unsigned char);

// Copy message buffer to application buffer (alternative: is to work
    directly with the data of the message)
if (keysPressedReceviedCount > 0)
{
  pKeysPressed = new unsigned char [keysPressedReceviedCount];
  memcpy((void*)pKeysPressed, msgRead.data.readAccess(),
      keysPressedReceviedCount);
}
// Message content not useful anymore: clear the message
msgRead.clear();
```

Remark The actual codes of *capture* (share/flowvr/examples/primes/src/capture.cpp) and *visu* (share/flowvr/examples/primes/src/visu.cpp)) use a higher level layer for handling key messages: Chunk Events (see section 7.3).
A VOIR *a revoir* FIN

7.2.3 BufferPool: Reusing Old Buffers for Better Performance

Reference file

- *include/flowvr/bufferpool.h: The buffer pool header file.*
- *share/flowvr/examples/primes/src/compute.cpp : The* `compute` *module code from Primes uses a buffer pool.*

A VOIR *The flowvr::BufferPool* FIN

7.2.3.1 Constant Size Buffers

As the shared memory is concurrently accessed by several processes, allocation operations can become quite expensive (lock contentions, memory fragmentation). To remove this potential bottleneck, a simple class lets you reuse old buffers not used anymore. This only works for the case of repetitive allocations of buffers with the same size, which is quite common.
There is no major difference compared to a regular buffer handling except that you have to:

- Create a `BufferPool` object after the component initialisation and before entering the main loop. The `BufferPool` stores a cache of old buffers. This cache size has a default value that you can change if required (constructor argument).
- Call the `alloc` method from the `BufferPoll` class rather than the regular allocation methods like `alloc` (see 6.2.1.6). Allocate a new buffer if it fails to reuse an old one (feature that can be turned off)

This mechanism of buffer pools enables reducing or even removing dynamic allocations with a bounded memory cost (buffer size $\times$ cache size). Performance improvement can be significant.

7.2.3.2 Bounded Size Buffers

If the size of buffers is not constant but bounded, a `BufferPool` can still be used with a little extra effort to resize the buffer provided. When allocating a buffer, use the `BufferPool` allocation method `alloc` and request the maximum size. If you only need a part of this buffer, resize it using a `BufferWrite` copy constructor or the `expand` method (see subsection 7.2.1). If this buffer needs to be sent to a different host, only the relevant sub-part will be sent.
The code below shows how to derive a buffer of size 256 from an original buffer of greater size :

```
BufferWrite originalBuffer; // Buffer retrieved from the buffer Pool whose
    size will be >= 256

... // Deal with originalBuffer

BufferWrite* pSubBuffer = 0; // pointer to the sub-buffer

// Get a reference to a subset of the original buffer (the 256 bytes at
    offset 0) :
pSubBuffer = new BufferWrite(originalBuffer, 0, 256);
```

7.2.3.3 Example

The *compute* component sends computed prime numbers by packets of constant size. Thus it can use a buffer pool (share/flowvr/examples/primes/src/compute.cpp):

```
flowvr::BufferPool* pOutPool = 0; // BufferPool object
unsigned int tempPrimeNumbersMaxCount = ...; // Constant count of prime
    numbers
unsigned int *tTempPrimeNumbers = 0; // Calculated prime numbers by
    iteration

// Create a pool of buffers :
pOutPool = new flowvr::BufferPool();
...
flowvr::MessageWrite msgWrite; // MessageWrite object

// Request for a new buffer from the pool to send new computed prime
    numbers.
msgWrite.data = pOutPool->alloc(pFlowVRModule, tempPrimeNumbersMaxCount*
    sizeof(unsigned int));

// Fill message data :
memcpy((void*)msgWrite.data.writeAccess(), (void*)tTempPrimeNumbers,
    tempPrimeNumbersMaxCount*sizeof(unsigned int));
```

7.3 Chunks: Structuring Message Content

Related files:

- *include/ftl/chunk.h : The base chunk definition.*
- *include/ftl/chunkwriter.h : Building new messages from chunks.*
- *include/ftl/chunkreader.h : Reading chunks from a message.*

FlowVR provides a utility library to store and retrieve data in message buffers as structured chunks. This library helps to pack and unpack data into a message buffer. It is useful in particular when a message is the concatenation of several sub-messages.
Beside the base chunk definition, FlowVR provides chunk serializations for mouse and keyboard events. We present here how to use these specializations.

7.3.1 Chunk for Keyboard and Mouse Events

Related files:

- *include/ftl/chunkevents.h: The chunk events header file.*
- *share/flowvr/examples/primes/src/capture.cpp and share/flowvr/examples/primes/src/visu.cpp: Examples of module using chunk events*

The flowvr-ftl/include/ftl/chunkevents.h header file defines a set of data structures to encode keyboard and mouse events. To each event type is associated a class with various public fields that define a chunk event.

Encoding an event is done when calling the appropriate `addEvent` that first build the appropriate chunk event and next add it to a `ChunkEventWriter` object, the chunk event version of a `BufferWrite`.
Once all chunks have been stored in the `ChunkEvenWriter`, calling its `put` method on a given port will emit the expected message. There is no need to explicitly take care of buffer allocation and data packing.

```
// ChunkEventWriter
ChunkEventWriter *keysMsgs = new ftl::ChunkEventWriter();

// Add first chunk of EventButton type
keysMsgs->addEventButton(FLOWVR_KEY_F1,true);

// Add second chunk of EventKeyboard type
keysMsgs->addEventKeyboard('a',0,false,0 );

// Put message (stamps are automatically set)
keysMsgs->put(pPortOut);
```

At reception, the procedure is similar to message reading, except that a chunk iterator is defined to iterate on the message to extract each chunk. As there is different types of chunks, it is necessary to first probe the chunk `type` to cast the retrieved chunk to the correct type and read its content.

```
flowvr::Message msgRead;

// Get the message
pFlowVRModule->get(pPortIn, msgRead);

// Use iterator to extract the chunks
for (ChunkIterator it = chunkBegin(msgRead); it != chunkEnd(msgRead) ; it++ )
     {

      // The chunk
   const Chunk* c = (const Chunk*) it;

       // Probe its type
   switch (c->type & 0x0F) {

      case ChunkEvent::BUTTON:
                  // Cast and read the chunk fields
                  ChunkEventButton * cc = (ChunkEventButton *)c;
         key = cc->key;
         val = cc->key;
      case ChunkEvent::KEYBOARD:

                  // Cast and read the chunk fields
                   ChunkEventKeyboard * cc = (ChunkEventKeyboard *)c;
         key = cc->key;
         val = cc->val;
                  special= cc-> special;
                  modifier= cc->modifier;
      break;
   }
}
```

The events supported:

- **ChunkEventButton**: It supports one id for each key and one digital value (0/1) pressed or released
- **ChunkEventSlider**: events for sliders, it supports one id for each key and one analog value (0 to 1)
- **ChunkEventKeyboard**: events for keyboard. It supports one id for each key, one key values (0/1) pressed or released, one tag specifying if it is a special button and a modifier that tell if CTRL, SHIFT, etc. where pressed together with the key
- **ChunkEventMouse**: events for mouse. It supports the x and y coordinates and digital values for each mouse key (0/1) pressed or released
- **ChunkEventString**: events for strings. It supports any string and can be used for commands like speech recognition
- **ChunkEventPosition**: events for positioning devices. It supports a 4 X 4 matrix for transformation.

The `addEvent` methods also support a last optional parameter to encode a device id. It is useful to distinguish between devices when two or more a re used.

7.4 Stamps

Related files:

- *include/flowvr/stamp.h: the stamp API*
- *share/flowvr/examples/primes/src/compute.cpp share/flowvr/examples/primes/src/visu.cpp: example of user defined stamp.*
- *flowvrd/src/plugins/filters/flowvr.plugins.FilterIt.cpp: Example of filter merging messages based on their* `it` *stamps.*

Each `FULL` FlowVR message is composed of a stamp list and the data buffer (payload). A stamp is a small piece of information related to the message. Some stamps are automatically set by FlowVR. The user can also define new stamps if required. To some extent, stamps enable to perform computations on messages without having to read the message data buffer. The stamp list can be sent without the message payload if the destination does not need it. It improves performance by avoiding useless data transfers. Such a message is called a `STAMP` message.

7.4.1 Predefined Stamps

$\boxed{\text{A VOIR}}$ *Clarify the stmaps with -1 values, 0 values, etc.* $\boxed{\text{FIN}}$
The following system stamps are part of each message stamp list:

- `source` (`string`): This stamp identifies where the message was created (component and output port).
- `it` (`int`): iteration number of the component when the message was created. It increases monotically for the sequence of messages emitted on a given output port (if a message is not sent at each iteration `num` stamps are not consecutive for instance). A module iteration starts each time a `wait` is executed. Because filters do not have such explicit iteration counter, setting the `it` value is left to the developper.

- `num` (`int`): message number at a given message source (component and output port). `num` is incremented by one each time a new message is sent.

These stamps carry the base identification data of each message. For example, `flowvr` commonly relies on the `source` stamp to route messages to their destination.
When a module calls the `put()` (see 6.2.1.3) a stamp list with the predefined stamps is automatically attached to the message.
For filters, the call to `put()` on an `outputmessagequeue` sets the `source` and `num` stamps automatically. The `it` stamp must be explicitly set by the user. Notice that the `put` has an optional parameter to set `num` to a specific value instead of the system assigned one if required. $\boxed{\text{A VOIR}}$ *link to put of message queue once documented* $\boxed{\text{FIN}}$
Reading a system stamp on a message is simple:

```
// Get a new message from input port
pFlowVRModule->get(pPortPrimesIn, msgRead);

// Read the it stamp value and store it in myit
msgRead.stamps.read(pPortPrimesIn->stamps->it, myit);
```

7.4.2 Stamp List Specification

A list of stamps, called the **stamp list specification** is attached to each input and ouput port of modules and filters. This list defines the list of stamps that is attached to each message send or received on that port.
This stamp list is first defined for each output port. No action is required from the programmer for predefined stamps. If extra stamps need to be attached to messages, they need to be added to the corresponding output port stamp list specification.
At initialization, a very first message is sent from each ouput port with this stamp list specificition (special message with a stamp `num=-1`). Input ports set their stamp list specification from the one they receive through this message. This message is received at the first call to the `wait` (see 6.2.1.1) for modules or at the `newStampListSpecification` calls for filters (see subsection 8.2.3).
Ports of modules and filters have a `stamps` field that gives direct access to the stamp list specification. This is a classical stamp list.

7.4.3 User Defined Stamps

7.4.3.1 Adding a new Stamp

A stamp list specification (see subsection 7.4.2) is attached to each port. By default this list contains only the predefined stamps. To add a new stamp to an output port, it is first necessary to define a new `StampInfo` object with a name and type. A `StampInfo` is build from the stamp's name and type. This type is created by calling the `create` method of the appropriate `BaseType` subclass. Next, this new stamp specification need to be added to the port stamp list with the `add` method (both filter and module ports have this `stamps` field).
Every time a new message is built, the user has the responsability to fill the values of the extra stamps using the `write` method on the message stamp list, providing again the stamp specification as a parameter, to correctly store the stamp value:

```
 // Output port declaration :
 flowvr::OutputPort* pPortPrimesOut = new flowvr::OutputPort("primesOut");

 // New stamp specification :
 flowvr::StampInfo *pStampComputeTime = new flowvr::StampInfo("
     computationTimeIt", flowvr::TypeInt::create());

 // Add the new stamp to the output port stamp list.
 pPortPrimesOut->stamps->add(pStampComputeTime);

 flowvr::MessageWrite msgWrite;
 int lastIterationComputeTime;
 ....
 // Set the value of the new stamp
 if ( !msgWrite.stamps.write(*pStampComputeTime, lastIterationComputeTime))
      std::cout << "Error writing computationTimeIt"<<std::endl;

// Transfer the message
 pFlowVRModule->put(pPortPrimesOut, msgWrite);
```

7.4.3.2 Reading a Stamp Value

On input ports the stamp list specification (see subsection 7.4.2) is set automatically according to the stamp list contained in a special initialization message. If extra stamps have been defined on the emitter side they will be available.
Call the `read` method on the received message stamp list, providing a stamp specification as a parameter. This stamp specification can be retrieved from the port stamp list specification, using the stamp name. Of course, the container of the stamp value should match the actual stamp type (`int` in the example bellow).

```
pPortPrimesIn = new flowvr::InputPort("primesIn");

flowvr::Message msgRead;
int lastIterationComputeTime;
 ...
 // Receive the message
pFlowVRModule->get(pPortPrimesIn, msgRead);

// Get a pointer on the stamp specification (need to be called after the
    first wait for modules)
flowvr::StampInfo * pStampComputeTime = (*(pPortPrimes.stamps))[std::string(
    "computationTimeIt")];

// Read the value of stamp "computationTimeIt". Assign -1 if error.
if (! m.stamps.read(*pStampComputeTime, lastIterationComputeTime))
    lastIterationComputeTime = -1;
```

An alternative approach consists in declaring the extra stamp like for an output port:

```
 pPortPrimesIn = new flowvr::InputPort("primesIn");

 // New stamp specification
 flowvr::StampInfo * stampComputeTime= new flowvr::StampInfo("
     computationTimeIt", flowvr::TypeInt::create());
 // Add new stamp specification to the port stamp list
 pPortPrimesIn->stamps->add(stampComputeTime);

 flowvr::Message msgRead;
 int lastIterationComputeTime;
 ...
 // Receive the message
 pFlowVRModule->get(pPortPrimesIn, msgRead);

// Read the user defined stamp value. Assign -1 if error.
 if (!msgRead.stamps.read(*stampComputeTime, lastIterationComputeTime))
   lastIterationComputeTime = -1;
```

7.4.4 Stamps Forwarding

If you had several modules chained one after the other, keeping and forwarding stamps with the previous API can become cumbersome. It would require to define the stamplist the exact same way in each module, and to copy each stamp manually.

```
int put(OutputPort* port, const StampList * )
```

With this method, you can easily forward a stamplist to an output port. It can only be used at the very first iteration, before putting the first message.

```
int nit = 0;
while ( module->wait() ) {
   if ( nit == 0 ) {
      module->put( &outPort, inPort.stamps );
   }
   flowvr::Message in;
   module->get( &inPort, in );
   // stamp forwarding
   flowvr::MessageWrite out;
   out.stamps.clone( in.stamps, outPort.stamps );
   /* ... */
   put( &outPort, out );
   nit++;
}
```

7.4.4.1 Forwarding a modified stamplist

You might also want to add a new stamp to the previous stamplist. This can also be done but has one important restriction: you must not use variable length strings. That is, if one of the user-defined stamp is a string, then it must always have the same length from one iteration to the other. If so, then you can *clone* the input stamplist, add a new stamp then forward it.

```
int nit = 0;
while ( module->wait() ) {
   if ( nit == 0 ) {
      StampList * newList = inPort.stamps.clone()
      newList->add( newStamp );
      module->put( &outPort, newList );
      delete newlist;
   }
```

```
    flowvr::Message in;
    module->get( &inPort, in );
    // stamp forwarding
    flowvr::MessageWrite out;
    out.stamps.clone( in.stamps, outPort.stamps );
    /* ... */
    out.stamps->write( outPort.stamps["newStamp"], val );
    module->put( &outPort, out );
    nit++;
}
```

Once its done, at each iteration, incoming stamps can be cloned using the existing API.

Chapter 8

Filters and Synchronizers

Related files:

- *include/flowvr/plugd/messagequeue.h: input message queue.*
- *include/flowvr/plugd/outputmessagequeue.h: output message queue.*
- *include/flowvr/plugins/baseobject.h: daemon plugin base class.*
- *include/flowvr/plugins/filter.h: filter base class.*
- *include/flowvr/plugins/synchronizor.h: synchronizer base class* $\boxed{\textit{A VOIR}}$ rename synchronizer $\boxed{\textit{FIN}}$
- *include/flowvr/plugins/flowvr.plugins.PreSignal.cpp: a very simple filter that emit* `nb` *initial empty messages and next just forward incoming messages.*
- *flowvrd/src/plugins/filters/flowvr.plugins.FilterIt.cpp: filter example.*

Filters are primitive components having input and output ports (also called input and output message queues for filters). They perform tasks on input data and produce new output messages. There are two main differences with modules:

- Filters are plugins loaded and executed by the FlowVR daemon. They need to be written in C++ (not the case for modules). Being close to the daemon they have access to internal structures for a better performance and more flexibility.
- Filters have full access to the full message queues while modules only have access to the last message available. Thus filters are commonly used to merge or resample messages for instance. This extra flexibility make them a little more complex to program.

Synchronizers are a special case of filters having only `STAMP` input and output ports. In this document we usually do not distinguish between filters and synchronizers ans usually use the general term filter, unless explicitly stated.
$\boxed{\text{A VOIR}}$ *doStart useless for non threaded fitlers ? See greedy sync where it is used. fitler== 1 thread indépendant du noyau ?* $\boxed{\text{FIN}}$
$\boxed{\text{A VOIR}}$ *Dan la calsse de base des synchronizers: int advance; ///< Number of order to send before waiting for the end of an iteration int nbOrder; ///< Number of order sent virtual void doStart(plugd::Dispatcher* dispatcher);* $\boxed{\text{FIN}}$

8.1 Inheritance and Plugin Loader

Filters must inherit from the `Filter` class (`synchronizor` for synchronizers).
Filters (and synchronizers) must implement a special virtual method and create an instance of a `GenClass` templated object that the daemon uses to get an handle on the filter class to create an instance and execute it.
Mandatory method to implement in each filter class:

```
virtual Class* getClass() const
{
  return &FilterItClass;
};
```

Mandatory creation of an instance of `GenClass`:

```
flowvr::plugd::GenClass<FilterIt> FilterItClass("flowvr.plugins.FilterIt", //
    must corespond to the name of the plugin once compiled.
                           "free form text about the filter ", // filter description
                           &flowvr::plugins::FilterClass// name of the class the
                               filter inherit from concatenated with the string "Class
                               "
                            );
```

8.2 Filter Callbacks

A VOIR *synchronizer in .net.xml (and not synchronizor) ?* FIN
Programming a filter relies on filling three main callback functions (virtual methods) we detail in the following.

8.2.1 Dispatcher

The callback methods all provide a pointer to the `dispatcher`. The dispatcher is a special plugin in charge of executing some special actions for the daemon. Some filter related methods, on message queues for instance, need to provide this pointer to the dispatcher to trigger some actions.

8.2.2 `init`

`virtual flowvr::plugd::Result init(flowvr::xml::DOMElement* xmlRoot, flowvr::plugd::Dispatcher* dispatcher):`

- `xmlRoot`: parameter xml tree.
- `dispatche`: pointer to the dispatcher.

Called at filter initialization (after plugin loading). This is in this method that should be implemented any action that needs to take place at initialization before any message reception.
This method should first start by calling the parent init method. For a filter for instance:

```
  flowvr::plugd::Result result = Filter::init(xmlRoot, dispatcher);
  if (result.error()) return result;
```

A VOIR *liens vers la definition de parametres si on en fait une* FIN Each filter can define parameters that get values at starting time. When starting the application, the parameter values are forwarded to the filter by the daemon as an XML tree. This tree has a simple structure, a `<parameter>` root having as children the list of parameters in the form `<paraname>value</paramname>`. These xml trees are part of the `.net.xml` intermediate file generated for each application. Here is a tree example:

```
<parameters>
  <info>....some text about the component...</info>
  <trace>0</trace>
  <freq>1</freq>
</parameters>
```

The `<parameter>` tree always contain 2 special parameters that should be ignored at this point:

- `<info>`: a string containing some information about the filter.
- `<trace>`: set to 1 if this object is traced (see chapter 14), 0 otherwise.

Parsing the tree using the XML DOM parse provided with FlowVR (include/flowvr/xml.h and include/flowvr/tinyxml.h) enables to retrieve the parameter values. Here is a short example to retrieve a frequency parameter:

```
// read the freq parameter node
xml::DOMNodeList* lfreq = xmlRoot->getElementsByTagName("freq");

if (lfreq != NULL && lfreq->getLength()>=1)
{
    // Get the freq parameter value (string)
 std::string fr = lfreq->getTextContent();
    // Convert to float
 if (!(fr.empty())) freqHz = flowvr::ftl::convertTo<float>(fr);
    if (freqHz > 1000000.0f || freqHz < 1.0f )
    {
      return Result(flowvr::plugd::Result::ERROR,"Incorrect frequency value
          ");
    }
}
else
   return Result(flowvr::plugd::Result::ERROR,"frequency parameter not
       found");
delete lfreq;
```

Next, in the method the filter must declare its message queues, i.e. its ports. First call the `initInputs` and `initOutputs` for initializing the vector queues vectors (argument is the number of ports). Next, give a name to each port (`setName` on each queue):

```
//initialization of the input message queue (rely on a enum to identify
    each one)
initInputs(NBPORTS);
inputs[IDPORT_IN]->setName("in");
inputs[IDPORT_ORDER]->setName("order");


//initialization of the output message queue (just one port no enum)
initOutputs(1);
outputs[0]->setName("out");
```

If known it is also convenient to set the output message type, `STAMP` or `FULL` at this point:

```
outputs[0]->msgtype = Message::FULL;
```

The output message type is sometimes set in the `newStampListSpecification` method, when it is defined according to the type of message received on a given input port.
The input message type is actually defined
$\boxed{\text{A VOIR}}$ *terminer: plutot au niveau du component non ?* $\boxed{\text{FIN}}$
$\boxed{\text{A VOIR}}$ *(msg.data.valid()* $\boxed{\text{FIN}}$ $\boxed{\text{A VOIR}}$ *pointer to themessage queue api* $\boxed{\text{FIN}}$. $\boxed{\text{A VOIR}}$ *type du port vraiment important. A priroi non pour les outputs. Important pour les inputs (drive the auto connection seting. C'est une commodite pour du code de type if (outputs[0]->msgtype == Message::FULL) m.data = alloc(0);* $\boxed{\text{FIN}}$

8.2.3 `newStampListSpecification`

`newStampListSpecification(int mqid, const Message& msg, plugd::Dispatcher* dispatcher)`:

- `mqid`: the index in the input message queue vector the message comes from.
- `msg`: the received message.
- `dispatcher`: pointer to the dispatcher.

This method is called once per connected input message queue at initialization when receiving the system message containing the stamp list specification (see subsection 7.4.2) of future incoming messages on that queue (this special message has the `num` stamp set to -1).
This is in this method that should be implemented any action that needs to take place at initialization but that need information contained in the first system messages (stamp list specification for instance). For instance filters that forward messages on their output queue with the same stamp list specification than incoming messages, retrieve the stamp list specification from the received message, copy it for the output message queue and set this new specification by handling it to the dispatcher:

```
// System message received on first input port.
if (mqid==0)
  {
    // Copy the stamp list for the first output message queue.
    outputs[0]->stamps = inputs[0]->getStampList();
    // Tell the dispatcher to take into account this specification
    outputs[0]->newStampSpecification(dispatcher);
  }
```

8.2.4 `newMessageNotification`

`newMessageNotification(int mqid, int msgnum, const Message& msg, plugd::Dispatcher* dispatcher)`.

- `mqid`: the index in the input message queue vector the message comes from.
- `msgnum`: the `num` stamp of the recieved message.
- `msg`: the received message.
- `dispatcher`: pointer to the dispatcher

This method is called each time a new message is recieved on an input port. This method usually implements the main task filters are designed for.
Because one input port can receive messages while an other input port has not yet received its stamp list specification, it can be useful in this method to proble the state of a given input port calling:

```
if (!inputs[IDPORT_IN]->stampsReceived()) return; // stamp specification not
    yet received on that port.
```

A VOIR *peut etre une pointeur vers la manipulation des messages* FIN

8.3 Ports: Input and Output Message Queues

Filters, like modules, have input and output ports. To gain in flexibility compared to modules, these ports are handled through a different API. A filter handles **input and output messages queues**. A message queue is a buffer of messages that can be viewed as FIFO queue. Filters do not have any predefined port.

8.3.1 Port Vectors

Input messages queues (resp. output message queues) are stored separately in an `inputs` (resp. `ouputs`) vector.
Here are the main methods filters support to handle their two vector of ports (include/flowvr/plugins/baseobject.h):

- `void initInputs(int num) initOuputs(int num)` : initialize the vector of message queues with `num` ports. Must be called before any operation on any message queue.
- `int addOuputs(int num)` or `addInputs(int num)`: add `num` queues to the existing vector.
- `inputs` and `outputs`: the input and output message queue vectors. Access to a given queue using the `[]` operator.

8.3.2 Messages

Message queues handle classical FlowVR messages composed of a stamp list and data buffer (see chapter 7).

8.3.3 Input Message Queues

An input message queues is a `queue` (see Figure 8.1). Received messages are accumulated at the back. Oldest messages are at the front. Message queue indexing is based on the `num` stamp of the received messages. The stamp starts at `1` ans increases monotically for each new message. In case a message has a `num` stamp strictly greater that the stamp of the latest received message, the queue is filled with `Message::Null` messages to ensure the received message is stored at the `num` position in the queue as expected (this mechanism should never be triggered unless some filter miss use the `num` filter).
The main methods on an input message queue (include/flowvr/plugd/messagequeue.h) are the following:

- `void setName(const std::string &myname)`: set the name of the input message queue.
- `int frontNum()` : get the `num` stamp value of the oldest message in the queue.
- `const Message& frontMsg()` : get the oldest message.
- `const Message& backMsg()` : get the newest message.
- `void eraseFront()` : erase the oldest message.

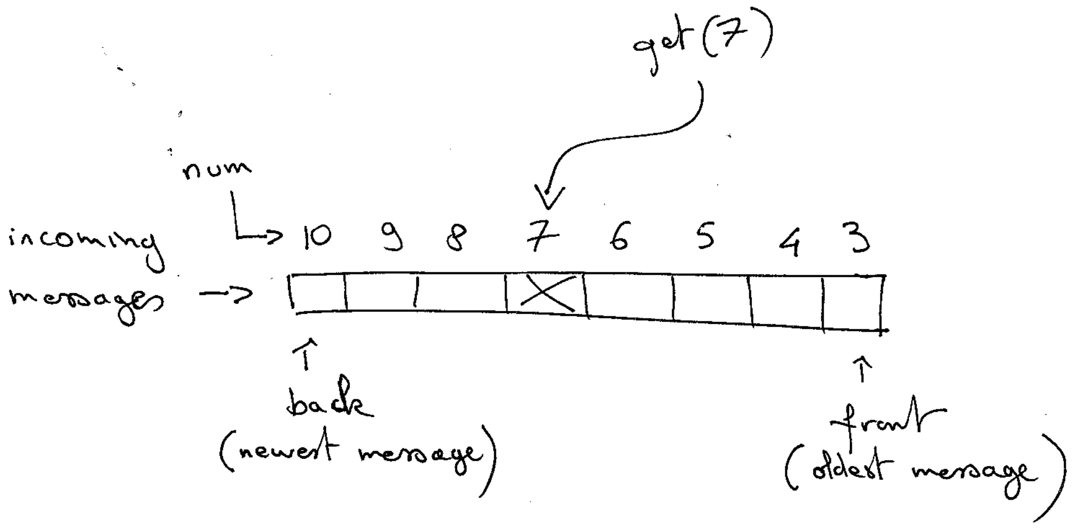


Figure 8.1: Input Message Queue Structure

- `void setFront(int num)` : arase all messages having a `num` stamp value stricly smaller than `num`. If the queue is empty `num` becomes the new front index.
- `void erase(int num)` : free the message in the queue having the specified `num` stamp value (message still in the queue but is content not valid any more).
- `int size()` : return the queue size (number of messages currently stored)
- `int backNum()` : return the `num` stamp value of the most recent message in the queue.
- `int frontNum()` : return the `num` stamp value of the oldest message in the queue.
- `int empty()` : check if the queue is empty.
- `const Message& get(int num)` : get message with `num` stamp value. Return a `Message::Null` message if not available.
- `bool stampsReceived()` : return *true* if the stamp list specification (see subsection 7.4.2) message has been received (system message with `num=-1`).
- `StampList& getStampList()` : return the stamp list specification associated with this port.
- `bool isConnected()` : return `true` if the input port is connected to an other component in the current application.

A VOIR *work to do on valid/clear/ Message::Null: for instance why message.clear() does not set the message to null ?* FIN

8.3.4 Output Message Queues

Here are the main methods to access the messages of an output message queue (include/flowvr/-plugd/outputmessagequeue.h):

- `void setName(std::string portname)` : set the name of the output message queue. Mandatory for each output message queue.
- `StampList stamps`: the stamp list specification (see subsection 7.4.2) for this port.
- `void newStampSpecification(Dispatcher* dispatcher, int it=0)` : set the stamp list specification for this queue. Must be called once, before processing any message. The stamp list specification must be properly set before to make this call. Usually caled in the `init()` method of the filter or in the `newStampListSpecification()`. The `it` optional parameter enables to overwrite the default value assigned to the `it` stamp in the system message built form the specification.
- `void put(MessagePut msg, Dispatcher* dispatcher, int num=-1)` : put the message `msg` into the output message queue and keep the dispatcher informed. The message can be of type `messagePut` or `messageWrite`. The `num` message stamp is set automatically unless overwritten by the `num` value given as the third argument.
- `int getNextNum()`: return the `num` stamp value to be used for the next message.
- `const string& getName()`: return the message queue name.
- `Message::Type msgtype`: output message queue type (`STAMP` or `FULL`). If an output port is set to `STAMP`, it will emit only `STAMP` messages and any attempt to send messages to a input port expecting `FULL` messages will produce an error. A `FULL` output port must emit messages with a payload (can be of size 0). A `STAMP` message can always be extracted from a `FULL` message depending on the destination needs (see **??**), thus declaring output ports as `FULL` is more general.

A VOIR *msgtype: is this really necessary or just syntactic sugar ?* FIN

A VOIR *getNextNum(): reading the code it seems that it returns the num of the next sent message* FIN

8.4 Standard filters

The standard filters are defined (and commented!) in the `filters.py` library. There are examples for many filters in `flowvr-examples/filters`.
A `<filterclass>` node must be specified containing the class name of the filter in a java-like notation (full class name with namespace components separated by periods). If the filter requires some initialization parameters they must be specified within a `parameter`.
Each time a filter is instantiated, it is added to the `.net.xml` and `.cmd.xml` files, indicating to the flowvrd daemon it has to load the corresponding plug-in.
FlowVR provides several predefined filters. Here is a short description for each of them and the corresponding plug-in:

- FilterDiscard: Discard all messages from input if filter not open, forward them all on output otherwise. Corresponding plug-in loaded by daemon : flowvr::plugins::Discard
- FilterIt: Forward on its output the messages received on its input having the same 'it' numbers than the ones received on the order port. Discard other messages (filter usually used with a synchronizer) Corresponding plug-in loaded by daemon : flowvr::plugins::FilterIt
- FilterLastOrNull: Forward on output messages the last input message or null message if there is no input. Discard other messages. Corresponding plug-in loaded by daemon : flowvr::plugins::FilterLastOrNull

- FilterMergeIt: Merge messages received on input port into one message sent on output port. More precisely, when it receives a message on the 'order' port, it inspects the icomming message queue ('in' port), discard all messages with a non null 'scratch' stamp value, concatenate all other messages in one message sent on 'out' port. This message has its 'scratch' stamp set to 0 and its 'stamp' stamp set to the sum of all 'stamp' stamps of the concatenated messages. The name of the 'scratch' and 'stamp' stamps are read from the omponent parameter list. These parameters are optionals Corresponding plug-in loaded by daemon : flowvr::plugins::MergeIt

- FilterMerge: Merge nb successive messages received and send on output. Messages can be discarded using the stamp given with the scratch parameter. The stamp given with the stamp parameter is combined in the merged message. Corresponding plug-in loaded by daemon : flowvr::plugins::MergeMsg

- FilterMultiplex: Forward messages received on inputs in0,...,innb-1 on output. Change the it and num stamps to 'hide' multiplexing Corresponding plug-in loaded by daemon : flowvr::plugins::MultiplexFilter

- FilterPreSignal: Filter that adds nb (a paramater) inital messages, then forward incoming messages to output (used to boot cycles). 1 parameter. 'nb': number of initial messages Corresponding plug-in loaded by daemon : flowvr::plugins::PreSignal

- FilterRotate: Alternatively dispatch messages received on in0, in1, ..., inNb-1 on output. 1 Corresponding plug-in loaded by daemon : flowvr::plugins::RotateFilter

- FilterScatter: Split input message in nb parts of size elementsize and send each part to output out0, out1, ..., outNb-1 Corresponding plug-in loaded by daemon : flowvr::plugins::Scatter

- FilterSignalAnd: Wait for receiving one message per input (nb inputs), and then send on output (use stamp read from message on in0) Corresponding plug-in loaded by daemon : flowvr::plugins::SignalAnd

- FilterUnrotate: Alternatively receives an message on each input and forward it on output Corresponding plug-in loaded by daemon : flowvr::plugins::UnrotateFilter

Extra filters can be programmed if required (see chapter 8).

8.5 Standard synchronizers

A synchronizer is an other special type of filter that receives and send only stamp messages. It is dedicated to messages synchronization.
It is instantiated using a syntax similar to filters (see section 8.4) by adding a `<synchronizer>` node and specifying the synchronizer to use in a `<synchronizerclass>` node.
FlowVR provides the following synchronizer (see flowvrd/src/plugins/sync for an up-to-date list)::

- GreedySynchronizer: A synchronizer which chooses the last available message. Every time it receives a message on endIt port, it looks for the most recent message received on stamps port and sends to order port the stamp of this message. Used for implementing greedy (subsampling) filtering. Corresponding plug-in loaded by daemon : flowvr::plugins::GreedySynchronizor

- MaxFrequencySynchronizor: Send a first message and then send output messages at min (incoming message rate, freq rate) Corresponding plug-in loaded by daemon : flowvr::plugins::MaxFrequencySynchronizor

Chapter 9

Application Execution

9.1 The FlowVR Daemon: `flowvrd`

FlowVR requires a daemon on each host involved in a FlowVR application execution, including the host where the application is launched from.

9.1.1 Launching the FlowVR Daemon

A FlowVR daemon must run on each host running modules. The command line for launching a daemon on a host is:

```
flowvrd --top
```

Each daemon can be launched in any order with different user logins (though not recommended).
The `--top` option request the daemon to monitor and display some data about the plugins and modules it controls. In particular it gives the frequency of each running module or filter.
To obtain the list of available options for `flowvrd` :

```
flowvrd -h
```

9.1.2 The FlowVR Daemon Command Language

A FlowVR daemon is controlled by a XML based command language. See flowvr-parse/dtd/commands.dtd for the authorized syntax.
The commands for a given application are stored in the `.cmd.xml` file.
A FlowVR application developer only have to launch and stop daemons. He should not have to directly use the command language.
If you are keen to learn more, please refer to the developer manual's daemon section (see section 20.1).

9.1.3 Application Launching

Simply append the application name after FlowVR's aptly-named launch command :

```
flowvr Primes
```

9.1.4 Start/Pause/Stop

Once the application is running, `flowvr` listens on the command line for user input. In a nutshell you can suspend, stop and restart the application.
To suspend the execution:

```
pause
```

To resume the execution:

```
start
```

To stop the execution (all modules receive an exit signal through the *wait* call and all plugins the daemons may have loaded are unloaded):

```
stop
```

This command does not stop the daemons.
For a list of all available commands:

```
help
```

9.1.5 `flowvr-kill`

If the application does not start or stop correctly, it can result in zombie processes polluting the machines. The `flowvr-kill` utility gets rid of those lingering processes. If your application runs locally, simply call:

```
flowvr-kill
```

If you are running a distributed application, simply provide the list of hosts involved:

```
flowvr-kill --list 'host1 host2 host3'
```

It is also possible to list the hosts in a file and provide the file to `flowvr-kill`. You only have to list the hosts, one per line, and save the file as txt.

```
flowvr-kill --hostfile mycluster.txt
```

When a module calls the init() method, it creates a temporary file locally in the `/.flowvr` directory. This files stores the PID of the module and the name of the machine. When `flowvr-kill` is called it just inspect on each machine the `/.flowvr` directory and kill all listed processes.

9.2 Run-time Environment

FlowVR run-time environment (see chapter 20) relies on a daemon that runs on each host of the target machine. This daemon is in charge of:

- registering each module running on its host,
- implementing all message exchanges transiting through its host,
- executing all local filters.

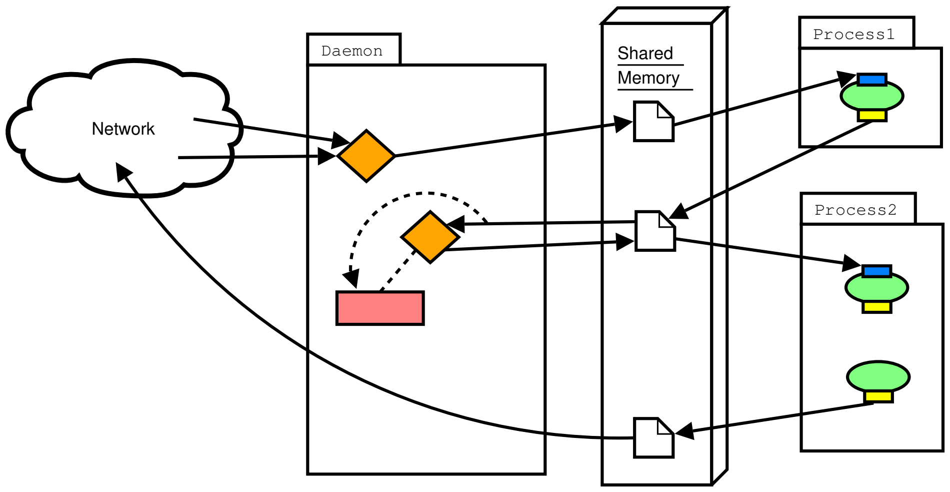


Figure 9.1: The FlowVR daemon acts as a broker between modules (green ellipses), using filters (orange diamonds) for resampling and simple data processing, and synchronizers for synchronization policies (pink rectangle).

Modules are launched directly using their own command. FlowVR does not impose a specific launching mechanism to ease portability of existing code. For instance, MPI code can use `mpirun`. Once launched, each module registers at the daemon of its host.
The daemon manages shared memory segments used to store all messages it handles. Local modules and filters have a direct access to this memory under the control of the daemon. The daemon takes care of memory allocation, ensures safe read and write accesses and garbage collection. Having direct access to the shared memory segment saves data copies.
When a module executes a *put*, it tells the daemon that the message is ready to be forwarded to its destination. If the destination module runs on the same host, the daemon provides a pointer to the module that directly reads the message from the shared memory. If the message has to be forwarded to a distant module, the daemon sends it to the daemon of the distant host. The target daemon retrieves the message, stores it in its shared memory segment and provides a pointer to the destination module, which processes it exactly as for local communications.
Each daemon stores a table of actions to perform on messages according to their origin. An action can be a simple message routing, or the execution of a filter. Filters are plugins loaded by the daemons when starting the application. Here, an important difference between a filter and module appears. A filter is necessarily a daemon plugin, while a module is an external process or thread. This design is required to provide the best flexibility for programming modules, while providing minimum overhead in complex filtering networks.
The daemon and its plugins are multi-threaded, ensuring a better performance scalability on multi-core and multi CPU architectures.

9.3 Application Deployment, Execution and Debugging

Application processing is controlled with a Python script, and deployment and execution is controlled from the FlowVR command called `flowvr` (see section 5.6). `flowvr` follows several steps (see **??**). Starting an application usually requires the user to set the values for several parameters, eg. the the mapping of modules on the target host machines.
`flowvr` produces several intermediate files. Three of them are worth to inspect when debugging:

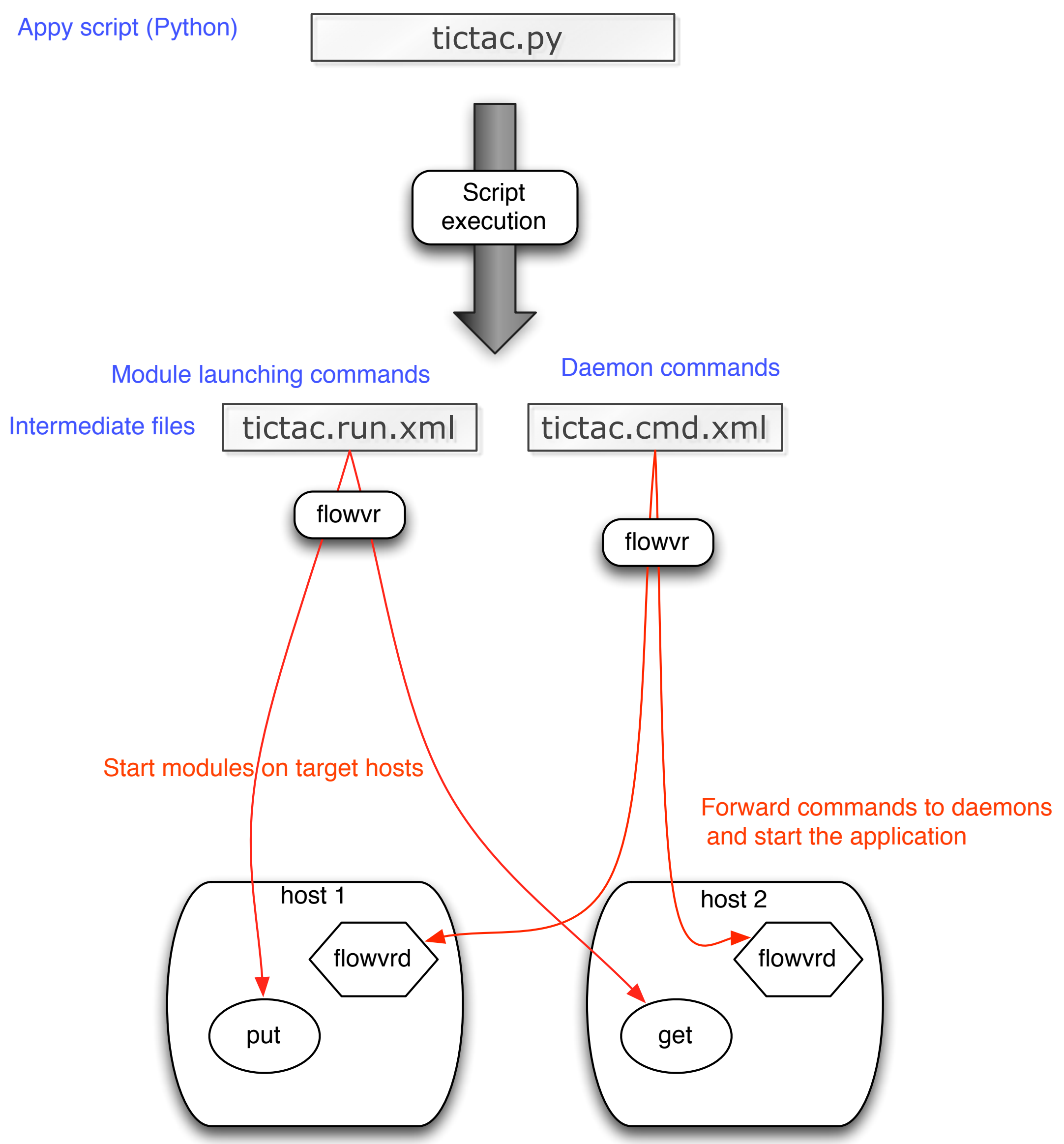


Figure 9.2: The FlowVR front-end command `flowvr` starts from a parametrized high level description of the application (top) to produce intermediate files (xml) and launch the application.

- The `.net.xml` file that contains the description of the network graph. The `flowvr-glgraph` utility (see section 16.1) enables to produce a browsable network graph image.
- The `.run.xml` file that contains the command line used to start each module.
- The `.cmd.xml` file that contains commands to the `flowvr` to launch filters and set up links.

FlowVR comes with a built-in trace capture tool (see chapter 14) that is simple to activate. From these traces, the tool `flowvr-gltrace` computes a visual representation (see Figure 14.1) of the activity of the various components and the message flows.
Often when starting for the first time an application on a cluster, users run into basic cluster configuration issues. Make sure that you have a `ssh` access to the various nodes of your target machine. It also very convenient if all the nodes share the same file system (the norm on most clusters), easing access to the required binaries and libraries. Often you also have to change the system settings to enable the allocation of a large shared memory segment. If some components open display windows, make sure you set accordingly the authorizations of the windowing system. FlowVR is daemon based. Before starting an application, you need to run one FlowVR daemon on each machine host involved in the application.

9.4 Using an MPI network layer

By default, the FlowVR daemon communicates over TCP/IP networks (`flowvr.plugins.NetTCP`). Relying on the TCP/IP protocol may not give the best performance on high performance networks such as Infiniband or Gemini. In this case, we provide two plugins that rely on MPI to transport messages. These plugins enable better communication performance if your MPI implementation supports native communication over such networks.

9.4.1 `NetMPI` vs `NetMPIm` plugins

- flowvr.plugins.NetMPI relies on a mix of TCP and MPI communication using the `MPI_THREAD_FUNNELED` option (well supported by all MPI distributions). So far this plugin gave us the best performance.
- flowvr.plugins.NetMPIm fully relies on MPI communications using the `MPI_THREAD_MULTIPLE` option (not well supported by all MPI distributions)

9.4.1.1 `NetMPI` and `NetMPIm` plugins Work with

- OpenMPI (no high performance Infiniband support, must be preloaded using the `LD_PRELOAD` environnement variable)
- mpich, mpich2, mpich3 (but no high performance Infiniband support)
- mvapich (supports Infiniband, tested thoroughly)

OpenMPI suffers from a bug that makes the daemon crash when using a NetMPI(m) plugin (https://bugzilla.redhat.com/show_bug.cgi?id=801945). It is triggered when the library is loaded when loading the plugin with dlopen. This can be prevented by preloading the OpenMPI library when launching the daemon using the `LD_PRELOAD` environement variable.

9.4.2 Running flowvrd with MPI

The only change required to use these plugins is to launch the daemons appropriately.
Sample command:

```
mpirun -np nbDaemon -machinefile /path/to/machinefile -env
   MV2_ENABLE_AFFINITY 0 flowvrd -s 1G --network flowvr.plugins.NetMPI
```

`MV2_ENABLE_AFFINITY` is required with the mvapich implementation. It enable having either `MPI_THREAD_FUNELED` or `MPI_THREAD_MULTIPLE` at MPI initialization. Other options may be required for other MPI implementations.

9.4.3 Enabling top output with MPI

To obtain a readable "top" output, you need to create the following script `launchDaemon.sh`:

```
flowvrd -f --top -s 1G --network flowvr.plugins.NetMPIm > /tmp/logsDaemon.
   txt 2> /tmp/logsDaemon.err.txt
```

and launch the daemons using:

```
mpirun -np nbDaemon -machinefile /path/to/machinefile -env MV2\_ENABLE\
   _AFFINITY 0 ./launchDaemon
tail -f /tmp/logsDaemon.txt
```

Chapter 10

Language bindings

10.1 Python Module Programming

10.1.1 Interface

For Python modules, there is a single entry point:

- lib/flowvr/python/flowvr.py

10.1.1.1 `initModule`

A module must first be registered and initialized to connect to the flowvr daemon.

```
module = flowvr.initModule(ports)
```

This method takes as argument:

- `ports`: a vector of user defined input and output ports (Section 6.2.2).
- `instancename` and `modulename`: these optional arguments are usually omitted. If set they overwrite the module name automatically set by FlowVR using `modulename/instancename` as new name.

10.1.1.2 `wait`

A classical python module loop looks like this :

```
while module.wait():
    # work to be done during an iteration

# wait returns zero: end the module execution now
module.close()
# more things to do specific to this module to properly stop it.
```

10.1.1.3 `get`

```
message = inport.get()
```

See the original section (see 6.2.1.2) for more information.

10.1.1.4 `put`

```
outport.put(message)
```

See the original section (see 6.2.1.3) for more information.

10.1.1.5 `getStatus`

This method returns the status of the module, zero if an error occurred or if the module needs to be stopped. All other methods also return the status after completion.

10.1.1.6 `close`

This method should be called to properly exit the module.

10.1.1.7 `alloc`

```
x = module.allocString('hi there!')
```

If performance isn't your first priority, it is easier to construct a string and transfer it to the buffer afterwards.
See the original section (see 6.2.1.6) for more information.

10.1.2 User Defined Ports

The information related to each port is:

- its direction (input or output)
- whether it's a full or stamps port
- its blocking state (blocking or non-blocking)
- for output ports, the type of the stamps it plans to send out.

10.1.2.1 Port Vector

Ports need to be declared before the module initialisation. All user defined ports need to be stored in a vector passed as argument of initModule (see subsection 6.2.1).

10.1.2.2 Output Port

```
   outport = OutputPort(myname)
```

Output port constructor:

- `myname`: port name
- `mystamps`: a stamp list specification (see subsection 7.4.2) if the messages sent on that port will carry user defined stamps. This argument is optional and the stamp list specification can be specified latter.

- `bOwn`: only used when passing a stamp list specification in `mystamps`. If set to `true` the stamplist is deleted upon destruction of the port, not otherwise.

10.1.2.3 Input Port

```
inport = InputPort(myname)
```

Input port constructor:

- `myname`: port name
- `mystamps`: The stamp list specification usually does not need to be specified here (leave the default NULL value) as it will be set by the system at initialisation according to the stamp list specification of the incoming messages.
- `bOwnStampList`: only used when passing a stamp list specification in `mystamps`. If set to `true` the stamplist is deleted upon destruction of the port, not otherwise.
- a stamp list specification (see subsection 7.4.2) if the messages sent on that port will carry user defined stamps. This argument is optional and the stamp list specification can be specified latter.

10.1.2.4 Example

In this example we create an extra input and output port. We also add stamps to the output port:

```
# declare an input and output port
outport = flowvr.OutputPort('out')
inport = flowvr.OutputPort('in')

# add an int stamp
outport.addStamp("mycounter", int)

# prepare the ports vector
ports = flowvr.vectorPort()
ports.push_back(inport)
ports.push_back(outport)
```

Caevat in Python: storage for `inport`, `outport` or `ports` is managed by Python, so they may be garbage-collected when they get out of scope, crashing the application. This can be avoided by making them global variables or by telling the garbage collector to forget them with `x.this.own(False)`.

10.1.3 Event Ports

It is possible to have non-blocking input ports called **event ports**. In opposite to classical input ports, the `wait` unlocks even though no new message might be available on event ports. Thus, if no message was available when `wait` was called, `get` returns an invalid message (make sure to test it).
Event ports can be convenient for grabbing messages that are sent episodically, i.e. at a rate significantly lower than the expected module iteration rate. Event ports are typically used to receive control messages from a GUI.
If a module has only event ports and the `beginIt` port is not connected, it will never block at the `wait` call. This may lead to a free running module that consumes much more CPU resources than expected.
Messages can be accumulated on input ports. For instance, if many messages are sent between two iterations of the event port holder, it will possibly lead to buffer overflows. Inserting a filter to erase or merge accumulated messages might be required to avoid such pitfall.

The same behavior can be obtained by putting an adequate filter before a classical input port. This approach has the advantage of changing the module behavior for a given application without having to change the module code. Generally we advice not to abuse of event ports that tend to affect the genericity of modules.
Being an event port or not need to be set before calling the `initModule` method (see subsection 6.2.1), either:

- At port creation (see subsubsection 6.2.3.3)
- Setting to true the port non blocking flag (call after `initModule` has no effect):

```
InputPort::setNonBlockingFlag(bool bBlock)
```

A VOIR *Et l'initialisation des modules paralléle ?* FIN

10.1.4 Probing Ports State

- `bool Port::isConnected()`: return `true` if the port is connected. Call valid only after the first `wait`.
- `bool Port::isInput()` and `bool isOuput()`: return `true` if match the expected port type.
- `bool InputPort::isNonBlockingPort()`: return `true` if event port (see 6.2.3.4).

10.1.5 Running modules

For Python modules, the cmdline (see Section 5.7) contains "`python module_path.py`" if the module path is absolute or relative to the current directory, or "`python -m module_name.py`" if it can be found in the PYTHONPATH.
Remember that for a distributed execution each module binary needs to be accessible from the nodes where the module is executed (copy the binaries on these nodes or have rely on a distributed file system like NFS).

10.1.6 Messages and buffers

The Python API is simpler and more limited, because it does not need to be as optimized as the C++ version.

10.1.6.1 Stamps

Stamps in messages are accessed by name. The StampInfo structure is not manipulated explicitly. They are declared as:

```
outport = flowvr.OutputPort('text')

outport.addStamp("mycounter", int) # integer stamp
outport.addStamp("myarray", int, 3) # fixed array of 3 ints
outport.addStamp("mystring", str) # string
```

Setting the stamps of an output message:

```
# outport.stamps must be passed to the constructor if non-standard stamps will be
    output.
message = flowvr.MessageWrite(outport.stamps)

message.setStamp("mycounter", 123)
message.setStamp(("myarray", 1), 456) # set element 1 of the array
message.setStamp("mystring", "toto")

outport.put(message)
```

Reading stamps from an input message:

```
message = inport.get()
mycounter = message.getStamp('mycounter')
arr = message.getStamp('myarray') # get the full array

stamps = message.getStamps() # dictionay with all stamps
mycounter2 = stamps['mycounter']
```

10.1.6.2 Data

The Buffer data is exposed as Python strings.
Setting the data of a message:

```
message = flowvr.MessageWrite()
message.data = module.allocString("toto")
```

Reading from a buffer:

```
message = inport.get()
message.data.asString()
```

10.1.6.3 Python "chunks": converting binary data to/from strings

Since binary data in FlowVR is exposed as strings by the API, some functions are required to convert data from/to Strings. These are standard Python functions.
Homogeneous arrays:

```
import array

# array -> string
a = array.array('f') # 32-bit floating point array
a.append(1.2)
a.append(3.4)
s = a.tostring() # s='\x9a\x99\x99?\x9a\x99Y@'

# string -> array
x = array.array('f')
x.fromstring(s)
```

Structured data:

```
import struct

# structured data -> string
s = struct.pack('if', 1, 4.5)

# string -> structured data
x, y = struct.unpack('if', s) # x = 1, y = 4.5
```

Numpy is the standard matrix manipulation library in Python. It can also convert to/from binary data in strings:

```
import numpy

# numpy array -> string
a = numpy.zeros((10,2))
a[3,0] = 1.54
s = a.tostring()

# string -> numpy array
numpy.fromstring(s, dtype = 'float64')
a.reshape((10,2))
```

Part IV

Examples

Table of Contents

Chapter 11

Primes

This is the main example we use through this manual. Source code is installed in share/flowvr/examples/primes. Primes can be used as a template to start developing your own application.
This application shows a classical structure coupling a simulation, a visualization and an interaction interface. The simulation iteratively computes prime numbers. The visualization displays them over a spiral. The user can interactively rotate the view using arrow keys.
More precisely:

- '*compute*' : a parallel computation. Each process iteratively tests some integers for primality, and sends its latest results each time a certain amount of prime numbers have been found. Only a few numbers are calculated by iteration in order to maintain a rather smooth cooperation between the components of the application.
- '*visu*' : a visualization process based on OpenGL and Glut. A spiral is drawn, with points plotted only when the angle in radian is prime. This condition let appear a 'Galaxy-like' pattern well known in Mathematics and used as an heuristic for very big prime numbers calculation. The display is constantly updated as new numbers are computed, and a zoom is automatically performed to fit the window area. As the application is running in an infinite loop, the process is limited to the last million of numbers received.
- '*capture*' : an input dedicated process retrieving the keyboard state using classical Glut function calls. Cursor keys hit information is transmitted to the visualization system which rotates the display accordingly.

As this example is primarily a case study, code and algorithms have been deliberately kept unoptimized to favor simplicity and clarity.

11.1 Application Instantiation and Execution

Refer to the Getting Started Section for compiling, instantiating and executing Primes (see chapter 2). Make sure you have atleast ran the primes application on your local machine.

11.2 Directory Structure

The share/flowvr/examples/primes directory contains:

- The share/flowvr/examples/primes/make-app.sh utility script file for compiling and installing the application.
- The share/flowvr/examples/primes/CMakeLists.txt cmake project configuration file.
- The share/flowvr/examples/primes/primes.py the flowvr network generation script.
- The share/flowvr/examples/primes/src directory that contains the module source files as well as a CMakeList.txt file.
- The share/flowvr/examples/primes/config input files for cmake to produce configuration files.

All the other examples from share/flowvr/examples have the same structure.

11.3 Compiling FlowVR modules

FlowVR modules are plain executables built from C++ code (or other languages for which there exists flowvr bindings). The compilation flags are:
`-I ${FLOWVR_PREFIX}/include`
The link flags:
`-L ${FLOWVR_PREFIX}/lib -lflowvr-base -lflowvr-mod`
Cmake is the build system used in FlowVR and the examples.
Cmake processes `CMakeLists.txt` files to prepare the compilation of the application. It performs tasks like searching for FlowVR installation directories, definition of the library and binaries to compile, files to install, etc. Then it generates appropriate Makefiles.
Primes relies on two `CMakeLists.txt files`, written to be generic enough to be easily reused for other applications. These files are (hopefully) commented enough to be self-explanatory:

- The share/flowvr/examples/primes/CMakeLists.txt
- The share/flowvr/examples/primes/src/CMakeLists.txt

Running share/flowvr/examples/primes/make-app.sh calls cmake to compile the modules.

11.4 Environment Variables

The `primes-config.sh` file generated when compiling Primes sets several environment variables (share/flowvr/examples/primes/bin/primes-config.sh).

- **PATH:** Add a path to the binary search path to give access to the module binaries. It makes application more portable by avoiding to provide access path to module launchers like `flowvr-run-ssh` (see subsection 20.8.2).
- **primes_DIR:** Cmake specific environment variable that gives access to the share/flowvr/examples/primes/cmake/primesConfig.cmake. It enables to reuse this application from other applications using Cmake. The `FIND_PACKAGE(primes)` command will succeed and set the variables `primes_FOUND`, `primes_INCLUDE_DIR`.

The `primes-config.sh` script does is not required to run the application, because relative paths to the exectuables (bin/capture, etc.) are used.

11.5 Modules

Primes relies on three modules. Each module is implemented in a source file.
The 3 modules are:

- `compute`: loops over integers, and checks whether they are prime numbers. Those that are found to be prime (with a simple brute-force algorithm) are sent on its output port `primesOut`. It can be started in several instances, where each instance is in charge of computing different prime numbers in parallel. Modules are named `compute/0`, `compute/1`, etc., each one having a `primesOut` output port.
  - share/flowvr/examples/primes/src/compute.cpp
- `visu`: opens a graphical window and displays the primes received on its `primesIn` input port as dots on a spiral. It launches a single module with two input ports `keysIn` and `primesIn` where it receives interaction events and primes numbers.
  - share/flowvr/examples/primes/src/visu.cpp
- `capture`: launch a single module with one `keysOut` output port where the module sends at each iteration the arrow key states .
  - share/flowvr/examples/primes/src/capture.cpp

11.5.1 Module implementation

Modules are plain executables. They are started by the `flowvr` executable, that calls `flowvr-run-env`, `flowvr-run-ssh` and/or `mpirun`.
At startup, a module:

1. sets up a list of input and ouput ports (`std::vector <flowvr::Port*>` in C++).
2. connects to the localhost's FlowVR daemon. It registers with a module identifier given by the `FLOWVR_MODNAME` environment variable. This is taken care of by `flowvr-run-env` and the module source does not need to know about it. The daemon grants access to the shared memory segment for the module (Section 9.2).
3. enters an event processing loop.

Several modules can be implemented in the same executable. In this case, the steps above are duplicated for each module.

11.5.2 The event processing loop

The event processing loop has the following structure:

```
// initialize API

...
while (!module -> wait())
{
    // read input message
    module -> get(input_port, input_message);
```

```
      // handle messge
      ....

      // write output message
      module -> put(output_port, output_message);
  }
```

The `wait()` call returns when there is a message on all its input ports that are connected (non-connected ports are ignored). `wait()` returns true when the application should be stopped.
A message is automatically sent on *endIt* every time the module enters `wait()`. Symmetrically, the processing cannot start if *beginIt* is connected and there is no message on it.
The source of the Primes modules is generously commented, so it is also a good starting point for module programming.
A detailed description of module programming is in Section 11.5.

11.6 Component Assembly

These modules are assembled in the application root component, `primes`, defined in the files share/flowvr/examples/primes/primes.py.
For the sake of pedagogy, Primes defines the `example` parameter used to browse between several examples of component assembly. The example can be selected from the command line, like:

```
python primes.py 0 && flowvr primes
```

This selects example 0, builds the graph and runs the application. The `&&` ensures that the application is start up only if the graph building succeeds.
In the following, we describe the different examples, increasing the complexity with each.

11.6.1 Without synchronization

The simplest version is a producer-consumer with the `compute` and `visu` modules. This corresponds to the data-driven policy (see 4.1.1).
The flowvrapp code for this application is simply (example 0):

```
from flowvrapp import *

computerun = FlowvrRunSSHMultiple('bin/compute', hosts = 'localhost', prefix = '
    compute')
compute = Module("compute/0", run = computerun)
compute.addPort("primesOut", direction = "out")

visu = Module("visu", cmdline = "bin/visu")
visu.addPort("primesIn", direction = "in")

compute.getPort("primesOut").link(visu.getPort("primesIn"))

app.generate_xml("primes")
```

A `FlowvrRunSSHMultiple` object is required for the compute module because it is designed to be run in several instances.
Depending on the relative speed of the CPU and display, `compute` may produce prime numbers faster than `visu` module can display them. When messages are not consumed, they stack up into flowvrd's memory, which eventually fills up and crashes the application.

11.6.2 Synchronization basics

We have to dive a little more into how FlowVR handles messages.

11.6.2.1 Stamps

A stamp (see section 7.4) is a dictionary that maps names to values, it contains at least an iteration number (named `it`). Ports are defined to transmit either the message and its stamps (*full port*), or just the stamps (*stamps port*).
The connection rules are:

- ports of the same type can be connected
- full output ports can be linked to stamps input ports: they get only the stamps part of the messages
- stamps output ports cannot be linked to full input ports

11.6.2.2 Controlling the module's output rate

In our case, in order to limit the throughput of `compute`, `visu`'s `endIt` can be linked to compute's `beginIt`. This way, `compute` will not start an iteration before `visu` has ended processing the previous (example 1):

```
...
visu.getPort("endIt").link(compute.getPort("beginIt"))
....
```

The output rate of the module is effectively controlled in this way.
This example happens to work because `compute`'s event loop is not exactly as described above: a message is sent on the `primesOut` port before the first call to `wait()`. If the two modules would use the standard loop above, they would deadlock.
Also, computation and visualization are not simultaneous, because at any moment at least one of the two modules is waiting for a message to arrive on its input port: they are not executing in parallel.

11.6.2.3 The presignal filter

Intuitively, making them run in parallel just requires that `compute` is run two times instead of one before it has to wait for `visu`'s signal. This could be coded in the compute module. However, hard-coding the number of `puts` before the first `wait()` does not scale: if more modules are in the loop, this number has to be adjusted also, so it depends on the FlowVR graph topology.
Therefore, there is a special kind of primitive, the *PreSignal*, that transmits stamps messages, and also sends a specified number of messages before its first wait.
The pre-signal is not a module, it is a *filter*. Filters are primitives that run as a thread in the FlowVR daemon, and that have access to the message queues of their ports. They do not necessarily have a `beginIt` and an `endIt`.
In Flowvrapp, standard filters are defined in the library `filters.py`. The PreSignal can be used with:

```
from flowvrapp import *
from filters import *

...

# nb is the number of messages to send before the first wait
```

```
presignal = FilterPreSignal("presignal", nb = 1)

visu.getPort("endIt").link(presignal.getPort("in"))
presignal.getPort("out").link(compute.getPort("beginIt"))
...
```

This implements a demand-driven policy (section 4.1.2).

11.6.3 Synchronizing multiple inputs

11.6.3.1 An example

Now the display of `visu` will be controlled by the user: a new module, `capture`, opens a window and waits for key events on the window. It sends messages at a predefined frequency (1 kHz) containing the state of the arrow keys.
The `visu` module now has two connected inputs, `keysIn` and `primesIn` (example 3):

```
...
capture = Module("primes/capture", cmdline = "bin/capture")
capture.addPort("keysOut")

capture.getPort("keysOut").link(visu.getPort("keysIn"))
...
```

It processes the messages only when there is at least one available on each port. In this case, `capture` is producing messages faster than `compute`. Therefore, the capture messages are queueing up in flowvrd, eventually crashing it.

11.6.3.2 Synchronizers

To avoid this, we use a *synchronizer*. Similar to a filter, synchronizers are primitives that run in the FlowVR daemon.
Here, we use a *GreedySynchronizer* synchronizer. It has two stamps input ports: `endIt` (yes, it is an input port) and `stamps`. Port `endIt` shoud be connected to a port that outputs messages slowly, and `stamps` should be connected to a fast port. Every time a slow message arrives, the last fast message that arrived is sent on the GreedySynchronizer's output port, `order`. Previous fast messages are discarded.
The GreedySynchronizer handles stamps messages, but here the messages to be subsampled, from `capture`'s `keysOut`, are full: the `order` port will output only the stamps part of the message.
To get the full message, we can use a *FilterIt* filter. A FilterIt transmits messages from its `in` port to its `out` port. Transmitted messages are selected on the `it` field of their stamps, other messages are dropped. The selected `it`'s are those of the stamps messages received on the `order` port of the FilterIt.
By chaining the `order` ports of the GreedySynchronizer and the FilterIt, it is possible to apply the same filtering on the stamps and the full message of the "fast" port.
In our case, the slow port is that comming from `compute` via `presignal`m and the fast one, that of `capture`. Therefore, we can synchronize them with (example 4):

```
...
sync = GreedySynchronizor("sync")
presignal.getPort("out").link(sync.getPort("endIt"))
capture.getPort("keysOut").link(sync.getPort("stamps"))

filterit = FilterIt("filterit")
capture.getPort("keysOut").link(filterit.getPort("in"))
sync.getPort("order").link(filterit.getPort("order"))
```

```
filterit.getPort("out").link(visu.getPort('keysIn'))
...
```

11.6.3.3 Further reading

More information on the available filters and synchronizers can be found in the Doxygen documentation, generated in FlowVR's build directory by

```
make doc-flowvr
```

Then point a browser to `flowvr/doc/html/index.html` in the build directory.

11.6.4 Composites

When FlowVR applications get larger, a linear description of the graph becomes harder to read. It becomes useful to group primitives together into *composites*.

11.6.4.1 Grouping primitives

It is convenient to handle a set of related primitives as if it was a single object. Therefore, composites have a `getPort` method that can be used to expose the ports of the enclosed primitives. They do not have an `addPort`, though: ports can only be created on primitives.
It is conventional to see composites like directories in a file hierachy. Therefore, primitives in a composite are of the form `compositeName/primitiveName` (the slash indicates the structure). Composites can be nested also.
An object that is either a primitive or a composite is called a *component*.
In our example, a `Display` composite will group together `visu` and `capture`, and simply expose `visu`'s `primesIn` and `endIt` (example 5):

```
class Display(Component):
  """ Displays primes in a window, and a secondary control window """

  def __init__(self, prefix):
    Component.__init__(self)

    visu = Module(prefix + "/visu", cmdline = "bin/visu")
    ...
    capture = Module(prefix + "/capture", cmdline = "bin/capture")
    ...
    presignal = ...
    sync = ...

    ### expose input and output ports

    self.ports["endIt"] = visu.getPort("endIt")
    self.ports["primesIn"] = visu.getPort("primesIn")


display = Display("display")
compute.getPort("primesOut").link(display.getPort("primesIn"))
```

The `prefix` is the prefix of all primitives in the composite. Note that the `presignal` that was used for both `compute` and `sync` has to be duplicated.

11.6.4.2 The greedy

In fact, the synchronization pattern described above, that re-samples messages from a fast port to match the speed of a slower port is classical: it is called a *Greedy*.
The composite encapsulating the primitives of a Greedy is a `Greedy` object (defined in the `filters.py` library), so the code above can be simplified to (example 6):

```
...
class Display(Composite):

  def __init__(self, prefix):
    ...
    # greedy that samples captures's keysOut at the speed of visu's endIt
    greedy = Greedy(prefix + "/greedy")

    visu.getPort("endIt").link(greedy.getPort("sync"))
    capture.getPort("keysOut").link(greedy.getPort("in"))
    greedy.getPort("out").link(visu.getPort("keysIn"))
    ...
```

11.6.5 Multiplying compute modules

Now we can run several instances of compute, to exploit mulitple CPUs or modules on other machines. We will group the computation modules in a `Compute` composite.

11.6.5.1 Running more instances

The `FlowvrRunSSHMultiple` object we saw previously runs several instances of an executable. It is called like

```
computerun = FlowvrRunSSHMultiple("bin/compute", hosts = "mohawk mohawk opata
```

Where:

- the first argument is the command line of the module to run
- `hosts` is a string with host names, separated by spaces. If a host appears several times, the module is run in several instances the specified machine.
- `prefix` is the prefix of the name of the modules. </ul>

In this case, the corresponding modules must be constructed like:

```
module = Module("compute/0", run = computerun)
module = Module("compute/1", run = computerun)
module = Module("compute/2", run = computerun)
```

The name of the module must match the prefix given to `FlowvrRunSSHMultiple`, and they are numbered from 0 to n-1 (n is the number of instances).

11.6.5.2 Merging results

The `FilterMerge` filter has an arbitrary number of input ports, `in0`, `in1`,..., `inn-1`. When a message is available on all of its ports, they are concatenated into a single message that is sent on the `out` port. New input ports are added to the `FilterMerge` with `newInputPort()`.
We have the necessary components for our parallel compute (example 7):

```
class Compute(Composite):

  def __init__(self, prefix, hosts):
    Component.__init__(self)

    computerun = FlowvrRunSSHMultiple('bin/compute', hosts = hosts, prefix = prefix
        )

    # hosts_list: convert hosts to a list
    hosts_list = hosts.split()
    ninstance = len(hosts_list)

    merge = FilterMerge(prefix + '/merge')
    all_beginIts = []

    for i in range(ninstance):
      compute = Module(prefix + "/" + str(i), run = computerun)
      compute.addPort("primesOut", direction = "out")

      compute.getPort("primesOut").link(merge.newInputPort())
      all_beginIts.append(compute.getPort("beginIt"))

    self.ports["primesOut"] = merge.getPort("out")
    self.ports["beginIt"] = tuple(all_beginIts)

...

compute = Compute("compute", hosts = "localhost " * 4)
display = Display("display")
compute.getPort("primesOut").link(display.getPort("primesIn"))
```

A few comments:

- The `host = "localhost " * 4` specifies that the `compute` modules should be run on the local host in four instances (in Python a string muliplied by an int returns this number of concatenated strings).
- `all_beginIts` is a list of all `beginIt` ports that must be connected. For an input port `inC` of a composite, it is acceptable to have a tuple of primitive input ports. This means that a port connected to `inC` should be connected to all the primitive ports.

11.6.5.3 On multiple hosts

Running this example in 4 instances on the machine mohawk (which is the localhost) plus 4 more instances on opata should be a matter of setting

```
compute = Compute("compute", hosts = "mohawk " * 4 + "opata " * 4)
```

Unfortunately a few more adjustments are necessary (example 8):

```
class Compute(Component):
...
    for i in range(ninstance):
      compute = Module(prefix + "/" + str(i), run = computerun, host = hosts_list[i
          ])
...
```

```
class Display(Component):
...
    visu.run.options += "-x DISPLAY "
    capture.run.options += "-x DISPLAY "
...
app.default_host = "mohawk"

compute = Compute("compute", hosts = "mohawk " * 4 + "opata " * 4)
```

Where:

- `app.default_host` is set to localhost, but FlowVR is confused because localhost is the same as mohawk. Therefore, all primitives should be mapped by default on mohawk.
- since the modules that use X (the graphical interface) are not running on the localhost anymore, they don't have access to the `DISPLAY` variable. It must be explicitly propargated to the modules with `visu.run.options += "-x DISPLAY "`.
- the `compute` modules need to be mapped explicitly on the hosts they are running on, via `host =` options. Order does matter.

11.6.5.4 Tree merge

We are in a context where computations are run on many "slave" machines, and the results of the computations must be merged on a central "master" machine. A simple way of doing this is to send all the bits to the master and do the merging on it. However, due to the message fragmentation and the cost of merging results, this may overload the master, while the slaves are idle.
A solution to this is to make sub-groups of slaves and merge the results on these subgroups. This can be done recursively, in a tree of merges (example 9):

```
class Compute(Component):

  def __init__(self, prefix, hosts, out_port):
    ...
    all_primesOut = []

    for i in range(ninstance):
      compute = Module(prefix + "/" + str(i), run = computerun, host = hosts_list[i
          ])
      all_primesOut.append(compute.getPort("primesOut"))
      ...
    make_filter_tree(prefix + '/tree', all_primesOut, out_port)

...

display = Display("display")

hosts = "mohawk " * 4 + "opata " * 4
compute = Compute("compute", hosts = hosts, out_port = display.getPort("primesIn")
    )
```

The function `make_filter_tree` needs to know where the input ports and the output port come from, to decide where to map the tree nodes.

Chapter 12

Fluid

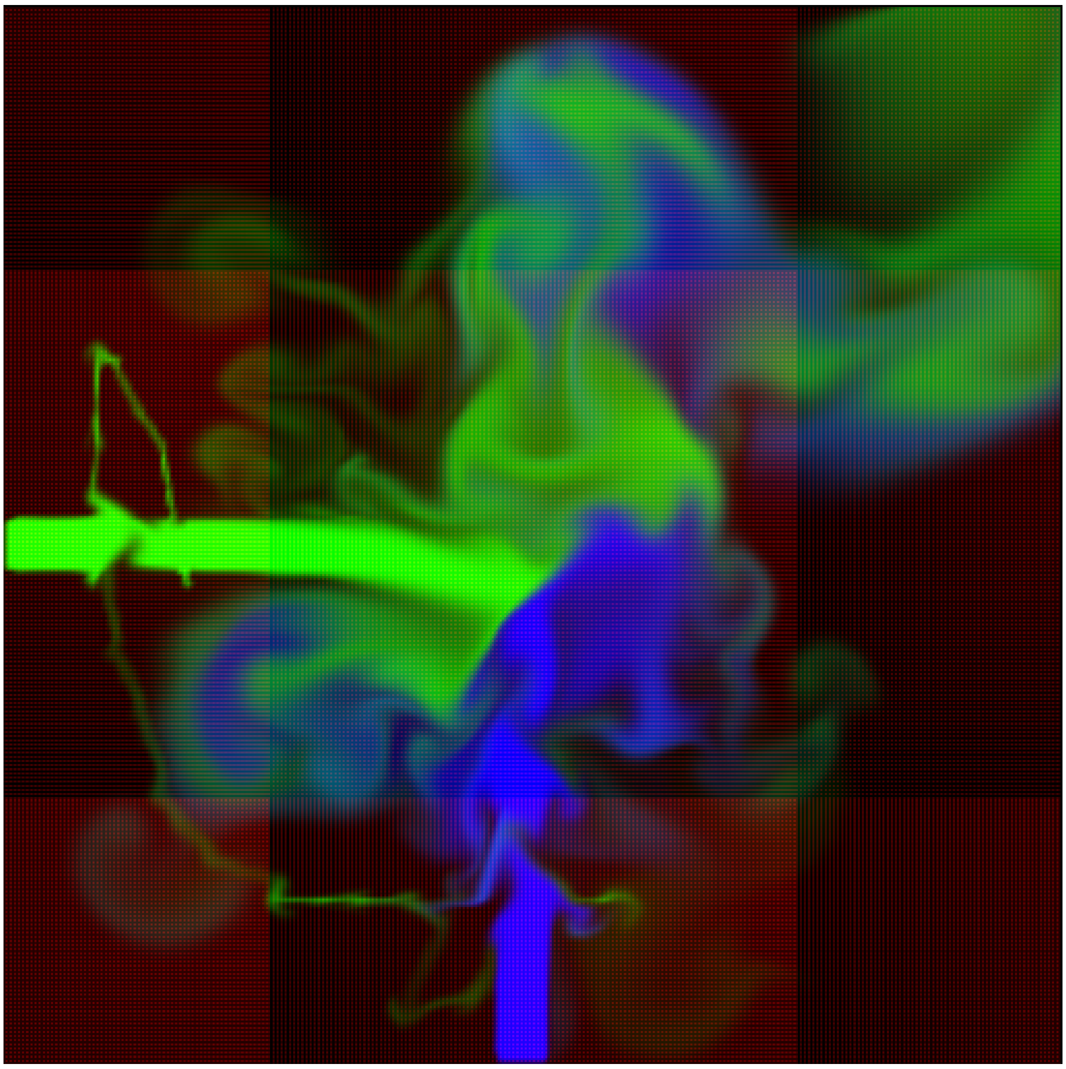


Figure 12.1: The Fluid example. User can mix the fluid using the mouse. Use the `v` key to switch between density field and velocity field visualization. Simulation is parallelized with MPI.

The Fluid application (see Figure 12.1) shows an example of module parallelized with MPI. Source code is installed in share/flowvr/examples/fluid. It fellows the same organization as the Primes example (see chapter 11) (file organization as well as compilation, instantiation and execution is similar).
The Fluid example is also one of the FlowVR test demo (launched executing `flowvr-demo-fluid`).
The Fluid example is built from 2 modules, a `fluid` module that perform a 2D fluid simulation and a `gldens` module that capture mouse events and display the fluid. `gldens` sends the captured mouse position to `fluid` that treats it as an obstacle for the fluid. `fluid` sends the fluid density grid to `gldens` for visualization.

12.1 Compilation

fellows the same organization as the Primes example (see chapter 11) (file organisation as well as compilation, instantiation and execution is similar).
Go to the `share/flowvr/examples/fluid` directory.
Compilation:

```
. /make-app.sh
```

A sequential version of the fluid simulation module is always built (the `fluid` application). The parallel version is only built if you have OpenMPI installed (the `fluidmpi` application)
Source the configuration script to set your environment variables :

```
source bin/fluid-config.sh
```

12.2 Instantiation and Execution

12.2.0.1 `Fluid` (sequential)

The sequential simulation is associated to a simple application network with direct 1-to-1 communications between both metamodules.
To start it locally on your machine:

```
python fluid.py && flowvr fluid
```

You can switch between the velocity or density filled visualization using the `v` key.
The fluid simulation is based on a grid discretization of the 2D space. The resolution of this grid is controlled by 2 parameters `nx` and `ny` (default $256x256$) that can be adjusted from the command line:

```
python fluid.py 128 128 && flowvr fluid
```

12.2.0.2 `FluidMPI` (parallel)

$\boxed{\text{A VOIR}}$ *adapter pour appy* $\boxed{\text{FIN}}$
The parallel MPI simulation is associated to a more advanced network. Because the fluid metamodule spawn several modules, collective communications (`Com1ToN` and `Com1ToN`) are used (share/flowvr/examples/fluid/include/fluid/components/fluidmpi.comp.h and share/flowvr/examples/fluid/src/fluidmpi.comp.cpp).
To start it locally on your machine you must rely on the `fluidmpi.csv` file (share/flowvr/examples/fluid/fluidmpi.csv) that requests to start 4 of the fluid module on the local host:

```
flowvr -x fluidmpi
```

If you have a quad-core processor you should notice some performance gain compared to the sequential version. If started with the `-t`, the daemon will display the frame rate of each module. You should here to notice a higher frame rate compared to starting only one instance of the fluid module. You can also modify the `fluidmpi.csv` to distribute the different instances on different machines.
Again the grid resolution can be changed:

```
flowvr -L -Pfluidmpi:nx=128,fluidmpi:ny=128 -x fluidmpi
```

Remember that when using the `-L` option only one instance of each module, even if parallelized, is started on the local machine.

12.3 The Simulation Module

12.3.1 Module and Metamodule Components

The module component (share/flowvr/examples/fluid/include/fluid/components/modulefluid.comp.h) is classical. We use the same module for the sequential code and the MPI one: the module ports are the same.
We have 2 different metamodule components. One for the sequential version using the `flowvr-run-ssh` launching command (share/flowvr/examples/fluid/include/fluid/components/metamodulefluid.comp.h). It inherits form the `MetaModuleFlowvrRunSSH` metamodule class. One for the MPI code using the `mpirun` launching command (share/flowvr/examples/fluid/include/fluid/components/metamodulefluidmpi.comp.h). It inherits from the `MetaModuleOpenMPI` metamodule code. Because `mpirun` launchers are not standard across different MPI implementations, a new metamodule should be developed if an MPI implementation other that OpenMPI is used.
These metamodules define the grid resolution parameters `nx` and `ny` parameters (default values $256x256$). In their constructor they add as arguments of the launching command line the value of these parameters using the instructions :

```
getRun()->addArg(TokenParameter(*this,"nx"));
getRun()->addArg(TokenParameter(*this,"ny"));
```

tcTokenParameter tells to get the value from the parameter `"nx"` hold by the component `this` (the current metamodule) when building the launching command line. You can see the final command line used to start the modules inspecting the generated `fluidmpi.run.xml` file.

12.3.2 The Code Module

The simulation code is distributed amongst the following files:

- share/flowvr/examples/fluid/include/fluid/Turbulent.h
- share/flowvr/examples/fluid/src/Turbulent.cpp
- share/flowvr/examples/fluid/src/TurbulentBase.cpp
- share/flowvr/examples/fluid/src/fluid.cpp

In the same code we mix the sequential and parallel version. At compilation if OpenMPI is detected, the `USE_MPI` variable is defined, activating all code sections delimited by `#ifdef USE_MPI`. Compilation produces 2 binaries, a sequential one (share/flowvr/examples/fluid/bin/fluidmpi), and a parallel MPI based one (share/flowvr/examples/fluid/bin/fluidmpi).
The fluid simulation is based on a 2D grid of cells. At each new iteration a new state is computed for each grid cell. This state depends on the state at the previous iteration of the considered cell and its four neighbors. The mouse location on the grid is also required as it is treated as obstacle for the fluid (received on the `positions` port). At each iteration the simulation sends the density data on port `density` and the velocity data on port `velocity`.
The fluid is an implementation of the algorithm introduced in the article "Stable Fluids" from J. Stam, SIGGRAPH 99.
The simulation is parallelized by splitting the grid cell into blocks distributed amongst the different MPI modules. For instance, a $2^k x 2^k$ grid is split into 4 $2^{(k-1)} x 2^{(k-1)}$ blocks on 4 modules, into 8 $2^{(k-2)} x 2^{(k-2)}$ blocks on 8 modules, etc. To perform the necessary computations the state of the cells on the block borders must be exchanged between neighbors at each iteration. These communications are managed by MPI (share/flowvr/examples/fluid/modules/src/TurbulentBase.cpp). They are transparent for FlowVR. The mouse position needs to be forwarded to each module. The FlowVR network takes care of it. After each iteration, the results (velocity and density data) must be sent to the visualization module for rendering. But the blocks containing the results are distributed amongst the different MPI module. Once again we delegate to FlowVR the responsibility of assembling the partial block results from each MPI module to forward to the visualization module a full grid of density data and a full grid of velocity data.
Each module sends on its `density` (resp. `velocity`) output port its density (resp. velocity) block result. To help FlowVR assemble correctly the different blocks into a full grid, the `fluid` module define the additional stamps (see section 7.4) `P` and `N`. `P` stores the coordinates of the first block cell in the full grid and `N` the block size :

```
flowvr::StampInfo StampP("P",flowvr::TypeArray::create(2,flowvr::TypeInt::
    create()));
flowvr::StampInfo StampN("N",flowvr::TypeArray::create(2,flowvr::TypeInt::
    create()));
```

These stamps values (stamps are additional data attached to messages that can be easily accessed) are used by the filters in charge of merging the block to forward to the visualization module a full grid.
The various instances of the `fluid` module have a different rank defined by MPI at starting time (the `MPI_Comm_rank`). They also have can retrieve the total number of running instances through `MPI_Comm_size`). Each module instance uses theses values to identify the block it has to process. The rank is also the index each module instance uses to build its FlowVR id (see subsection 20.7.6):

```
 MPI_Comm_size(MPI_COMM_WORLD,&nb_proc);
 MPI_Comm_rank(MPI_COMM_WORLD,&local_proc);
 flowvr::Parallel::init(local_proc, nb_proc);
```

The id is built from the module prefix names returned by the environment variable `FLOWVR_MODNAME` suffixed by the rank (`fluid/0` for rank 0 module). The variable is automatically set by FlowVR and forwarded to each module instance by `mpirun` (see `fluidmpi.run.xml`).

12.4 The visualization and Interaction Module

A single module takes care of rendering (using OpenGL and Glut), and mouse position capture (Glut based).

12.4.1 The Event Capture and Visualization Code

See share/flowvr/examples/fluid/src/gldens.cpp for the module code.
This module relies on Glut. It has a `positions` output port (the mouse position), a `velocity` and `density` input port where it receives the data to be displayed.
The loop structure is hidden behind the callback style imposed by Glut.
The module *gldens* reads the stamp *N* associated to each message received on the `velocity` and `density` ports to get the size of the received grid. It uses the class `GridInputPort` to define this user stamp (see share/flowvr/examples/fluid/src/gldens.cpp):

```
/// Input Port for 2D Grids with stamp "N" for size
class GridInputPort : public flowvr::InputPort
{
public:
  GridInputPort(const char* name="grid")
    : InputPort(name),
      stampN("N",flowvr::TypeArray::create(2,flowvr::TypeInt::create()))
  {
    stamps->add(&stampN);
  }
  flowvr::StampInfo stampN;
};
GridInputPort portDensity("density");
GridInputPort portVelocity("velocity");
```

Note that you are not required to define a new class to simply add a stamp. For example, the same module *gldens* defines a stamp *B* on the output *positions* port to send the state of the mouse's buttons using the following code:

```
OutputPort portPosition("position");
StampInfo stampButtons("B",TypeInt::create());
// .... //
  portPosition.stamps->add(&stampButtons);
```

To add a stamp in a new message, use the `write` method:

```
    MessageWrite mpos;
    // .... //
    mpos.stamps.write(stampButtons,mouse_down);
```

Similarly, a stamp is read using the `read` method. Note that this method returns `true` if the operation succeeded and `false` otherwise. In the case where the stamp is not present in the received message an error is returned when the module reads the stamp. Then you can either decide to ignore the error and use a default value instead, or stop the execution of the module and warn the user. Our *gldens* example module uses the default dimensions if the stamp is not present. The code to read the stamp *N* in this case is:

```
  int nx,ny;
  if (m.stamps.read(portDensity.stampN[0],nx) && m.stamps.read(portDensity.
      stampN[1],ny))
  {
    // Store the given dimension
  }
```

12.4.2 The Event Capture and Visualization Modules and Metamodules Components

The module and metamodule components are classical (use `flowvr-run-ssh` as launcher):

- share/flowvr/examples/fluid/include/fluid/components/modulegldens.comp.h
- share/flowvr/examples/fluid/include/fluid/components/metamodulegldens.comp.h

12.5 Component Assembly

Related files:

- share/flowvr/examples/fluid/include/fluid/components/fluid.comp.h
- share/flowvr/examples/fluid/src/fluid.comp.h
- share/flowvr/examples/fluid/include/fluid/components/fluidmpi.comp.h
- share/flowvr/examples/fluid/src/fluidmpi.comp.h

12.5.1 Sequential Fluid

Component assembly is trivial for the sequential version (simply direct links between the velocity, density and positions ports:

- share/flowvr/examples/fluid/include/fluid/components/fluid.comp.h
- share/flowvr/examples/fluid/src/fluid.comp.h

12.5.2 Parallel Fluid

The MPI version requires a more advanced network. Because the fluid module can run several instances, collective communications patterns must be used. Positions sent by the `gldens` module need to be broadcasted to the different instances of `fluid`. For that purpose we use a `Com1toN` broadcast pattern.

```
Com1toN<FilterRoutingNode> * c = addObject<Com1toN<FilterRoutingNode> >("
    com1toNPositions");
```

This broadcast tree (tree arity is controlled by the `TREE_ARITY` parameter) uses a `FilterRoutingNode` filter as intermediate node. This type of filter simply duplicates each message received on its input ports on all its outputs (the number of output ports is automatically defined according to the tree arity). A `Com1toN` pattern has simply two ports `in` and `out` that need to be connected to the source and destination metamodules. Notice that the pattern is connected to the metamodules and not the modules. Connection between the actual primitive components of `Com1toN` and the metamodules is performed automatically once the number of module instances to launch is known (the tree shape also depends on the number of modules):

```
Com1toN<FilterRoutingNode> * c = addObject<Com1toN<FilterRoutingNode> >("
    com1toNPositions");
setParameter<unsigned int>("TREE\_ARITY",2,"com1toNPositions");
link(metamodulegldens->getPort("positions"), c->getPort("in"));
link(c->getPort("out"), metamodulefluid->getPort("positions"));
```

The mapping of the filters is performed automatically according to the mapping of the connected modules. Map the modules on different hosts to see how it affect the the filter mapping (flowvr-glgraph (see chapter 13) can color the nodes according to the machine they are mapped on).
To assemble the velocity or density blocks, we use a `ComNto1` pattern with a specific filter `FIlterMerge2D` that has been designed to interpret the `P` and `N` stamp and reassemble the received

blocks into a new larger block (flowvrd/src/plugins/filters/flowvr.plugins.Merge2D.cpp). The filter has 2 input ports `in0` and `in1` and one output port `out`. But the `Com1toN` only work with filters exposing one `in` port and one `out` port (many FlowVR patterns work this way). For that purpose the merge component is actually built from the primitive filter `filtermerge2dprimitive` (include/flowvr/app/components/filtermerge2dprimitive.comp.h) and an encapsulating composite component `filtermerge2D` (include/flowvr/app/components/filtermerge2d.comp.h) that simply hides the filter interface by connecting the `in0` and `in1` ports to its `in` port (idem for the `out` ports). `filtermerge2d` creates an instance of `filtermerge2dprimitive` and connect it to its ports in the specific virtual method `execute`. It also checks that there is actually only 2 primitives components connected to its `in` port and forces each of theses primitive to be each one connected to a different input port of `filtermegre2dprimitive` (flowvr-app/src/filtermerge2d.comp.cpp). Because this merging filter has a fixed number of input the tree must have an arity of 2.

Part V

Utilities

Table of Contents

Chapter 13

Flowvr-glgraph: Interactive Graph Visualization

13.1 Introduction

FlowVR provides an interactive graph visualizer called *flowvr-glgraph (see Figure 13.1)* to help users develop and debug their applications. It computes a 2D graphical visualization displaying a graph of all the components of an application. It can generate graphs based on two types of files: *.net.xml* (network-based) and *.adl.out.xml* (hierarchical).

13.1.1 Network-based .net.xml

Vertices (nodes) correspond to modules, filters or synchronizers. Edges correspond to full connections (full line) or stamp connections (dashed line). The data is extracted from `.net.xml` files generated by `flowvr` (see section 5.6).

13.1.2 Hierarchical .adl.out.xml

Representation based on composites. A composite can contain other components. All components in a composite are inside it's box. Composites have input and output ports, which are located at the top and the bottom of the box, respectively. This will be further explained later (see section 13.3).
Be careful not to confuse the meaning of edges (arrows going from one component to another) in network-based graphs and hierarchical graphs. In the hierarchical graphs, edges merely express a "channel" through which messages can be sent to another component. In such graphs, connections and stamp connections are separate components.

13.2 Launching

To launch flowvr-glgraph:

```
flowvr-glgraph [file.net.xml] [file.gltrace.xml]
```

13.3 Component representation

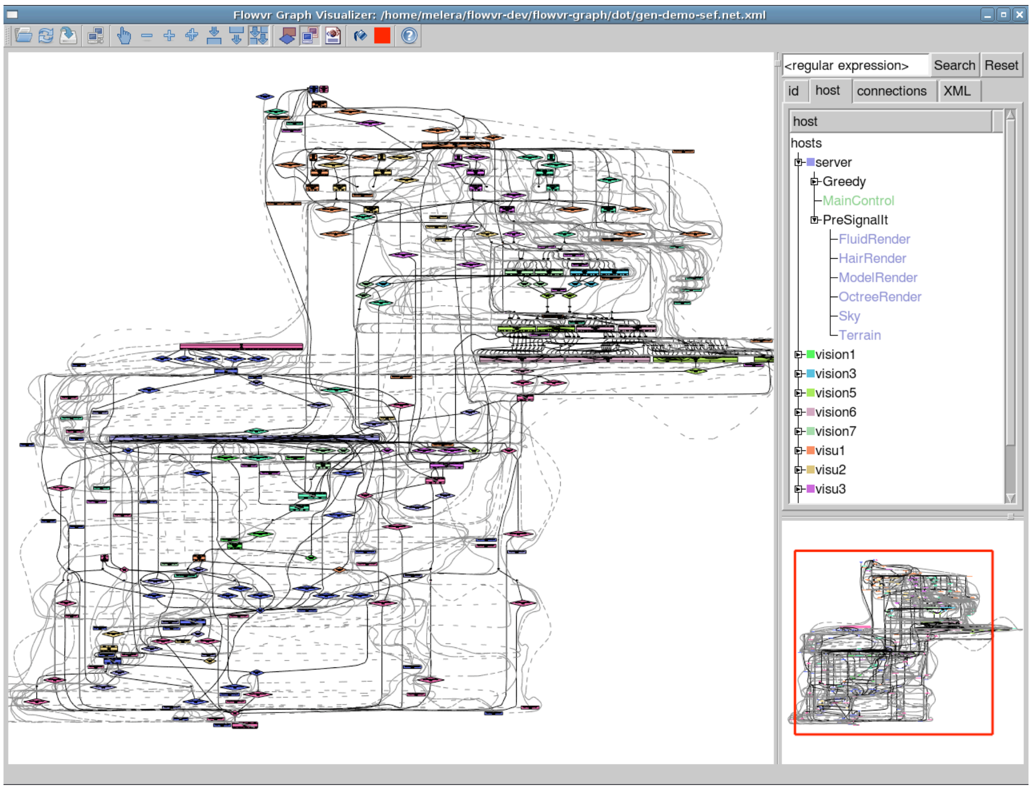


Figure 13.1: The `flowvr-glgraph` interface, network-based graph.

13.3.1 .net.xml

Vertices (nodes) correspond to modules, filters or synchronizers. Edges correspond to full connections (full line) or stamps connections (dashed line). Modules are green, filters are blue, synchronizers are red.

13.3.2 .adl.out.xml

13.3.2.0.1 Composite Each composite component is represented by a "cluster". Clusters have a grey outline and a label in the top/middle. The label follows this pattern: "parent/current". They also have input ports at the top (green), and output ports at the bottom(light blue). The mini-viewer on the right shows the cluster outline.

13.3.2.0.2 Composite ports Input ports: green rectangles. Output ports: light blue rectangles.

13.3.2.0.3 Module (primitive module) Modules are purple. They are one block vertically separated in 4 sub-blocks. The first block contains the input ports. The second block, the name. The third block contains the host, plus where that information came from (if available). The fourth and final block contains the output ports.

13.3.2.0.4 Filter Filters are orange diamonds.

13.3.2.0.5 Connection Connections are blue rectangles with a full black outline.

13.3.2.0.6 Stamp connection Stamp connections are blue rectangles with a dashed outline.

13.4 Shared functionalities

This is what you can do with both formats:

- Load a file, generate a graph
- Move and zoom
- Name, host and XML treeviews
- Snap to component/cluster from treeview
- Search for component/cluster, snap to result in viewport
- Export viewport as an image

13.5 The toolbar

From left to right:

📂	Open a FlowVR XML file. It goes through a layout computation that could take a few seconds with large graphs. You can also load a file from command prompt by running flowvr-glgraph and append the file name as an argument.
🔄	Reload the current file.
📥	Export the viewport in a graphical file format. Available formats are: jpg, png, eps, ps, ppm, bmp and fig.
🖧	(.net.xml) If this button is toggled, only the connections between two different hosts are made visible.
👆	(.net.xml) By toggling this button, you enter in selection mode.
−	(.net.xml) Decrement visualization depth.
+	(.net.xml) Increment visualization depth.
✚	(.net.xml) Set visualization depth to the max depth.
⬇	(.net.xml) Visualize only the upstream elements.
⬇	(.net.xml) Visualize only the downstream elements.
⇅	(.net.xml) Visualize both upstream and downstream elements.
🖥	(.net.xml) Use one color for one node type (green for modules, blue for filters, red for synchronizers and gray for routing nodes)
◆	(.net.xml) Use one color for one host.
📄	(.net.xml) Use the colors defined in the file.
🖌	(.net.xml) Replace the clicked color by the current color.
🟥	(.net.xml) Change the current color.
❓	Open the help dialog.

13.6 The view

13.6.1 Normal Mode

All the graph nodes are visible. Hold left mouse button to move around the graph. A right-click on a node or a connection magnifies it (.net only).

13.6.2 Selection Mode (.net.xml)

The visualization is depth-limited. i.e. only the nodes not farther from the selected element than the depth limit will be set visible. A click on a node or a connection will select it.

13.7 The lists

Four trees are filled during loading.
id: nodes id.
host: nodes id, grouped by host.
connections: connections id. (.net only)
XML: the whole xml tree.

13.7.1 Normal Mode

A click on a list element will center the view on it.

13.7.2 Selection Mode (.net.xml)

In the selection mode, the clicked element is centered and selected. If it is a branch, all its leaves will be selected.

13.8 Searching by Regular Expressions

This text field allows to make advanced research in the nodes id list. The syntax used is fully described on the QT's web site: `http://doc.trolltech.com/3.3/qregexp.html`. The 'Search' button makes the matching nodes visible and hide the rest. The 'Reset' button allows to come back to the normal visualization.

13.9 Clustered Layout (.net.xml)

Use the `c` key to toogle the clustered layout that improves the layout by taking benefit of the component hierarchies. This layout is very helpfull, but crashes may occur (layout algorithm not totally stable).

Chapter 14

Trace Capture and Visualization

Capturing the trace of an execution and visualize can greatly help to understand the behavior of a distributed application. For that purpose FlowVR comes with tools to record particular events during an execution, and to analyze and visualize them off-line.

14.1 Trace Capture and Code Instrumentation

The application should be instrumented (include specific function calls into the code) to record *events*. An event corresponds to an *id*, an *occurrence time*, and a *data*. We call a *trace* the list of events related to the same *id*.
There are two kinds of events:

- *FlowVR pre-defined events* that require no coding effort (just enable/disable the related trace capture).
- *User defined events* related to function calls included by the user. This enables the user to customize event capture according to its needs.

Recording these events takes place through a specific filter called a *logger*. At least one *logger* is required by host that want to record traces. Care has been taken to minimize the overhead related to event recording (pre-allocation of buffers, reduced number locks). When trace capture is not activated (*logger* not loaded), calls to event capture functions is of negligible cost. It is therefore not necessary to remove these calls from the code.
Each *logger* as one output port `log` where it periodically sends the traces it has recorded. For recording these traces into files, the `log` port of each logger is connected to a specific metamodule `fwrite`

14.1.1 FlowVR Defined Events

FlowVR pre-defines traces to capture for each module :

- when the `wait` starts (the `waitBegin` trace)
- when the `wait` ends (the `waitEnd` trace)
- when the put on the `endIt` port occurs (the `endIt` trace)
- when the get on the `beginIt` port occurs (the `beginIt` trace)
- when the `get` occurs for each input port (the trace has the name of the input port)

- for each output port when the `put` occurs (the trace has the name of the output port)

The data associated with each of these traces is the current module iteration number.

14.1.2 User Defined Events

There are 2 things to do to enable a module to record a new trace:

- In Flowvrapp, declare that you want to add a user trace (NB that the type of the trace is currently not used):

```
mymodule.traces["myTrace"] = int
```

- Then in your source code, declare a new vector of trace object, and pass it to the method initModule() (as the optional second argument). Each trace has a name, and declare the type of the data to be recorded (use a c++ template):

```
//create a vector of Traces:
std::vector <flowvr::Trace*> myPersonnalTraces;
...
//declare your Traces
TypedTrace<DataTypeforTrace> myTrace("myTrace");
...
//add them to the vector.
myPersonnalTraces.push_back(myTrace);
...
//and finally pass this vector to the initModule Method (which registers
     and initializes the module to the FlowVR daemon)
if (!(pFlowVRModule = flowvr::initModule(ports, myPersonnalTraces)))
{
return -1;
}
```

- Now you can call the function 'write' to record an event:

```
myTrace.write(data);
```

In share/flowvr/examples/primes, the *visu* module declares a user trace that emits an event each time a message from the *capture* module is not of size zero, which means the user is pressing one or several keys (the number of keys simultaneously pressed is taken as the message data, of type `int`). The user trace is defined and used as below (see share/flowvr/examples/primes/modules/src/visu.cpp) :
For advanced users, refer to the flowvr-base/include/trace.h file that defines the API for the FlowVR traces.

14.1.3 Traces for Filters and Synchronizers

Recording Traces within Filters and Synchronizers is similar to recording traces within modules.
By default each filter and each synchronizer declare a trace for each of its ports. These traces are automatically included in the application network when generated.
User defined events related to function calls can also be included in filters and synchronizers for specific needs.
The procedure is similar to adding *User defined traces* to a module.

- as for module, the trace should be declared in Flowvrapp

- Then in the source code of your filter, the trace should simply be added to the trace vector *outputTraces* (which is already declared in object base class).

```
//declare your Traces
TypedTrace<DataTypeforTrace> myTrace("myTrace");
...
//add them to the vector.
outputTraces.push_back(myTrace);
...
```

- finally, the calling of the write method is similar to the earlier one.

Filter base class declares 2 custom traces iddleBegin and iddleEnd, that are for instance called in filter Merge. Refer to flowvrd/src/plugins/filters/flowvr.plugins.Merge.cpp and flowvrd/src/plugins/sys/flowvr.plugins.Filter.cpp

14.1.4 Launching Trace Capture

14.1.4.1 Generating the network

Tracing an application implies to add Loggers(plugin) and FWrite modules, to write events informations in a file, and to generate specific files. This is done in Flowvrapp with

```
import traces
traces.add_traces(traces.select_primitives(), 'appName')
```

The `traces.select_primitives()` selects the primitives to trace. By default, all the primitives declared so far are traced.
The appName argument sets the prefix for sevral files:

- First, two files are created, containing commands to start and stop logging events : [appName].prolog.xml (for start commands) and [appName].epilog.xml (for stop commands).
- Then, a result file is created for each host to log. By default, those files are written in `/tmp/trace_log_[hostname]`. You can override the files' prefix using an argument to `add_traces`.
- We advise to set a NFS directory on your cluster, to ease trace visualization. Indeed, the 'flowvr-gltrace' utility needs to access to all of those files to construct the trace graph. Else, you'll have to gather them in a local directory to be able to launch trace visualization.
- finally, a description file is generated : [appName].gltrace.xml that stores information about the events traced (name, color, links...)

14.1.4.2 Command Details

Once you have launched your application, `flowvr` accepts commands `trace` / `notrace` in the telnet console, to respectivly start/stop tracing the application. Events are logged between this two calls.

```
 ...
 Processing STDIN...
 >trace
```

wait the time you need, then :

```
 >notrace
```

14.2 Trace Visualization

FlowVR comes with the `flowvr-gltrace` tool to analyze and visualize traces.
Visualizing a trace can be as simple as:

```
flowvr-gltrace [your_app].gltrace.xml
```

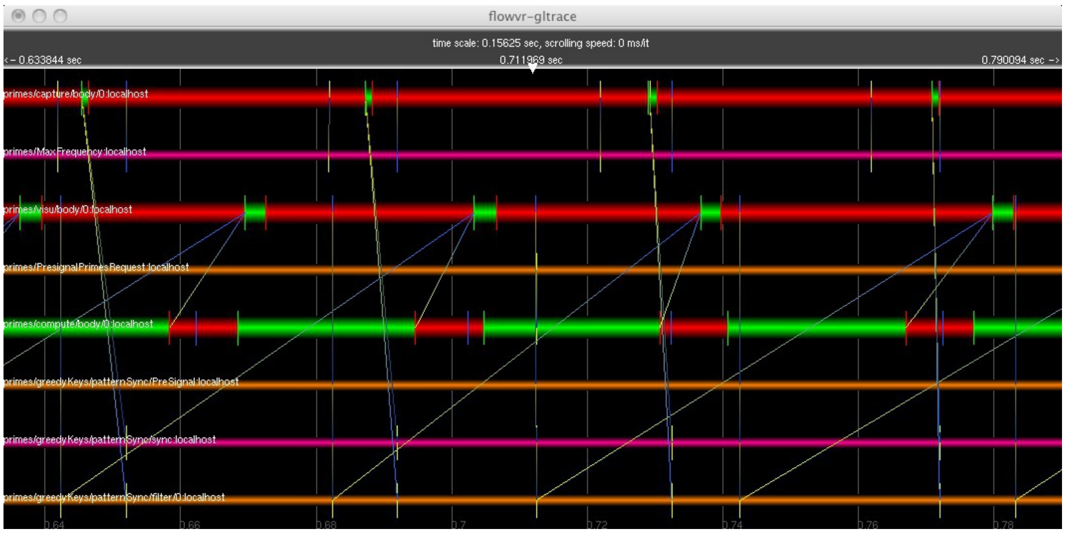

flowvr-gltrace
time scale: 0.15625 sec, scrolling speed: 0 ms/it
<- 0.633844 sec
0.711969 sec
0.790094 sec ->
primes/capture/body/0:localhost
primes/MaxFrequency:localhost
primes/visu/body/0:localhost
primes/PresignalPrimesRequest:localhost
primes/compute/body/0:localhost
primes/greedyKeys/patternSync/PreSignal:localhost
primes/greedyKeys/patternSync/sync:localhost
primes/greedyKeys/patternSync/filter/0:localhost
0.64
0.66
0.68
0.7
0.72
0.74
0.76
0.78

Figure 14.1: Visualization of a trace generated from the Primes example. Modules alternate between red (waiting) to green (computing). We can see for instance that the visualization module is activated once it gets a message from the `compute` module and from the `capture` (indirectly through the `patternsync/filter` as we are using a greedy pattern here.)

14.2.1 User Interface commands

To navigate through the trace the following functions are available:

- key `q` or `Q` : quit `flowvr-gltrace`
- key + and – or mouse sroll : horizontal zoom in and out the trace
- key `ctrl` + mouse scroll : vertical zoom in and out the trace
- Left mouse click: grab the image and scroll it on the left or on the right
- key `ctrl` + Left mouse click : grab the image and scroll the image vertically
- Right mouse click: draw an isochronous line
- key `4` and `6` : decrease/increase the speed of automatic scrolling (start at 0)
- key `3` and `9` : decrease/increase the ratio of speed increase/decrease
- key `5` : stop automatic scrolling

14.2.2 Graphical Representation description

Events are ordered on an horizontal time line.
Each line corresponds to a different object (module, filter, synchronizer). The name of this object is displayed on the left of each line. Objects are differentiated by the color and the size of the line.

- Modules are represented by thick lines, being either *green* or *red* :
  - when `waitBegin` event occurs the line turns *red* (an inactive period starts);
  - when `waitEnd` event occurs the line turns *green* (an active period starts).
- Filters are represented by thin lines, being either *orange* or *blue* :
  - when `iddleBegin` event occurs the line turns *blue* (an iddle period starts - only for specific filter such as merge or signalAnd, that are waiting for several messages on their input ports)
  - when `iddleEnd` event occurs the line turns *orange*
- Synchronizers are represented by thin *pink* lines

Each time an event occurs (put, get, or any other event) a short colored vertical line is displayed on the line where it took place:

- `beginIt`: *white* line
- `endIt`: *blue* line
- `order`: *purple* line (for filters/synchronizer)
- `put` or `get` on other ports: *yellow* line
- `user defined events` : *yellow* line

Message transmissions are displayed as lines going from one horizontal line to the other

- `full message` : line going from *red* (message emission) to *blue* (message reception)
- `stamp message` : line going from *green* (message emission) to *pink* (message reception)

14.2.3 Customizing Graphical Representation

The graphical representation can be customized by editing and modifying the `.gltrace.xml` file. The main sections for customization are:

1. `<objectlist>`: the objects (modules, filters, synchronizers)
2. `<eventlist>`: the events traced
3. `<linklist>`: the links between events that `flowvr-gltrace` can compute (for instance a message transmission between objects)

In each of these section you have the possibility to enable/disable a display (attribute `active`) for some data, to change the text displayed (attribute `text`) or a color (attribute `color`).
One can also customize the way the `.gltrace.xml` description file is generated. Refer to flowvr-app/src/applicationTracer.cpp and flowvr/include/flowvr/app/core/applicationTracer.h

14.2.4 Visualization in flowvr-glgraph

Traces can also be visualized in `flowvr-glgraph`, when the `.gltrace.xml` is passed as the second argument. The flowvr-glgraph has a current time, and a slider can be used to set it.
Currently active modules (not waiting) are displayed in bold, and messages are shown as red dots ont the links. The traces can be replayed in real time with the "play" button. The "play events" button does the same, but slows down the events as much as necessary so that all messages are visible (Fig 14.2).

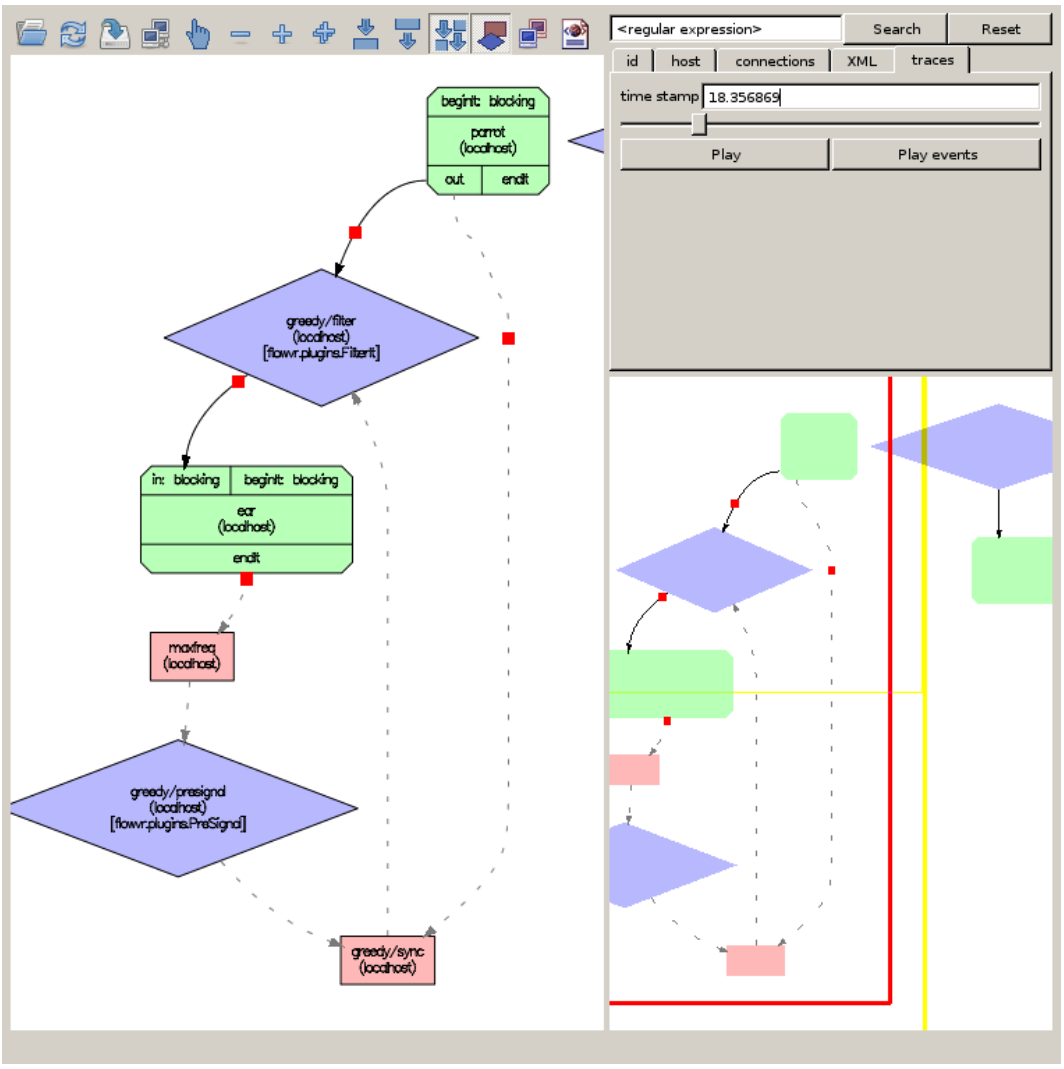


Figure 14.2: Visualization of a trace generated from one of the filter examples. Messages are shown in red.

Chapter 15

FlowVR Template Library

Related files

- *urlflowvrinclude/ftl/: contain all ftl header files.*

15.1 Overview

The FlowVR Template Library provides some generic representations for frequently used objects: vectors, matrices, quaternions, etc. as well as a utility tool to parse the command line arguments.

15.2 Vectors

```
#include <ftl/mat.h>
```

flowvr-ftl/include/ftl/vec.h
The class `Vec<N, Type>` represents fixed-size vectors of `N` elements of type `Type`. Elements are stored in a compact contiguous manner: a FTL vector can thus be casted from a standard C array. Standard mathematic operations are available directly by redifinition of the corresponding operators. Different predefined vector types are available as aliases: `VecNT` where `N` is 2, 3 or 4 and `T` is b (unsigned char), i (int), f (float), d(double).

15.3 Matrices

```
#include <ftl/mat.h>
```

The class `Mat<L, C, Type>` represents fixed-size matrices with `L` lines and `C` columns, each element being of type `Type`. Elements are stored in a single buffer in line order. Standard mathematic operations are available through redefined operators.

15.4 Quaternions

```
#include <ftl/quat.h>
```

The class `Quat` represents a quaternion. It can be constructed from different rotation representation and can be converted to and from rotation matrices.

15.5 Command Line Parsing

```
#include <ftl/cmdline.h>
```

To ease the parsing of command line options (a very frequent task for modules), FlowVR proposes an easy command line parsing API.

15.5.1 Declaring Options

Declaring an option simply consists in creating a variable of type:

- `Option<Type>` for an option with an argument (potentially optional) of type `Type`.
- `FlagOption` for an option with no additional argument (a flag).

The option's short name (a letter) indicates which short command ('-' followed by this letter) triggers the option. The option can also be triggered with a long command ('–' followed by the long name) if a non `NULL` long name is given. The description corresponds to the help given by the help option ('-h', '–help').

15.5.2 Parsing the Command Line

Parsing the command line first requires the creation of an object of type `CmdLine`. The description then given corresponds to the usage line given by the help option. Parsing the option is then done by calling the fonction `parse`.

15.5.3 Retrieving Values

Testing if an option is present is simply done by testing if its `count` attribute is non nul. An option is automatically casted into its value (a boolean for a flag).

Chapter 16

Other Tools

16.1 `flowvr-graph`: static network images

To generate graphical representations of FlowVR networks `flowvr-graph` uses two external tools: `xsltproc` (a *XSLT* parser) and `dot` (*Graphviz*).
`xsltproc` is the *XSLT* parser associated with `libxslt`. On Debian it can be installed using `apt-get install libxslt1-dev`
`dot` is a graph layout tool from the *Graphviz* package available at `http://www.graphviz.org/`. On Debian you can use `apt-get install graphviz` to install it.
Once these tools are installed you can use the provided script `flowvr-graph`. It takes as input the `.net.xml` file. If no argument is given it will print a `.dot` graph description. This graph can be customized and then displayed using `dot`. You can also directly specify arguments to `flowvr-graph` which will invoke `dot` using these arguments.
For example to compute the image of the Primes application:

```
cd share/flowvr/examples/fluid
# Compile the application
make-app.sh
# Process he application:
flowvr Fluid
# and this command will create a graphical representation
flowvr-graph -Tps -o ./primes.net.ps < ./primes.net.xml
```

You can generate different types of files (PS, PNG, FIG) using the `-T` argument. You can also pass some graph attributes using `-G` (such as `-Grankdir=LR` to have an horizontal layout.
If you have chosen to install it, you can also use the graph visualizer flowvr-glgraph (see chapter 13).

16.2 `flowvr-shmdump`: Dump Shared Memory Content

`flowvr-shmdump` is a useful utility to know the state of the shared memory created by a FlowVR daemon. It takes two arguments : the SHM ID (0 by defaults), to specify which shared memory area to explore if several FlowVR daemon are launched on the node ; and a position given in hexadecimal, to have a look on the data stored at this position.
There is the result of the command `flowvr-shmdump`, when there is just the daemon launched and no application.

```
flowvr-shmdump 0
FlowVR Shared Memory Dump version CVS
```

```
Opened shared memory area with ID=51337: internal id=3014667 size=10000000
    attached 1 time(s)
Mapped shared memory area with ID=51337 at 0x402a0000
Shared memory area mapped at 0x402a0000
Daemon header at 0x40
Head bloc at 0x1e0
  Buffer pos=0x1f0 length=0x18 nbref=1
  Buffer pos=0x208 length=0x10 nbref=1
Bloc pos=0x218 length=0x989468 prevDP=-0x38 nextDP=0x989468
Free memory 9999456, found 9999456 (99%)
```

This utility can be very useful for debugging some memory lack or to find a better size for the shared memory area.

16.3 `flowvr-run-ssh`: a Simple Module Launcher

A VOIR *may be move it to a upper section as it became a central tool for flowvr* FIN
`flowvr-run-ssh` (bin/flowvr-run-ssh) is a useful tool for launching FlowVR modules through ssh. It alleviates the difficulty of handling environnment variables with ssh and automatically propagates the FlowVR variables (FLOWVR_DAEMON, FLOWVR_PARENT, FLOWVR_MODNAME).
`flowvr-run-ssh` is intentionally kept simple. If it does not fit your needs look at more advanced launchers like TakTuk or mpirun.
The `MetaModuleFlowvrRunSSH` metamodule class (include/flowvr/app/components/metamoduleflowvr-run-ssh.comp.h) eases the use of `flowvr-run-ssh`. You just need to provide the executable name of you module and customize some options if required (include/flowvr/app/core/run.h). You will be able to start several instances of your module on distant machines.

```
flowvr-run-ssh [-v] [-d path] [-m] [-l login] [-e VAR VALUE] [-x VAR] [-s]
    [-p] hostlist command
```

- `-x VAR` : propagates the variable VAR through ssh.
- `-e VAR VALUE` : sets the variable VAR to VALUE.
- `-s` : sequential mode (does not set FLOWVR_RANK nor FLOWVR_NBPROC). This is the default behaviour if only one host is used.
- `-p` : parallel mode (sets FLOWVR_RANK and FLOWVR_NBPROC). This is the default behaviour if several hosts ares used.
- `-l login` : runs ssh with a different login.
- `-path path` : path is changed to `path` before launching command. By default the program is launched from the Home directory of the user.
- `-v` : Verbose mode.
- `-m` : multi-platform mode. All occurences in the path of the value contained in $PLATFORM will be set to the correspondant value of PLATFORM on the distant host.

`flowvr-run-ssh` needs a valid bash account with flowvr environment variables set at login.

16.4 `flowvr-fread` and `flowvr-fwrite` Modules: Save/Replay Messages

$\boxed{\text{A VOIR}}$ *a reviser: par antoine* $\boxed{\text{FIN}}$
FlowVR provides modules for writing messages to a file and for reading them.

- flowvr-fwrite: writes all messages on its input port to a file.
- flowvr-fread: reads messages from a message and sends them on its output port.

16.5 `flowvr-joypad` Module

$\boxed{\text{A VOIR}}$ *a reviser: par antoine* $\boxed{\text{FIN}}$
FlowVR also provides a module for retrieving events from a joypad. It opens two ports per button and axes: one for the event itself and one for the accumulated value on this axe (or state of the button). Note that the desc files should be modified according to your device for the number of buttons and axes.

Chapter 17

Developping Tools

17.1 File Searching in Path

Related files

- *include/flowvr/utils/filepath.h*

A very frequent task performed by FlowVR programs and modules is to search for a file given a set of directory locations. The class `FilePath` implements this functionnality. A path is considered as a string composed of directory locations separated by `':'` (as for the standard environment variable `$PATH`). Files are searched as the concatenation of the `filename` given and one of the directory location of the path. This class can be used in two different ways.

17.1.1 Basic search

```
bool FilePath::findFileIn(std::string& filename, const std::string& path);
```

This static function looks for file `filename` in the path given by the parameter `path`. If the file is found with one of the directory location, the function returns `true` and sets `filename` to the correct path for the file (mainly the concatenation of this directory and the file name). Otherwise it returns `false`.
This function simply tests the concatenation of each directory location (in the order of the path) and the `filename`. It does not test in the current directory and does not take a special care for files starting by `'/'`, `'./'` or `'../'`

17.1.2 Advanced search

To handle those cases correctly, we need to first build an object `FilePath` with the different paths to search before searching for the file.

17.1.2.1 Construction of a FilePath

```
FilePath::FilePath(const char *envVar = "FLOWVR_DATA_PATH");
```

This function builds a `FilePath` object by loading its path from an environment variable `envVar`. If no parameter is passed the variable `FLOWVR_DATA_PATH` is then used. For not loading the path from an environment variable, one must set `envVar` to `NULL`.

17.1.2.2 Addition of a Path

```
void FilePath::addPath(const std::string& path);
```

Adds a path for the search.

17.1.2.3 File Search

```
bool FilePath::findFile(std::string& filename);
```

File `filename` is searched as follows:

1. Using the specified file name in current directory.
2. In the directory path specified using addPath method (in reverse order: last added first searched).
3. In the directory path specified using an environment variable.

For file name starting with `'/'`, `'./'` or `'../'` only the first step is applied. The results of the search is identical to the basic file search method.

17.2 Stream Buffer accessors

Related files

- *include/flowvr/utils/streambuf.h*
- *include/flowvr/utils/iostream.h*

As it is, it may be tedious to access a buffer to write/read any data structure which is more complex than a mere array. Indeed, doing so involve some type-casting while keeping track of the offset used to write the next piece of data, both prone to careless mistakes. However, a few helper classes are supplied to help the user perform such accesses.

17.2.1 `std::streambuf`

This is the lowest-level of those helper classes. This abstract class of the C++ STL (Standard Template Library) gives a manner of accessing data coming from/to some *controlled input/output sequence*. It care care of both the buffering and the actual data accessing.
Thus the `flowvr::utils::streambuf` class implements `std::streambuf` to help accesssing a `flowvr::Buffer` however it's segmented, and with no redundant buffering. It enables to do so with a shorter code while reducing the risk of careless mistakes.

17.2.2 `std::stream`, `std::istream`, `std::ostream`

Atop of that, you can directly use the streams from the STL to make profit of their error handling and the overloaded extraction operators.
To avoid having to create the streambuf yourself, you can use the `flowvr::utils::bufferstream`, `flowvr::utils::ibufferstream` and `flowvr::utils::obufferstream` classes that inherit the standard streams in the fashion of `std::filestream` and `std::stringstream`.
Keep in mind that those streams serialize the data in ASCII text instead of simply copying memory.

17.2.3 Streambuf Usage examples

Streams from the STL serialize the data in ASCII, but using `streambuf` directly, you can also copy binary data.

```
flowvr::utils::streambuf inb( message.data );       // wrap an incoming data
std::ofstream out( "/tmp/some_file" );              // open some output file
out << & inb;                                       // dump all data
```

```
std::ifstream in( "/tmp/some_file" );               // open some input file
size_t size = in.rdbuf()->pubseekoff( 0, in.end ); // get size
in.rdbuf()->pubseekpos( 0 );                        // back to the begining
flowvr::BufferWrite data = module->alloc( size );   // pre-alloc output buffer
flowvr::utils::streambuf outb( data );              // wrap output buffer
in >> & outb;                                       // dump all data
flowvr::BufferWrite full, part;
full = outb.gBufferWrite();       // 'get-buffer', full internal buffer
part = outb.pBufferWrite();       // 'put-buffer', written bytes only
```

17.2.4 streams Usage examples

```
// per-alloc reasonnable amount of memory. Can be insufficient
flowvr::BufferWrite data = module->alloc( size );
flowvr::utils::obufferstream out( data );
out << some_int << '\n';
out << some_float << '\n';
out << some_object << '\n';
out << some_other_object;
// some allocation may have been necessary, get modified buffer
data = out.rdbuf()->pBufferWrite();
```

```
flowvr::utils::ibufferstream in( message.data );
in >> some_int;
in >> some_float;
in >> some_object;
in >> some_other_object;
```

Chapter 18

PortUtils – supporting tools for module creation

A VOIR *outdated, has to be brought up to work with flowvr-appy* FIN

This part describes an extension to Flowvrapp and the module code on top of FlowVR itself, called 'PortUtils'. It is part of the FlowVR library and may be used in conjunction with existing code. There is no 'binary' decision of whether to use PortUtils or not, but rather a new set of tools that you can use to address specific issues when creating applications with FlowVR.

Throughout the document, it is assumed that the user has good understanding of FlowVR, Flowvrapp and is thinking in a modular way, maybe even a bit object oriented. We will not spend too much time to explain C++ or template tricks or how dynamic library loading works. As well, there will be no discussion of `cmake` or make files and linker problems. So all this should be familiar, too.

18.1 Motivation

Writing a FlowVR application consists of a many-stage assembly procedure where the authors of an application already have some module or write new module code to define the final functionality. The semantics of the application are defined by the flow of data between the modules. These elements consist of connected filters, synchronizers or and a set of modules. One of the main goals of FlowVR is to keep the portion of code that has to be written to integrate already existing code into the middleware quite minimal. Typically, input- and output-ports have to be defined, and they have to be given to a module interface that thereafter is able to run the module update loop using the `wait()` (for message), `get()` (a message) and `put` (a message) cycle. From a module point of view, that is about it. Programmers have the freedom to choose more on the internal structure of the module binary, for example if there is more than one named module in the same binary or not, or whether the binary is a multi-threaded application itself. Whatever the specific choices here are, for FlowVR there is only the module interface and its ports that are given by the binary after launching it.

The network of modules, filters and synchronizers is created using the Flowvrapp layer. In this layer, programmers produce and use a set of C++ objects where each one can create a sub-graph of modules that is connected to other sub-graphs of the application network. For the scope discussed here, it is not interesting to reason about the network itself, but more on the interface to integrate a given module. For that, we start by discussing the 'standard' way of integrating a module first. After that, we will discuss some of the shortcomings of the approach, and where errors can occur. The PortUtils try to tackle these shortcomings, so we give an outline of the possibilities that arise when using PortUtils at the end of this section.

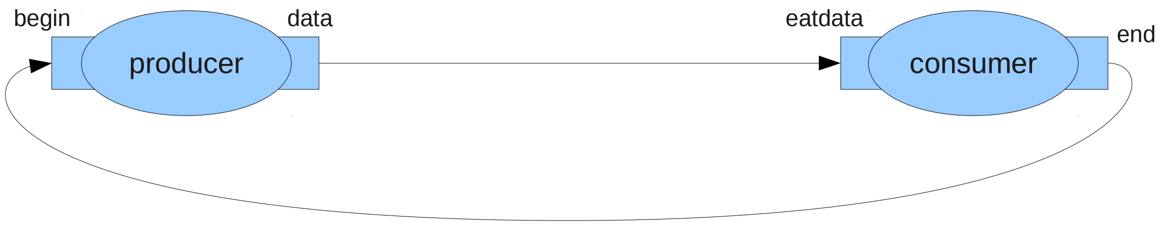


Figure 18.1: Conceptual network for the simple example.

18.2 Integration of custom code with FlowVR

This section will, very briefly, discuss the steps that have to be taken to integrate custom code into FlowVR by defining a module. It will not discuss what to do if you want to filter anything (see FlowVR documentation for that). We will use a very small example that might not be the standard use case when thinking about using FlowVR, as it is written for larger contexts.

18.2.1 An example problem.

For our discussion, we will use a simple producer–consumer problem. A number of modules will produce some data in parallel, the data will be transmitted to a number of consumers where it is used. For the sake of simplicity, we assume that the data is processed in a push-pipeline that is running at the speed of the consumer. In this discussion, we will limit the number of producers and consumers to 1, resulting in a 1:1 pipeline[1]. So the resulting conceptual network (see Figure 18.1) is simple.

18.2.2 Defining module level.

We will take care of the definition of this graph later, first we go and define discuss the module structure: we have two modules here, and we call them **P**(roducer) and **C**(onsumer). Before we dive into the code let us think a bit about the relation of the two.
The producer has 1 output port while the consumer has 1 input port[2]. P creates a blob of memory with an internal structure not important to FlowVR (it is just the size that counts), and C has to know that internal structure well in order to process on it[3]. The modules itself define *names* they give to the ports as well as a *direction* (in or out), and this information will be passed to the FlowVR-daemon upon a call to `flowvr::initModule()`.
The code of P looks probably a bit like this[4].

```
OutputPort p("data");
vector<Port*> v;
v.push_back(&p);
[...]
module = initModule( v );
// initialize the internal state machine here...
[...]
// now the flowvr-loop
while( module->wait() )
{
```

[1]Accustomed readers might see the relation to the TicTac example shipped with FlowVR.
[2]Let's not forget, but not discuss, that both have `beginIt` and `endIt` given by the system.
[3]There might be cases, where a module is only interested in the 'outer shape' of a message that is defined by FlowVR, e.g. its size and so on. But typically, this is an aspect that is more often covered by a filter, so we omit that case here.
[4]For the sake of simplicity, we omit namespace qualifiers and other syntactic burdens needed to compile the code.

```
   // call inner state: produce
   Data date = produce();
   size_t n;
   char *pdate = serialize(date, n); // make it a memory blob
                                      // write length to n
   // wrap the produced blob into a message
   MessageWrite mw;
   // create new blob of data with size n
   mw.data = flowvr->alloc(n);
   memcpy( m.data.writeAccess(), pdate, n ); // copy to message
   // now put
   module->put(&p, m);
} // while
```

There are some things to note on this example already here.

- the name of the port ('data') is 'hard-coded' as a string into the binary.
- the ports are given in a vector of ports to FlowVR, so they have to be explicitly added to that vector after creation.
- the data object produced has either to have a serial structure or has to be serialized explicitly[5].
- the port to use has to be known by pointer in the call to `put()`.
- the order of calls to `put()` is coded into the `while` loop.
- the code for creating a message is always the same: get a container, re-size the container, fill the container, put it to FlowVR and forget it.

The code of C looks similar in a way.

```
InputPort p("eatdata");
vector<Port*> v;
v.push_back(&p);
[...]
module = initModule( v );
// initialize internal state machine
[...]
while( module->wait() )
{
   // get a container
   Message m;
   // fill it
   module->get(&p, m);
   // consume...
   Data date = deserialize(m.data.readAccess(), m.data.getSize());
   consume(date);
}
[...]
```

Again there are things to note.

[5]For this document, we use the term *serializing*, which sometimes is named *marshalling* or has other names. Simply spoken: serialization produces a flat representation (e.g. a string) from a possible graph-like representation (e.g., an object graph).

- the name of the input port is again 'hardcoded' into the binary. It even differs in name from the one defined at the producer. That is by design of the data flow paradigm not a problem, as the name is only important for routing of the data.
- again we have to pass the ports to the module initializer by a vector of ports.
- for getting data, we have to know which port to query[6].
- the message has to be decoded from the message's flat structure into something meaningful to the inner state of the module (e.g., de-serialize the data).

The two codes need to be compiled into a binary. For this example, we decide to use the two modules in two different binaries, called `producer` and `consumer`. We have to create the make files for these files and deploy all libraries needed for linking and execution.
As an additional note here: we assume that producer and consumer can work without any externally given parameter. Normally these are given by command line options to change the behaviour of a module dynamically. This will be discussed later in the context of PortUtils again.

18.2.3 Defining network level.

The simple application graph was already shown (see Figure 18.1). After the module code is written, the connection between the module from `P.data` to `C.eatdata` needs to be defined on the level of Flowvrapp[7].
In Flowvrapp, we have to define an interface wrapper for the module that defines the ports and the inner, logical, sub-graph of the component. So we need two classes that inherit from `Module` and that define the ports in their constructor.

```
class ModuleProducer : public Module
{
public:
   ModuleProducer( const std::string &id_ )
   : Module(id_)
   {
      addPort( "data", OUTPUT, FULL );
   }
};
class ModuleConsumer : public Module
{
public:
   ModuleConsumer( const std::string &id_ )
   : Module(id_)
   {
      addPort( "eatdata", INPUT, FULL, "", Port::ST_BLOCKING );
   }
};
```

There are again some points to note here.

- as we wrap the final module, the port names and directions and state have to match *exactly* the definition that is given in the module code.
- we have to rightfully chose the type of the ports.

[6]In this example, this is clear: there is only one port, but we still need the pointer to that port.
[7]In the following, most of the interfaces used are in the namespace `flowvr::app`, which we will omit in the text.

- typically this stage can be used to forward user given parameters from the parameter-file to the command line of the launcher. For the given example, we do not need any parameters, so the code is as plain as above.

These modules have to be associated by a launcher, or in the definition of Flowvrapp, have to be wrapped by a `metamodule` instance. As we have no specific need for the MPI launcher, we use the FlowVR ssh launcher to start the binaries for us.

```
class MetaModuleProducer
: public MetaModuleFlowvrRunSSH<ModuleProducer>
{
public:
   MetaModuleProducer( const string &id_ )
   : MetaModuleFlowvrRunSSH<ModuleProducer>(id_, CmdLine("producer")) {}
};
class MetaModuleConsumer
: public MetaModuleFlowvrRunSSH<ModuleConsumer>
{
public:
   MetaModuleProducer( const string &id_ )
   : MetaModuleFlowvrRunSSH<ModuleConsumer>(id_, CmdLine("consumer")) {}
};
```

Again some things to note here.

- the launcher itself does only define the specific requirements on the launching process, for example command line parameters to the launcher to forward the display.

- the name of the binary to launch has to be given as a parameter to the `CmdLine` object, e.g., the ssh wrapper.

- by inheritance, the launcher will act as proxy to the module, as it forwards all ports defined by the module as-is.

At this stage, we come to a total of 4 classes for two modules: 1 for `producer`, 1 for `consumer`, and two launchers, as we have two different binaries to launch.
The resulting connection between the two still has to be defined in a separate class[8].

```
void ProducerConsumer::execute()
{
   MetaModuleProducer *p = addObject<MetaModuleProducer>("producer");
   MetaModuleConsumer *c = addObject<MetaModuleConsumer>("consumer");
   link( p->getPort("data"), c->getPort("eatdata") );

   FilterPreSignal *ps = addObject<FilterPreSignal>("ps");
   link( c->getPort("endIt"), ps->getPort("in") );
   link( ps->getPort("out"), p->getPort("beginIt") );
}
```

The things to note here are as follows.

- we connect the metamodules instead of the modules.

- we link the modules directly by the port names given in the code above.

[8]In this case, this is even the resulting application, but in the general case, this is just some subgraph of the final network.

- as we want to drive the application by the frequency of the consumer, we introduce a special cycle breaking filter here, though this is not important for the overall example.

Finally we arrive, after having defined 5 classes in Flowvrapp to create and launch the network as a FlowVR application. The application can now be launched by a call to `flowvr` in the command line, with all paths been set.

18.3 Shortcomings and sources of error

First, we will describe the most common errors that can happen on module level.

- forget to add a port to the vector of ports that is passed to `initModule()`.
- we give a name twice (input and output ports share the same namespace, so it is illegal to have a name for a port multiple times).

That is not overwhelmingly much, but these errors can be tedious to find when the applications get bigger. Especially the first point can become really annoying, for example when in the process of re-working the module code and adding new ports.
The internal structure of a module tends to show large bodies of the `module->wait()` while loop. Typically there is a larger state machine which become unreadable very quickly. If there is more than one port used for input and output passing, each port-polling results in more code and introduces more constraints on the order of putting and getting messages. More sophisticated integrations of the FlowVR API into custom modules is typically done for this module only and resulting code base is hard to compare to other approaches.
For the level of Flowvrapp, there is more errors that can occur.

- mis-spelling of a port name for the module declaration.
- mis-typing the ports during module declaration. For modules port types can only be `FULL`. But on Flowvrapp level, we can mark module ports as `STAMPS` without error, but the resulting network will not function properly.
- mis-spelling of the binary name of the application.
- mis-spelling port-names on the level of the `Composite` that defines the application[9].
- giving a wrong parameter set to the launcher (but that is beyond scope of this document to discuss that).

While all the errors depicted above are quick to solve, they are hard to find when the `Composite` code becomes big or there are more than two or three element in the resulting network.
General shortcomings of the approach are obvious. For even a very simple problem, we create a number of new classes and their instances can be regarded as 'parametric objects', that is they define mostly the strings to be passed and used for searching, while the code is generally quite comparable. For each new class, probably new include and linker paths have to be defined for the projects that want to use them. This makes building and deployment of the application more complicated. The relation between `metamodules` and `modules` may not be quite clear to beginners and it can confuse when to give which type to which other layer of Flowvrapp.

[9]that is: we mis-spell the name in a call to `getPort`*name*.

18.4 PortUtils – Overview

The idea of PortUtils was to create an API that implements a more sophisticated approach to module definition in the first hand. By that, we wanted to have more maintainable code and modules that can be better compared and exchanged. Another aspect was to reduce the number of error sources when integrating a module into the FlowVR network when using Flowvrapp.
With respect to the example given (see section 18.2), we learned the following.

- module writers have to create the main-loop and co-responding make structures for each module again and again.
- some arrangements we might want to have for modules have to be implemented by the one who creates the module code. For example the reaction of the module upon an internal error (stack-trace print) or the graceful exit and cleanup has to be reprogrammed every time.
- port names are used as strings throughout several levels, and they are typically hard-coded.
- maintaining ports and using them is easy, but error prone.
- reading and submitting messages to a port is a repetitive process that should be abstracted. In the end, we are interested in getting messages in and out, but usually do not care how this is done (means: the port itself is no longer an important structure to maintain).
- the current API does not give any hint on how to structure the module better. As such, there will be lots of big while loops. The module code tends to be hard to maintain and extend.
- there is no hint on how to support the re-use of serialization code for messages. This is a rather important aspect, as the data that flows through the network has to be interpreted by many modules, so a factorized serialization code simplifies application development and helps to maintain things.
- there is no direct support to factorize the code that works on specific types of messages. This, again, is an important aspect, for when we replicate functionality in the data flow network, we might not always use the full module code for that, but just a portion. As the module code determines when and how the ports are read and that there have to be other connections, that code is hard to factorize.
- even the thin-waved structure of FlowVR already leads to complex state machines and ofter hinders that code can be re-used in other contexts, for example when FlowVR types are passed throughout the system to finally deploy the data to the data-flow network. It would be more interesting to be able to separate code needed to set-up 3rd party code from the receiving and dispatching code working on the data.
- for long-lasting applications, often we have to do upgrades of 3rd party libraries. In that case we would like the work imposed by the updgrades to be minimal, only affecting small portions of our module code.
- with respect to the network, the flow of information is the same, while module code can change drastically. For example when a user decides to switch an underlying API. Having FlowVR code intertwined with module code, this typically leads to a total re-write of the module code.
- the Flowvrapp code introduces a lot of intermediate classes that are or 'parametric' type. The only really interesting class is the one that defines the connectivity of the sub-graph. The relation between the classes may not be too clear to users at first sight.

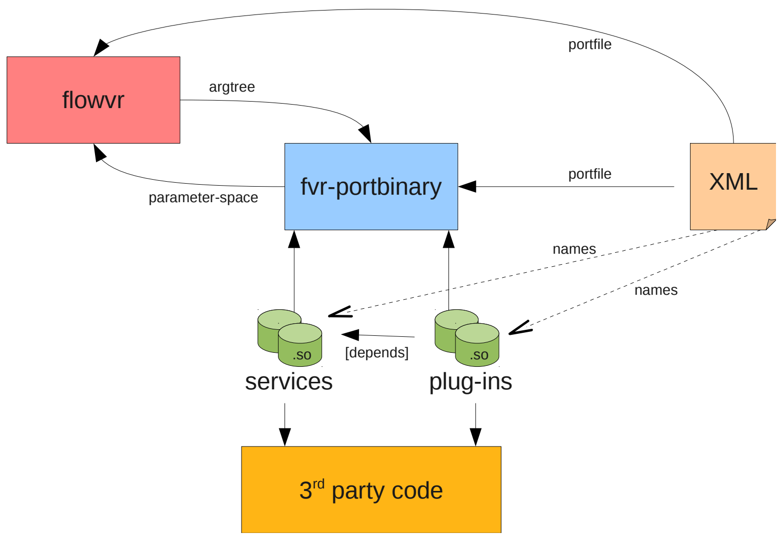


Figure 18.2: Physical structure of a PortUtils assembly.

Some of the points above can be approached by the usage of scripts that create skeletons for the code to write. This simplifies the first steps in the project but is totally useless when the code evolves and new requirements have to be introduced.
We will present the features of PortUtils in a more structured way now. The basic ideas of PortUtils in a nutshell are as follows.

- PortUtils maintained modules can be used along with already existing, normal flowvr and Flowvrapp elements. It builds completely on top of the existing API.
- for the modules: port-names, their 'semantics' and the relation to 3rd party code is described in a module-extern resource, currently an XML file.
- the 'semantics' of a module is defined by ports, their relation to 'handlers' and 'services' and an order of execution. User code focuses on defining 'handlers' and providing them as plugins to the PortUtils aware binary module.
- there is (normally) just one binary that is launched as a module and that takes care of setting up the module code according to the structure defined. This binary also defines some common mechanisms we want to support on system level, for example error handling and better debugging. It also tries to support the common 'tricks' that flowvr programmers do on the API, like 'pre-wait messages', 'more-than-one-message per loop' and 'one-shot' messages as well as 'no-port execution of code'.
- the external (XML) resource is used in Flowvrapp to create instances of parametric classes that reflect the defined structure. By that, it is no longer necessary to create the intermediate classes (module / launcher) manually, but instead focus more on the application graph.
- PortUtils unifies the way that parameters are defined and passed between Flowvrapp and the module[10].

This results in a special physical structure of a PortUtils assembly (see Figure 18.2).

[10]Up to now we did not cover parameter passing, this is done below.

All the above benefits come with a price. Dynamic module interfaces are not well supported, as the interface structure is defined by a static resource (the XML file). Of course between different runs one can modify the file, but there is less possibility to change the module's signature during run-time[11].
Another aspect is that deployment of the application gets more complicated. While the old approach needed to have just the execution path to be set-up correctly to find the binary, we now have to declare where to find the interface descriptors and extend the dynamic loader's run-time-search path to find plugin structures.
The structure of the code is now more fragmented, as it resides in separated plugin structures, and there is not a single main that can be inspected with a debugger[12].
The parameter passing is more complicated, as a parameter tree is passed in an encoded form to the application, so manually launching of modules is more complicated[13].
Finally said, there are just two locations now where strings are formulated that build the glue between the run-time components: on the level of the XML file descriptor and for the composite in Flowvrapp that still has to be formulated knowing the strings defined for the module. As the strings can change between iterations in the XML file, the co-responding Flowvrapp component has to be adapted to the changes. Even worse, we can pass the name of the interface descriptor to the Flowvrapp component by parameter and totally changing the signature of the module, causing a Flowvrapp component to fail during execution[14]. In that sense, the Flowvrapp component now defines the constraints that the interface descriptor has to obey. This dependency already exists in the normal approach of Flowvrapp, but it was by a three level compile time dependency (in the module, the Flowvrapp primitive description and the Flowvrapp composite description). Compared to a one-time compile time dependency (Flowvrapp composite description), this is already some kind of progress.

18.5 Using PortUtils

The following section will describe a user-level point of view on PortUtils by describing how to use the library. It will on purpose not explain some automagic facilities of PortUtils, but more focus on an example and an outline of the deployment structure.
From now on, we call the 'external XML interface descriptor' simply a *portfile*. The part of a module that actually performs a decoding of messages, or the encoding of new messages is called a *handler*. Handler code is defined by *plugins* and 3rd party code can be represented by *services* and it executed in a *code path*. The relation between the plugins and the services is strongly coupled from plugin to service, but there is not coupling from service to plugin structure[15]. *Ports* simply stay ports as they are defined by flowvr.
A *module* is now defined by reading a portfile that describes the ports to allocate, including their name and direction and flags. It will find there a definition of the code paths to execute, and each code path is bound to a plugin. Each plugin consists of a shared object that may be bound to exactly one service. Each service itself is a shared object, too.
There is already one executable shipped with flowvr, called `flowvr-portbinary` that does the bulk work, so the user does not have to worry about setting up the details and passing arguments and parameters.

[11]There is, however, a mechnism to dynamically duplicate existing ports, using the `multiplicity` attribute of a port declaration.

[12]Note that we do consider that actually a benefit, as we can focus on smaller code fragments for debugging. However, there are situations, especially in a `service` to `plugin` communication that might now be harder to overlook.

[13]There is, however, better tool-support for creating the encoded tree and parameter files.

[14]For the current version of Flowvrapp (1.8) this is the case, we hope to come up with a clever way of avoiding this in the future.

[15]At least there should be not coupling in that direction. It will lead to trouble.

18.5.1 Pickung up the example again

Let's see how to re-create the above given example with PortUtils. First of all, we care about the modules, we have two, called `producer` and `consumer`. We have to define two portfiles now, we can give them any name, but for the sake of simplicity, we call them `producer.xml` and `consumer.xml`. First of all, we will not care about the code that we want to execute within the module, but focus on the interface. The producer file just containing the port definition looks like this.

```
<config>
 <ports>
  <port name="data" direction="out"/>
 </ports>
</config>
```

The consumer file is defined comparable, we have to change port names and direction according to out needs.

```
<config>
 <ports>
  <port name="eatdata" direction="in"/>
 </ports>
</config>
```

Let us for now focus on the Flowvrapp level to connect the co-responding application graph, based on PortUtil modules. We define one instance that will reason just about the connection between `producer` and `consumer`. For now, we will focus on the implementation of the `execute` method of the composite. The full code will be shown later.

```
void ProducerComsumerExample::execute()
{
   FlowvrRunSSH ssh(this);
   ssh.setVerbose();
   ssh.setParallel();

   PortModuleRun *producer
      = new PortModuleRun( "producer", "producer.xml", ssh, this );
   PortModuleRun *consumer
      = new PortModuleRun( "consumer", "consumer.xml", ssh, this );

   link( producer->getPort("data"), consumer->getPort("eatdata") );

   PreSignalFilter *ps = addObject<PreSignalFilter>("pf");
   link( consumer->getPort("endIt"), ps->getPort("in") );
   link( ps->getPort("out"), producer->getPort("beginIt") );
}
```

That is about it. If we compile the application graph using the normal flowvr facility we get an application graph (see Figure 18.3). We could already run the application using FlowVR, but nothing will happen, as we did not yet define what the entities are about to do. For that, we have to define some plugins (also called 'handler') to the `flowvr-portbinary`.
In the example given, we want to produce some data with the producer. We can see that the producer does not react on input, it is a *source* for data as it just writes messages upon every iteration. PortUtils offers a special base class as a plugin to use for that case, called `SourcePortHandler`. Our special handler inherits from that handler and has to define a full virtual function to work. Again we will just focus on the relevant part, as the full code will be presented later.

```
// handle message to overload for a SourcePortHandler
```

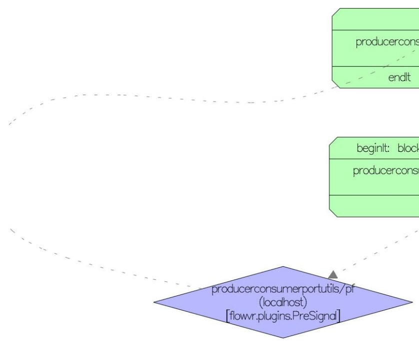

beginIt: blocking
producerconsumerportutils/producer/body/0
(localhost)
endIt
data
beginIt: blocking
eatdata: blocking
producerconsumerportutils/consumer/body/0
(localhost)
endIt
producerconsumerportutils/pf
(localhost)
[flowvr.plugins.PreSignal]

Figure 18.3: FlowVR graph representation shown by `flowvr-glgraph`.

```
// @param out the message container to fill
// @param stampsOut the stamps list associated with
//      the message / port for sending
// @param allocate the allocator to use to get memory from flowvr
virtual eState handleMessage( MessageWrite &out,
                              StampList *stampsOut,
                              Allocator &allocate )
{
   Data date = produce();
   size_t n;
   char *pdate = serialize(date,n);
   out.data = allocate.alloc(n);
   memcpy( out.data.writeAccess(), pdate, n );

   return E_OK;
}
```

As you can see, this code inside the method is quite close to the original code we devloped for the standard FlowVR way. Some things to note are as follow.

- There is no reference to a port from which we get the message.
- We already get passed a container (`MessageWrite`) to be filled.
- We can use the `Allocator` to claim memory for the message.
- We get, however, the `StampList` that is used on the output port later that we can use for writing stamps.
- The code returns `E_OK`, indicating that everything was all right and this handler is still active in the next iteration.

In the following, we assume that the handler is compiled into the shared object named `ProducerPlug.so`.
Now let us take a look at the co-responding code used for the consumer. We see that the consumer simply takes a message, decodes a data from it and consumes the message. This means that a plugin needs an input, but no output port. PortUtils offers a base class called `SinkPortHandler` for that pattern. Again, we will just focus on the part of the code that does the work, the whole code is presented later.

```
virtual eState handleMessage( const Message &m, const StampList *sl )
{
   Data date = deserialize(m.data.readAccess(),m.date.getSize());
   consume(date);
   return E_OK;
}
```

The observations are close to what is already described above, the signature of the handling method changed a bit. So we get a reference to the message that came in, as well as a pointer to the `StampList` associated with the original port that received the message. The consumer code we expect to be compiled into a shared object file called `ConsumePlug`.
We now need to hook the handling code into the co-responding XML files. For that we extend the above given XML files as follows.

```
<config>
 <ports>
  <port name="data" direction="out"/>
 </ports>
 <!-- this is the new code to insert: one for the code-paths to take. -->
 <code>
  <path name="produce" plugin="produce" out="data"/>
 </code>
 <!-- ... and one for the plugin that does the job. -->
 <plugins>
  <plugin name="produce" so="ProducePlug"/>
 </plugins>
</config>
```

We separate the plugin declaration from the code-path declaration. The reason for this is obvious: a plugin could be used more than once in different code paths. The one path given above is called '*produce*', just as the plugin. This may or may not be done like this. The strings merely have to co-respond, so that during creation of the code-path a co-responding plugin can be found by the string given. The plugin needs an attribute called '`so`' that tells PortUtils which shared object to link from disc. As a simplification for the procedure on different platforms, users must omit the latter part of the so-name, typically this is `.so` or `.dylib`. By naming convention, the shared object that is created lacks the prefix `lib`. This is, however, not mandatory. As a rule of thumb, give the whole name excluding the suffix as parameter to the `so` attribute. During execution time, this library has to be found in the dynamic library path of your system[16].
The XML file for the consumer is extended just as like the producer file.

```
<config>
 <ports>
  <port name="eatdata" direction="in"/>
 </ports>
 <code>
  <path name="consume" plugin="consume" in="eatdata"/>
 </code>
 <plugins>
  <plugin name="consume" so="ConsumePlug"/>
 </plugins>
</config>
```

When we run the application, we see that the modules are now iterating quite fast to exchange messages and we see messages of output for the producer and consumption for the consumer.
The programmer still can decide to have the serialization code integrated into the plugin, or use it as an external dependency to both, the producer and consumer plugins. For this example, we have chosen

[16]We leave out more details on this and come back to deployment later.

the first, by duplicating the class to exchange and separating the serialization code.

18.5.2 Deploying the example

We briefly mentioned that the `flowvr-portbinary` needs to read an existing portfile in order to function properly. During application launch-time, the portfiles have to be found in either the local path or in one of the paths that are defined by the environment variable `FLOWVR_PORTFILE_PATH`. The latter closely mimics the behavior that is already used in other FlowVR areas, for example`flowvr-render`, where the `FLOWVR_DATA_PATH` for example defines the location of models, shaders and textures. Additionally, the `LD_LIBRARY_PATH` has to be adjusted in a way it can find

- all the plugin and service shared objects.
- all relevant 3rd party libraries that are needed to load the plugins.
- all relevant utility libraries, for example serialization codes, that are needed.

There is no need to alter the run-time `PATH` variable, as the `flowvr-portbinary` will be part of a normal FlowVR installation[17].

18.5.3 A more complex example

The first example simply showed two participants, one sending and one receiving data. It can be seen that a re-write using PortUtils is quite straightforward. Obviously, the most benefit can be seen in the reduction of interfaces that have to be defined on the level of Flowvrapp. We will now make it a bit more complex by adding parameters. First we will see how this works in the standard way to compare it against the PortUtils approach.

18.5.3.1 Parameter in the regular approach

Parameters in flowvr are passed from a parameter file or the command line to Flowvrapp which re-parses the `Parameter`-run-time structure and creates arguments to augment the command line for startup. For this example, we will add a simple parameter, that decides on the number of `Data` objects that are put into one message. First, we have to modify the `main` routine of the producer. To simplify parameter handling, we use the `CmdLine` class which is part of flowvr already. Again, we will here just focus on the important details.

```
[...]
CmdLine line("producer");
bool error=false;
Option<int> num("num", 'n', "Number of Data instances to put in one message",
    false );
// parse the command line and put values into place
if(!line.parse(argc,argv, &error))
{
   if( error )
      cerr << line.help() << endl;
   return 1;
}
// now read off the given argument
int nnum = num.value();
[...]
// change in the wait-loop:
```

[17]Unless the application itself has a requirement on the state of the PATH variable.

```
while (flowvr->wait())
{
   // container to store entities
   vector<Data> vProduce;
   for( size_t n=0; n<nnum; ++n)
      vProduce.push_back(produce());
   MessageWrite mw;
   mw.data = flowvr->alloc( nnum * 256 );
   // now serialize, assuming a fixed size of 256
   // for simplifying the example.
   char *dt = mw.data.getWrite<char>();
   for( vector< Data >::iterator cit = vProduce.begin();
        cit != vProduce.end();
        ++cit )
   {
      size_t n;
      memcpy(dt, serialize((*cit),n), 256 );
      dt += 256;
   }

   // Send message
   flowvr->put(&p,mw);
}
```

The binary now has to be launched with the proper command-line, passing the correct argument. This command-line argument has to be constructed on the level of Flowvrapp again. The change will affect the `metamoduleproducer.comp` code, where we augment the constructor to `addParameter()` with a default value of 1.

```
MetaModuleProducer::MetaModuleProducer(const string& id_ )
: MetaModuleFlowvrRunSSH<ModuleProducer>(id_, CmdLine("producer") )
{
   setInfo( "Launcher for Module-Producer" );
   addParameter("nb", 1);
}
```

Note that we change the name of the parameter, as we will have to re-parse the command line in the `configure` method. This is the body of the `configure` method.

```
void MetaModuleProducer::configure()
{
   int n = getParameter<int>("nb");
   getRun()->addArg( "--num " + toString<int>(n) );
}
```

As you can see here, we rename the Flowvrapp variable `nb` to the `producer` argument name `num`. To alter the value of `nb` we have to pass it as a parameter in the command-line when launching FlowVR[18].

```
flowvr -x -L \
      -Pproducerconsumerregular/producer:nb=2 \
      --complib install/components/libproducerconsumerregular.comp.so \
      ProducerConsumerRegular
```

By doing this, the command-line will be augmented to hold the parameter `-num 2` that is passed to the `producer`. Things to note here are as follows.

[18]Or alternatively we add it to a parameter file.

- we have to give the parameter by name and type in the module code.
- we have to modify the Flowvrapp`metamodule` representation, declare the parameter and update the value of the parameter in the `configure()` method. Here, we have to know the correct name of the parameter as it is named in the module.
- there is no direct way to distinguish between parameters for the Flowvrapp level and the parameters that will be passed to the module. The `metamodule` code has to take care of that.

18.5.3.2 Parameters with PortUtils

It is clear that the plugin code for the producer has to be changed to respect the parameter. In the `producer` code we find a global function called `getParameters()`. This method is an entry point to *fetch* the parameter structure for this plugin.

```
extern "C" void getParameters( ARGS &args )
{
   args["num"] = Parameter("1",
               "Number of data entities to produce per iteration",
               Parameter::P_NUMBER );
}
```

Hereby, we declare that this plugin has a parameter *n*um with a default value of 1 that is considered a number type. In order to use the passed parameter in the module, we have to decode it, and there is a chance to do that in the constructor of the plugin.

```
ProducePlugHandler( const PortUtils::ARGS &args )
: SourcePortHandler()
, m_nNum(1)
{
   // note that we use a special operator here to
   // access the parameter ( 'args(string)' instead of 'args[string]' )
   m_nNum = args("num").getValue<int>();
}
```

The `handleMessage` routine has to be modified to produce now more than 1 `Data` item per iteration, similar to the code in the regular example, so we omit it here. All that is left now to do is to pass the parameter to the flowvr binary upon execution. To do so, the parameter is added to the command line as follows.

```
flowvr -x -L \
     -Pproducerconsumerportutils/producer:plugins-produce-num=2 \
     --complib install/components/libproducerconsumerportutils.comp.so \
     ProducerConsumerPortUtils
```

The parameter is automatically updated and passed to the executable. At this point in the tutorial, we will explain the structure of the parameter name in more detail. As can be seen above, we do not pass the parameter `num` just as this, but pass it as `plugins-produce-num`. This encoding means that the parameter named `num` is meant to be passed to a `plugin` of a PortUtils module. The name of the plugin is `produce`. Note that this is the name we defined in the `producer.xml` for the plugin. This routing scheme allows to define parameters for specific plugins, even if different plugins share some names in their parameter name-space.
PortUtils offers a way to check what parameters can be passed to a module, it can even create a template parameter file to be filled. The tools that aid in the process are discussed for showing parameters (see paragraph 18.5.3.2.1), creating parameters (see paragraph 18.5.3.2.2) and checking actually passed parameters (see paragraph 18.5.3.2.3).

18.5.3.2.1 Showing parameter-space (flowvr-pups) The flowvr-**p**ort **u**tils **p**arameter **s**how utility parses a portfile and outputs a string that shows the parameter space.

```
> pups -p config/producer.xml
[plugins]:
 [produce]:
  [num]: 1 # Number of data entities to produce per iteration [NUMBER ;
     NONE ; DEFAULT ; OPTIONAL]
[services]:
 <EMPTY-SUB-CONTAINER>
```

We can see in this example, that the producer module has 1 plugin, called `produce`, and no services. The `produce` plugin takes one parameter, called `num`. It is of type `NUMBER`, the default value of 1 is shown here and the user does not have to specify it, as there is a default value.

18.5.3.2.2 Creating parameter template files (flowvr-cpff) A template parameter file can be created with the flowvr-**c**reate **p**ort **f**ile **f**ile.

```
> cpff -p config/producer.xml -c producerconsumerportutils/producer
# plugins-produce -- num:
# Number of data entities to produce per iteration
# TYPE: NUMBER ; MODE = OPTIONAL
producerconsumerportutils/producer:plugins-produce-num=1
```

The '-c' option defines the prefix that is determined by the final hierarchy in the Flowvrapp representation. That is the prefix string necessary to identify the components in the application hierarchy. For the example above, that is `producerconsumerportutils/producer`.

18.5.3.2.3 Checking the parameters passed to an application (flowvr-spl) The flowvr-**s**how **p**arameter **l**ist can be used to parse the argument tree that is constructed on behalf of PortUtils and passed in the commandline via the `argtree` argument. The `argtree` in the command line is base64 encoded and can be found in the flowvr output or log files. The above example passes an argument tree as follows.

```
flowvr-run-ssh -v -p ' localhost ' flowvr-portbinary \
           --portfile producer.xml --idstring producer \
           --argtree 122:YyA0IHJvb3QgMyBjIDYgbW9kdWxlI\
                     DAgYyA3IHBsdWdpbnMgMSBjIDcgcH\
                     JvZHVjZSAxIHAgMSAtMSAzIDEgMyB\
                     udW0gMSAyIGMgOCBzZXJ2aWNlcyAwIA==
[...]
> spl -a 122:YyA0IHJvb3QgMyBjIDYgbW9kdWxlIDAg\
         YyA3IHBsdWdpbnMgMSBjIDcgcHJvZHVjZSAxIHA\
         gMSAtMSAzIDEgMyBudW0gMSAyIGMgOCBzZXJ2aW\
         NlcyAwIA==

[module]:
 <EMPTY-SUB-CONTAINER>
[plugins]:
 [produce]:
  [num]: 2   [NUMBER ; NONE ; FILE ; OPTIONAL]
[services]:
 <EMPTY-SUB-CONTAINER>
```

As can be seen, the `produce` plugin is passed a `num` parameter with a value of 2, while services and the module itself gain no parameter. The paramter was changed on behalf of a parameter file or the command line.
Things to note are as follows.

- The parameter space is defined by the plugin (or service) only.
- PortUtils offer tool support to check dynamic parameter-space, create configuration files and check on the parameters that are really passed to the application.
- The parameter space now has a syntax for PortUtil binaries.
- Adding new parameters to the application has no effect on the Flowvrapp layer.
- Flowvrapp components can still add parameters, but they are typically needed to configure the Flowvrapp behavior only.

18.5.3.3 Using PortUtils services

A *service* is a passage of code that is related to 3rd party code or, more general, any state that can be shared between plugins. As it is hard to describe that concept with the regular approach of creating module in flowvr, we omit this for the discussion to follow. In an abstract way, one can think of a data structure that is accessed in a sequence of `put` and `get` calls insider the `wait`-loop.
For the example defined so far, we define a `DataService` that is able to `produce()` and `consume()` data. The PortUtils interface for a service is used by inheritance again.

```
class DataService : public IServiceLayer
{
public:
   // [... portutil stuff snipped here ...]
   class Data
   {
   public:
      Data();
      char data[256];
   };

   Data produce();
   void consume( Data & );
   const char *serialize( const Data &, size_t &n );
   Data deserialize( const char *, size_t n );
};
```

We clearly see that the `DataService` interface now contains all the code we had duplicated in the plugins before, including the `serialize` and `deserialize` methods. We see also that we decided to give the `DataService` the possibility to do both: `produce` and `consume` the data. This is for the sake of simplicity of the example, and in a real scenario could be a different decision.
The plugin code is modified as well. The handler now gets passed a pointer to the `DataService` to operate on. This is a play on defines and part of the low-level PortUtils documentation, so we will skip a detailed presentation here, but only show the modification for the `producer`.

```
ProducePlugHandler( DataService *service, const PortUtils::ARGS &args )
: SourcePortHandler()
, m_nNum(1)
, m_service(service) // store service pointer
{
 // [... snip ...]
}
```

Note that we have to change the linking structure of the project now, as the plugins now depend on the service directly. The service itself does not see the plugins, so it has just to link 3rd party code if needed. For this example there is no external code. See the `cmake` files in the examples to know how to do the linking properly.
Finally, we modify the portfile structure to reflect for the `producer` and `consumer` to be depending on a service now.

```
<config>
 <ports>
  <port name="data" direction="out"/>
 </ports>
 <code>
  <path name="produce" plugin="produce" out="data"/>
 </code>
 <plugins>
  <!-- add a reference to a section of the services, see below... -->
  <plugin name="produce" so="ProducePlug" service="dataservice"/>
 </plugins>
 <services>
      <!-- define a service to be loaded from an .so -->
       <service name="dataservice" so="DataService"/>
 </services>
</config>
```

The `DataService` must, just like the plugins, be in the search path of the run-time linker, naturally. The portfile for the consumer is changed accordingly. If we launch the application now, there is apparently not any externally notable change in behavior, as only the internal structure of the module changed completely. There is no constraint on the way that the plugin / service communication is structured by PortUtils.
The service / plugin pattern can be used to create new modules from existing functionality without the need to write and deploy more modules. For example a simple filter chain can be produced by changing the Flowvrapp layer of the given example in the following way.

```
ProducerConsumerPortUtils::ProducerConsumerPortUtils( const string &id_ )
: Composite(id_)
{
   addParameter("chain-length", 1);
}

void ProducerConsumerPortUtils::execute()
{
   FlowvrRunSSH ssh(this);
   ssh.setVerbose();
   ssh.setParallel();

   PortModuleRun *producer
     = new PortModuleRun("producer", "producer.xml", ssh, this );
   PortModuleRun *consumer
     = new PortModuleRun("consumer", "consumer.xml", ssh, this );

   int chain_length = getParameter<int>("chain-length");
   if( chain_length < 0 )
      throw CustomException("Chain-length must be > 0", __FUNCTION_NAME__);
```

```
    PortModuleRun *pred = producer;
    for( size_t n = 0; n < chain_length; ++n )
    {
       PortModuleRun *c
          = new PortModuleRun("filter"+toString<int>(n),
                              "consumerproducer.xml",
                              ssh,
                              this);
       link( pred->getPort("data"), c->getPort("eatdata") );
       pred = c;
    }

    link( pred->getPort("data"), consumer->getPort("eatdata") );

    FilterPreSignal *ps = addObject<FilterPreSignal>("pf");
    link( consumer->getPort("endIt"), ps->getPort("in") );
    link( ps->getPort("out"), producer->getPort("beginIt") );
}
```

We use the old `producer` and `consumer` to build the head and tail of a filter chain. We use a portfile variant *consumerproduct.xml* to combine the existing pieces to a new module that has both: an input as well an an output port.

```
<config>
 <ports>
  <port name="eatdata" direction="in"/>
  <port name="data" direction="out"/>
 </ports>
 <code>
  <path name="consume" plugin="consume" in="eatdata"/>
  <path name="produce" plugin="produce" out="data"/>
 </code>
 <plugins>
  <plugin name="produce" so="ProducePlug" service="dataservice"/>
  <plugin name="consume" so="ConsumePlug" service="dataservice"/>
 </plugins>
 <services>
        <service name="dataservice" so="DataService"/>
 </services>
</config>
```

By passing the `-Pproducerconsumerportutils:chain-length=10` we can scale the pipeline to use 10 infix filters of the given structure. They all work on the `DataService` that is represented by `DataService`. It should also be obvious that the modules in the middle share the same state of the `DataService`. That mean that any `consume` of an incoming data can change the state that is reflected in a subsequent `produce`. Pipes-and-filter chains can be built quite straightforward with the approach.

Part VI

Developer Manual

Table of Contents

This chapter is intended to advanced users that need to program parallel applications (MPI), new filters or synchronizers, to add new functionalities to the flowvr deamon, etc.

Chapter 19

Custom filters

??

19.1 Compiling and loading a Filter

In FlowVR sources filters code is located in flowvr/flowvrd/src/plugins/filters/ and synchronizers in flowvr/flowvr-daemon/src/plugins/sync/, while other system plugins are in flowvr/flowvr-daemon/src/plugins/sys/.
Basic rules when developping a filter or synchronizer:

- The name of filters and synchronizers should always starts with `flowvr.plugins.MYFILTERNAME.cpp`.
- They must be compiled individually as dynamic libraries and stored in an accessible directory path (use the environment variables `LD_LIBRARY_PATH` or `DYLD_LIBRARY_PATH` on mac), in the subdirectory `flowvr/plugins` directory. For instance the `FilterIt` plugin is located in lib/flowvr/plugins/flowvr.plugins.FilterIt.so. You can simply put your plugins in the same directory. For distributed execution, all plugins should be accessible from each node that runs a daemon (duplicate them if necessary).
- At compilation, provide access to FlowVR header files located in include.
- At link edition, link to FlowVR libraries, mainly `libflowvr-base` and `libflowvr-plugd`, located in lib.

For compilation and installation, we strongly advice to rely on `cmake` (see **??**) (`pkg-config` is also supported).
If this is properly done, at execution each daemon involved in the application will load the required plugins looking for the libraries `flowvr.plugins.FILTERNAME.so` in the subdirectory `flowvr/plugins` from accessible library paths.
$\boxed{\text{A VOIR}}$ *make a section about cluster execution* $\boxed{\text{FIN}}$

Chapter 20

FlowVR Run-Time Architecture

20.1 The Daemon

The daemon is the main component of the FlowVR architecture. It can be viewed as a collection of objects and plugins. Some of these objects are automatically loaded (like the commander), others need a control command (filters). The objects loaded by the FlowVR daemon can be passive objects (routing table, filters) or threads (regulator, net).
When the daemon starts it:

- sets the daemon internal name ("*flowvrd*" by default),
- generates the shared memory area, defined by a size and an ID,
- loads a `Dispatcher` object,
- loads a `Commander` object,
- loads a `Net` object (NetTCP by default).

20.1.1 Message Handling

20.1.2 Routing Table

Each daemon owns a unique routing table common to all the FlowVR objects running on the node. This object contains a list of routes. A route is specified by a source and an action. Several actions can be linked to the same source and the number of routes for a source is equal to the number of actions to take when a message comes from this source. A lock is required to protect the completion of control command like adding or removing a route.
The different actions possible when a message must be parsed are :

- Put a message on the input message queue of a module running on this node.
- Send the message to another node. The source of the message does not change.

Main functions of the routing table :

- `bool RoutingTable::addRoute(const string& id, const string& source, Message::Type msgtype, Action* action)` : add a route (a source/action couple) to the routing table. Note the message type must be specified.

- `bool RoutingTable::removeRoute(const string& id)` : remove a route.
- `std::vector<Action*>* RoutingTable::getAction(string source,Message::Type msgtype)` : send the list of action linked to a source.
- `Result RoutingTable::getRouteCount(const string& id)` : give the number of times a specified route has been used. An atomic count linked to the action is used.

20.1.3 Filters and Synchronizers

See programming (see chapter 8) filters and synchronizers.

20.1.4 Regulator

The regulator can be viewed as a *'box'* encapsulating the modules of the application. This is the object which deals and interacts with the others components of the FlowVR daemon. By hiding message handling, access to the shared memory and management of the internal loop of modules, the regulator makes modules simpler to develop.
Main characteristics :

- One regulator class per application (all modules have the same regulator class).
- One default regulator (`flowvr.plugins.Regulator`).
- Its is possible to define and load a different regulator.

20.1.5 Net

The Net is the communicating object needed to allow messages passing between different nodes running a FlowVR daemon. This object is now loaded automatically when starting a daemon. In the current release the default Net object is based upon the TCP protocol (See flowvrd/src/plugins/sys/flowvr.plugins.NetTCP.cpp).
Another implementation of the Net object can be made. To specify the Net object loaded when startting a daemon :

```
flowvrd --network classnameobject
```

Those currently available are `flowvr.plugins.NetMPI` and `flowvr.plugins.NetMPIm`. (see Figure 9.3)

20.1.5.1 Threading

On one node, the `NetTMP` implementation relies on two threads per distant node to communicate with. One for receiving messages and the other for sending messages. There is an additional thread for receiving connexions from new distants deamons.
`NetMPI` relies on the same TCP threads to send/receive meta-data about incoming messages, as well as stamps. Data itself is however sent and received within one single MPI thread, so it merely requires `MPI_THREAD_FUNELED` at MPI initialization. (see Figure 9.3)
The second one, `NetMPIm`, send and recieve all data through MPI, within one sending thread and sending through another one.

20.1.5.2 Internal

When you create a new route leading to another node, the Net plugin is asked for an `Action`. It's a functor that will get messages to be sent on a peculiar distant node. The `Net` plugin defines and instanciates the `Action`, usually one per distant node. The `NetMPI` does it at initialization, but `NetTCP` does it on demand since it spawns a thread on both machines.

20.2 The Controller

The controller is a special module. Its role is to launch and control the application. There is only one controller for each FlowVR application. The controller directly accesses every FlowVR daemon involved in the application (and as such, these nodes must be made visible to the Controller). The controller does not use the routing table. It sends control commands (start a module, add a connection, pause or stop the application) and receives status reports. The controller fills the gap between the user and the application.

20.3 The Controller and Daemon Interactions

The interface between the controller and the daemon is the commander. There is exactly one commander per daemon. The commander receives commands from the controller, process the commands and sends the results to the controller. The commander can reach any FlowVR object present on the same node.

20.4 The Command Language

The command language is the protocol used for communications between the controller and the daemons. It follows the `XML` syntax. Complete descriptions can be found via the DTDs.
Commands processed by the controller :

- `<addobject class="classname" id="name" > parameters </addobject>` : add an object of the given class and indexed by the `id` value. Note that when a module is added, the corresponding class is a regulator (i.e. flowvrd/src/plugins/sys/flowvr.plugins.regulator). `parameters` are specific data needed for the initialization of the object (i.e the number of node for a MPI module or the frequency for a control frequency filter).
- `<removeobject id="name" />` : remove the object identified by the given `id`.
- `<addroute id= num><source id= name1 port= PORT messagetype="full" /><action id= name2 messagetype="full">parameters</action></addroute>` : add a route with the given `id`. The `source` tag specifies the object id, the port connected and the type of message (full or stamps only). Depending on the `action` id value, `parameters` can be :
  - if `id` is equal to the id of a module, a filter or a synchronizer, the parameters will be a `port` tag with the name of the destination port of this object.
  - if `id` is equal to the id of the net object (*NET* by default), then parameters will be a `dest` tag (i.e the name of the destination node).
- `<removeroute id="name" />` : remove the route corresponding to the given `id`.

- `<action id=name>actiontotake</action>` : send an action to the commander of the current `dest` node. `action` command needs an id variable to specify the object concerned by this action. The different actions possible are :
  - start : start or restart the specified object. Each object created for a FlowVR application must wait this command before sending any message. This applies when the object is paused or just created and initialized.
  - pause : pause the specified object. If the object is already in pause, it makes nothing. An object in pause will not send any messages, generated messages are buffered. When the object is re-starting (after a *start*), these buffers are emptied before completing any new message.
  - close : to close and destroy the selected object. This command is only used to finish a FlowVR application.
- `<group id="groupname" >` : this command contains an action command to process on a group of objects or a command applied on routes (like removeroute). The value of `id` corresponds to the namespace where the set of objects is locate. If the namespace is null then all objects/routes of the application on each node are concerned. When the commander on a node receives this command, it builds the list of objects concerned by the command and process the action on each of them. It contributes to reduce the number of commands sent to the commanders.

NOTE : the `action` tag also contains a `messagetype` variable. This is the combinaison of the two values given in this command that determines the type of the connection (stamps only or full).
Two types of message can be sent by the commander to the controller :

- Result
- News

20.5 Application Deployment

20.5.1 States components

Depending on whether the component is a module or a daemon plugin (filter or synchronizer), it passes through successive statements when the application is launched.

20.5.1.1 Modules

A module is not a plugin loaded by the daemon, it is an entity run in its own separate process. It has four different states : 'not launched', 'launched', 'paused' and 'started'.

- `not launched`: the module is not running yet. The process running the module may have been started but did not contact the deamon yet
- `launched`: the module is running.
- `paused`: the module stopped to iterate and is awaiting a signal from the daemon
- `started`: the module received a regular signal from the daemon, which allows him to start its iteration loop.

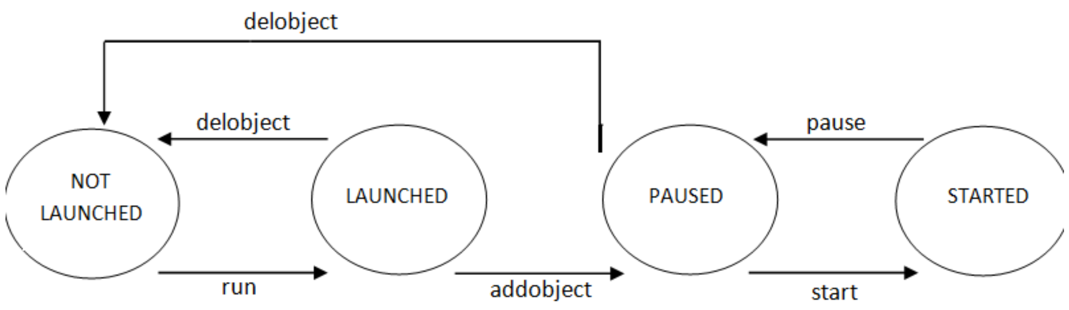


Figure 20.1: The differents states of a module

20.5.1.2 Plugin states

A plugin running in a thread and loaded by the daemon has three differents states : 'launched', 'paused' and 'started'.

- `launched`: plugin file loaded
- `paused`: the plugin is stopped and awaiting a signal from the daemon
- `started`: the plugin receives a regular signal from the daemon, which allows him to make iteration loop.

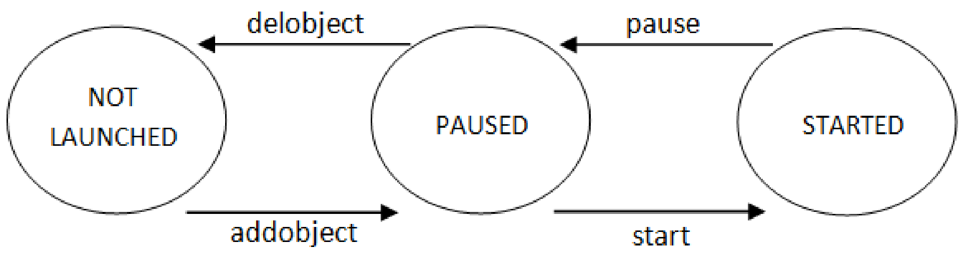


Figure 20.2: The differents states of a plugin

20.5.2 The 4 Deployment Stages

An application is deployed in four stages:

1. `Starting Modules`: retrieve the launching commands from the `.run.xml` file and execute them. It starts the metamodules (and its associated modules). Each module can go through the different states (see Figure 20.1): 'not launched', 'launched', 'paused' and 'started'.

2. `Plugin loading`: The daemon receives plugin loading orders from the controller through the `<addobject>` commands (stored in the `.cmd.xml` files). Each plugin can go through thethree states (see Figure 20.2): 'launched', 'paused' and 'started'.

3. `Route Creation`: Routes, which represent connexions between components, are added to the routing table of the deamon. The daemon receives route creation orders from the controller through the `<addroute>` commands (stored in the `.cmd.xml` files).

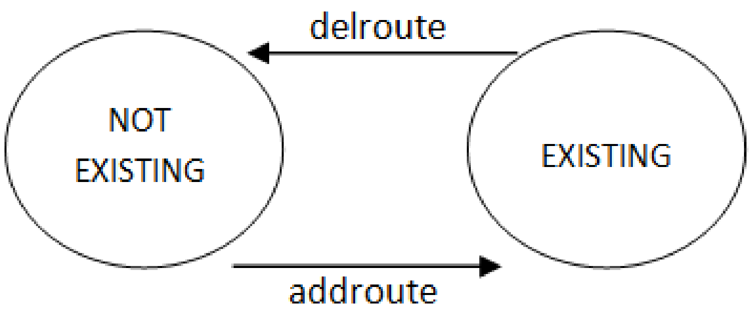


Figure 20.3: The differents states of a route

4. `Start Commands`: The controller sends start commands to the daemons. The daemon sends starting orders to its modules and plugins in the order defined by their priorities (from the highest to lowest).

20.5.3 Application launching: Interactive or Batch

When starting an application `flowvr` sends commands to the daemons to setup the application. From the `flowvr` prompt, the user may also issue his own commands through the console.
Flowvr objects are ordered hierarchically. They follow a tree structure : the root of the tree is the application name, and branches represent prefixes identifiers objects.
Thanks to this representation, it is possible to apply commands to specific objects, so commands can be applied to a subtree.

- `load` : This command allows you to browse the tree components such as the change directory of Linux, it can apply commands to a set of components that have this prefix. These orders are: start, pause, stop, addObject etc. 'load' alone loads all the components from the current position in the tree. 'load prefix' followed by a prefix can load all the components on this prefix (if any). 'load ..' return a shot back in the tree. 'load ../..' return two shots back in the tree.

- `loadDest` : This command loads components from a host.

- `launchMetamod` : This command launches metamodules from the current position in the tree. If a metamodule is already launched it does nothing

- `addRoute` : This command adds routes from the current position in the tree. A route is added if and only if both ends components are already added. If a route is already added it does nothing.

- `addObject` : This command adds objects from the current position in the tree and adds at the same time the road when both ends components are already added (it uses addRoute). If an object is already added it does nothing.
- `launch`: This command does in order: "launchMetaMod", "addObject", "addRoute." metamodules, all components and all roads have been added all that remains is to start the application with the command "start".
- `delObject` : This command removes objects from the current position in the tree. If an object is already deleted it does nothing.
- `delRoute` : This command removes routes from the current position in the tree. If an object is already deleted it does nothing.
- `<dest>hostname</dest>` : specify the hostname concerned by the following control commands (until a new `dest` command is processed). This command is just to set the state of the controller for the next commands.
- `<flush/>` : the flush command forces the controller to wait the result message of each command sended previously before processing other commands. This command can be useful when debugging the launch of a FlowVR application or for avoiding errors if some commands need that previous commands be completed.
- `<wait duration = time />` : the controller waits *time* seconds before processing the next commands.

There are other commands available that can provide information on the application progress :

- `pwd` : This command shows where you are in the tree.
- `getPaused` : This command returns components that have been paused.
- `getState` : This command returns the state of all components.
- `route` : This command shows routes added.

20.6 Shared Memory

20.6.1 Basics

A key to the performance of FlowVR is the shared memory. Using shared memory allows very fast communication between two modules residing on the same node. This section present details about the FlowVR allocators down to some pretty low-level details.
First, The FlowVR daemon allocate one big shared memory segment from the system (eventually several if both needed and authorized by the user). Then one can use a supplied allocator to obtain blocks from this shared segment.

20.6.2 Allocator

The allocator is a singleton through which one allocate shared memory. It's role is to get access to the shared memory segments from the system. It is instancied when initializing the first FlowVR module within a process.
This is the class allowing the user to allocate a high-level `flowvr::Buffer`, wrapping implementation details. If none of the shared memory segments can be used to allocate a buffer of the requested size, the allocator will ask the daemon to create a new shared memory segment. The daemon can either accept or deny it.

20.6.2.1 Custom Allocator

Furthermore, one can replace the default allocator by a custom one. One you can already find in flowvr is a pooled allocator giving better performance when allocating block of the very same size.
It can also be use when your block have roughly the same size. To do so, you can allocate blocks of the same size, larger than you can expect, then use the window constructor to prune the overflow.

20.6.2.2 Daemon Allocator

This specialized allocator, only used by the FlowVR daemon itself is responsible for allocating shared memory segments from the system. It is instancied within the daemon, right after the command line parsing, creating the *main memory segment* in which will live the *daemon header*.
It's also responsible of the allocation of new memory segments upon request, though it decides if it should either do it or not depending on the given command line options.

20.6.3 Shared Memory Area

The shared memory area don't have to be manipulated by the FlowVR user. It gives an interface for allocating 8 bytes aligned *blocks*[1] within one shared memory segment.
In the memory segment lives a chained list of free chunks from which you allocate. Both them and the allocated blocks are preceeded by a small header describing them. The header of the free chunk also contains the offest of the following chunk whilst the header of an allocated block contains a reference counter.
Chunks are stored as a double chained list. Each chunk header remembers the previous and next chunk while the area header keeps track of the first one. The list is ordered by the offset of the chunks within the memory segment.
When allocating, you lock the entire chunk list then search for a chunk large bigger than the requested size. This search is performed from both ends at once and the very first suitable chunk is used.

20.6.4 Buffers

This class wraps the memory block allocated by the shared memory area, notably performing the reference counting. It actually contains a vector of those memory blocks, each block beeing reference counted separatly, and freed using the allocator it as been allocated with.

[1] not quite a buffer yet

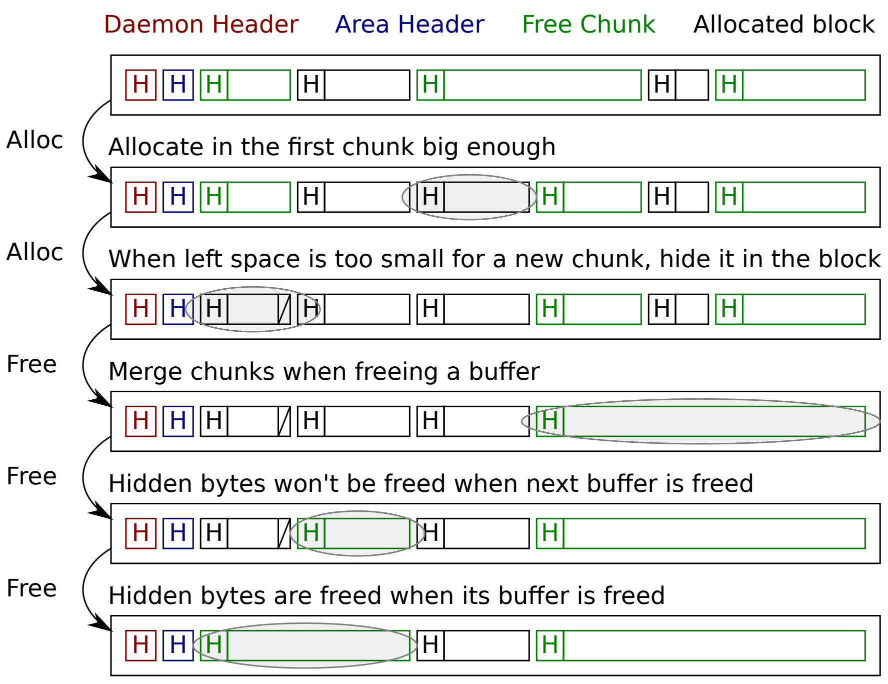

Daemon Header
Area Header
Free Chunk
Allocated block
Alloc
Allocate in the first chunk big enough
Alloc
When left space is too small for a new chunk, hide it in the block
Free
Merge chunks when freeing a buffer
Free
Hidden bytes won't be freed when next buffer is freed
Free
Hidden bytes are freed when its buffer is freed
H

Figure 20.4: Illustration of the allocation and freeing of a block within a shared memory area.

Since those buffer can be *segmented* one can easily concatenate huge buffers while avoiding costly memory copies but add some burden when manipulating them. However, the segmentation is always introduced by the user (through modules and filters), and we furthermore supply way of accessing a buffer regardless its eventual segmentation (see section 17.2).

20.7 Interprocess communications

This section explains some details about the way processes within a FlowVR node, and pecularily modules, communicate with each other.

20.7.1 MPChannel

Within a single node, processes communicate with each other almost exclusively using an `MPChannel`. This object living in the shared memory is a one-way FIFO communication channel. It is implemented as a fixed-length file using a lock and two conditions from the `pthread` library. Its maximum capacity is fixed once instancied, the writer process push messages on one end while the reader pop them from the other.

20.7.2 Put

The `put` command, when executed inside a module, send the message through an `MPChannel` to the `Regulator` associated with the module. The `Regulator` performs passive wait on the channel and processes incoming commands sequencially, delivering the message to the destination module if it's local, or to the `Net` object, using the `Dispatcher` object as a mere abstraction layer in both cases.

20.7.3 Wait

In the fashion of the `put` command, the `wait` one communicate with the `Regulator` but then waits for an answer comming through a second `MPchannel` (going from the `Regulator` to the Module). This means there is a cost even when there is no blocking port since you always perform four locks during the call.
Furthermore, since (`wait`/`put`/`close`) commands are serialized within the `Regulator`, a `wait` need all preceeding commands (notably `put` commands) to be performed by the `Regulator`. This is one reason why one should never put costly operations either in a filter nor a synchronizer, whose work is serialized by the `Regulator` as it processes the `put` command.

20.7.4 Alloc

`Buffer` allocation doesn't need inter-process communication but it uses the shared memory area, which accessible from every FlowVR module on the node. Data race are avoided by locking the entire memory area using a single mutex living in the area header. This enforces the serialization of both allocation et freeing of the shared memory blocks.

20.7.5 Get

This command is different than the previous commands since it doesn't involve any inter-process communication, not even a lock.

Part VII

MISC

20.7.6 Module Name

By default the module name is set automatically and the user does not have to worry about the naming mechanism. We detail below how this mechanism behaves in case the user needs to control module naming, necessary for instance when using a non supported command for module launching (`flowvr-run-ssh` and `mpirun` are currently supported $\boxed{\text{A VOIR}}$ *link* $\boxed{\text{FIN}}$).
The name of a module can be set through different ways:

- `FLOWVR_MODNAME`: set the module name from the value of this environement variable if set.
- When calling `initModule` (see subsection 6.2.1).

This may be useful to integrate a number in the name of modules belonging to the same group (`toto/0`,`toto/1`, ...). FlowVR provides such mechanism, it is called the **parallel interface** (include/flowvr/parallel.h). When this mechanism is activated, the affected modules have their name suffixed with a number (using a `/` separator). There are two ways to set this number:

- When executing `initModule`, the module detects the `FLOWVR_RANK` (the module rank in the group) and `FLOWVR_NBPROC` (the total number of modules in the group) environment variables. It thus automatically uses the value of `FLOWVR_RANK` as suffix. The `flowvr-run-ssh` utility set such values when launching the same instance of a module several times (see the -p option) $\boxed{\text{A VOIR}}$ *a prÃ©ciser* $\boxed{\text{FIN}}$
- `void Parallel::init(int rank, int nbProc)`: This method needs to be called before `initModule`. It sets the module rank number and the total number of module in the group through the `rank` and `nbProc` parameters . Then `rank` is used as a suffix by `initModule`. This enables to control module numbering from the module code. For instance the `mpirun` launcher command from MPI $\boxed{\text{A VOIR}}$ *un lien vers openmpi* $\boxed{\text{FIN}}$ numbers the various processes it starts. If these processes are also FlowVR modules, they can use the rank set by MPI (retreived calling `MPI_comm_rank` as a name suffix. The fluid example works this way (share/flowvr/examples/fluid/modules/src/fluid.cpp).

The parallel interface also provides the following methods that can be called in a module:

- `bool Parallel::isInitialized()`: Return `true` if the parallel interface is initialized. Initialized by `initModule` if not done before.
- `bool Parallel::isParalle()`: Return `true` if the parallel Interface is activated.
- `bool Parallel::getRank()`: Return the module rank (0 if parallel interface not activated).
- `bool Parallel::getNbProc()`: Return the total number of modules in the group (1 if parallel interface not activated).
- `bool Parallel::close()`: close the parallel interface.

`FLOWVR_MODNAME`, `FLOWVR_RANK` and `FLOWVR_NBPROC` are automatically propagated by the `flowvr-run-ssh` $\boxed{\text{A VOIR}}$ *link* $\boxed{\text{FIN}}$ module launcher. $\boxed{\text{A VOIR}}$ *a completer quand on aura ecrit une section sur les support des commandes de lancement* $\boxed{\text{FIN}}$
$\boxed{\text{A VOIR}}$ *Put here everything you think it is important to include in the docuemention, but that you do not know where to put* $\boxed{\text{FIN}}$
$\boxed{\text{A VOIR}}$ *ajouter de la doc sur: ou placer les codes (binaires des modules par exemple) et les variables d'environnement associÃ©es pour leur accessibilitÃ© (en mode cluster en particulier) parler de la ligne de commande ou placer les sources et les exÃ©cutables des filtres* $\boxed{\text{FIN}}$
Here is the list of source codes a user may find useful to inspect:

- Examples of applications: share/flowvr/examples,
- The various components provided with FlowVR: include/flowvr/app/components/.
- The source code of filters and synchronizers released with FlowVR: flowvrd/src/plugins/sync and flowvrd/src/plugins/filters,

20.7.7 The Application Controller

The control of the execution of a FlowVR application is managed by one special module called a *controller*, automatically loaded when launching the application. The controller is in charge of launching the metamodules and setting the network.
The controller first starts the application's metamodules. Once the modules launched, they register to their local daemon that sends an acknowledgment to the controller. Then, the controller sends to each daemon the list of plugins to load required to implement the FlowVR network.
A VOIR *bruno 2010: still valid but to detailled for user manual.* FIN

20.8 The Module API Factory: registerModule

```
static ModuleAPI* ModuleAPIFactory::registerModule(std::string instancename=
    std::string(""))}
```

This method will register a new module and return the appropriate module API implementation. The module gets the name composed of the concatenation of:

- the name returned by the `FLOWVR_MODNAME` environment variable (see **??**),
- the / separator,
- the value of the `instancename` parameter.

Most of the time the `instancename` parameter can be omitted, and the module gets the name returned by `FLOWVR_MODNAME`.

20.8.0.1 init

```
int ModuleAPI::init(std::vector<Port*>& ports)
```

This method initializes the module and its ports. This method must be called once before entering the loop. The list of ports used by the module can not be modified after this call. No other method of the module API should be called before calling `init`.

20.8.1 Module Binary and Launching Commands

We first need to explain how a module is started. One goal when designing the module API was to be as little intrusive as possible to easily turn any piece of code into a FlowVR module.
Thus the main way to transmit data to a module is through the argument of its command line or through environment variables.
Each time `flowvr-run-ssh` starts a module it sets several environment variables that need to be propagated

- `FLOWVR_MODNAME`: the modules's name inside the application graph.
- `FLOWVR_RANK` and `FLOWVR_NBPROC`: rank is used to identify multiple instances of the process running in parallel.
- `FLOWVR_PARENT`: the PID of the process to attach to. (i.e. the daemon)

These environment variables are automatically calculated by the metamodule at launch time.
As said before (see ??), the `flowvr` executable parses the description of your application's network, and produces intermediate files, when called without the "-x" option.
Those files are named as follows :

- ***applicationName*.net.xml** : An xml file describing the network of the application : the modules, filters, synchronisers, and how they are connected together. You can use the **flowvr-glgraph** utility to view this file as a graph.
- ***applicationName*.cmd.xml** : An xml file containing internal commands for the flowvr deamon, read at launch time.
- ***applicationName*.run.xml** : An xml file containing the launch commands for every module. By default, the launcher is the executable flowvr-run-ssh. It uses an ssh connection to launch distant and local executables. (This is why you should allow incoming ssh connections on your machine for flowvr to run properly). The launcher can also be mpirun.
- ***applicationName*.adl.out.xml** : An xml file reflecting the whole architecture of your application (useful to inspect for instance to check the actual parameter values). You can use the **flowvr-glgraph** utility to view this file as a graph.

Now if you specify the "-x", commands in the *applicationName*.run.xml are parsed, and executed using the flowvr-run-ssh (see subsection 20.8.2) script. Here are commands generated for the Primes example:

```
<commands>
   <run metamoduleid="primes/capture/body">flowvr-run-ssh -v -p -e DISPLAY
       :0 ' host1 ' capture </run>
   <run metamoduleid="primes/visu/body">flowvr-run-ssh -v -p -e DISPLAY :0
       ' host1 ' visu </run>
   <run metamoduleid="primes/compute/body">flowvr-run-ssh -v -p ' host1
      host2 host3 host4 ' compute </run>
</commands>
```

At runtime, those commands become:

```
Command params: -v -e DISPLAY ":0" -e FLOWVR_PARENT "/oneida/xxxx/read:P"
    -e FLOWVR_MODNAME "primes/capture/body" -e FLOWVR_NBPROC "1" -e
    FLOWVR_RANK 0
Command params: -v -e DISPLAY ":0" -e FLOWVR_PARENT "/oneida/xxxx/read:P"
    -e FLOWVR_MODNAME "primes/visu/body" -e FLOWVR_NBPROC "1" -e
    FLOWVR_RANK 0
Command params: -v -e FLOWVR_PARENT "/oneida/xxxx/read:P" -e
    FLOWVR_MODNAME "primes/compute/body" -e FLOWVR_NBPROC "4" -e
    FLOWVR_RANK 0
Command params: -v -e FLOWVR_PARENT "/oneida/xxxx/read:P" -e
    FLOWVR_MODNAME "primes/compute/body" -e FLOWVR_NBPROC "4" -e
    FLOWVR_RANK 1
Command params: -v -e FLOWVR_PARENT "/oneida/xxxx/read:P" -e
    FLOWVR_MODNAME "primes/compute/body" -e FLOWVR_NBPROC "4" -e
    FLOWVR_RANK 2
```

```
Command params: -v -e FLOWVR_PARENT "/oneida/xxxx/read:P" -e
    FLOWVR_MODNAME "primes/compute/body" -e FLOWVR_NBPROC "4" -e
    FLOWVR_RANK 3
```

Once it has been launched, each module connects to the daemon using the `FLOWVR_PARENT` variable, which was previously set by the launcher.
A VOIR *FLOWVR_PWD FLOWVR_PREFIX* FIN
Then, the `flowvr-telnet` script parses the *applicationName*.cmd.xml file and issues commands to the concerned daemons. (start/stop/pause/addobject ... etc)

20.8.2 `flowvr-run-ssh`: a Simple Module Launcher

`Flowvr-run-ssh` is the default FlowVR launcher. It connects to given distant machines, set environment variables, and launches the binary it has been given as argument.

```
Usage: flowvr-run-ssh [-v] [--path path] [-l login] [-e VAR VALUE] [-x VAR
    ] [-s] [-p] [-m] [-b] hostlist command
  -v          : verbose
  --path path : changes to path before executing the command.
  -d path     : (DEPRECATED) changes to path before executing the command.
  -e VAR VALUE : sets the variable VAR to VALUE in the environment of the
      command.
  -x VAR      : propagates the variable VAR in the environment of the
      command.
  -s          : does not set FLOWVR_RANK and FLOWVR_NBPROC (sequential mode)
       [default if one host].
  -p          : sets FLOWVR_RANK and FLOWVR_NBPROC (parallel mode) [default
      if several hosts].
  -l login    : specifies the user to log in as on the remote machine.
  -m          : disable FLOWVR_PLATFORM substitution in path.
  -b          : run in background.
```

A VOIR *Put here everything you think is important to include in the documentation, but that you do not know where to put* FIN

Acknowledgment

The development of FlowVR has been partly funded by the RNTL project GEOBENCH, the ANR grant ANR-07-CIS-003 (FVNANO project) and the ARN grant ANR-06-MDCA-003 (DALIA project).