Norwegian Defence Research Establishment

24/00556

FFI-RAPPORT

Superflow

– an efficient processing framework for modern C++

Ragnar Smestad
Martin Vonheim Larsen
Trym Vegard Haavardsholm

Superflow

– an efficient processing framework for modern C++

Ragnar Smestad
Martin Vonheim Larsen
Trym Vegard Haavardsholm

Norwegian Defence Research Establishment (FFI) 8 April 2024

Keywords
C++ (Programmeringsspråk)
Rammeverk
Autonomi
Systemarkitektur

FFI report
24/00556

Project number
1688

Electronic ISBN
978-82-464-3529-9

Approvers
Rikke Amilde Seehuus, *Research Manager*
Håkon Storli Andersen, *Director of Research*
Source code and license are approved by Ragnar Smestad

The document is electronically approved and therefore has no handwritten signature.

Source code
Source code is published at https://github.com/ffi-no/superflow.

(U) Summary

This report describes *Superflow*, a generic data processing framework written in C++.

Superflow is made for creating and running flexible processing graphs, where the nodes are individual processing stages, and the edges are data flows between nodes.

Processing stages are represented by concurrent processing elements, called *proxels*. Each proxel is an abstraction that encapsulates a self-contained part of the processing pipeline, such as specific algorithms, file writers or even parts of a graphical user interface (GUI). Data flows between proxels are realised through connected *ports*, which are objects that provide different, type-safe communication schemes through a common interface. A proxel typically has input ports for receiving or requesting data and output ports for providing results. The proxels and ports are managed in a container class called *Graph*, which offers a convenient way to start and stop the processing graph, add and connect proxels, monitor the status of the processing graph, and more.

In order to simplify the creation of a graph and to be flexible to changes in content and structure, Superflow provides tools for parsing configuration files that contains lists of proxels, parameters and connections. These can be used to create and start graphs automatically without recompiling any code.

The design of Superflow makes it simple to combine different sensors, algorithms, and processing stages and to dynamically reconfigure established processing pipelines. The framework supports parallel processing, branching, and merging of pipelines as well as synchronisation through barriers and latches. This is all performed in an efficient, type-safe, and extensible communication scheme based on modern C++.

This report contains a description of the main components in Superflow, followed by a short tutorial that will get you started on using Superflow in your own applications.

(U) Sammendrag

Denne rapporten beskriver *Superflow*, et generisk rammeverk for dataprosessering skrevet i C++.

Superflow er laget for å konstruere og kjøre dataprosesseringsgrafer, hvor nodene i grafen er individuelle prosesseringssteg, og kantene representerer dataflyt mellom nodene.

Prosesseringsstegene representeres ved hjelp av en egen datatype, kalt *proxel* (processing element), slik at hver proxel er en abstraksjon som innkapsler en selvstendig del av prosesseringskjeden. En proxel kan for eksempel inneholde en spesiell algoritme, lesing og skriving til fil eller presentasjon av et grafisk brukergrensesnitt (GUI). Hver proxel kan kjøres i en egen tråd, slik at ulike prosesseringer foregår samtidig og i parallell. Dataflyten mellom proxler realiseres gjennom sammenkoblede *porter*. Porter er egne klasser som tilbyr ulike former for multitråd- eller multiprosesskommunikasjon bak et felles grensesnitt. En proxel har typisk input-porter for å motta eller å be om data og output-porter for å levere fra seg resultater. Proxlene administreres ved hjelp av en egen klasse kalt *Graph*, som gjør at man enkelt kan starte og stoppe prosesseringen, legge til og koble sammen proxler, overvåke gjeldende status for grafen osv.

Superflow tilbyr verktøy for å lese inn konfigurasjonsfiler, noe som forenkler oppsettet av grafer og gjør det enkelt å tilpasse grafens innhold og struktur. En konfigurasjonsfil definerer hvilke proxler som skal inngå i grafen, hvordan portene kobles sammen, og eventuelle parametere som justerer proxlenes virkemåte. Dette kan brukes til å generere og starte en graf automatisk, uten å måtte rekompilere noe kode.

Virkemåten til Superflow gjør det enkelt å sette sammen ulike sensorer, algoritmer og prosesseringssteg på en dynamisk måte. Rammeverket støtter parallell prosessering med forgrening, sammenfletting og synkronisering av prosesseringsløyper. Alt dette er kodet inn i et effektivt, typesikkert og utvidbart system basert på moderne C++.

Rapporten inneholder en beskrivelse av hovedkomponentene i Superflow, etterfulgt av et kort praktisk eksempel som hjelper deg å komme i gang med å bruke Superflow til dine egne anvendelser.

Contents

The Superflow is the informational space between universes. It also serves as the place where dreams and ideas come from, and from where telepathy operates. (. . .). The laws of physics do not apply in the Superflow [1].

1 Introduction

This report describes *Superflow*, a generic data processing framework written in C++.

Superflow is developed as the core software component in research and experimentation with perception for autonomous vehicles at FFI. A very important part of this research is to explore different approaches, and to run tests in realistic scenarios on real autonomous platforms. In such exploratory work, it is crucial that the software architecture is lean and efficient, robust to changing conditions, flexible to new ideas and (almost) always working.

The research follows an *agile* development approach [3], which embraces changing requirements and encourages solutions to evolve over time by developing software in small increments between working versions. In software development terms, *coupling* is a measure of how strongly one software element depends on other elements, while *cohesion* is a measure of how functionally related the operations of a software element are. We want to reduce the impact of change through *low coupling*, while keeping objects focused, understandable, and manageable through *high cohesion* [4, 5]. These were the main design principles in the development process that lead to the framework presented here.

In Superflow, data flow and data processing are decoupled and focused through *port* and *processing element (proxel)* abstractions. This makes it simple to combine different sensors, algorithms and processing stages, and to dynamically reconfigure established processing pipelines. The framework supports parallel processing, branching and merging of pipelines, and synchronisation through barriers and latches. This is all performed in an efficient, type safe and extensible communication scheme based on modern C++. There are other methods and other libraries available for data processing, which also offer low coupling between the software elements in the pipeline. In our opinion, however, the main reasons to select Superflow are:

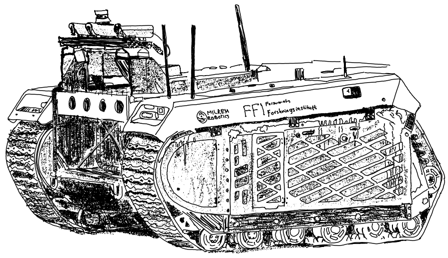
- High level of flexibility and efficiency
- Few additional requirements (the core module has no dependencies)
- Strongly typed, non-serialised, possibly zero-copy data transfer through ports
- No custom build tool or bloated ecosystem, just pure modern CMake

Although originally developed for real-time processing on autonomous vehicles, we are certain that Superflow will be useful in a multitude of other interesting applications as well.

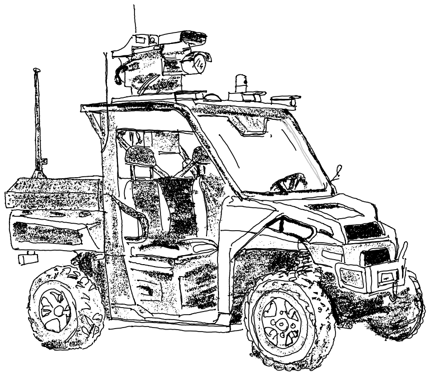
We will in the following chapters first give an overview of the main components in Superflow, before we finish with an example of how it can be applied through a short and straightforward tutorial. You are welcome to contact the authors for more information on Superflow and further developments.

The code is open-source with MIT license, and available on https://github.com/ffi-no/superflow.

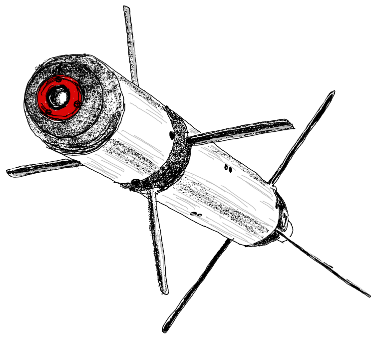
(a) *The 'Valkyrie' swarm system with multiple* Flamingo *UAVs.*



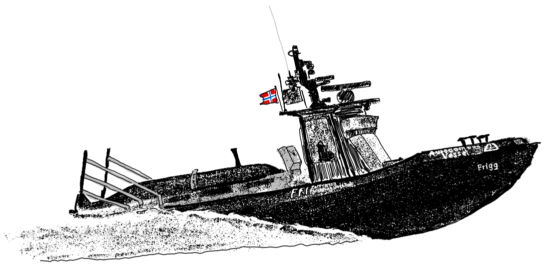
(b) The UGV 'Tor'



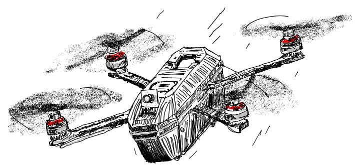
(c) The UGV demonstrator 'Olav'



(d) The portable short range missile 'Ravn'



(e) The Odin *class USVs 'Frigg' and 'Odin'*



(f) The UAV countermeasures 'Interceptor' drone

Figure 1.1 Superflow was originally developed as the backbone of the scene analysis framework 'Warpath' [2], the perception module in FFI's work with autonomous systems. Here are examples of platforms and hardware where Superflow and Warpath has been deployed.

2 Superflow

In this chapter we go through the specifics of Superflow. We will explain the basic concepts you need to understand before using it for the first time. In some of the subsections we will take a closer look at the internal workings, which is useful if you need to extend Superflow to suit your own needs. For most users, however, the built-in features will probably suffice.

2.1 Overview

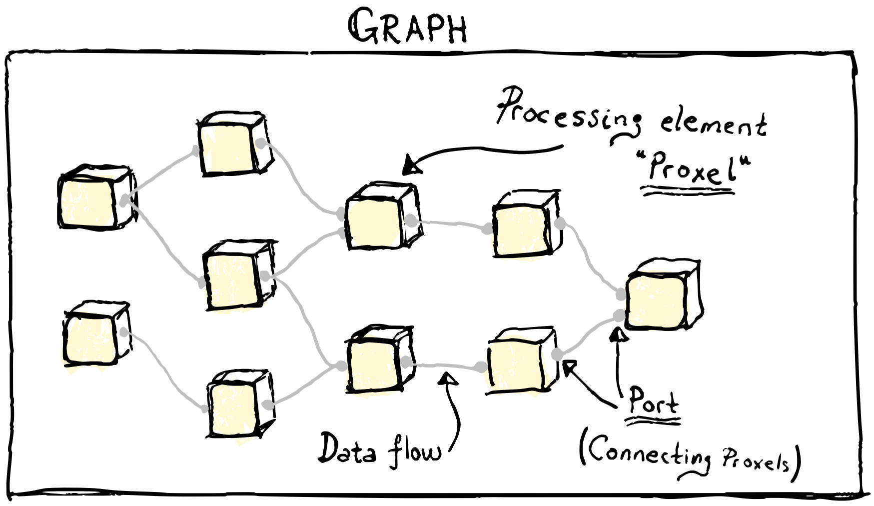


Figure 2.1 Example of a Superflow processing graph. The nodes are the processing stages, called proxels, and the edges represent the data flow through ports.

Superflow is a C++ library made for creating and running flexible processing graphs, where the nodes are the individual processing stages in the overarching processing system, and the edges represent the data flow between stages.

The processing stages are represented by concurrent *proxels*, that encapsulate self-contained parts of the processing system, such as specific algorithms, file I/O or even graphical user interfaces (GUI). The data flows are represented by connected *ports*, which provide different, type-safe communication schemes. A proxel may typically have input ports for receiving or requesting data, and output ports for providing results. By connecting output ports to compatible input ports, we can construct complex processing graphs, with branches, merges and even feedback loops. Since the proxels are conditionally independent through ports, it is easy to change or add new proxels, without needing to know or change how the other proxels are implemented.

Figure 2.1 shows an example of a processing graph with a non-linear pipeline structure, where each processing stage runs concurrently, and data flows from *producers* to *consumers* of data. We will later see that many other types of data flow is also possible.

To use Superflow in your application, you will need to implement your own `Proxel`s (section 2.2) and connect them with `Port`s (section 2.3), both typically managed in a `Graph` object (section 2.4.1).

You might want to implement a `Factory` for each `Proxel` (section 2.4.2), so that you can utilise the included *yaml* module (section 2.5) for automatically setting up `Graph`s with yaml-files, and you can proceed with the *loader* module (section 2.6) to enable dynamic loading of precompiled proxel libraries. In some cases, you might also want to extend the functionality in Superflow by implementing new `Port` functionality to work with your proxels. By reading this chapter, you will hopefully be able to understand how the main components in Superflow work, and how you may use them to your advantage. The tutorial chapter will build upon these basics to give an example of how you may implement your own minimal Superflow processing graph.

2.2 Proxel

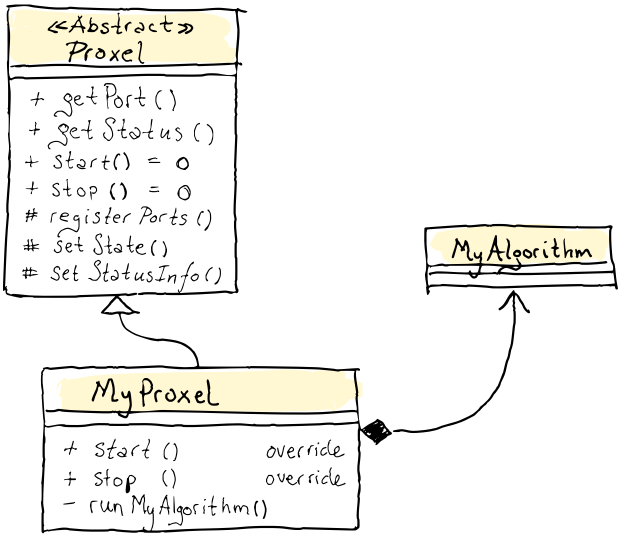


Figure 2.2 A custom proxel extending the abstract Proxel class. The internal processing is typically maintained outside the proxel as a free function or in a library.

The processing encapsulation provided by a proxel is beneficial from at least two points of view. Externally, we require the proxel to take input data and produce output data. The internal processing may be replaced if desired, but the inputs and outputs should stay the same. Internally, however, the source and destination of data is irrelevant. The only requirement is that valid input is available for processing and that the expected result is produced. The proxel's purpose is thus to isolate the inner processing from the rest of the graph.

In Superflow, all proxels must be derived from the abstract `Proxel` class. A typical way to create a new proxel is to develop the inner processing as a standalone library or function, and then create a thin wrapper class that extends `Proxel`. A diagram showing this relation is shown in fig. 2.2. The new proxel is required to override the methods `start()` and `stop()`. When someone calls `start()`, the proxel is expected to prepare and launch the actual processing job, wait for data and to keep processing until `stop()` is called.

In order to facilitate easy connection of ports, `Proxel` provides the methods `getPort()` and `registerPorts()`. When extending `Proxel`, you should call `registerPorts()` to assign a string-based name to each of your ports. When someone wants to connect to your proxel, they call `getPort(port_name)` which returns a pointer to the matching port.

The abstract `Proxel` class also defines methods for updating and retrieving the current status of the proxel. These are explained in section 2.2.1.

A simplified example of a custom proxel is given in listing 2.1, showing the most important aspects without going into details. Notice the use of `start()`, `stop()` and `registerPorts()`, along with a processing loop that utilises a function that is maintained outside of the proxel. The details about ports are omitted, as this will be the topic of section 2.3. You should look to the tutorial in chapter 3 for more comprehensive (and compiling) examples.

2.2.1 Monitoring the state of a proxel

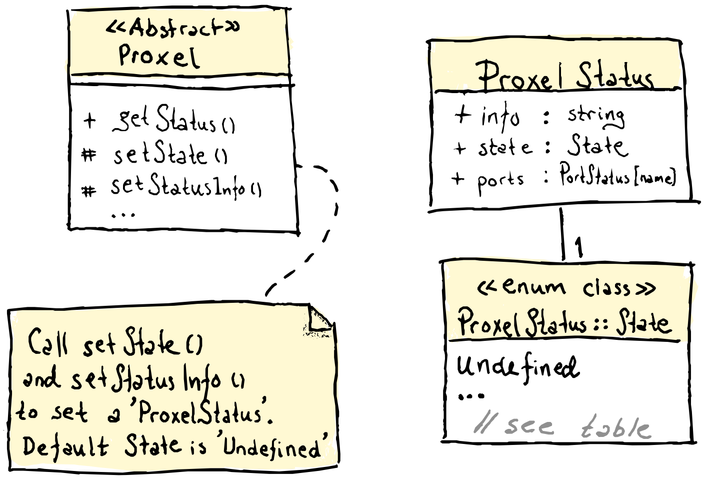


Figure 2.3 ProxelStatus and relation to protected `Proxel`-methods.

In order to facilitate monitoring of a processing graph, the `Proxel` class contains three methods that are used to set and query the current state of a proxel:

- `public getStatus()`
- `protected setState(State state)`
- `protected setStatusInfo(std::string info)`

The `public` method `getStatus()` returns a `struct ProxelStatus`, which contains status attributes of the proxel. The method relies on receiving data via the two other `protected` `Proxel`-methods, namely `setState()` and `setStatusInfo()`. These are meant to be used by the proxels themselves whenever new information is available. A schematic overview is shown in fig. 2.3.

The method `setState()` accepts a `ProxelStatus::State`. This enumerated type defines a set of predefined states that a proxel can be in. Before `setState()` is called for the first time, the default state is `State::Undefined`. Whenever the state of the proxel changes, it should call `setState()` with *(a) Header file*

```
#pragma once
#include "superflow/proxel.h"

class MyProxel : public Proxel
{
public:
  MyProxel();
  void start() override;
  void stop() noexcept override;

private:
  void runMyAlgorithm();

  // Declare ports here
  /* ... */
};
```

(b) Source file

```
#include "my_proxel.h"
#include "my_algorithm.h"

void MyProxel::start()
{
  runMyAlgorithm();
}

void MyProxel::stop() noexcept
{ /* make sure to stop 'runMyAlgorithm' here */
}

MyProxel::MyProxel()
    : my_port1_{/* create my_port1_ here */}
    , /* initialize other ports and stuff here */
{
  registerPorts(/* register ports here*/);// Important!
}

void MyProxel::runMyAlgorithm()
{
  while (/* new data is available */)
  {
    /* fetch new data */
    const auto result = myAlgorithm(data);// Available from my_algorithm.h
    /* send result */
  }
}
```

Listing 2.1   A simplified example of a custom proxel.

an appropriate state. The possible values are listed in table 2.1. A common pattern is to call `setState(State::AwaitingInput)` when the proxel is started, `setState(State::Running)` when the first input is received, and `setState(State::Paused)` when the proxel is stopped. See section 3.1.2 for an example of this.

With `setStatusInfo(std::string info)`, the developer can provide a textual description of the proxel's current status. A typical usage is to report the average load of the proxel, given as an active/idle ratio, together with the average active processing time, given in seconds. This type of timing stats can be computed by the `ProxelTimer` helper class (see section 2.7.1). Other examples of usages are to report the number of current tracks held by an object tracking proxel, or the number of connections held by a radio communication proxel.

The final field of `ProxelStatus` reports the status of the ports as a `PortStatus`, which tells the number of connections for a given port, together with the number of 'transactions' performed by that port. These are provided automatically by the port, without any action needed from the proxel developer.

`AwaitingInput`	Active state, but is waiting for input
`AwaitingRequest`	Idle state, waiting for requests
`AwaitingResponse`	Request sent, waiting for response
`Crashed`	Process died
`NotConnected`	None of the ports are connected
`Paused`	Stopped working
`Running`	Currently working
`Unavailable`	Not able to function normally (but is not in an error state)
`Undefined`	No state is set.

Table 2.1 ProxelState

2.3 Port

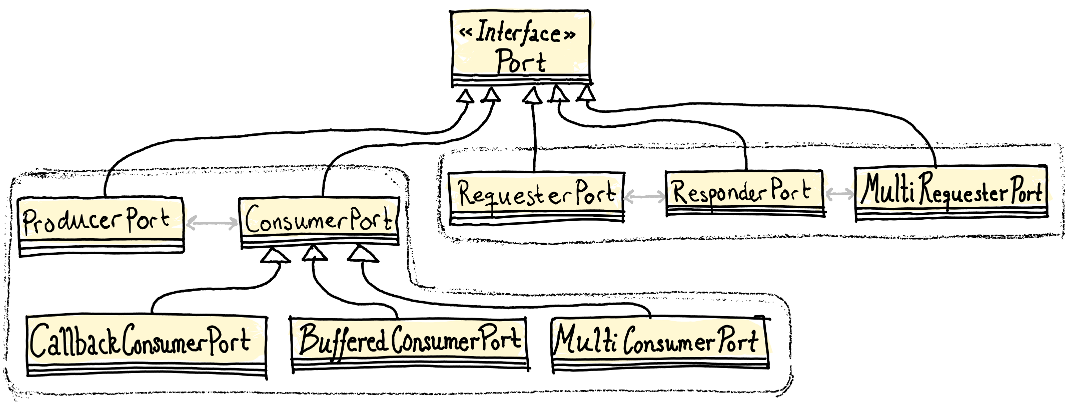


Figure 2.4 Ports in Superflow, grouped by data exchange pattern. All ports are derived from the `Port` interface.

In order for proxels to exchange data with other proxels, they utilise another key component of Superflow, namely `Port`s. At the top level, `Port` is just an interface, but Superflow comes ready with quite a collection of implemented `Port`-variants. These ports are grouped into two main categories. The first category focuses on stream based data exchange, commonly known as a producer/consumer pattern [6]. The other category is built around a request/response type of behaviour, which means that one proxel can actively request something from another proxel. The available port types and concepts are listed in tables 2.2 to 2.4, and the details about the different ports will be the topic of the next sections. The relation between available ports is shown in fig. 2.4.

The built-in ports should satisfy most needs, but it is also possible to create completely custom ports. The classes in Superflow that extend `Port` must all know what kinds of other derived `Port`s they are compatible with, and they should throw an error if an unsupported connection is attempted. Which specific mechanisms in a port pair that makes it possible to connect or to exchange data is of no concern to either the `Proxel` class or to the `Port` interface. The generic concept of ports is illustrated in fig. 2.5, showing a derived `Port`-pair that first connects using the required `connect()` method and then communicates by using some agreed-upon scheme.

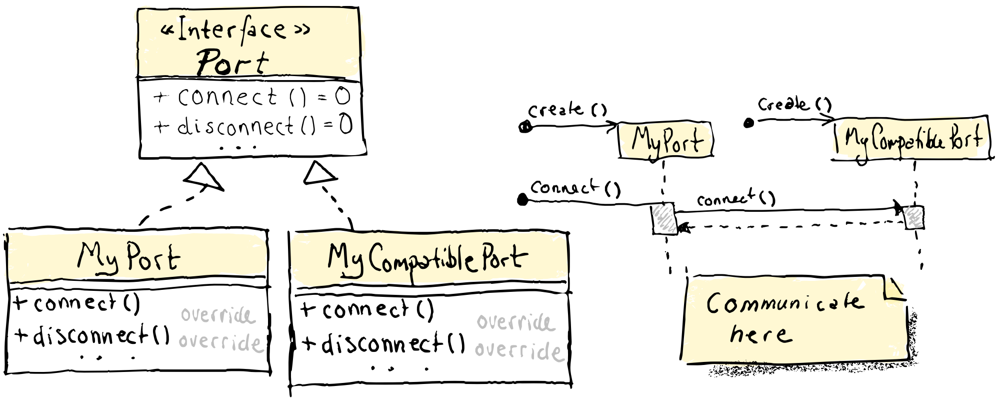


Figure 2.5 Example of derived `Port`-types. When a connection is set up using the required `connect()` method, the ports are ready to communicate using some agreed-upon scheme.

`ProducerPort<T>`	Push data to one or many consumers with no response required
`CallbackConsumerPort<T, P>`	Receive data using a callback function
`BufferedConsumerPort<T, P, M>`	Receive data by popping from a buffer of configurable size
`MultiConsumerPort<T, M>`	Receive data from several producers with a single pop
`RequesterPort`	Request data by providing arguments if necessary
`MultiRequesterPort`	Request data from several responders simultaneously
`ResponderPort`	Respond to a request by returning a value of a given type

Table 2.2 Port types

ConnectPolicy::Single	Only allow one ProducerPort to connect.
ConnectPolicy::Multi	Allow multiple ProducerPorts to connect.

Table 2.3 ConnectPolicys *for* ConsumerPorts

GetMode	Description	Available for
GetMode::Blocking	Attempts to retrieve data from the buffer are blocking, so that the consumer will wait until new data is added to the buffer. The wait will be aborted only if the consumer is deactivated.	All ConsumerPorts
GetMode::Latched	If the buffer is empty, latched mode will not block but instead return latest data again.	All ConsumerPorts
GetMode::ReadyOnly	When connected to multiple producers one would usually wait for all of them to produce data before returning. This mode fetches only from ready producers, i.e. not necessarily all.	MultiConsumerPort
GetMode::AtLeastOneNew	Similar to latched, but blocks until at least one of the producers have new data.	MultiConsumerPort

Table 2.4 GetModes *for* ConsumerPorts

2.3.1 Producer/consumer ports

In a typical processing pipeline, data will flow along a sequence of processing stages. In Superflow, this flow is implemented using `ProducerPort`s and `ConsumerPort`s.

The `ProducerPort<T>` delivers data of a given type `T` from some producer of that data, which is typically (but not necessarily) a proxel. You will send data through the `ProducerPort` using the `send()` method, which then passes the data on to any connected consumers. The `ProducerPort<T>` is able to connect and pass on data to any port that extends the `ConsumerPort<T>` interface. It is required that `T` is the same at both the sending and receiving side. There are currently three types of `ConsumerPort`s, with behaviours further customisable through template parameters:

- `CallbackConsumerPort<T, ConnectPolicy>`
- `BufferedConsumerPort<T, ConnectPolicy, GetMode>`
- `MultiConsumerPort<T, GetMode>`

CallbackConsumerPort

With a `CallbackConsumerPort`, a given *callback function* will be called each time a `ProducerPort` pushes a reference to new data. The function to be called is passed to the constructor of `CallbackConsumerPort`. A `ConnectPolicy`, which can be either `Single` or `Multi`, defines whether one or more `ProducerPort`s may connect and push data to the `ConsumerPort`. The default is `ConnectPolicy::Single`, in which case the port will throw an exception if more than one port attempts to connect.

The following example shows two common usage patterns, both using a lambda as the callback function.[1]

```
// Example of CallbackConsumerPort.
// Assume that we have declared a port pointer
std::shared_ptr<CallbackConsumerPort<std::string>> my_port;

// (1) Process the data directly in a lambda:
my_port = std::make_shared<CallbackConsumerPort<std::string>>(
  [](const std::string& input) { /* process input */ });

// (2) Use a lambda to pass the input to a separate function:
void myCallbackFunction(const std::string&)
{ /* process input */ }

my_port = std::make_shared<CallbackConsumerPort<std::string>>(
  [](const std::string& input) { myCallbackFunction(input); });
```

A sequence diagram with the typical usage of a `CallbackConsumerPort` is shown in fig. 2.6. The proxel that owns the `CallbackConsumerPort` is usually passive, meaning that no processing loop is actively fetching data. Instead, all work performed in the proxel happens through invocations of the callback function. No thread safety mechanisms are built into the port itself, so that must be handled explicitly. The `CallbackConsumerPort` will block the upstream `ProducerPort` while the callback is executed. Therefore it is good practice to keep the callback function as lightweight as possible, unless such blocking is intended. If you want to perform time-consuming work in a separate thread in order to avoid such blocking, consider using a `BufferedConsumerPort` instead.

A complete example of a `Proxel` with a `CallbackConsumerPort` can be found in the tutorial (section 3.1.4).

[1] Read about why you should prefer lambdas to `std::bind` in Scott Meyer's "Effective Modern C++", item 34. [7]

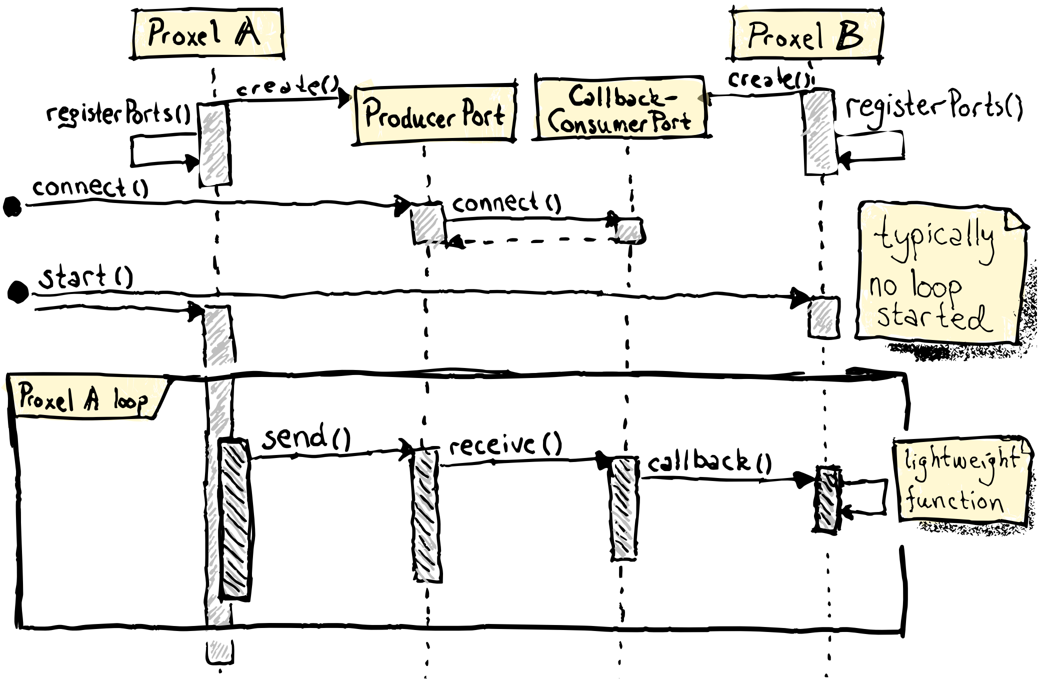


Figure 2.6 CallbackConsumerPort sequence diagram.
Data from the ProducerPort in 'Proxel A' triggers a callback in 'Proxel B'.

BufferedConsumerPort

A `BufferedConsumerPort` has its own thread safe FIFO[2]-queue of configurable size, that data from the `ProducerPort` is continuously added to. If the buffer gets full, the oldest data will be discarded.

On the consuming end, you can extract data in three different ways: Either using the `getNext()` method, using the <<-operator[3], or using the built-in *iterator*. We definitely recommend that you use the iterator approach in combination with a range-based `for`-loop, as it takes care of all the edge cases of empty or deactivated buffers. However, some use cases may require direct usage of `getNext()`. Examples of all usages can be found below.

The behaviour of an empty buffer is configured using the last template parameter, `GetMode`, which can be `Blocking` or `Latched`. When `Blocking` mode is selected, the consumer will be blocked until the buffer is no longer empty, the producer disconnects or the port is deactivated. `Latched` mode prevents blocking by returning the last message again and again, until new data is available. This means that `Latched` only blocks if no data has ever been inserted into the buffer. The default `GetMode` is `Blocking`.

The `ConnectPolicy`, which can be `Single` or `Multi`, controls whether one or multiple `ProducerPorts` are allowed to push data to the `BufferedConsumerPort`. In `Single` mode, the `BufferedConsumerPort` will throw an exception if multiple ports attempt to connect. If `Multi` mode is used, data from all connected `ProducerPorts` will be added to the same queue. In this case the data elements will be extracted in the order they arrive, regardless of which producer they came from.

[2] First in, first out.
[3] This is typically called a *stream operator*, and commonly used throughout the C++ standard library.

The example below illustrates different ways to retrieve data from a `BufferedConsumerPort`. First, usage of the stream operator is shown at line 6. Then we see how to use the range-based for loop to continuously retrieve data from the buffer using the built-in iterator (line 11). Since we always use a `Port::Ptr` (a `std::shared_ptr`), it must be dereferenced. Finally, we see that the `getNext`-method could also be called directly (line 15). A sequence diagram with typical usage of a `BufferedConsumerPort` is shown in fig. 2.7.

```
1  // Different ways to retrieve data from a BufferedConsumerPort.
2  constexpr size_t buffer_size{10};
3  auto my_port =
4    std::make_shared<BufferedConsumerPort<std::string>>(buffer_size);
5
6  // (1) Stream operator
7  std::string input;
8  input << *my_port;
9  const bool success = *my_port;// using the port's operator bool()
10
11 // (2) Range-based for loop
12 for (const std::string& input: *my_port)
13 { /* process input */ }
14
15 // (3) Nothing fancy
16 std::string input;
17 const bool success = my_port->getNext(input);
```

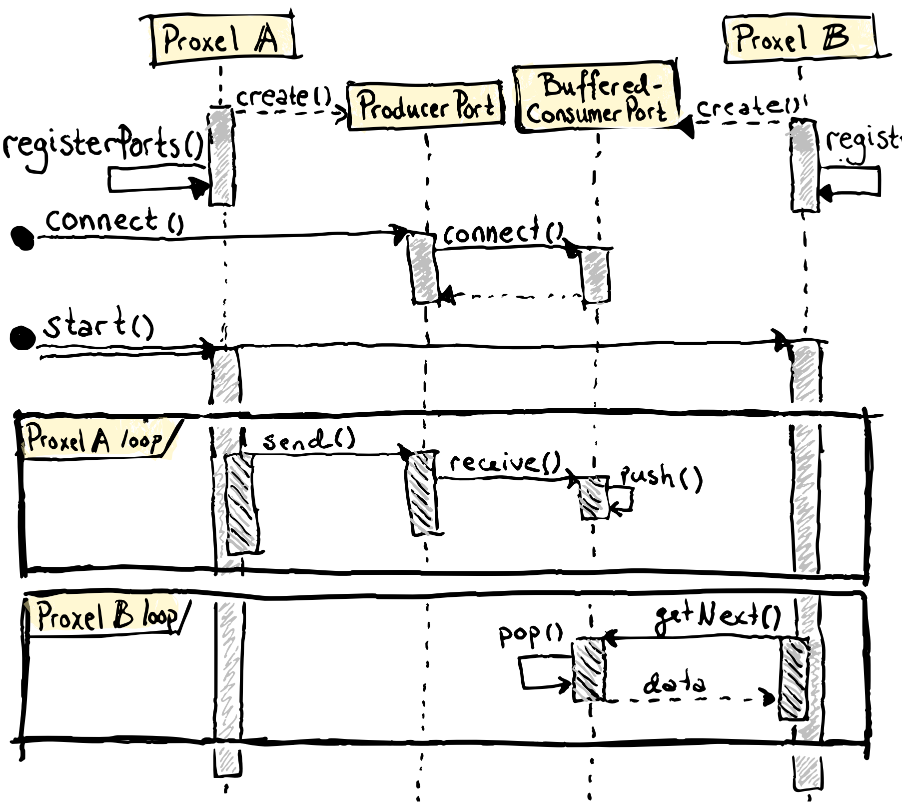


Figure 2.7 BufferedConsumerPort sequence diagram.
Data from the ProducerPort in ‘Proxel A’ is pushed onto a queue in ‘Proxel B’, which pops data from the queue for processing.

MultiConsumerPort

The `MultiConsumerPort` is a special implementation of `BufferedConsumerPort` with multiple producers, using a *dedicated* buffer for each connected `ProducerPort`. One data element from each producer can then be retrieved simultaneously as a `std::vector` of data elements. In contrast, a normal `BufferedConsumerPort` would only return a single element at a time.

`MultiConsumerPort` supports `GetMode`s `ReadyOnly` and `AtLeastOneNew` in addition to `Blocking` and `Latched`. When `Blocking` is used, `getNext()` (and other variants) will yield a vector with *one* element from *each* producer, blocking until such elements are available. No element will be returned twice. In `Latched` mode, `getNext()` will also return a vector with *one* element for *each* producer, but it will return the last element again from a given producer if no new element has been received. Thus, in `Latched` mode, `getNext()` will only block initially, until one element has been received from each connected producer. Subsequent calls to `getNext()` will then return immediately. The `AtLeastOneNew` mode is similar to `Latched`, but causes `getNext()` to block until at least one of the connected producers have provided a new element. The `ReadyOnly` mode does not block. It will return a vector with one element from each producer that has sent a new element since the last call to `getNext()`. If none of the producers has sent a new element, the vector will be empty.

Listed below is a short example with a `MultiConsumerPort`. Like `BufferedConsumerPort`, it supports the range-based for loop for retrieval of data (line 11). Since the data is contained in a `std::vector`, another for-loop is utilised for processing the elements from each `ProducerPort` separately.

```
1  // Different ways to retrieve data from a MultiConsumerPort.
2  auto my_port = std::make_shared<MultiConsumerPort<int>>();
3
4  // (1) Using getNext
5  std::vector<int> items;
6  my_port->getNext(items);
7  for (const int item: items)
8  { /* process item */ }
9
10 // (2) Using range-based for loop
11 for (const auto& data: *my_port)
12 {
13   for (const int input: data)
14   { /* process input */ }
15 }
```

The sequence diagram in fig. 2.8 shows a setup with two `ProducerPort`s connected to a `MultiConsumerPort`. Data gets pushed to the buffers asynchronously, and is retrieved simultaneously by the receiving proxel.

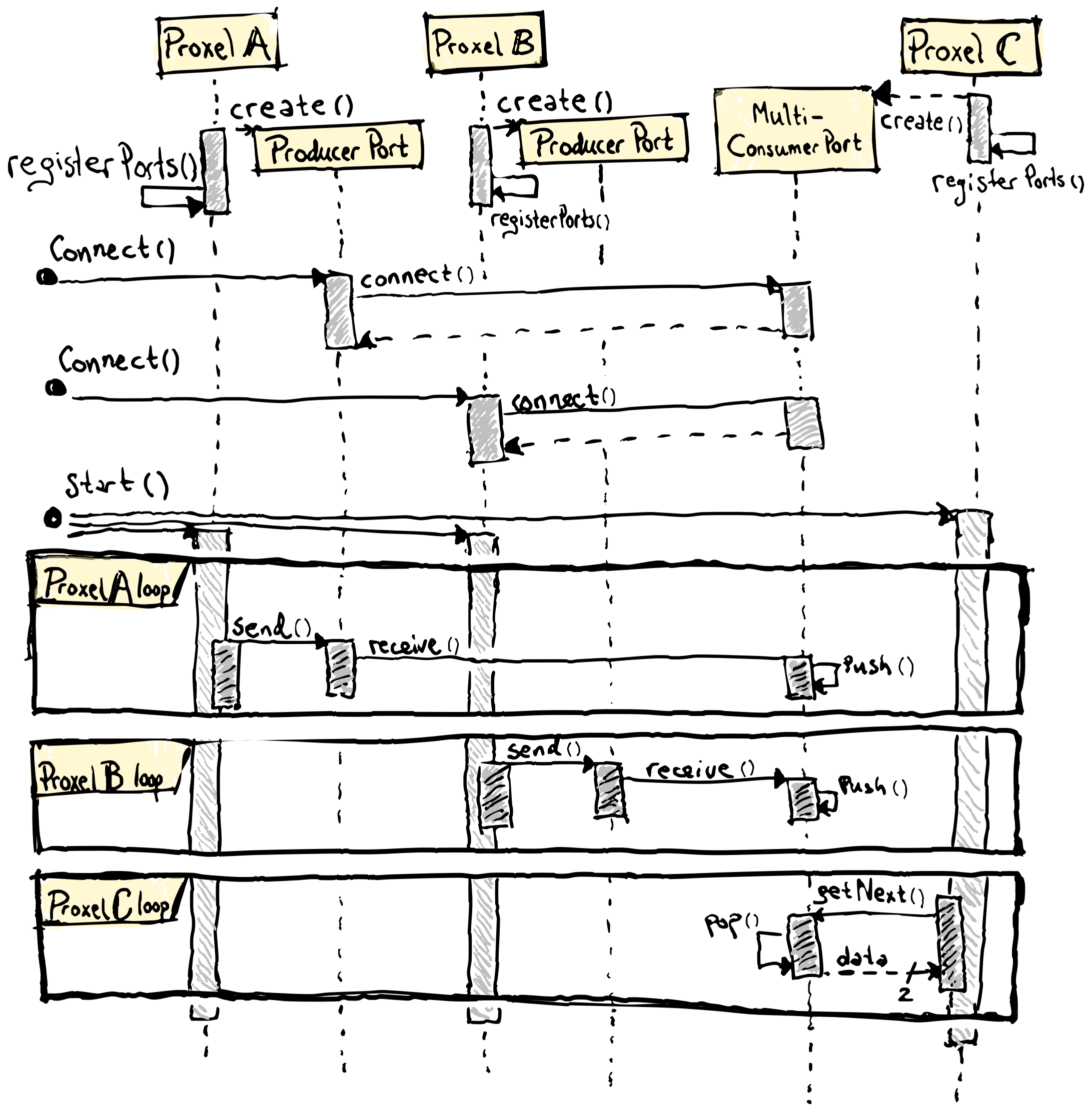


Figure 2.8 MultiConsumerPort sequence diagram.
Data from several ProducerPorts (here in 'Proxel A' and 'Proxel B') is pushed onto separate queues in 'Proxel C', which pops a vector of data for processing.

2.3.2 Requester/responder ports

The producer/consumer pattern described in the previous sections implements a clear separation where the producer should have no concerns about who, how or when someone receives the data. In other situations though, it is useful to actively request specific data through a port connection, in what is commonly known as a *request/response* pattern. This kind of communication is well suited for calls that you only need to do occasionally and that take a bounded amount of time to complete. Examples may be to configure the behaviour of a proxel through parameter updates, or to reset its internal state. Another typical use case is to extract some information that lies within the domain of a specific proxel, but which is not available or reasonable to publish through a `ProducerPort`. The requester can e.g. perform a query with a timestamp, and the responder can return an unambiguous result for that given timestamp only, instead of, say, continuously publishing values corresponding to *all* possible timestamps.

Superflow offers three types of ports for request/response communication:

- `RequesterPort<ReturnValue(Args...)>`
- `MultiRequesterPort<ReturnValue(Args...)>`
- `ResponderPort<ReturnValue(Args...)>`

The implementation of the communication is simply that the requester calls a given function within the responder. This is similar to the *callback function* used in a `CallbackConsumerPort`, but this time with a function signature that allows multiple arguments as well as a return value.

Listed below is an example where a `RequesterPort` requests a `std::string` by passing an `int` to a `ResponderPort`. Note how the templated function signature `<std::string(int)>` must match on both `Port`s. The function that produces the response is passed to the constructor of `ResponderPort`.

```
// Example that shows usage of Requester- and ResponderPort
auto requester = std::make_shared<RequesterPort<std::string(int)>>();
auto responder = std::make_shared<ResponderPort<std::string(int)>>(
  [](int a) { return std::to_string(a); } // the responder's response
);

requester->connect(responder);
const std::string value = requester->request(42); // value is "42"
```

Figure 2.9 shows a sequence diagram with two proxels connected by a `RequesterPort` and a `ResponderPort`. While ‘Proxel A’ goes about its business, ‘Proxel B’ requests a value that is computed with some function that lies inside of ‘Proxel A’.

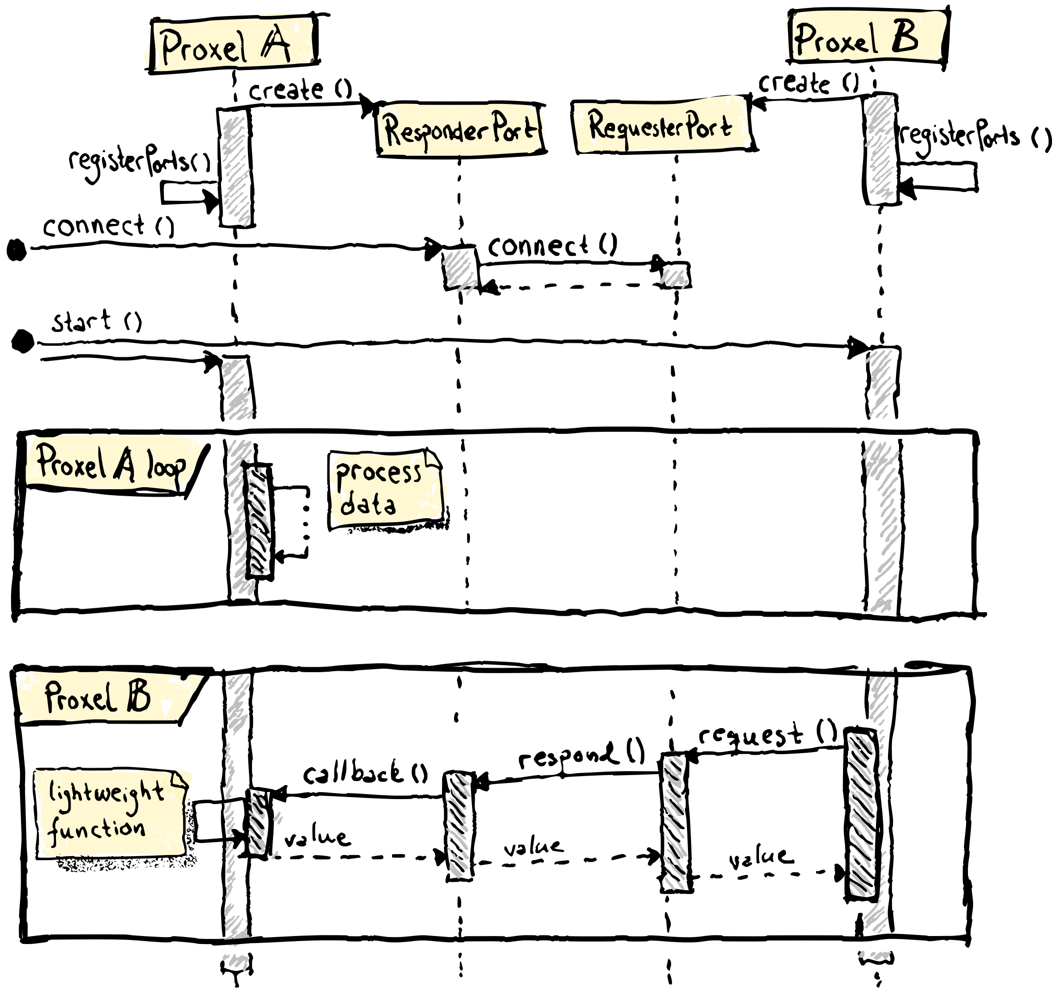


Figure 2.9 Requester-/ResponderPort sequence diagram.
While 'Proxel A' goes about its business, 'Proxel B' requests a value that is computed with some function that lies inside of 'Proxel A'.

MultiRequesterPort

The request/response relation is one-to-one, unless a `MultiRequesterPort` is used. A `MultiRequesterPort` is able to simultaneously request data from several `ResponderPorts` and retrieve the result as a `std::vector`. The call is blocking until all `ResponderPorts` have responded.

A sequence diagram is displayed in fig. 2.10. 'Proxel B' and 'Proxel C' is continuously doing some unrelated work, while 'Proxel A' sporadically requests a value from both of them, again calling a function within the respective proxel. When both requests are completed, 'Proxel A' receives the returned values in a `std::vector`.

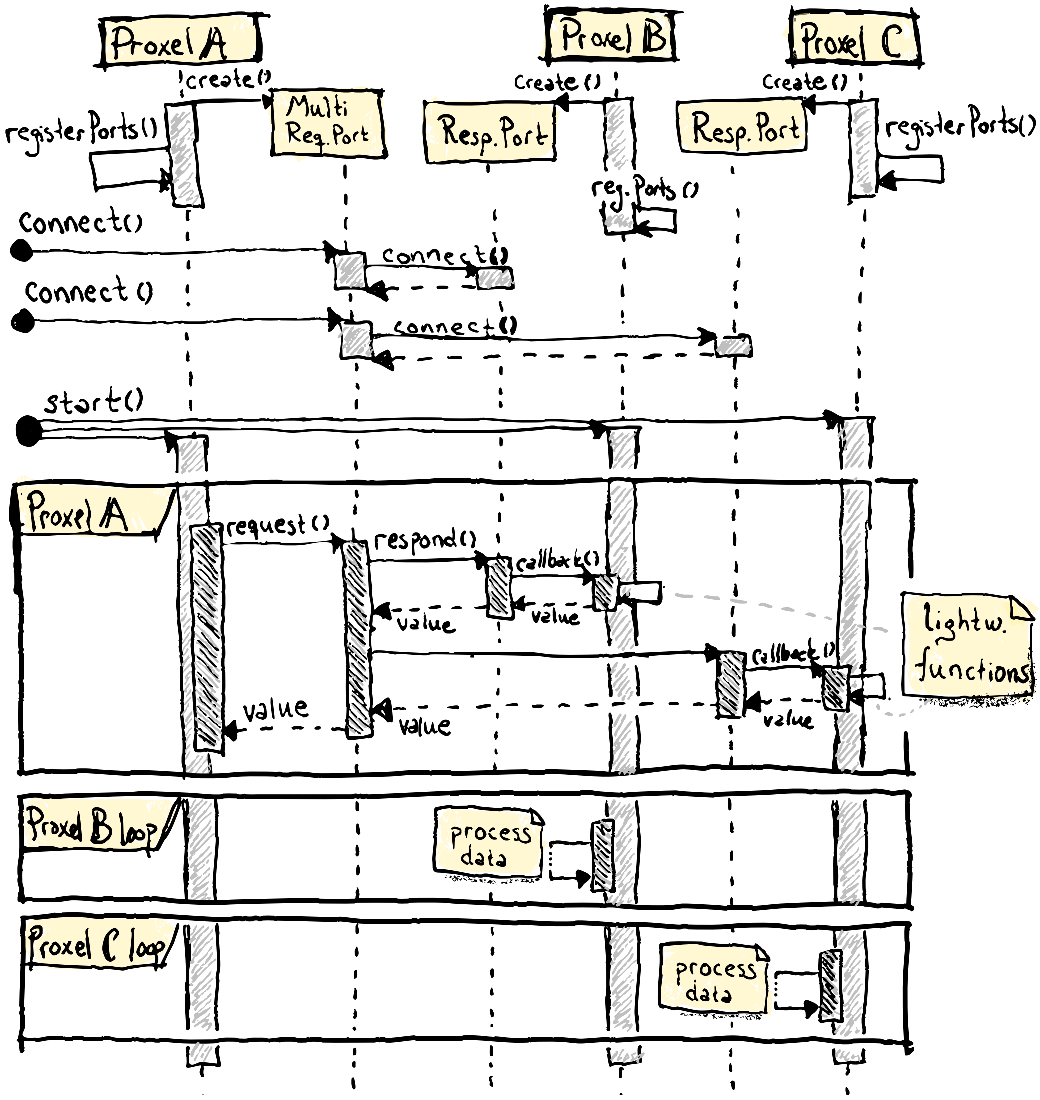


Figure 2.10 *MultiRequesterPort sequence diagram.*
'Proxel A' sporadically requests values from 'Proxel B' and 'Proxel C', which are concurrently doing unrelated work.

2.4 Graph

Proxels and ports are the very core of Superflow, and even when creating the most complex processing graphs, they are all that you really need. When using these building blocks, however, it soon becomes clear that some tasks are common across many applications. Examples of such tasks are:

- Starting and stopping proxels
- Monitoring the status of each proxel
- Handling crashed proxels
- Connecting specific ports on specific proxels
- Handling attempts to connect mismatching port types

This motivates functionality that can be used to manage and interact with the processing graph built up of proxels and ports. Superflow provides this functionality through a class called `Graph`, which is the topic of section 2.4.1.

The actual construction of the graph, with creation of proxels and specification of connections, is still up to the programmer. Doing this manually is trivial for small projects, but will quickly result in inflexible and cumbersome code as the projects grow larger and proxel types are reused. In order to simplify the creation of a graph and be flexible to changes in contents and structure, we have divided the task into the following parts, each corresponding to utilities in Superflow given in parentheses:

- How to create a specific type of proxel. (`Factory`)
- How to represent a set of parameters required to create a given proxel. (`PropertyList`)
- Which proxels to create and with which parameters. (`ProxelConfig`)
- Which ports on which proxels to connect to each other. (`ConnectionSpec`)
- How to create a `Graph` using these components. (`createGraph()`)

This approach is suitable for integration with external tools that load a pipeline configuration and create the graph automatically. This will be the topic of section 2.4.2, where each of the utilities are also documented. A typical example of using an external tool is to parse a configuration file that defines the processing graph and then use the provided utilities for setting up the `Graph`. This is implemented in the *yaml* module, which is described in section 2.5.

2.4.1 The Graph class

The `Graph` offers a convenient way to add proxels to the processing graph and kick-start the entire process in one go by starting all proxels. It handles crashed proxels, and performs a clean teardown of the graph when stopped. The `Graph` can be queried for status information regarding proxel workload, processing time, activity status and more. When connecting ports, type checking is done and errors are handled gracefully. The `Graph` provides each proxel with a dedicated worker thread.

When adding proxels to a `Graph`, each proxel is associated with a unique proxel name. The names make it possible to interact with the proxels through the `Graph`, and are used in the process of

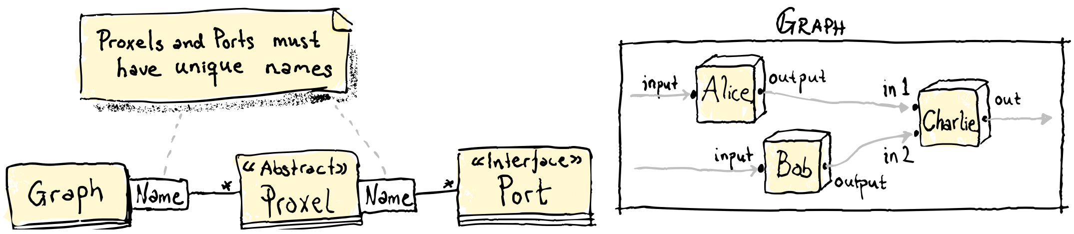


Figure 2.11 *The relations between* `Graph`, `Proxel` *and* `Port`. *Proxels are associated with unique names within the graph, and ports are uniquely identified with names within each proxel.*

connecting proxels through ports. Inside each proxel, ports are also given unique names by calling the owning proxel's method `registerPorts()` (see section 2.2). The relations between `Graph`, `Proxel` and `Port` are illustrated in fig. 2.11. Port names are unique only *within* each proxel. It is thus fine to e.g. name a port 'input' in several proxels.

2.4.2 Building a Graph

This section documents the utilities that Superflow offers to facilitate an automatic creation of a `Graph`. The most important parts are the `PropertyList` and the `Factory`, as these are necessary also when using the *yaml* module library (section 2.5). The rest is mostly useful if you are going to create your own tool for building a graph.

PropertyList

A `PropertyList` is a templated type that must support extraction of values by referring to their names. It is required that a valid `PropertyList` defines the following functions:

- `bool hasKey(const std::string& key)`, tells whether the given key exists or not.
- `T convertValue(const std::string& key)`, retrieves a value from the list.

A skeleton for a `PropertyList` class can look like the `struct` in the following example. Only the declarations are shown, as the implementation of the methods would be application specific, i.e. depend on the source and parsing of the actual property data.

```
// Example of a custom PropertyList
struct MyPropertyList
{
  bool hasKey(const std::string& key) const;

  template<typename T>
  T convertValue(const std::string& key) const;

  // Constructor and other structure would go here...
};
```

A function called `value` is then used to extract elements from a `PropertyList`. Default property values are also supported, in case the property is absent in the list. The usage of `value` with a valid `PropertyList` is as simple as:

```cpp
#include "superflow/value.h"

// Using the value function to extract values from a PropertyList.
MyPropertyList plist(/* ... */); // Initialization of plist.

const auto power = value<int>(plist, "power");

constexpr int default_power{2};
const auto power = value<int>(plist, "power", default_power);
```

Factory

In Superflow, a `Factory<PropertyList>` simply refers to a function that creates a `Proxel` from a given `PropertyList`. This is defined as

```cpp
template <typename PropertyList>
using Factory = std::function<Proxel::Ptr(const PropertyList&)>;
```

As we can see, a `Factory` is a function pointer to any function that takes a `PropertyList` as argument and returns a `Proxel::Ptr` (`std::shared_ptr<Proxel>`). It is commonly declared alongside the proxel. The `value` function described in the previous paragraph extracts values that can be used to create and configure a new proxel. A short example of a `Factory` is shown below:

```cpp
/// This is the Factory for MyProxel. In this case it is called 'create'.
template <typename PropertyList>
static Proxel::Ptr MyProxel::create(const PropertyList& plist)
{
  // Extract properties, regardless of the type of PropertyList
  const auto x = value<std::string>(plist, "x");
  const auto y = value<std::string>(plist, "y", "z"); // z is default
  return std::make_shared<MyProxel>(x, y);
}

// The method is a valid Factory.
const Factory<MyPropertyList> my_factory = MyProxel::create<MyPropertyList>;

const MyPropertyList plist(/* ... */); // Initialization of MyPropertyList
Proxel::Ptr my_proxel = my_factory(plist);
```

FactoryMap

Since each `Proxel` type typically comes with its own `Factory`, the factories should be organised in a `FactoryMap<PropertyList>` which maps factory names to actual `Factory` objects. This makes it easy to retrieve a specific `Factory`. An example of a `FactoryMap` is shown below:

```
// Assume MyPropertyList is a valid implementation of PropertyList...
const FactoryMap<MyPropertyList> factory_map{
  {
    {"MyProxel", MyProxel::create<MyPropertyList> },
    {"MyOtherProxel", MyOtherProxel::create<MyPropertyList> }
  }
};
```

ProxelConfig

When working with factories, there will typically be one `PropertyList` for each proxel. This `PropertyList` can be bundled with the unique proxel name (see section 2.4) in a struct called `ProxelConfig`. Such structs will typically be created by parsing a configuration file, but of course, nothing is stopping you from handcrafting the `ProxelConfig`s you need.

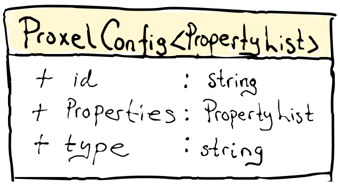


Figure 2.12 ProxelConfig contains all the necessary information to create a specific proxel.

ConnectionSpec

A *connection* is made when a named `Port` of a named `Proxel` is coupled with a named `Port` in another named `Proxel`. The Superflow struct `ConnectionSpec` is a data type made to store one such four-component relation.

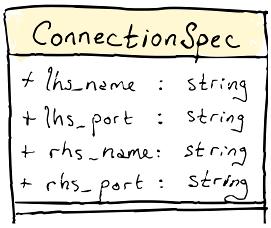


Figure 2.13 ConnectionSpec specifies a connection between two ports on two corresponding proxels.

createGraph()

From the different components described above, a `Graph` can be created using the function `createGraph()` from Superflow. It relies on a set of `ProxelConfig`s to create all the `Proxel`s, and uses `ConnectionSpec`s to connect them afterwards. A quick summary of what `createGraph()` does and what arguments it needs is as follows:

1. For each `ProxelConfig` in the `configs` argument, get the appropriate `Factory` from the `FactoryMap` using the field `ProxelConfig.type`. Note that the `FactoryMap` in the example contains only one proxel type.
2. Call the `Factory` function with the field `ProxelConfig.properties` to create the actual `Proxel`s.
3. For each `ConnectionSpec` in `connections`, connect the `Proxel`s using the specified ports.

A snippet of code is shown below, where we illustrate what kind information each argument to `createGraph()` contains.

```
// This example shows the use of the createGraph function.
// MyPropertyList is some arbitrary PropertyList.
Factory<MyPropertyList> factory = MyProxel::create<MyPropertyList>;
FactoryMap<MyPropertyList> factories{{{"MyProxel", factory}}};

// Assume that property lists for each proxel have been obtained
// as properties1 and properties2.
const auto configs = std::vector<ProxelConfig<MyPropertyList>>
{
  {"proxel1", "MyProxel", properties1},
  {"proxel2", "MyProxel", properties2}
};

// Assume MyProxel has ports named "inport" and "outport".
std::vector<ConnectionSpec> connections
{
  {"proxel1", "outport", "proxel2", "inport"}
};

// Create the graph.
Graph graph = createGraph(factories, configs, connections);
```

2.5 The yaml module

The *yaml* module implements the `PropertyList` interface and provides an automated workflow for setting up a `Graph`. Using a YAML-based [8] configuration format, the module creates and configures proxels, and connects specified ports. This lets a user create and adjust a complete processing graph of available proxels simply by editing a text file. The module uses the *yaml-cpp*[9] third-party library internally, which makes this one of the optional Superflow modules with external dependencies. In the next sections, we will go through the steps to build up a valid configuration file.

2.5.1 The configuration file

The structure and contents of the configuration file must follow a distinct pattern. This section describes these requirements, and ends with a concrete example of a valid configuration file.

Two lists are required in the file: `Proxels` and `Connections`. In the `Proxels` list, an entry consists of a unique name, followed by configuration parameters for that proxel. The only required parameter is `type`, which is used as a key to the `FactoryMap`, i.e. to retrieve the correct `Factory`. The parameter `enable` is optional. Its default value is `true`, but it can be set to `false` in order to temporarily disable a proxel. A minimal proxel entry will look like this:

```yaml
my_proxel_name:        # unique name of the proxel
  type   : "MyProxel"  # class name (key to the FactoryMap)
```

The `Connections` list defines how specific ports on specific proxels are connected together. An entry in the connections list will typically look like this:

```yaml
- [proxel_name1: 'port_name1', proxel_name2: 'port_name2']
```

The proxel name must be one of the unique names declared in the `Proxels`-list. The name is used for accessing the proxel via the `Graph`, and then the port name is used to fetch the actual port via that proxel's `getPort()` method (see section 2.2). Connections are skipped for disabled proxels. The listing below is a valid example of a configuration file for the *yaml* module.

```yaml
%YAML 1.2
---
Proxels:
  proxel_1:                 # unique name of the proxel
    type     : "MyProxel"   # class name
    my_param : 42           # proxel-specific parameter

  proxel_2:                 # unique name of the proxel
    type   : "YourProxel"   # class name
    enable : false          # leave this proxel out of the graph

Connections:
  - [proxel_1: 'out', proxel_2: 'in']
...
```

2.5.2 Replication

In some cases, you would want to have several proxels of the same type, all with the same configurations or only slight variations. This could be specified as:

```
Proxels:
  proxel1:              # unique name
    type: "MyProxel"
    param1 : 2
    param2 : "value"

  proxel2:              # unique name
    type: "MyProxel"    # same type as proxel1
    param1 : 3          # different
    param2 : "value"    # same
```

In order to minimise duplicated text, improve maintainability and to reduce the length of the configuration file, it is possible to specify the parameter `replicate: n` for a proxel. This will create n instances of that proxel, expanding the name of each instance to the base name of the replicated proxel plus an appended index number (e.g. `name_0, ..., name_n-1`). Additionally, any parameter prefixed with a dollar sign (e.g. `$param1`) is expected to be a list of values where the length of the list is equal to the number of instances. Each value in the list will then be assigned to one corresponding proxel instance. The following code snippet illustrates this:

```
Proxels:
  myprox:               # base name
    type: "MyProxel"
    replicate: 2        # actual unique names will be myprox_0 and myprox_1
    $param1: [2, 3]     # param1 will be '2' for myprox_0 and '3' for myprox_1
    param2 : "value"    # will be the same for both replicas
```

Connections are handled seamlessly. Consider a replicated proxel called `replicated` with a port `"out"` and a non-replicated proxel called `myproxel` with a compatible port `"in"`. The following specification requires that `myproxel: "in"` supports multiple connections.

```
Connections:
  - [replicated: 'out', myproxel: 'in']
    # replicated_0.out \
    #                    > myproxel.in
    # replicated_1.out /

    # equivalent to
  - [replicated_0: 'out', myproxel: 'in']
  - [replicated_1: 'out', myproxel: 'in']
```

It is also possible to assign ports in the replicas to specific ports in the non-replicated proxel. This would be specified as

```
Connections:
  - [replicated: 'out', myproxel: ['a', 'b']]
    # replicated_0.out -> myproxel.a
    # replicated_1.out -> myproxel.b

    # equivalent to
  - [replicated_0: 'out', myproxel: 'a']
  - [replicated_1: 'out', myproxel: 'b']
```

2.5.3 SectionPaths

Technically, the `Proxels`-list is strictly required, but it *may* be given another name and it is also possible to have multiple proxel sections in a configuration file. It may even be a subgroup of another list. If so, the so called `SectionPath`s must be specified as an argument to the function `yaml::createGraph`. This could be the case if you want to embed other configuration parameters in your config file, which you would typically load directly with *yaml-cpp*. The example below shows a config file with corresponding C++ code specifying the `SectionPath`s:

```yaml
Platform:
  Sensors:        # This is a proxel section
    camera1: {type: "MyCameraProxel" }

  Engine:         # Section ignored by the yaml module
    fuel : "gas"

ImgProx:          # This is another proxel section
  canny: {type: "CannyProxel" }
```

```cpp
// Specifying SectionPaths that list proxels in the configuration file.
// config_path defined elsewhere.
// factory_map defined elsewhere.
const std::vector<yaml::SectionPath> config_sections =
{
  {"Platform", "Sensors"},
  {"ImgProx"}
};
```

2.5.4 Creating a Graph from YAML-file

After learning about the details of the configuration file, we are finally ready to employ the function `yaml::createGraph` to load the file and create a `Graph`. In the snippet below, we show in context how the function can be used. For full examples with detailed explanations though, you should work your way to sections 3.2.2 and 3.3.2 of the tutorial.

```cpp
#include "myproxel.h"
#include "superflow/yaml/yaml.h"

int main(int argc, char **argv)
{
  const flow::yaml::FactoryMap factory_map{
    {{"MyProxel", MyProxel::create<flow::yaml::YAMLPropertyList>}}
  };

  const std::string config_path{argv[1]};
  flow::Graph graph = yaml::createGraph(config_path, factory_map, config_sections);

  graph.start();
  // ...
  graph.stop();

  return EXIT_SUCCESS;
}
```

2.6 The loader module

The *loader* module enables dynamic loading of shared proxel-libraries like plugins. A loader-compatible library has embedded a list of `Factory`s for the `Proxel`s it contains, which assists in the automatic creation of a corresponding `FactoryMap`. Let us start with a motivating example before going into detail. Notice how few lines that are actually required to set up a `Graph`, given that we have a handful of precompiled libraries with proxels and a *yaml* configuration file:

```
#include "superflow/loader/load_factories.h"
#include "superflow/yaml/yaml_property_list.h"

int main(int argc , char **argv)
{
  const std::vector<flow::load::ProxelLibrary> libraries{
    {{"path/to/libraries", "lib1"}},
    {{"path/to/libraries", "lib2"}}
  };

  const std::string yaml_config_file_path {argv[1]};
  const auto graph = flow::yaml::createGraph(
    yaml_config_file_path,
    flow::load::loadFactories<flow::yaml::YAMLPropertyList>(libraries)
  );
}
```

The beauty of what is presented to us here, is that the user does not have to include any specific header files from the proxel libraries in their consuming application. Since the configuration is loaded from a yaml-file, no hard coded proxel names are required in the compiled client code either. In the example, `"path/to/libraries"` is a directory that contains compiled library files, and `"lib1"` and `"lib2"` are library names. As we will see later, the prefix and suffix (`lib*.so` or `*.dll`) will be determined automatically.

There are three important aspects when working with a proxel library that has *loader* support:

1. For each `Proxel`, a corresponding `Factory` must be 'registered' using one of the macros from the header file `"superflow/loader/register_factory.h"`.
2. When compiling the proxel factories, you must have already decided on a concrete `PropertyList` (or 'value adapter'). At the time of writing, the `flow::yaml::YAMLPropertyList` is the only `PropertyList` we have implemented.
3. When loading the library into the consuming application, create a `flow::load::ProxelLibrary` for each shared library, and keep them in scope as long as their proxels are in use.

We will continue to elaborate on these points through the following sections.

If needed, you should turn back to section 2.4.2 for the details about `Factory`s and `FactoryMap`s. Section 3.3 contains a tutorial where you can practise how to use the *loader* module.

2.6.1 Registering the proxel factory

The functionality of the *loader* module is provided by Boost.DLL [10], where we employ its abilities to export *named symbols* to a *named section* in a binary, and later, in another program, to load the symbols dynamically by referring to those names.

When we say ‘register factory’, we mean to create such a *named symbol* so that the *symbol* points to a concrete implementation of a `Factory<PropertyList>`, and the *name* is set to be the actual type of `Proxel` that the `Factory` can create. Later, when we browse through symbol names in the binary and find one that matches a `Proxel` type, we can easily retrieve its corresponding `Factory` and construct a new proxel.

We want it to be a pleasant experience for the developer to register a factory. Our easiest and most common way, is to use the macro `REGISTER_PROXEL_FACTORY`. In the following example, the only *loader*-essential parts are on lines 2 and 13. The rest of the code contains nothing we haven't seen before.

```
1 #include "my/proxel.h"
2 #include "superflow/loader/register_factory.h"
3 #include "superflow/value.h"
4
5 template<typename PropertyList>
6 flow::Proxel::Ptr createMyProxel(const PropertyList& adapter)
7 {
8   return std::make_shared<MyProxel>(
9     flow::value<int>(adapter, "key")
10   );
11 }
12
13 REGISTER_PROXEL_FACTORY(MyProxel, createMyProxel)
```

That is all there is to it for the developer of the `Proxel`. A common pattern is to place the macro either at the bottom of the `Proxel`’s cpp-file, or in a standalone file such as `create-myproxel.cpp`.

If you want a more detailed understanding of the mechanisms behind the *loader* module, we refer to the Boost.DLL documentation.

Required `LOADER_ADAPTER_*` macros

Internally, `REGISTER_PROXEL_FACTORY` requires the existence of some additional macro values. These are:

- `LOADER_ADAPTER_HEADER`, path to the concrete property list header file.
- `LOADER_ADAPTER_NAME`, a short, unique identifier for the `PropertyList`.
  It must be equal to the corresponding `PropertyList::adapter_name`.
- `LOADER_ADAPTER_TYPE`, the concrete type of `PropertyList`.

We encourage that these values are predefined by the creator of the `PropertyList`. For the *yaml* module, values are defined through the target’s ‘interface compile definitions’. That means that when you link your proxel library to `flow::yaml`, they will be automatically defined.

If you consider implementing your own `PropertyList`, here are the concrete values defined in CMake as an example:

```
target_compile_definitions(flow::yaml
  INTERFACE
  LOADER_ADAPTER_HEADER="superflow/yaml/yaml_property_list.h"
  LOADER_ADAPTER_NAME=YAML
  LOADER_ADAPTER_TYPE=flow::yaml::YAMLPropertyList
)
```

Why does `REGISTER_PROXEL_FACTORY` have to be a macro?

Our macros are in fact just wrapping another macro, from the Boost.DLL library, and we have created ours just so that we can store and retrieve `Factory`s in a predictable way. If you examine the final, expanded macro, you see that we depend on Boost to generate code in a way that is arguably both simple and user friendly with the help of macros, and we have not found the incentives to pursue other feasible solutions, even though macros are often frowned upon.

Requirements for a PropertyList

The discussion of and requirements for a `PropertyList` is already written in section 2.4.2, but for the sake of context we will repeat it here.

The interface of a `PropertyList` is defined through the templated function `flow::value<T>` from `"superflow/value.h"`. It requires that a valid `PropertyList` has the following methods:

- `bool hasKey(const std::string& key)`, which tells whether the given key exists or not.
- `T convertValue(const std::string& key)`, which retrieves a value from the list.

The *loader* module requires in addition the existence of a string `PropertyList::adapter_name`. That adapter name defines a unique identifier of the concrete `PropertyList`, which plays a role in the process of storing and retrieving `Factory`s of the correct type from the shared library. Specifically, it states the *named section* we are searching for in the binary.

2.6.2 The ProxelLibrary class

A proxel library and its registered `Factory`s are accessed through the use of a `flow::load::ProxelLibrary` object. You construct it using the path to the shared library file, and fetch the `FactoryMap<PropertyList>` using the method `loadFactories<PropertyList>`:

```
const flow::load::ProxelLibrary library{"path/to/library"};
const auto factories = library.loadFactories<flow::yaml::YAMLPropertyList>();
```

Warning: An object of the `ProxelLibrary` class is your only reference to the shared library, and must not go out of scope as long as its proxels are in use. That will cause the shared library to unload and your application to crash!

You can collect factories from multiple libraries using the function `loadFactories`. Remember that the libraries must not go out of scope, hence we keep the vector as a variable. To reduce the risk of such errors, we have explicitly disallowed to construct the vector as a temporary within the function argument list.

```
const std::vector<flow::load::ProxelLibrary> libraries{
   {"path/to/library1"},
   {"path/to/library2"}
  };

const auto factories = flow::load::loadFactories<flow::yaml::YAMLPropertyList>(
  libraries
);
```

If you separate the name of the library and the path to the directory in which the library resides, Boost can automatically determine the prefix and suffix of the shared library file.

```
const std::string library_directory{"/directory"};
const std::string library_name{"myproxels"};
const flow::load::ProxelLibrary library{library_directory, library_name};

// Linux result: /directory/libmyproxels.so
// Windows result: /directory/myproxels.dll
```

2.6.3 Where to register the factory

Where should I ideally define and declare my `Factory`, and where do I typically call the macro `REGISTER_PROXEL_FACTORY`? Strictly speaking, it does not matter. Or, more precisely: it is up to you. It depends on how you want to design, create, use and share your `Proxel`s and `Factory`s and to what degree you strive to keep dependencies private and hidden from public interface. Just remember that if you want to use the *loader* functionality, you are at some point eventually required to link your library to the *loader* module (and probably the *yaml* module as well).

There are basically two factory registration patterns that we follow in our applications:

- The `Factory` is part of the proxel header file, defined side-by-side to the proxel `class` definition. The macro and `#include "superflow/loader/..."` may either be in the `*.h`-file or the `*.cpp`-file.
- The `Factory` is not visible in the `Proxel`'s public interface.
  It is defined alongside the macro elsewhere, either in the `*.cpp`-file of the `Proxel`, in a separate source file, or in a separate library as we will see in the tutorial.

2.6.4 Advanced factory registration

`REGISTER_PROXEL_FACTORY` can only support one type of `PropertyList`, namely the one it is bound to through `LOADER_ADAPTER_TYPE`. If you find yourself in a situation where you want to compile support for multiple `PropertyList`s into your proxel library plugin, you are off the beaten path. However, mechanisms are in place to facilitate such a need. In short, you would want to use `REGISTER_PROXEL_FACTORY_SECTIONED` directly or indirectly for each `PropertyList`. We can ignore the required `LOADER_ADAPTER_*` macros by defining `LOADER_IGNORE`, since we will not use plain old `REGISTER_PROXEL_FACTORY` this time.

In listing 2.2, we sketch up a skeleton for a new type, `JSONPropertyList`, and we assume that we repeat the process for some `XMLPropertyList`. Then, we apply both these types to a `Factory` for `MyProxel`. The main aspects of the example are that:

- We create a new type of `PropertyList` in `"json_property_list.h"`.
- The new type implements `convertValue<T>()` and `hasKey()` (not shown), according to the requirements of `flow::value<T>()`.
- We define `adapter_name` for the type, according to the requirements of `loadFactories`.
- We create a helper macro in `"json_register_factory.h"`, so that we will not have to remember the `adapter_name` or use it explicitly.

- We `#include "xxx_register_factory.h"` for both `PropertyList`-types in the file where we define the `Proxel`'s `Factory` function, and register both types with the library.

If you want to, you can compile your bare `Proxel`s initially into a static library, and later compile all the `Factory`s into a separate, shared factory-library that links to your proxels. The example includes an abbreviated CMake-snippet which illustrates that pattern.

Do not be confused: `superflow::json` and `superflow::xml` are hypothetical `PropertyList`s. They are not actually implemented in Superflow and not available at the time of writing.

(a) json_property_list.h

```cpp
#define JSON_ADAPTER_NAME JSON

namespace flow::json
{
class JSONPropertyList
{
public:
  // Implement the convertValue() and hasKey() functions ...

  static constexpr const char* adapter_name{BOOST_PP_STRINGIZE(JSON_ADAPTER_NAME)};
};
}
```

(b) json_register_factory.h

```cpp
#include "superflow/loader/register_factory.h"
#include "superflow/json/json_property_list.h"

#define REGISTER_JSON_PROXEL_FACTORY(ProxelName, Factory) \
        REGISTER_PROXEL_FACTORY_SECTIONED( ProxelName, \
          Factory<flow::json::JSONPropertyList>, JSON_ADAPTER_NAME \
        )
```

(c) my_proxel_factory.cpp

```cpp
#include "proxels/my_proxel.h"
#include "superflow/value.h"
#include "superflow/json/json_register_factory.h"
#include "superflow/xml/xml_register_factory.h"

template<typename PropertyList>
flow::Proxel::Ptr createMyProxel(const PropertyList& properties)
{ /* ... */ }

REGISTER_JSON_PROXEL_FACTORY(MyProxel, createMyProxel)
REGISTER_XML_PROXEL_FACTORY(MyProxel, createMyProxel)
```

(d) CMakeLists.txt

```cmake
add_library(my-proxel STATIC my-proxel.cpp)
target_link_libraries(my-proxel PUBLIC superflow::core)

add_library(my-proxel-factory SHARED my-proxel-factory.cpp)
target_link_libraries(my-proxel-factory PRIVATE
  my-proxels
  superflow::loader
  superflow::json
  superflow::xml
)

add_executable(main src/main.cpp)
target_link_libraries(main
  superflow::core superflow::loader superflow::json superflow::xml
)
```

Listing 2.2 Hypothetical example of embedding multiple property lists into a library.

2.6.5 Summary for the loader module

If you are building a proxel library with *loader* support, you should remember that:

- For each proxel, a `Factory` must be *registered*, typically by calling the `REGISTER_PROXEL_FACTORY` macro.
- You must create a *shared* library for the proxel factories. With CMake, it will typically look like this:
  `add_library(myproxels SHARED ...)`
- The templated type `PropertyList` must be defined compile time for the `Factory`s, as it is required to create the complete type for `Factory<PropertyList>`.
- The library with factories must thus be linked to `superflow::loader` *and* a concrete `PropertyList`. E.g., `target_link_libraries(myproxels PRIVATE superflow::loader superflow::yaml)`
- The chosen `PropertyList` must also be linked into the consuming application, in order to retrieve the `Factory`s.
  E.g., `target_link_libraries(main PRIVATE superflow::loader superflow::yaml)`

2.7 Utilities

As part of the *core* module, Superflow comes with a collection of utilities and helper classes, located in the `"superflow/utils/..."` directory. They are mostly well documented, so we will give just a brief introduction to a selection of utilities in this report.

- Load monitoring: `ProxelTimer`
- Timing based behaviour: `Metronome`, `Sleeper` and `Throttle`
- Protecting objects in multithreaded environments: `Mutexed` and `SharedMutexed`
- Hide implementation details: `pimpl`
- Control flow: `SignalWaiter` and `waitForSignal()`.

2.7.1 ProxelTimer

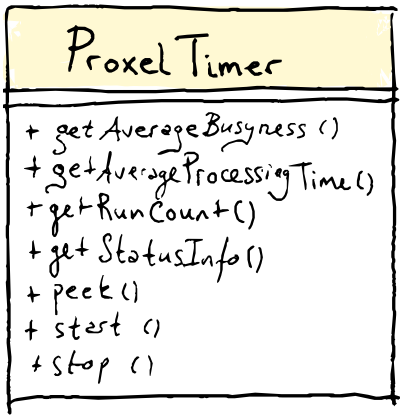


Figure 2.14 A ProxelTimer can be used for computing workload stats for a `Proxel`.

We often want to monitor the processing load of a `Proxel`. The `ProxelTimer` is a tool that can help with the task, as it can compute different stats based on timing of the `Proxel`'s active or idle states. See fig. 2.14 for available class methods.

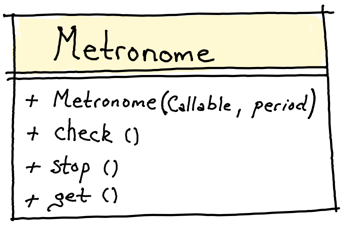
For each iteration of the `Proxel`'s processing loop, `ProxelTimer.start()` should be called at the beginning, and `ProxelTimer.stop()` should be called at the end. Subsequent calls to the `get`-functions of `ProxelTimer` will then return computed stats. The pseudo-code snippet in listing 2.3 summarises this pattern. You can call `peek()` to get processing time since `start()` without calling `stop()`. The function `getStatusInfo()` returns a formatted `string` with status information, suitable for the `Proxel::setStatusInfo()` method.

```
// Example using a ProxelTimer
#include "superflow/utils/proxel_timer.h"

ProxelTimer my_timer;

while (looping)
{
  my_timer.start();
  // do processing
  my_timer.stop();
}

std::cout << my_timer.getRunCount() << " loops completed" << std::endl;
std::cout << my_timer.getStatusInfo() << std::endl;// Formatted output.
```

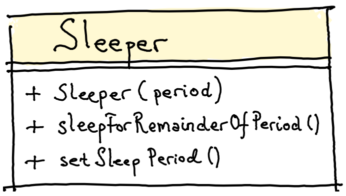
Listing 2.3 ProxelTimer, example usage.

2.7.2 Rate limiters: Metronome, Sleeper and Throttle

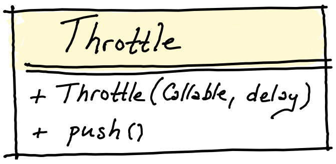
These classes will in different ways stall the execution of a thread, typically with the intention of carrying out a given task periodically.



(a) Metronome will repeatedly call a given function at fixed rate.



(b) Sleeper will stall execution of the current thread until a given point in time.



(c) Throttle will delay calling a function, with replaceable arguments, until a given point in time.

Figure 2.15 Three available rate limiters in Superflow utilities.

Metronome

Calls a given function on a separate thread at a specified interval, omitting the first (immediate) call. Will continue to do so forever, until `stop()` or destructor is called. A `Metronome` is useful for e.g. printing a status message while waiting for a task to complete, or generating data at a predictable rate. An example is provided in listing 2.4.

Sleeper

Typically intended to be used within a for-loop in order to stall execution. After the main processing is completed within the loop, call `sleepForRemainderOfPeriod` to stall further execution until until one `period` has passed since the previous iteration. See listing 2.5 for an example.

```cpp
#include "superflow/utils/metronome.h"

Metronome repeater{
  [](const auto& duration)
  {
    std::cerr
      << "my_func has been stalling for " << duration.count() << "s"
      << std::endl;
  },
  std::chrono::seconds{2}
};

my_func(); // we expect that this might stall
repeater.stop();

// `repeater` will print the above message every 2 seconds until
// `my_func()` has returned and `stop()` has been called.
```

Listing 2.4 Metronome, example usage.

```cpp
#include "superflow/utils/sleeper.h"

using namespace std::chrono_literals;
const Sleeper rate_limiter(10ms);

for (const auto& data : *latched_port_)
{
  do_work();
  rate_limiter.sleepForRemainderOfPeriod();  // Sleep for the rest of the period

  if(some condition)
  { rate_limiter.setNewSleepPeriod(xms); }
}
```

Listing 2.5 Sleeper, example usage. If execution time of `do_work()` is less than 10ms, it will not be called again until 10ms has passed since the previous call. If execution time is more than 10ms, it will be called again immediately, as `sleepForRemainderOfPeriod` time is already due.

```cpp
#include "superflow/utils/throttle.h"

using namespace std::chrono_literals;

void processFusedData(const T& data)
{ process(data); }

void functionWithThrottle()
{
  const Throttle throttle(processFusedData, 1s);

  for (const auto& data : *high_frequency_port_)
  {
    fused_data_ = fuse(data);
    throttle.push(fused_data_);
  }
}
```

Listing 2.6 Throttle, example usage. The idea is that `fuse(data)` will happen at a high rate, while `processFusedData` should happen at a lower, periodic rate. An example can be to fuse sensor data into a traversability map at high rate, while sending the most recently fused map to a motion planner at a lower rate.

Throttle

Ensures that a given function is called with data at most once per the given period of time. Arguments to the function are provided with the `push(data)` method. If `push(data)` is called before a period of time has elapsed since the previous call, `Throttle` will delay the function call. When the delay has expired, the registered function will be called with the most recent data as argument. Data that is `push`ed while the throttler is stalling, will thus by design be replaced. If a period elapses without any new data being available, the function will not be called. Later, when new data is provided, the function will be called immediately and a new period starts.

Note: Since data is copied for each call to `push` (unless you `std::move`), other strategies than `Throttle` should probably be considered for large data objects.

2.7.3 Multithreading: Mutexed and SharedMutexed

Mutexed

If you have a `class` with several members that each requires their own `mutex` for safe access in a multithreaded environment, `Mutexed` is a wrapper class that can be applied in order to save code duplication. Note that since `Mutexed` is derived from `T`, `T` cannot be pointer or reference.

```cpp
Mutexed<std::string> mutexed("hello");
{
  std::scoped_lock lock{mutexed};
  mutexed = "hello";
}
```

SharedMutexed

Similar to `Mutexed`, but with a `std::shared_mutex` in order to facilitate concurrent read operations and exclusive write operations, commonly known as a ‘reader/writer lock’.

Note: If you have frequent, but short read operations, a plain lock (`Mutexed`) will likely outperform reader/writer locks because of their extra complexity. A `SharedMutexed` is better suited for scenarios where read operations are frequent and expensive, but you should also note that performing expensive operations while holding a lock is often a bad sign. There may be better ways to solve the problem than using a `SharedMutexed` [11]. As always, you should test and measure the actual execution time of your alternatives in order to know for sure what yields the best performance in your application.

2.7.4 Hiding implementation details: The pimpl class

Helper class for the ‘PImpl’[4]pattern [12]. The code is taken from Herb Sutter’s ‘Guru of the Week (GotW)’ #101 [13], with slight modifications.

The `pimpl_h.h` file should be included from the header file of the class owning the `pimpl` object.

```cpp
#pragma once
#include "superflow/utils/pimpl_h.h"
class MyClass
{
public:
  MyClass(ctor args);
  // ...
private:
  class impl;      // forward declare
  pimpl<impl> m_; // instead of std::unique_ptr<impl>
  // ...
};
```

The `pimpl_impl.h` file should be included from the source file (cpp). Remember the explicit template instantiation!

```cpp
#include "mylib/my_class.h"
#include "superflow/utils/pimpl_impl.h"

// Easy to forget, but strictly required for the code to compile
template class flow::pimpl<MyClass::impl>;

// The impl.
class MyClass::impl
{
  impl(impl ctor arguments);
  void func();
};

MyClass::MyClass(ctor args)
  : m_{impl ctor arguments} // instead of std::make_unique<impl>
{
  m_->func(); // Access the impl
}
```

[4]Pointer to Implementation

2.7.5 Handling signals: SignalWaiter and waitForSignal

`SignalWaiter` is a RAII[5]-wrapper for thread-safe listening to one or more interrupt signals from the operating system. Instead of creating the `SignalWaiter` class explicitly, you will typically use the function `waitForSignal(const std::vector<int>& signals)` to block the execution of the current thread.

Note: If you are already using `std::signal()` or plain C `signal()` in your program, the `SignalWaiter` class might not work as expected.

```
#include "superflow/utils/wait_for_signal.h"

int main(int, char**)
{
  // init...
  flow::waitForSignal({SIGINT}); // < Wait for Ctrl+C
  // teardown...
}
```

[5]Resource Acquisition Is Initialisation

3 Tutorial

In this chapter, we will use Superflow to create a small processing graph that performs a very simplified processing on a sequence of numbers. The corresponding `Graph` will contain five `Proxels` of four different types. We will use the *yaml* module to set up the `Graph`. A view of the processing pipeline is shown in fig. 3.1.

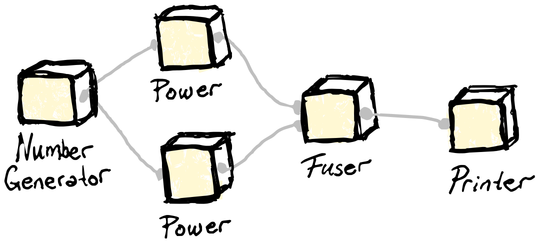


Figure 3.1 The tutorial processing graph.

First, the `NumberGenerator`-proxel will generate a number sequence. Next, the `Power`-proxels will receive the generated numbers and compute the value of the numbers raised to the power of `power`. Since `power` will be a configurable number, this illustrates the use of `PropertyList` to modify the behaviour of a proxel. Two instances will be created of this proxel, with different `powers`. Then, a `Fuser`-proxel will fuse data from the two `Power`-proxels, giving an example of fusing data streams into a proxel. Finally, the result is printed to a file by the `Printer`-proxel, which illustrates that not all proxels must have an output port.

3.1 Creating proxels

Let us start with writing the actual code for the proxels in our sample processing pipeline. Each proxel consists of a header file and a source file. The important points to notice about the code examples are the following:

- Inclusion of header files from the Superflow library
  `#include "superflow/xyz.h"`
- Inheritance from the `Proxel` class.
  `class MyProxel : public flow::Proxel`
- Overriding the pure virtual methods from `Proxel`.
  `void start() override;`
  `void stop() override;`
- The `static` factory method called `create`. Note that it must not necessarily be named `create`, nor be a static member function. It is the signature that is important.
  From the library:
  `Factory = std::function<Proxel::Ptr(const PropertyList&)>`
- Input/output through member variables of different `Port` types
  e.g. `flow::ProducerPort<int>::Ptr`.

Remember that the constructor of the proxel is used for setup and configuration before any processing starts. It is important to call `registerPorts()` in the constructor to expose any ports, so that connections can be made. As soon as `start()` is called, the main processing will begin. Also remember that the `Graph` calls each proxel's `start()` from a separate thread.

3.1.1 The NumberGenerator proxel

The first `Proxel`, `NumberGenerator`, is shown in listings 3.1 and 3.2. The constructor creates an instance of a `ProducerPort<int>`, and then exposes it by registering it with the name `"out"`. In `start()`, the loop for generating numbers is initiated with a call to the function `generateNumbers()`. When that function returns, this proxel will do no more work.

```
#pragma once

#include "superflow/producer_port.h"
#include "superflow/proxel.h"

class NumberGenerator : public flow::Proxel
{
public:
  NumberGenerator();

  void start() override;
  void stop() noexcept override;

  template<typename PropertyList>
  static flow::Proxel::Ptr create(const PropertyList&);

private:
  flow::ProducerPort<int>::Ptr output_port_;
  bool stopped_;
  void generateNumbers() const;
};

template<typename PropertyList>
flow::Proxel::Ptr NumberGenerator::create(const PropertyList&)
{
  return std::make_shared<NumberGenerator>();
}
```

Listing 3.1 number-generator.h

```
#include "number-generator.h"

using namespace flow;

NumberGenerator::NumberGenerator()
    : output_port_{std::make_shared<ProducerPort<int>>()}
{
  registerPorts(
    {{"out", output_port_}}
  );
}

void NumberGenerator::start()
{
  stopped_= false;
  generateNumbers();
}

void NumberGenerator::stop() noexcept
{ stopped_ = true; }

void NumberGenerator::generateNumbers() const
{
  constexpr int num_iterations{5};
  for (int i = 0; i < num_iterations && !stopped_; ++i)
  { output_port_->send(i); }
}
```

Listing 3.2 number-generator.cpp

3.1.2 The Power proxel

The `Power` proxel has a very similar header file, as shown in listing 3.3. The differences are that this proxel receives its input through a port of type `BufferedConsumerPort<int>` and that it produces `double`s with a `ProducerPort<double>`. Additionally, a parameter called `"power"` is extracted from the `PropertyList` in the factory method `create()`.

The source file is shown in listing 3.4. As before, ports are initialised in the constructor and then exposed by calling `registerPorts()`. In this example, the `BufferedConsumerPort` is configured with a buffer size of 10 in order to capture the burst of data sent from the `NumberGenerator`. In general, the choice of buffer size will depend on the how fast your consumer is able to process data. If you expect to receive bursts of data, and your consumer on average is faster than the producer, a buffer with reasonable size can be applicable. If, on the other hand, the receiver on average is slower than the producer, you will nevertheless eventually drop data. A smaller buffer should thus be selected if you prefer to work on the newest available data.

In `start()`, the range-based for loop is used for retrieving data from the `input_port_` (i.e. the buffer). Within the loop, the `process()` method will process the data, and the result is passed on through the `ProducerPort`. Note also that we see an example usage of `Proxel::setState( ... )` in `start()`.

When no more data is available, the `BufferedConsumerPort` port will block until it is deactivated through a call to `stop` and thereby `input_port_->deactivate()`.

Protip: If your program hangs and does not terminate when you quit your app, make sure that you are calling `deactivate()` for all buffered `Port`s.

```cpp
#pragma once

#include "superflow/buffered_consumer_port.h"
#include "superflow/producer_port.h"
#include "superflow/proxel.h"
#include "superflow/value.h"

class Power : public flow::Proxel
{
public:
  explicit Power(int power);

  void start() override;
  void stop() noexcept override;

  template <typename PropertyList>
  static flow::Proxel::Ptr create(const PropertyList& plist);

private:
  size_t buffer_size_;
  int power_;
  flow::BufferedConsumerPort<int>::Ptr input_port_;
  flow::ProducerPort<double>::Ptr output_port_;
  double process(int i) const;
};

template<typename PropertyList>
flow::Proxel::Ptr Power::create(const PropertyList& plist)
{
  return std::make_shared<Power>(
      flow::value<int>(plist, "power")
  );
}
```

Listing 3.3 power.h

```cpp
#include "power.h"
#include <cmath>

using namespace flow;

Power::Power(const int power)
    : buffer_size_{10}
    , power_{power}
    , input_port_{std::make_shared<BufferedConsumerPort<int>>(buffer_size_)}
    , output_port_{std::make_shared<ProducerPort<double>>()}
{
  registerPorts(
    {
      {"in",  input_port_},
      {"out", output_port_}
    }
  );
}

void Power::start()
{
  setState(State::AwaitingInput);

  for (const int i : *input_port_)
  {
    setState(State::Running);
    const auto result = process(i);
    output_port_->send(result);
  }

  setState(State::Paused);
}

void Power::stop() noexcept
{
  input_port_->deactivate();
}

double Power::process(const int i) const
{
  return std::pow(i, power_);
}
```

Listing 3.4 power.cpp

3.1.3 The Fuser proxel

In the `Fuser` proxel, a `MultiConsumerPort<double>` is used as an input port. As with the `BufferedConsumerPort` in the `Power` proxel, it is initialised with a buffer size. No other template parameters than `double` is specified, so we will get the default `GetMode::Blocking` behaviour. This means that all producers must push new data before the consumer unblocks.

In the `process()` method, we see that the data retrieved from the `MultiConsumerPort` comes as a vector. All elements in the vector will be multiplied together. If we get X^a from the first `Power` proxel and X^b from the other, the result from the `Fuser` proxel should be $X^{(a+b)}$.

```cpp
#pragma once

#include "superflow/multi_consumer_port.h"
#include "superflow/producer_port.h"
#include "superflow/proxel.h"

class Fuser : public flow::Proxel
{
public:
  Fuser();

  void start() override;
  void stop() noexcept override;

  template<typename PropertyList>
  static flow::Proxel::Ptr create(const PropertyList&);

private:
  size_t buffer_size_;
  flow::MultiConsumerPort<double>::Ptr input_port_;
  flow::ProducerPort<double>::Ptr output_port_;

  double process(const std::vector<double>&);
};

template<typename PropertyList>
flow::Proxel::Ptr Fuser::create(const PropertyList&)
{ return std::make_shared<Fuser>(); }
```

Listing 3.5 fuser.h

```cpp
#include "fuser.h"

using namespace flow;

Fuser::Fuser()
    : buffer_size_{10}
    , input_port_{std::make_shared<MultiConsumerPort<double>>(buffer_size_)}
    , output_port_{std::make_shared<ProducerPort<double>>()}
{
  registerPorts(
    {
      {"in", input_port_},
      {"out", output_port_}
    }
  );
}

void Fuser::start()
{
  // Remember that the Blocking MultiConsumerPort returns
  // a vector with values from all connected ProducerPorts.
  for (const auto& data : *input_port_)
  {
    const double product = process(data);
    output_port_->send(product);
  }
}

void Fuser::stop() noexcept
{ input_port_->deactivate(); }

double Fuser::process(const std::vector<double>& data);
{
  double product{1.};

  for (const auto number : data)
  { product *= number; }

  return product;
}
```

Listing 3.6 fuser.cpp

3.1.4 The Printer proxel

This proxel receives its input through a `CallbackConsumerPort`, which needs to be coupled with a callback function. As seen in listing 3.8, the actual function that gets registered is a lambda, which in turn calls the class method `print()`. In order to call a class method, the lambda must capture `this`. The `Printer` proxel does not have an output port. Typical examples where this is common are proxels that are attached to the processing graph in order to facilitate logging, visualisation or other sink-like behaviour. In this case we write the results to a file. The filename is loaded in the factory using the `value()` function, and passed on to the proxel's constructor.

It is also worth noting that this is a completely passive proxel, where both `start()` and `stop()` are totally empty. This means that the thread provided by `Graph` to run the proxel will return immediately, and all actions are performed by the calling thread of the callback function.

```cpp
#pragma once

#include "superflow/callback_consumer_port.h"
#include "superflow/proxel.h"
#include "superflow/value.h"
#include <fstream>

class Printer : public flow::Proxel
{
public:
  explicit Printer(const std::string& filename);

  void start() override;
  void stop() noexcept override;

  template<typename PropertyList>
  static flow::Proxel::Ptr create(const PropertyList&);

private:
  flow::CallbackConsumerPort<double>::Ptr input_port_;
  mutable std::ofstream file_;

  void print(double number) const;
};

template<typename PropertyList>
flow::Proxel::Ptr Printer::create(const PropertyList& plist)
{
  return std::make_shared<Printer>(
      flow::value<std::string>(plist, "filename")
  );
}
```

Listing 3.7 printer.h

```cpp
#include "printer.h"

using namespace flow;

Printer::Printer(const std::string& filename)
    : input_port_{std::make_shared<CallbackConsumerPort<double>>(
        [this](const double d)
        { print(d); }
      )}
    , file_{filename, std::ios::binary}
{
  registerPorts(
    {{"in", input_port_}}
  );
}

void Printer::start()
{}

void Printer::stop() noexcept
{}

void Printer::print(const double number) const
{
  file_ << number << std::endl;
}
```

Listing 3.8 printer.cpp

3.2 Creating the Graph

We have now created all the necessary building blocks for our sample processing graph. This section will describe the setup of the actual `Graph` using the *yaml* module, which was introduced in section 2.5. We will begin by creating a YAML configuration file, and then continue to set up our application in an actual `main` function.

3.2.1 The configuration file

The structure and requirements of a configuration file is discussed in section 2.5.1. The configuration file for this tutorial is shown in listing 3.9. We recognise the required section `Proxels` with our five proxels, where the 'replicate' functionality is utilised for the `Power` proxel. We can see that the `Printer` proxel will print to a file called 'results.txt'. The `Connections` section is straightforward, but notice the 'replicate' functionality that seamlessly connects replicated `Power` proxels to the `MultiConsumerPort` in the `Fuser` proxel.

```yaml
%YAML 1.2
---
Proxels:
  number_generator:
    type   : "NumberGenerator"

  power:
    type: "Power"
    replicate: 2
    $power: [2, 3]

  fuser:
    type   : "Fuser"

  printer:
    type     : "Printer"
    filename : "results.txt"

Connections:
  - [number_generator: 'out', power: 'in']
  - [power: 'out', fuser: 'in']
  - [fuser: 'out', printer: 'in']
...
```

Listing 3.9 config.yaml

3.2.2 Graph setup using the *yaml* module

Finally, we are ready to create the program that will build and run our processing graph. We will name the file 'main.cpp' and its contents are shown in listing 3.10. The header files for all the proxels are included at the beginning. This gives access to the factory for each type. The factories are inserted into a `yaml::FactoryMap` (line 18), which is an alias for a `FactoryMap<yaml::YAMLPropertyList>`. Each factory in the map must also be specified with the same template parameter `yaml::YAMLPropertyList`.

The rest is up to the *yaml* module. To create a `Graph`, all we have to do is to provide the path to the configuration file and pass it to `yaml::createGraph()` along with the `FactoryMap` (line 27). It is really that simple! Internally, the *yaml* module will parse the file and create the necessary objects to pass on to the Superflow function `createGraph()` described in section 2.4.2.

After calling `start()` (line 30), the processing will begin. We have utilised a helper function called `waitForSignal` to pause the main thread while the graph is working. When the user presses `Ctrl+C`, the main thread will continue and the processing is stopped (line 34).

```cpp
1 #include "fuser.h"
2 #include "number-generator.h"
3 #include "power.h"
4 #include "printer.h"
5
6 #include "superflow/graph.h"
7 #include "superflow/utils/wait_for_signal.h"
8 #include "superflow/yaml/yaml.h"
9
10 #include <csignal>
11 #include <fstream>
12 #include <iostream>
13
14 int main(int argc, char **argv)
15 {
16   const std::string config_path{argv[1]};
17
18   const flow::yaml::FactoryMap factory_map{
19     {
20       {"Fuser",           Fuser::create<flow::yaml::YAMLPropertyList>},
21       {"NumberGenerator", NumberGenerator::create<flow::yaml::YAMLPropertyList>},
22       {"Power",           Power::create<flow::yaml::YAMLPropertyList>},
23       {"Printer",         Printer::create<flow::yaml::YAMLPropertyList>}
24     }
25   };
26
27   flow::Graph graph = flow::yaml::createGraph(config_path, factory_map);
28
29   std::cout << "- Press Ctrl+C to stop." << std::endl;
30   graph.start();
31
32   flow::waitForSignal({SIGINT});
33
34   graph.stop();
35
36   return EXIT_SUCCESS;
37 }
```

Listing 3.10 main.cpp

3.2.3 Result

When running the program, include the path to the configuration file as a command line argument.

The output of the program is shown in listing 3.11. As we can see, the `Printer` proxel prints the expected fused results $X^{(a+b)} = X^{(2+3)}$: $0^5, 1^5, 2^5, 3^5$ and 4^5.

```
0
1
32
243
1024
```

Listing 3.11 Output of the tutorial processing graph (results.txt).

3.2.4 Monitoring status

In order to create a minimal working example, we utilised a helper function to pause the main thread while processing. A more common practice is to create a loop and let the main thread do other work, such as refreshing a GUI or printing graph status, while waiting for a condition that terminates the loop. Available through the Superflow library is an experimental ncurses GUI, that can be used to monitor the status of the processing graph from the command line. A sample output is shown in fig. 3.2.

```
┌-fuser--------------------┐ ┌-number_generator---------┐ ┌-power_0------------------┐
|state: UNDEFINED          | |state: UNDEFINED          | |state: RUNNING            |
|                          | |                          | |                          |
|                          | |                          | |                          |
|                          | |                          | |                          |
|                          | |                          | |                          |
└┌-in------┐┌-out-----┐-----┘ └┌-out-----┐---------------┘ └┌-in------┐┌-out-----┐-----┘
 |C:      2||C:      1|        |C:      2|                  |C:      1||C:      1|
 |T:      5||T:      5|        |T:      5|                  |T:      5||T:      5|
 └---------┘└---------┘        └---------┘                  └---------┘└---------┘

┌-power_1------------------┐ ┌-printer------------------┐
|state: RUNNING            | |state: UNDEFINED          |
|                          | |                          |
|                          | |                          |
|                          | |                          |
|                          | |                          |
└┌-in------┐┌-out-----┐-----┘ └┌-in------┐---------------┘
 |C:      1||C:      1|        |C:      1|
 |T:      5||T:      5|        |T:      5|
 └---------┘└---------┘        └---------┘
```

Figure 3.2 A terminal GUI for monitoring status is available through Superflow.

In the figure we recognise each proxel as a named box, with the ports attached at the bottom. The characters in the port boxes states the number of connections, ‘C’, and the number of transactions, ‘T’. At the time of capture, all ports had counted five transactions. In order to use the GUI, we include `"superflow/curses/graph_gui.h"`, create an object of `GraphGUI` and replace the `waitForSignal` we used earlier. The updated part of ‘main.cpp’ is shown below:

```
flow::curses::GraphGUI gui;

graph.start();
gui.spin(graph);
graph.stop();
```

3.2.5 CMakeLists.txt

For the sake of completeness, we have also included the file *CMakeLists.txt*. It is not in the scope of this tutorial to explain the language of CMake, but we show here how to link your application to the Superflow modules if you have a CMake based project.

```
1  cmake_minimum_required(VERSION 3.10.2)
2  project(tutorial)
3
4  find_package(superflow CONFIG REQUIRED)
5
6  add_executable(${PROJECT_NAME}
7    fuser.h
8    number-generator.h
9    power.h
10   printer.h
11   #
12   fuser.cpp
13   number-generator.cpp
14   power.cpp
15   printer.cpp
16   #
17   main.cpp
18 )
19
20 target_link_libraries(${PROJECT_NAME} PUBLIC
21   superflow::core
22   superflow::yaml
23 )
24
25 set_target_properties(${PROJECT_NAME} PROPERTIES
26   CXX_STANDARD_REQUIRED ON
27   CXX_STANDARD 17
28   )
```

Listing 3.12 CMakeLists.txt

In order to use the ncurses GUI, we must add `superflow::curses` to the `target_link_libraries` in *CMakeLists.txt* (lines 20 to 23).

3.3 Adding the loader module

In this part of the tutorial, we will utilise the *loader* module. We will reuse the code we have created so far, but it will be organised a little differently.

- The `Proxel`s are moved into a separate directory, and compiled into a library called `proxels`.
- We create a new library, `factories`, that adds the *loader* functionality to the `proxels` library.
- The application is also placed into a separate directory, called `app`.

Usually, we don't bother putting the factories in a separate library like this, but for the sake of the tutorial it makes a clear distinction between 'loader code' and regular 'proxel code'.

The structure of the project is shown in listing 3.13.

```
.
├── app
│   ├── CMakeLists.txt
│   ├── config.yaml
│   ├── main.cpp
│   └── usage.h
├── factories
│   ├── src
│   │   └── proxel-factories.cpp
│   └── CMakeLists.txt
├── proxels
│   ├── include
│   │   └── proxels
│   │       ├── fuser.h
│   │       ├── number-generator.h
│   │       ├── power.h
│   │       └── printer.h
│   ├── src
│   │   ├── fuser.cpp
│   │   ├── number-generator.cpp
│   │   ├── power.cpp
│   │   └── printer.cpp
│   └── CMakeLists.txt
└── CMakeLists.txt
```

Listing 3.13 File structure of the loader-tutorial.

3.3.1 CMakeLists.txt

By examining the CMake-files of the project, we get a good overview of how the different parts of the project plays together. Starting from the top, we will go through each of them and highlight the essential parts. We use the topmost file to tie the project together, as seen in listing 3.14.

Listing 3.15 is the CMakeLists for the `proxels` library. As we can see on lines lines 3 and 25, the library only depends on `superflow::core`. Besides that, the rest of the code is mostly boilerplate for creating a library. We do not include all the `export` and `install` CMake-statements that often follows when creating a relocatable library, as we simply rely on `add_subdirectory` in this tutorial.

```
cmake_minimum_required(VERSION 3.8)
project(tutorial)

set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_STANDARD 17)

add_subdirectory(proxels)
add_subdirectory(factories)
add_subdirectory(app)
```

Listing 3.14 Toplevel CMakeLists.txt

```
project(proxels)

find_package(superflow REQUIRED)

add_library(proxels
  src/fuser.cpp
  src/number-generator.cpp
  src/power.cpp
  src/printer.cpp
)
add_library(tutorial::proxels ALIAS proxels)

target_include_directories(proxels
  PRIVATE $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
)

target_include_directories(proxels
  SYSTEM INTERFACE
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
  $<INSTALL_INTERFACE:include>
)

target_link_libraries(proxels
  PUBLIC
  superflow::core
)

set_target_properties(proxels PROPERTIES
  POSITION_INDEPENDENT_CODE ON
)
```

Listing 3.15 Proxels library CMakeLists.txt

```
1  project(factories)
2
3  find_package(superflow REQUIRED)
4
5  add_library(factories SHARED
6    src/proxel-factories.cpp
7  )
8
9  target_link_libraries(factories
10   tutorial::proxels
11   superflow::loader
12   superflow::yaml
13 )
```

Listing 3.16 Factories library CMakeLists.txt

```
1  project(main)
2
3  find_package(superflow REQUIRED)
4
5  add_executable(main-loader main.cpp)
6
7  target_link_libraries(main-loader PUBLIC
8    superflow::core
9    superflow::curses
10   superflow::loader
11   superflow::yaml
12 )
```

Listing 3.17 Application CMakeLists.txt

Listing 3.16 is the CMakeLists for the `factories` library. Notice that we explicitly declare this a `SHARED` library. On lines 9 to 12 we see that the library links to both `superflow::loader` and `superflow::yaml` (as stated in section 2.6.5), since we will use the `REGISTER_PROXEL_FACTORY` macro to register `YAMLPropertyList` factories.

Listing 3.17 is the CMakeLists for our application. Again, we review the `target_link_libraries` on lines 7 to 11: The app links to `superflow::core` because, among other things, we want to create a `Graph`. We link to `superflow::curses` because we want visual feedback on the state of the processing, and lastly, we link to both `superflow::loader` and `superflow::yaml`, because we will use `flow::load::loadFactories<flow::yaml::YAMLPropertyList>` in the `main` function. The most interesting part, however, is perhaps that the list of libraries make no mention of the `proxels` library, nor the `factories`. We are thus creating an executable that, at compile time, knows nothing about the proxels it will run in its `Graph`.

3.3.2 Application

We move on to `main.cpp`, shown in listing 3.18. Compared to what we used earlier in the tutorial, there are a few differences. Firstly, we have added some error checking to our input arguments (lines 11 to 18), in order to pretty-print a helpful error message if arguments are not provided. We have isolated the actual `usage` function in its own header file for brevity. Feel free to create your own! The first argument to our program is the path to the *yaml* configuration file, and the second argument is the path to a directory containing the `factories` library. If our current working directory is the root of the project and we compile our binaries into a `build` subdirectory, the command to launch the program should be similar to

```
$ ./build/app/main ./app/config.yaml ./build/factories
```

The most important difference though, compared to the previous ‘main.cpp’ in listing 3.10, is that there are no references to any of the `Proxel`s. Inclusion of header files and hard coding of a `FactoryMap` are all gone! Everything is now taken care of by the *loader* module (lines 20 to 24). It might be confusing that even though we only load one library, we create a `std::vector<ProxelLibrary>`. The reason is simply that we decided to load `FactoryMap`s with the function `flow::load::loadFactories`, as it scales better if we later decide to load more libraries. For the record, an alternative for a single library can be

```
20 const flow::load::ProxelLibrary library{library_path, "factories"};
21 const auto factory_map = library.loadFactories<flow::yaml::YAMLPropertyList>();
```

The rest is similar to what we used before: create the `Graph` and GUI, start, spin, stop and return.

3.3.3 Factories

Now, we inspect ‘proxel-factories.cpp’, shown in listing 3.19. In this example, we decided to gather all macro calls in one file. The file is quite simple: The header file for each `Proxel` is included, in order to see the `class` declaration and the `static flow::Proxel::Ptr create()` that we created for all `Proxel`s in the previous part of the tutorial. Then, each `Factory` is simply registered with `REGISTER_PROXEL_FACTORY`. Remember that the `factories` library is linked to the `yaml` library, which defines all other values internally required by the macro.

3.3.4 Proxel files

Review the files created earlier for the `proxels` library in listings 3.1 to 3.8. The files are mostly unchanged for the current part of the tutorial, besides reorganising them as shown in the file hierarchy tree in listing 3.13, and adding some more status info through `setStatusInfo`.

```cpp
#include "usage.h"
#include "superflow/curses/graph_gui.h"
#include "superflow/graph.h"
#include "superflow/loader/proxel_library.h"
#include "superflow/yaml/yaml.h"

#include <iostream>

int main(int argc, char** argv)
{
  if (argc < 3)
  {
    std::cerr << usage(argv[0]) << std::endl;
    return EXIT_FAILURE;
  }

  const std::string config_path{argv[1]};
  const std::string library_path{argv[2]};

  const std::vector<flow::load::ProxelLibrary> libraries{
    {library_path, "factories"}
  };
  const auto factory_map =
    flow::load::loadFactories<flow::yaml::YAMLPropertyList>(libraries);

  flow::Graph graph = flow::yaml::createGraph(config_path, factory_map);
  flow::curses::GraphGUI gui;

  graph.start();
  gui.spin(graph);
  graph.stop();

  return EXIT_SUCCESS;
}
```

Listing 3.18 main.cpp

```cpp
#include "superflow/loader/register_factory.h"

#include "proxels/fuser.h"
#include "proxels/number-generator.h"
#include "proxels/power.h"
#include "proxels/printer.h"

REGISTER_PROXEL_FACTORY(Fuser, Fuser::create)
REGISTER_PROXEL_FACTORY(NumberGenerator, NumberGenerator::create)
REGISTER_PROXEL_FACTORY(Power, Power::create)
REGISTER_PROXEL_FACTORY(Printer, Printer::create)
```

Listing 3.19 proxel-factories.cpp

3.3.5 Running the program

Finally, we compile and run our project:

```
$ cmake -B build
$ cmake --build build

$ build/app/main app/config.yaml build/factories
```

```
┌-fuser----------------┐ ┌-number_generator-----┐ ┌-power_0--------------┐
|state: UNDEFINED      | |state: PAUSED         | |state: RUNNING        |
|prod(16*64) = 1024    | |4                     | |4^2 = 16              |
|                      | |                      | |                      |
|                      | |                      | |                      |
|                      | |                      | |                      |
└┌-in-----┐┌-out----┐--┘ └┌-out----┐------------┘ └┌-in-----┐┌-out----┐--┘
 |C:     2||C:     1|     |C:     2|               |C:     1||C:     1|
 |T:     5||T:     5|     |T:     5|               |T:     5||T:     5|
 └--------┘└--------┘     └--------┘               └--------┘└--------┘

┌-power_1--------------┐ ┌-printer--------------┐
|state: RUNNING        | |state: UNDEFINED      |
|4^3 = 64              | |1024                  |
|                      | |                      |
|                      | |                      |
|                      | |                      |
└┌-in-----┐┌-out----┐--┘ └┌-in-----┐------------┘
 |C:     1||C:     1|     |C:     1|
 |T:     5||T:     5|     |T:     5|
 └--------┘└--------┘     └--------┘
```

Figure 3.3 The curses GUI showing the state after a successful run of the loader *tutorial.*

With that, we now conclude our very brief but quite comprehensive guide on how you can apply Superflow in your application. We hope you find Superflow useful, and encourage your feedback, suggestions and pull requests to help us improve it over time!

References

[1] Marvel Database. *Superflow*. 2018. URL: http://marvel.wikia.com/wiki/Superflow (visited on 03/10/2018).

[2] Trym Vegard Haavardsholm, Ragnar Smestad, Martin Vonheim Larsen, Marius Thoresen and Idar Dyrdal. ‘Scene Understanding for Autonomous Steering’. In: *STO-MP-IST-127: Intelligence and Autonomy In Robotics*. NATO Science and Technology Organization, 2016.

[3] Kent Beck, James Grenning, Robert C. Martin, Mike Beedle, Jim Highsmith, Steve Mellor, Arie van Bennekum, Andrew Hunt, Ken Schwaber, Alistair Cockburn, Ron Jeffries, Jeff Sutherland, Ward Cunningham, Jon Kern, Dave Thomas, Martin Fowler and Brian Marick. *Manifesto for Agile Software Development*. 2001. URL: http://agilemanifesto.org/ (visited on 07/12/2018).

[4] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (2nd Edition)*. 2001. ISBN: 0131489062.

[5] Robert C Martin. *Agile Software Development, Principles, Patterns, and Practices*. 2003. ISBN: 0135974445.

[6] E. W. Dijkstra. ‘Information streams sharing a finite buffer’. In: *Information Processing Letters*. Vol. 1. 1972, pp. 179–180.

[7] Scott Meyers. *Effective modern C++: 42 specific ways to improve your use of C++ 11 and C++ 14*. " O’Reilly Media, Inc.", 2014.

[8] Oren Ben-Kiki, Clark Evans and Ingy döt Net. *Yaml ain’t markup language (yaml™) version 1.2*. 2009. URL: https://yaml.org/spec/1.2/spec.html (visited on 29/01/2018).

[9] Jesse Beder. *yaml-cpp*. URL: https://github.com/jbeder/yaml-cpp (visited on 12/12/2018).

[10] *Boost C++ Libraries*. URL: https://www.boost.org/doc/libs/master/doc/html/boost_dll.html (visited on 09/11/2023).

[11] *When is std::shared_timed_mutex slower than std::mutex and when (not) to use it?* URL: https://stackoverflow.com/a/56159440/14325545 (visited on 27/11/2023).

[12] *PImpl*. URL: https://en.cppreference.com/w/cpp/language/pimpl (visited on 27/11/2023).

[13] Herb Sutter. *Sutter’s Mill, GotW #101: Compilation Firewalls, Part 2*. URL: https://herbsutter.com/gotw/_101/ (visited on 27/11/2023).

About FFI

The Norwegian Defence Research Establishment (FFI) was founded 11th of April 1946. It is organised as an administrative agency subordinate to the Ministry of Defence.

FFI's mission

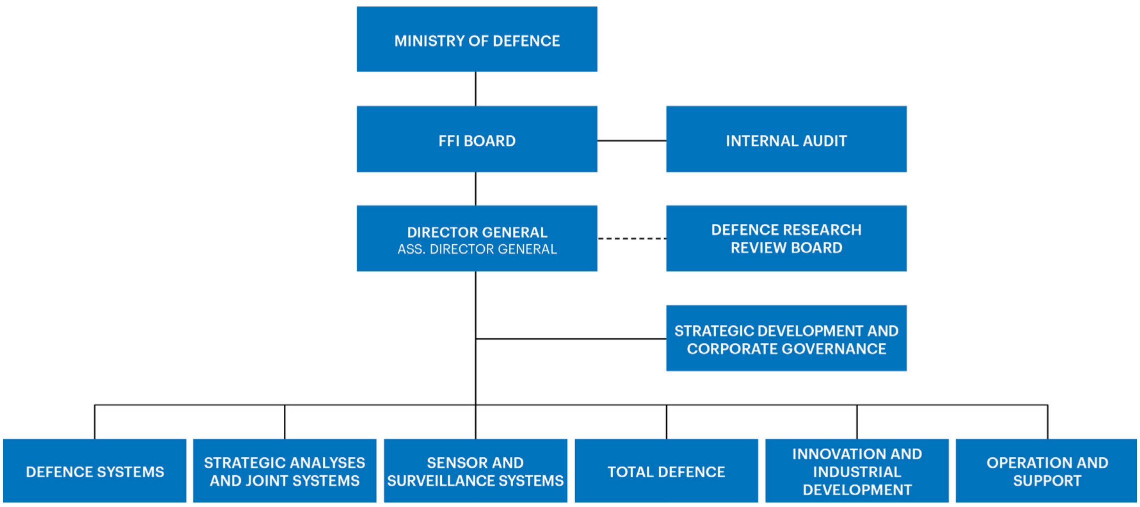
FFI is the prime institution responsible for defence related research in Norway. Its principal mission is to carry out research and development to meet the requirements of the Armed Forces. FFI has the role of chief adviser to the political and military leadership. In particular, the institute shall focus on aspects of the development in science and technology that can influence our security policy or defence planning.

FFI's vision

FFI turns knowledge and ideas into an efficient defence.

FFI's characteristics

Creative, daring, broad-minded and responsible.



Forsvarets forskningsinstitutt (FFI)
Postboks 25
2027 Kjeller

Besøksadresse:
Kjeller: Instituttveien 20, Kjeller
Horten: Nedre vei 16, Karljohansvern, Horten

Telefon: 91 50 30 03
E-post: post@ffi.no
ffi.no

Norwegian Defence Research Establishment (FFI)
PO box 25
NO-2027 Kjeller
NORWAY

Visitor address:
Kjeller: Instituttveien 20, Kjeller
Horten: Nedre vei 16, Karljohansvern, Horten

Telephone: +47 91 50 30 03
E-mail: post@ffi.no
ffi.no/en