Introduction to FastFlow programming

Massimo Torquati
<massimo.torquati@unipi.it>

SPM lecture 2

Master Degree in Computer Science
Master Degree in Computer Science & Networking
University of Pisa

Brief recap

```
virtual TOUT* svc(TIN* task) = 0;    // encapsulates user's code
virtual int   svc_init();            // initialization code
virtual void  svc_end();             // finalization code
```

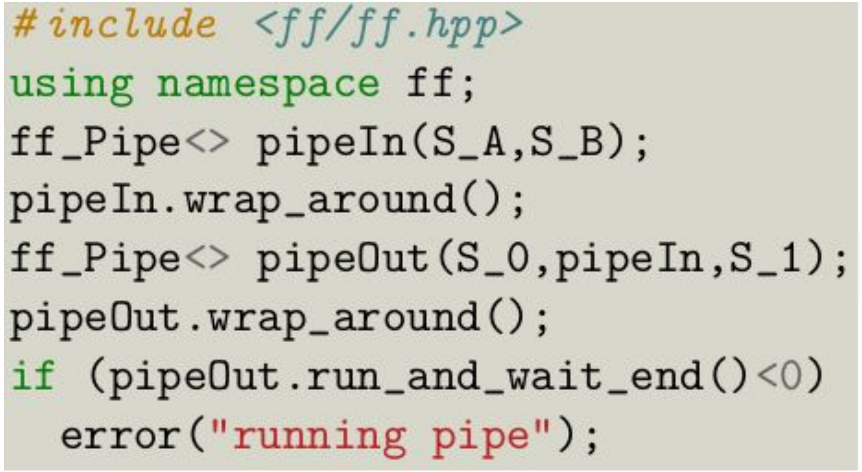
- **svc_init()** is called once at the beginning each time the thread associated to the node is (re-)started
- **svc()** is called for each input item present in one of the input queues. Its return value can be:
  - A reference to the result data
  - EOS (End-Of-Stream)
  - GO_ON
  - A few other special values can be returned
- **eosnotify()** is called each time an EOS message is received by the node. It is possible to send data into the output queues.
- **svc_end()** is called before terminating the node (or when the node is put to sleep). It is not possible to send any "final" result into the output queues.

Sequential node life- cycle (simplified):

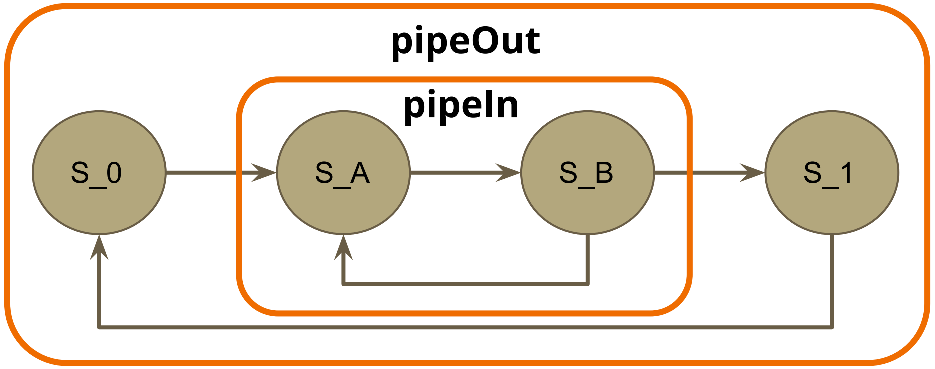
```
do {
  if (svc_init() < 0) break;
  do {
    in = input_channel.pop();
    if (in == EOS) {
      // if this method has been redefined, the user's method
      // is called and informed that the EOS has arrived
      eosnotify();
      output_channel.push(EOS);
    } else {
      out = svc(in);  // it calls the business logic code
      if (out == GO_ON) continue;
      output_channel.push(out);
    }
  } while(out != EOS);
  svc_end();
} while(true);
```

Feedback channels

- Feedback channel: its data flow is directed in the opposite direction than the standard data flow (backward channel)
- Used to route back results, or to provide control information to a previous node
- They always have unbounded capacity to avoid deadlock



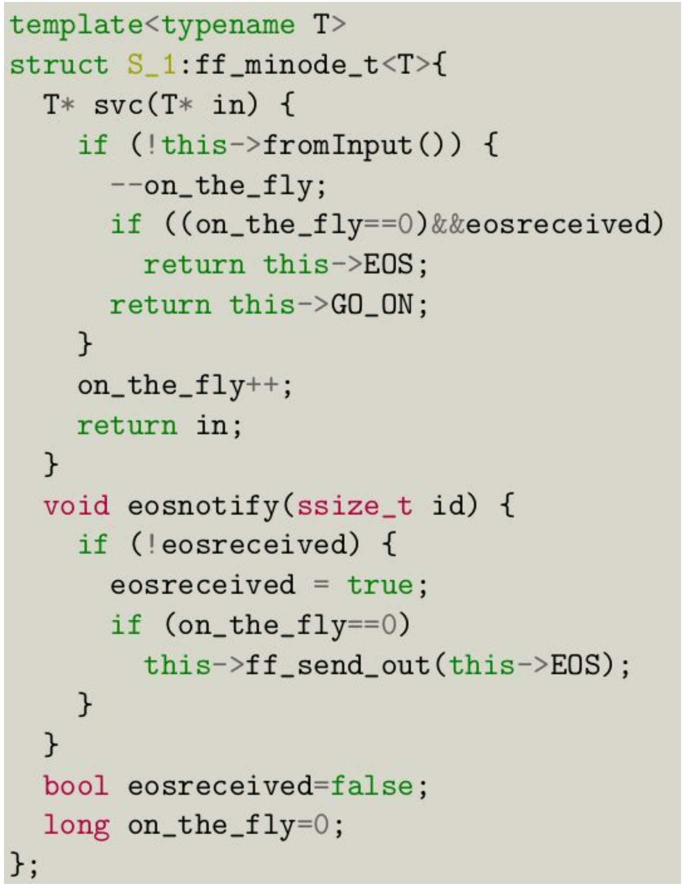
```
#include <ff/ff.hpp>
using namespace ff;
ff_Pipe<> pipeIn(S_A,S_B);
pipeIn.wrap_around();
ff_Pipe<> pipeOut(S_0,pipeIn,S_1);
pipeOut.wrap_around();
if (pipeOut.run_and_wait_end()<0)
  error("running pipe");
```



- Let’s have a look at a simplified version of this code (termination_with_loops.cpp), with only the internal loop

Handling termination with feedback channels

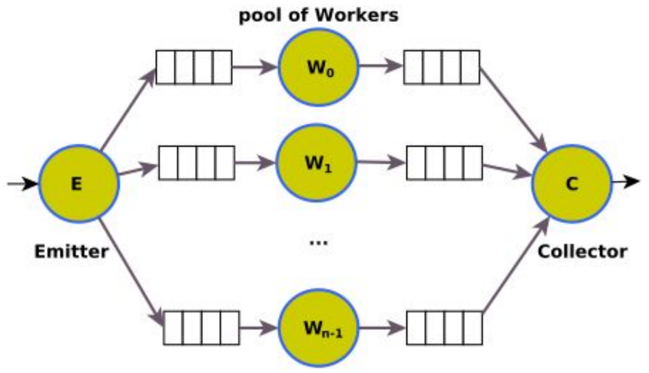
- Feedback channels complicate program termination
- **Standard termination:** A node terminates when it receives an EOS message from all its input channels. Then it forwards the EOS into all its output channels and then calls *svc_end()*
- **In case of feedbacks**, the previous condition might not hold. With cyclic graphs, we need to use the *eosnotify()* method to be notified each time an EOS is received, and depending on some application-specific conditions to properly forward the EOS at the right moment (see, for example termination_with_loop.cpp)



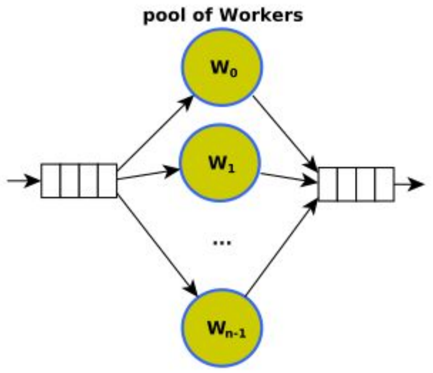
```
template<typename T>
struct S_1:ff_minode_t<T>{
  T* svc(T* in) {
    if (!this->fromInput()) {
      --on_the_fly;
      if ((on_the_fly==0)&&eosreceived)
        return this->EOS;
      return this->GO_ON;
    }
    on_the_fly++;
    return in;
  }
  void eosnotify(ssize_t id) {
    if (!eosreceived) {
      eosreceived = true;
      if (on_the_fly==0)
        this->ff_send_out(this->EOS);
    }
  }
  bool eosreceived=false;
  long on_the_fly=0;
};
```

Farm pattern

$(\texttt{farm}\ \Delta) : \alpha\ \mathsf{stream} \to \beta\ \mathsf{stream}$ where Δ is any pattern having input type $\alpha\ \mathsf{stream}$ and output type $\beta\ \mathsf{stream}$. The semantics is such that all the items $x_i : \alpha$ are processed and their output items $y_i : \beta$ where $y_i = f(x_i)$ computed. From the parallel semantics viewpoint, within the farm the pattern Δ is replicated $n \geq 1$ times (n is a non-functional parameter of the pattern called *parallelism degree*) and, in general, the input items may be computed in parallel by the different instances of Δ.

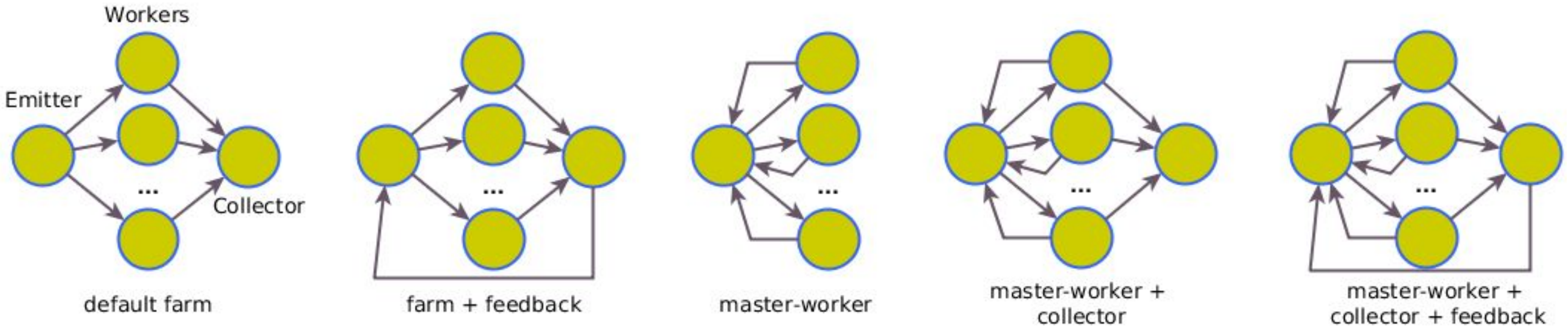


a)



b)

Farm skeletons in FastFlow



Farm

```
struct myNode: ff_node_t<myTask> {
   myTask *svc(myTask * t) {
      F(t);
      return GO_ON;
}};

std::vector<std::unique_ptr<ff_node>> W;
W.push_back(make_unique<myNode>());
W.push_back(make_unique<myNode>());

ff_Farm<myTask>
                  myFarm(std::move(W));

ff_Pipe<myTask>
    pipe(_1, myFarm, <...other stages...>);

pipe.run_and_wait_end();
```

- Farm’s Workers are ff_node_t objects
- The number of replicas (parallelism degree) is given by the size of the vector of node objects
- A farm itself is a node (farms can be nested)
- By default the ff_Farm has an Emitter and a Collector. The Collector can be removed by using *remove_collector()*:
  - my*Farm.remove_collector();*
- Emitter and Collector may be redefined by the user through suitable ff_node_t objects (also multi-input/output)
- Default task scheduling is **round-robin**. If the channels have bounded capacity is **loose round-robin**
- Auto-scheduling is provided if:
  - *myFarm.set_scheduling_ondemand()*
- Possibility to implement user's specific scheduling strategies by adding a **ff_monode_t** object as Emitter, and using the **ff_send_out_to** method to explicitly schedule tasks

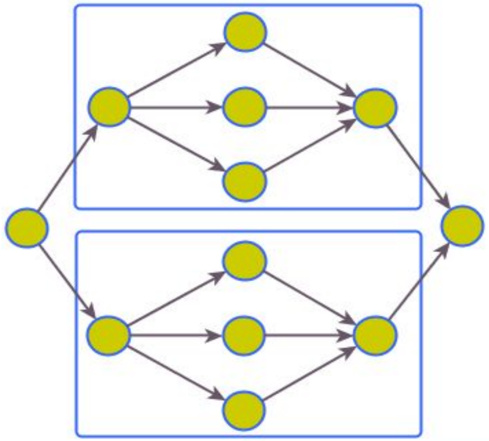
Farm

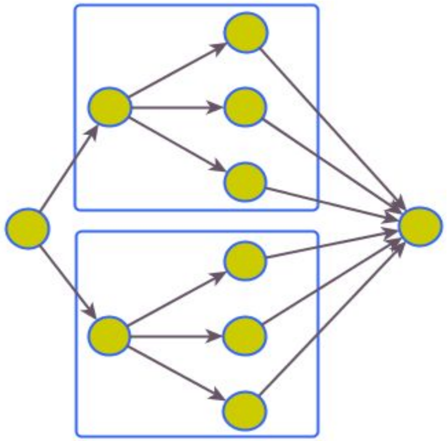
```
myTask *F(myTask * t,ff_node*const) {
    .... <work on t> ....
    return t;
}

ff_Farm<myTask> myFarm(F, 5);
```

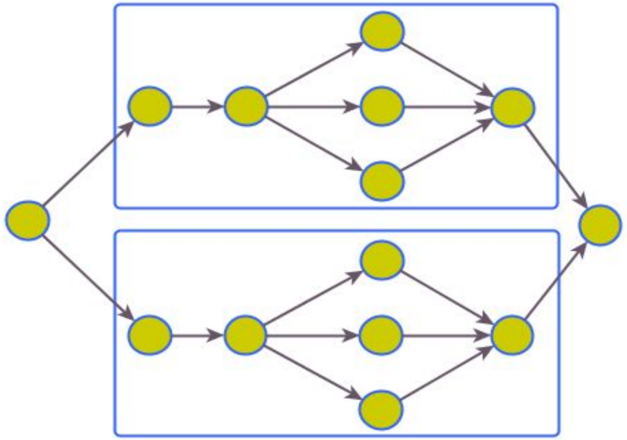
```
myTask *F(myTask * t,ff_node*const) {
    .... <work on t> ....
    return t;
}

ff_OFarm<myTask> myFarm(F, 5);
```

- Simpler syntax, it is possible to create a ff_Farm through a function (F) and setting the number of replicas
- The function must have a given signature!
- The ff_OFarm is a farm pattern that preserves input ordering (we will discuss ordering issues later)

Nesting of pipe and farm

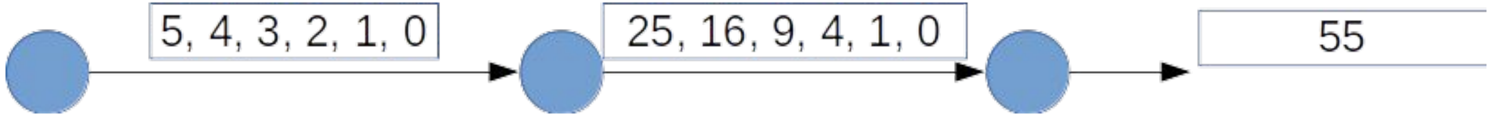




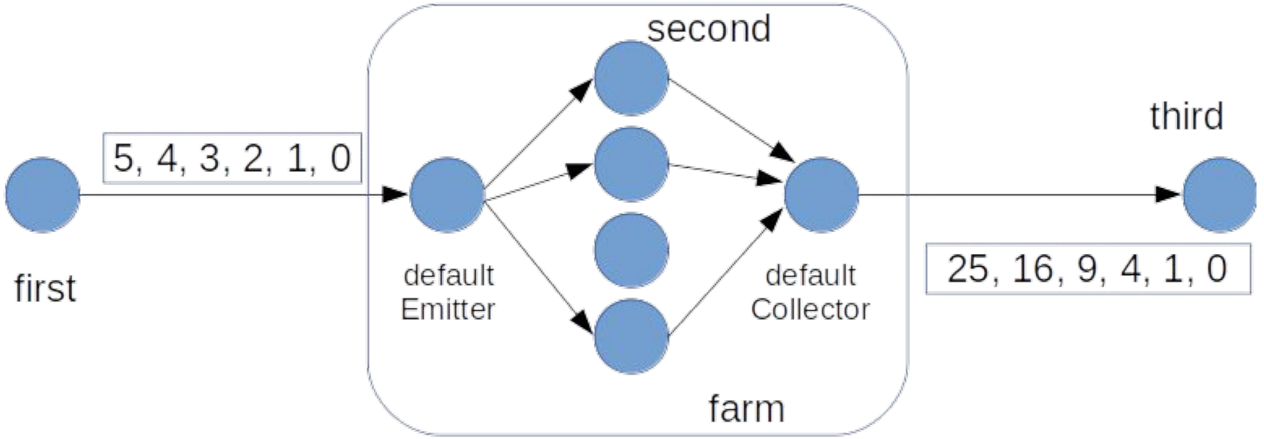


Farm square example

- Let's consider again the square example: pipe(seq, seq, seq)



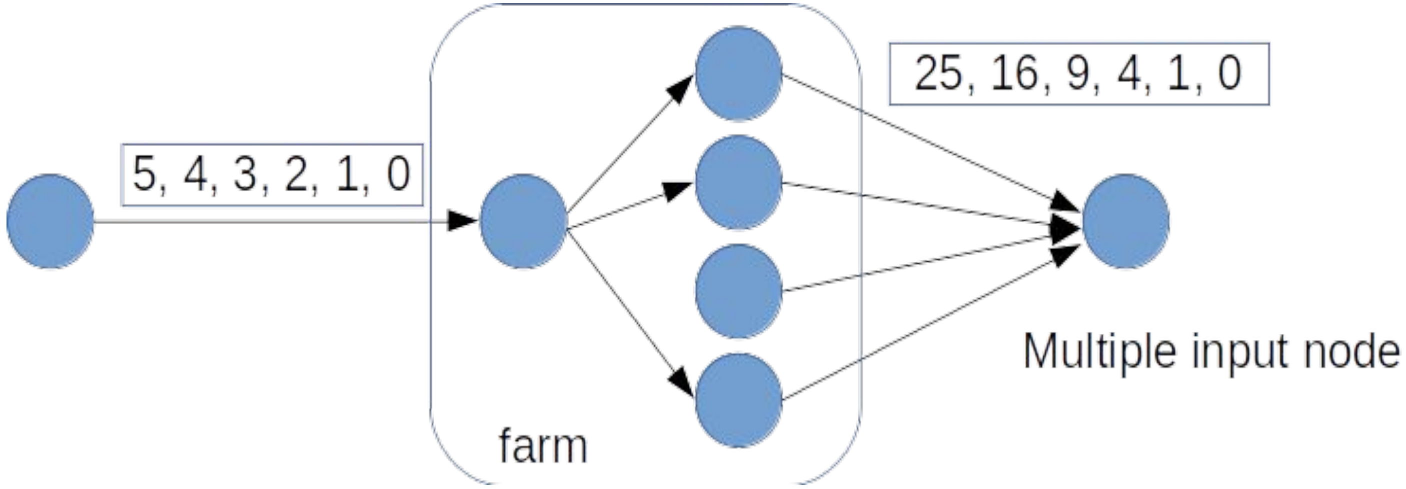
- The second stage can be replicated pipe(seq, farm(seq), seq)



- Let’s see the code (**farm_square1.cpp**)!

Farm square example revised

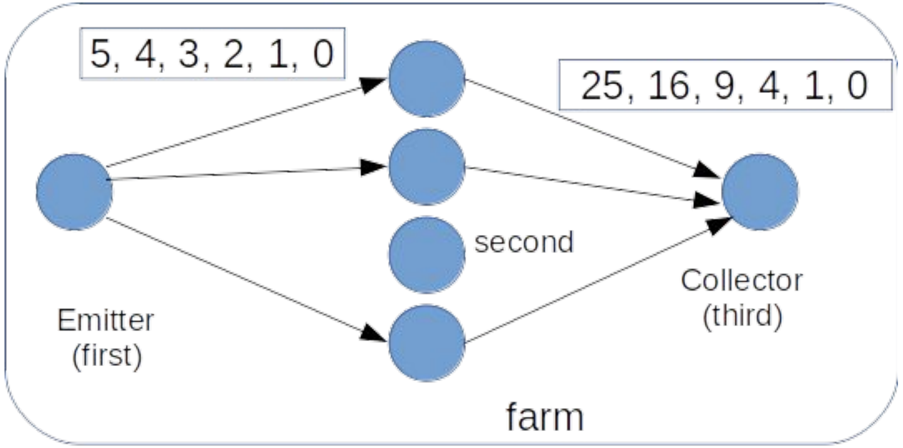
- The Collector of the farm can be removed but the third stage must be defined as multi-input node (**ff_minode_t**)



- Let’s see the code (**farm_square2.cpp**)!

Farm square example revised

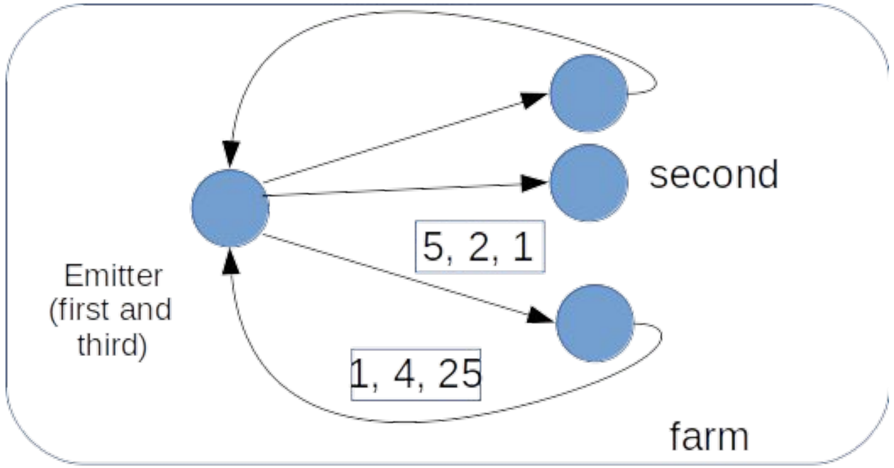
- We can fuse the first stage within the Emitter and the last stage within the Collector. The result is a single ff_Farm.



- The Emitter acts as a multi-output node, the Collector acts as a multi-input node
- Let’s see the code (**farm_square3.cpp**)!

Farm square example revised

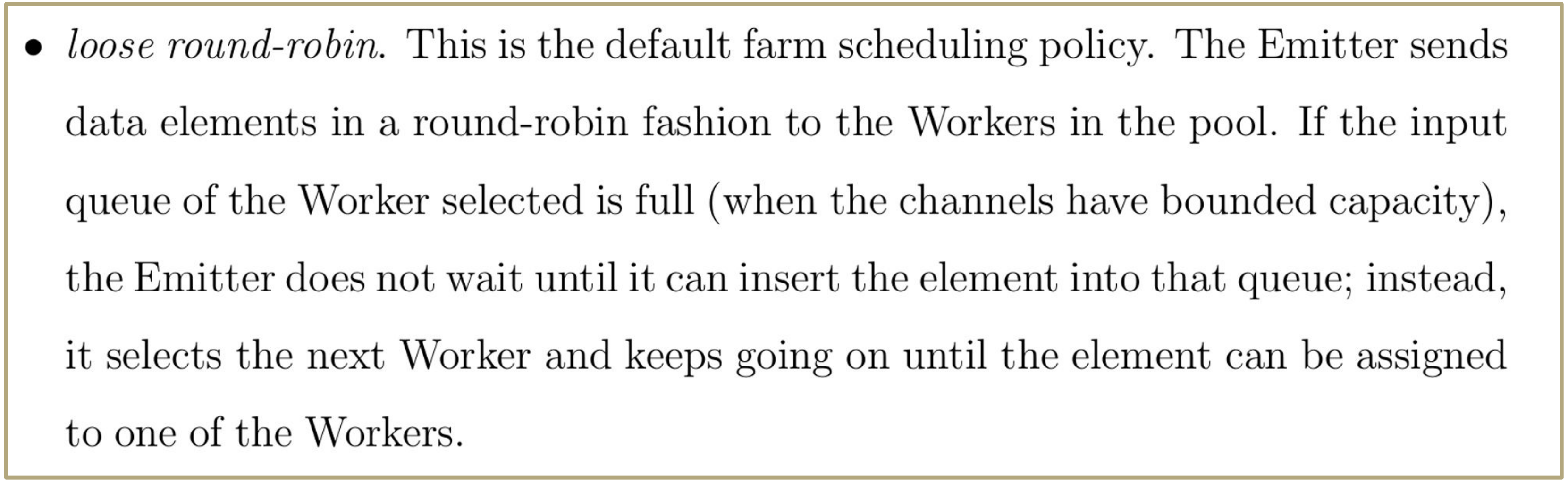
- Let’s build a master-worker version



- Let’s see the code (farm_square4.cpp)!

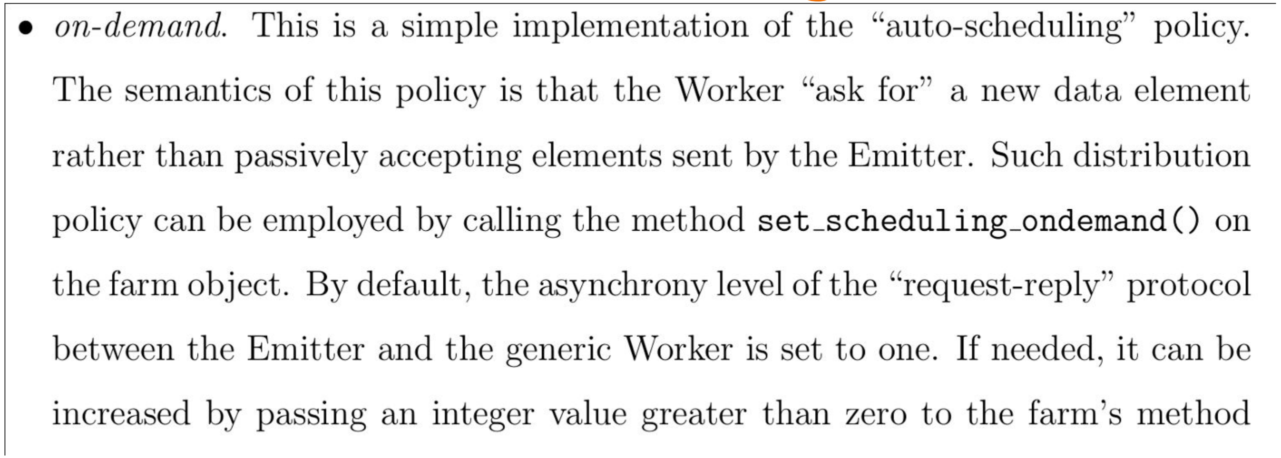
Round-robin and "loose" round-robin scheduling

- By default, FastFlow communication channels have unbounded capacity (see Lesson1 the part related to uSPSC queues)
- By compiling the program with -DFF_BOUNDED_BUFFER, all channels (but the feedback ones!) have *bounded capacity*
    - Buffer capacity can be controlled system-wide with -DDEFAULT_BUFFER_CAPACITY=<N>

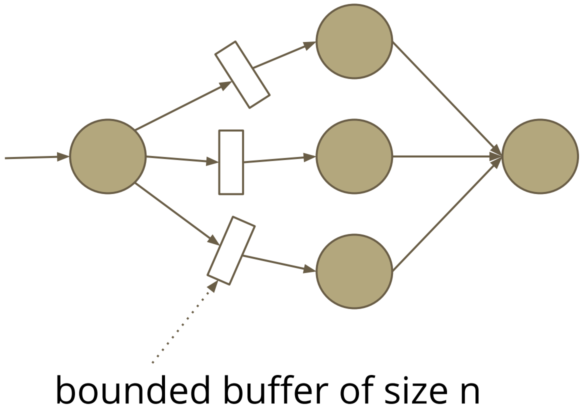


- *loose round-robin*. This is the default farm scheduling policy. The Emitter sends data elements in a round-robin fashion to the Workers in the pool. If the input queue of the Worker selected is full (when the channels have bounded capacity), the Emitter does not wait until it can insert the element into that queue; instead, it selects the next Worker and keeps going on until the element can be assigned to one of the Workers.

On-demand scheduling



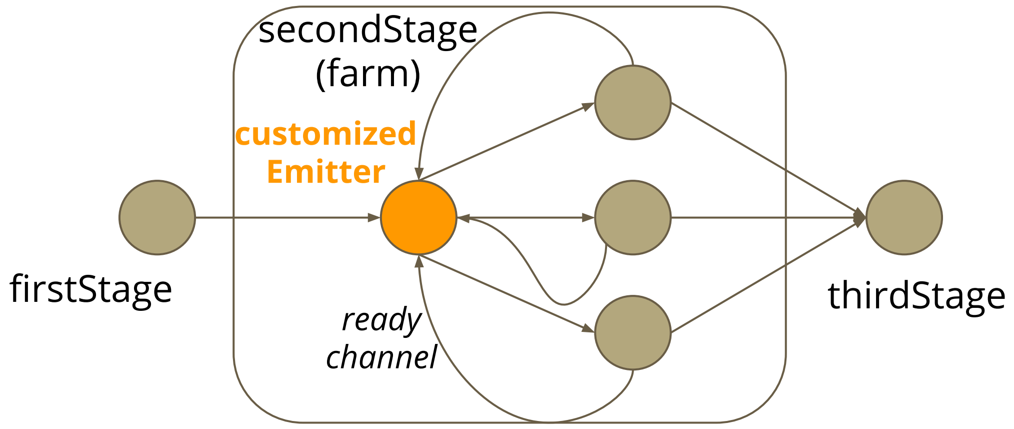
- *on-demand.* This is a simple implementation of the "auto-scheduling" policy. The semantics of this policy is that the Worker "ask for" a new data element rather than passively accepting elements sent by the Emitter. Such distribution policy can be employed by calling the method `set_scheduling_ondemand()` on the farm object. By default, the asynchrony level of the "request-reply" protocol between the Emitter and the generic Worker is set to one. If needed, it can be increased by passing an integer value greater than zero to the farm's method



- The set_scheduling_ondemand(n) call, emulates the on-demand scheduling policy
  - It uses a bounded queue of size n (by default n=1) between the Emitter and the Worker
  - If the queue has free space it means the Worker is "*ready*"

Implementing a real on-demand scheduling

- We need to customize the Emitter and we need a feedback channel from each Worker telling the Emitter whether it has completed the task assigned, so it is “ready” for a new task

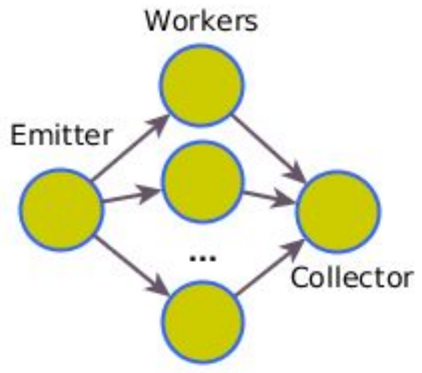


- Let’s comment the farm_square_multioutput.cpp example implementing a “real” on-demand scheduling policy

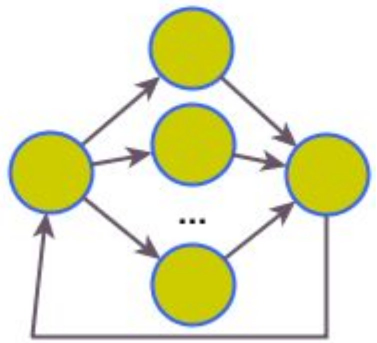
Ordering issues

- The default implementation of the farm pattern does not preserve the input ordering of data items
- In some applications preserving input ordering is mandatory (e.g. video streaming)
- FastFlow provides the **ff_OFarm** pattern, which preserves input ordering when producing results
- Limitations:
    - Workers cannot change the length of the stream
    - Workers can only be standard sequential nodes
- It is possible to use on-demand scheduling and also to define user-defined scheduling policies by defining the Emitter and the Collector
- Rule-of-Thumb: If you do not need a strict input/output ordering, it is generally better (from the performance standpoint) to use a standard farm and to implement your own policy by redefining the Emitter scheduling and the Collector gathering policies

Ordered Farm skeletons in FastFlow



default farm

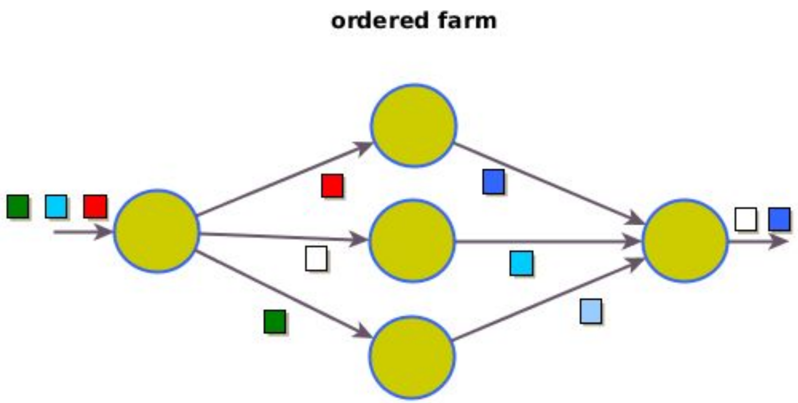


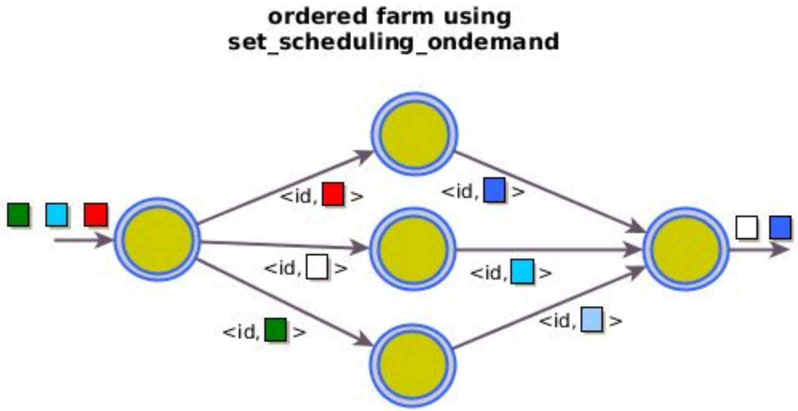
farm + feedback

Ordered Farm implementation

- Custom scheduler and custom collector used in ff_OFarm:
  - scheduling of data elements is strict round-robin (even with bounded queues)
  - gathering follows the same ordering used in data distribution
- This simple (and effective) policy does not work well if the workload is unbalanced
  - That is, a few data items (e.g., those ones that will be assigned to the same worker-id) have much higher computational costs
- For non-deterministic scheduling (e.g., on-demand, random) it is needed to tag input data
  - Data-tagging is automatically implemented in the ff_OFarm if *set_scheduling_ondemand* is used

Ordered Farm implementation





- Let’s have a look again at the square example (**farm_square_ordered_unbalanced.cpp**)