FastFlow: targeting distributed systems

Massimo Torquati

May 17th, 2012 (rev. 1)

November, 27th, 2013 (rev. 2)

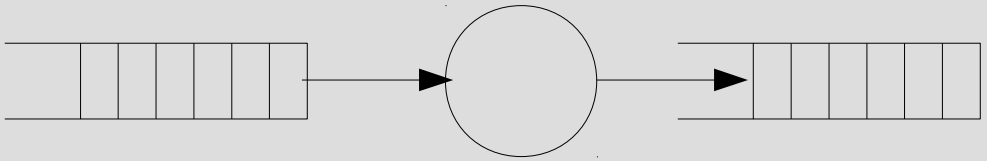
torquati@di.unipi.it

http://www.di.unipi.it/~torquati

http://mc-fastflow.sourceforge.net

FastFlow node

- FastFlow's implementation is based on the concept of node (ff_node class)
- A node is an abstraction which has an input and an output queue where another node can be connected to (queues can be bounded or unbounded)

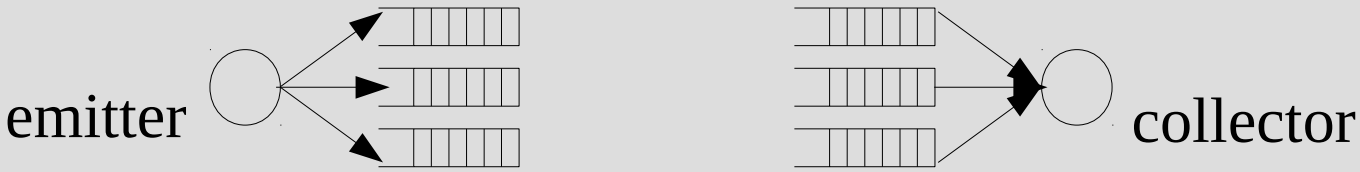


generic node

- Operations: ***get*** from the input queue, ***put*** to the output queue

FastFlow node (2)

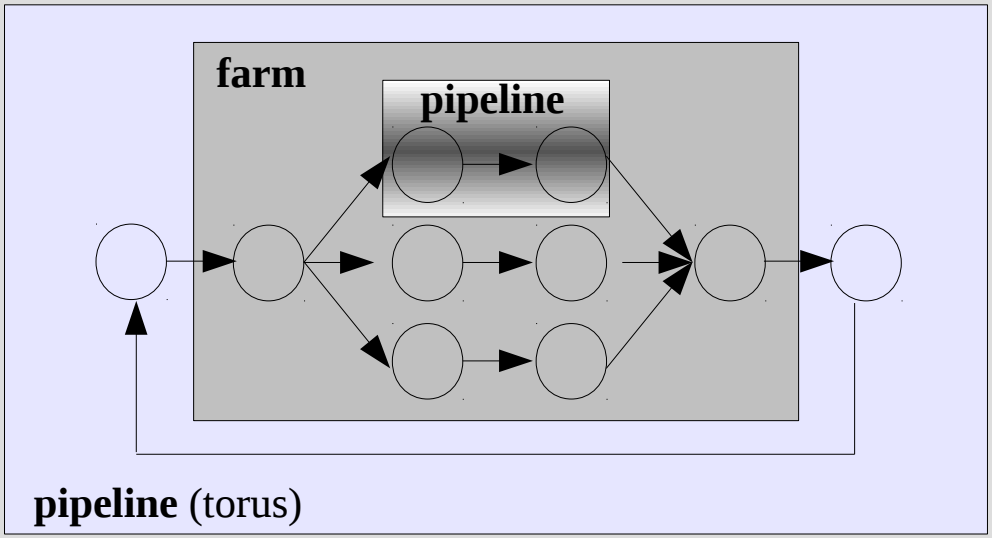
- A sequential node is eventually (at run-time) a ***posix-thread***
- There are 2 “special” nodes used in the *farm* skeleton which provide SPMC and MCSP queues using an active thread for scheduling and gathering policies control



- An ongoing work is to implement SPMC and MCSP as a lock-free CDS

Nodes composition

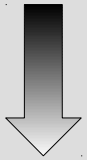
- A node can be a sequential node, a *pipeline*, a *farm* or a combination of them
  - The model exposed is a *streaming network* model



- NOTE: there are some limitations on the possible nesting of nodes when cycles are present

Scaling to multiple SMP workstations

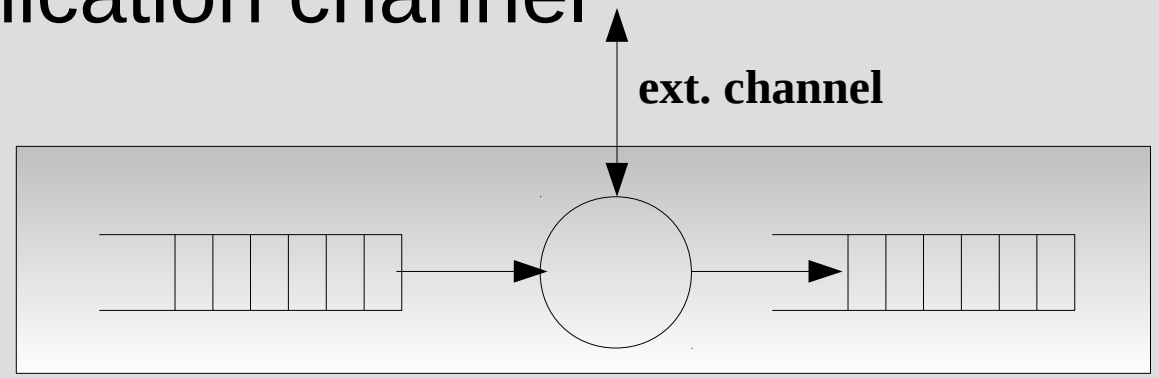
- We need to scale to hundreds/thousands of cores



- We have to use more than a single multi-core platform
- The streaming network model exposed by FastFlow, can be easily extended to work outside the single workstation

From node to dnode

- A dnode (class ff_dnode) is a node (i.e. extends the ff_node class) with an external communication channel

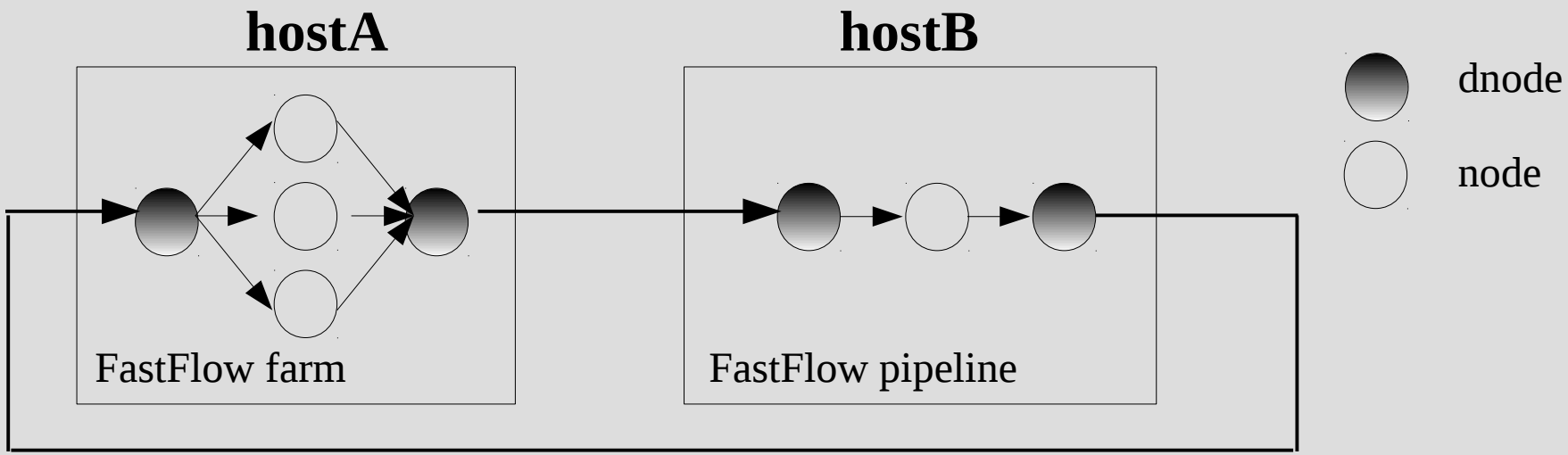


generic dnode

- The **ext. channel** is specialized to be an input or an output channel
  - Can be both input and output node in the newer version of FF : ff_dinout abstraction

From node to dnode (2)

- The main idea is to have only the ***edge nodes*** of the FastFlow network to be able to “talk to” the outside world



- Actually in the above scenario we have 2 FastFlow applications connected together

ff_dnode class sketch

- The ff_dnode offers the same interface of the ff_node
- In addition it encapsulates the external channel whose type is passed as template parameter
- The init method creates and initializes the communication end-point

emacs@pirotecnia

File Edit Options Buffers Tools C++ Help

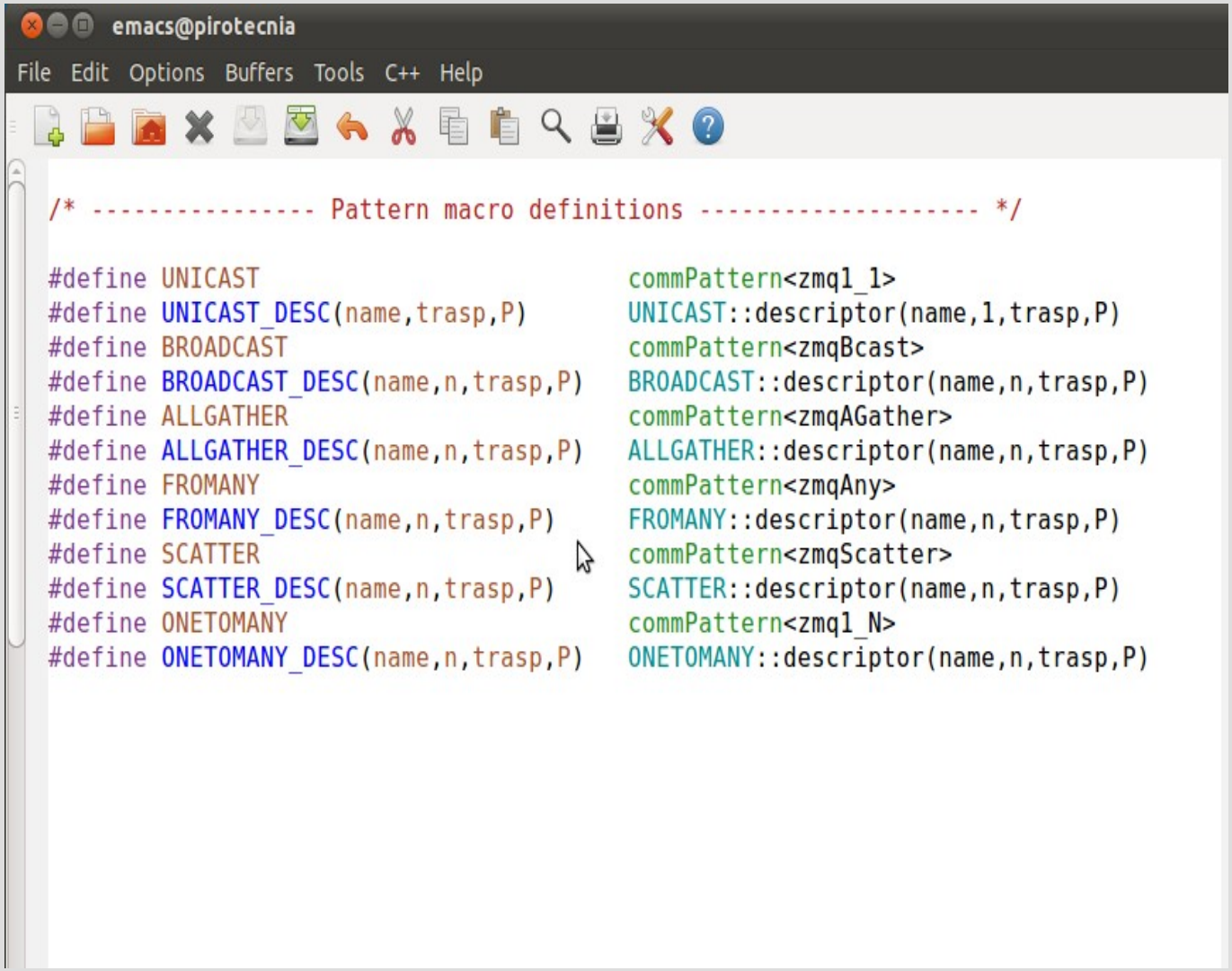
```
// dnode class
template <typename CommImpl>
class ff_dnode: public ff_node {
....

public:
  // initializes the dnode
  int init(const std::string& name, const std::string& address,
           const int peers, typename CommImpl::TransportImpl* const transp,
           const bool p, dnode_cbk_t cbk=0);

  int run(bool=false)  { return ff_node::run();}
  int wait()           { return ff_node::wait();}

...

protected:
  CommImpl com;
};
```

Available communication patterns

- Unicast (one-to-one)
- Broadcast
- Scatter
- One-To-Many
- All Gather
- Collect from Any
- On-demand

TODO

- Many-To-One (the opposite of One-To-Many)



Communication pattern interface

- ***init*** and ***close***
- *The descriptor contains all implementation details*
- ***get*** and ***put*** interface
- ***putmore*** used for multipart message (sender-side)
- **done** used for multipar message (receiver-side)

emacs@pirotecnia

File Edit Options Buffers Tools Minibuf Help

```
// Communication Pattern interface
template <typename Impl>
class commPattern  {
protected:
    Impl impl;
public:
    typedef typename Impl::descriptor      descriptor;
    typedef typename Impl::tosend_t        tosend_t;
    typedef typename Impl::torecv_t        torecv_t;

    // creates a communication pattern
    commPattern():impl() {}
    // @param D is an implementation-based descriptor for the pattern
    // it contains all the low-level implementation details
    commPattern(descriptor* D):impl(D) {}
    // sets the descriptor
    inline void setDescriptor(descriptor* D) {  impl.setDescriptor(D); }
    // returns the descriptor
    inline  descriptor* getDescriptor() { return impl.getDescriptor(); }
    // initializes communication pattern
    inline bool init(const std::string& address) { return impl.init(address); }
    // Specifies that the message being sent is just a part of the whole msg.
    inline bool putmore(const tosend_t& msg) { return impl.putmore(msg);}
    // sends one message
    inline bool put(const tosend_t& msg) { return impl.put(msg); }
    // receives one message part
    inline bool get(torecv_t& msg) { return impl.get(msg); }
    // all messages have been received
    inline void done() { impl.done();}
    // close pattern
    inline bool close() { return impl.close();  }
};
```

Communication patterns implementation

- At moment, the ***external channel*** of the dnode is implemented using the ZeroMQ library
- The implementation uses the TCP/IP transport layer
- We have planned to add more implementations based on different messaging framework
  - we have implemented a trasparent porting of distributed FastFlow over native infiniband transport (Linux verbs)

ZeroMQ messaging framework (1)

- ZeroMQ (or ØMQ) is a communication library
- It provides you a socket layer abstraction
- Sockets carry whole messages across various transports:
  - in-process (threads), inter-proess, TCP/IP, multicast
- ØMQ is quite easy to use
- It is efficient enough to be used in cluster environment

ZeroMQ messaging framework (2)

- ZeroMQ offers an asynchronous I/O model
- Runs on most operating systems (Linux, Windows, OS X)
- Supports many programming languages: C++, Java, .NET, Python, C#, Erlang, Perl, ……
- It is open-source, LGPL license
- Lots of documentation and examples available
  - take a look at: **www.zeromq.org**

ZeroMQ messaging framework (3)

- Sockets can be used with different communication patterns
  - Not only classical bidirectional communication between 2 peers (point-to-point)
- ØMQ offers the following patterns:
  - request/reply, publish/subscribe, push/pull
- Communication patterns can be directly used in your application to solve specific communication need:
  - take a look at ***zguide.zeromq.org*** for more details

ZeroMQ Hello World

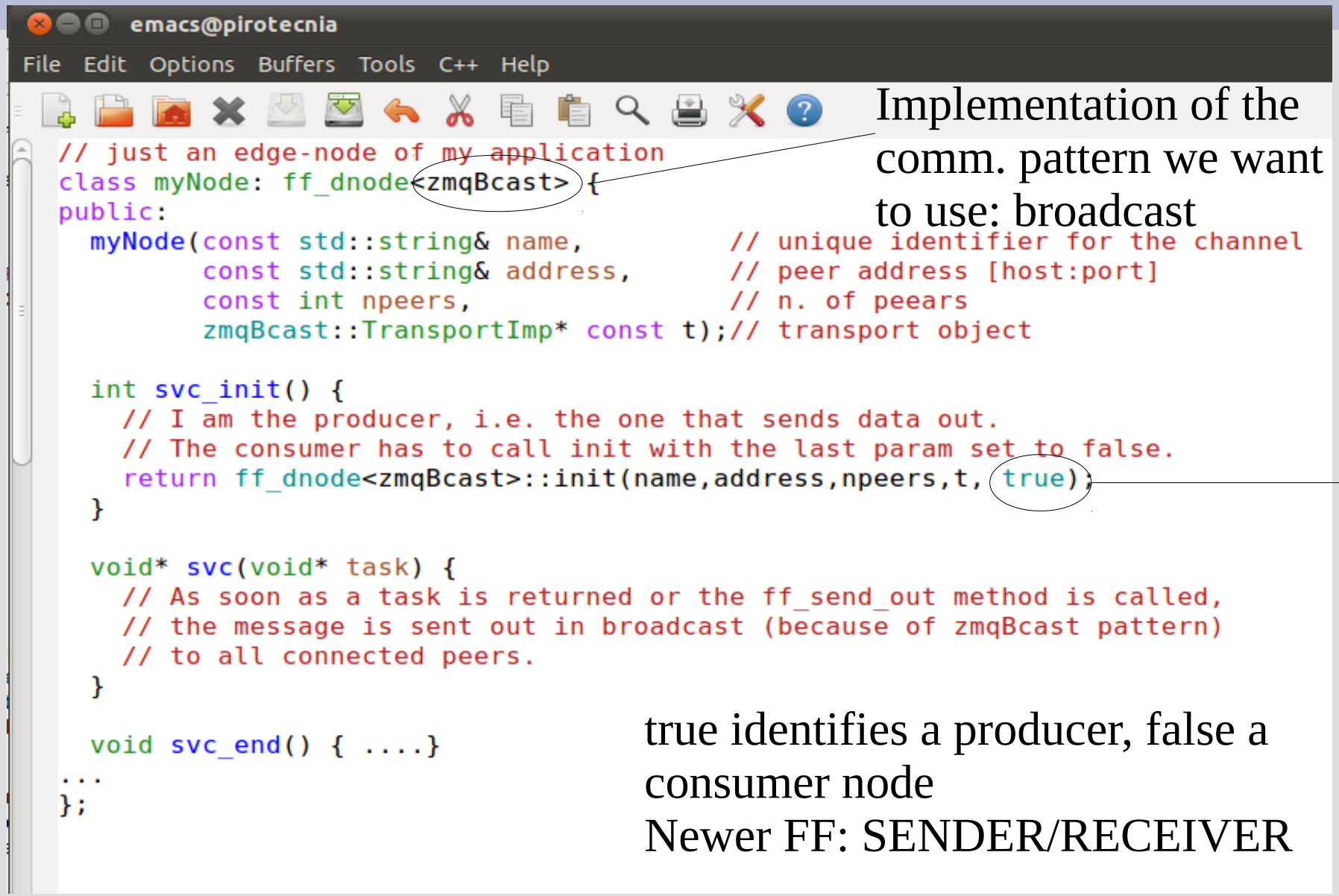
```c
int main (void)
{
    void *context = zmq_init (1);

    //  Socket to talk to clients
    void *responder = zmq_socket (context, ZMQ_REP);
    zmq_bind (responder, "tcp://*:5555");

    while (1) {
        //  Wait for next request from client
        zmq_msg_t request;
        zmq_msg_init (&request);
        zmq_recv (responder, &request, 0);
        printf ("Received Hello\n");
        zmq_msg_close (&request);

        //  Do some 'work'
        sleep (1);

        //  Send reply back to client
        zmq_msg_t reply;
        zmq_msg_init_size (&reply, 5);
        memcpy (zmq_msg_data (&reply), "World", 5);
        zmq_send (responder, &reply, 0);
        zmq_msg_close (&reply);
    }
    //  We never get here but if we did, this would be how we end
    zmq_close (responder);
    zmq_term (context);
    return 0;
}
```

From ØMQ on-line manual

ZeroMQ programming

- Minor pitfalls you may come across with ØMQ:
  - It is not possible to provide your pre-allocated buffer on the receiver side
    - The message buffer allocation is in charge of the ZeroMQ runtime
  - You must be carefull to mange multi-part messages
  - Some kind of ØMQ sockets, if not used properly, start dropping messages without any alert.

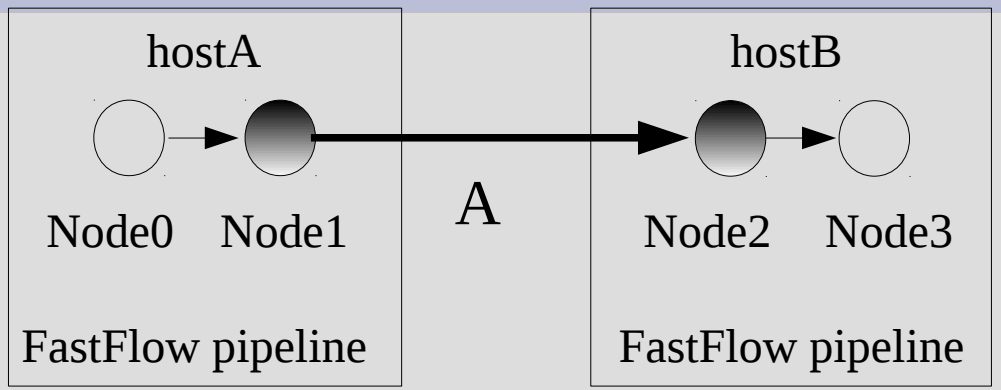
How to define a dnode



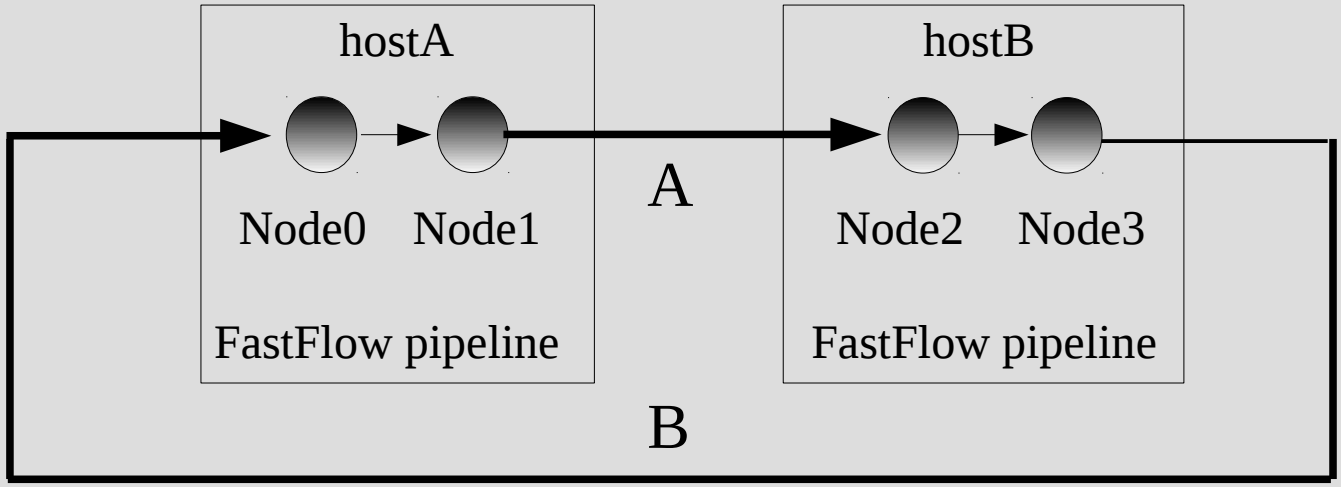
Implementation of the comm. pattern we want to use: broadcast

true identifies a producer, false a consumer node
Newer FF: SENDER/RECEIVER

First distributed example: pipeline

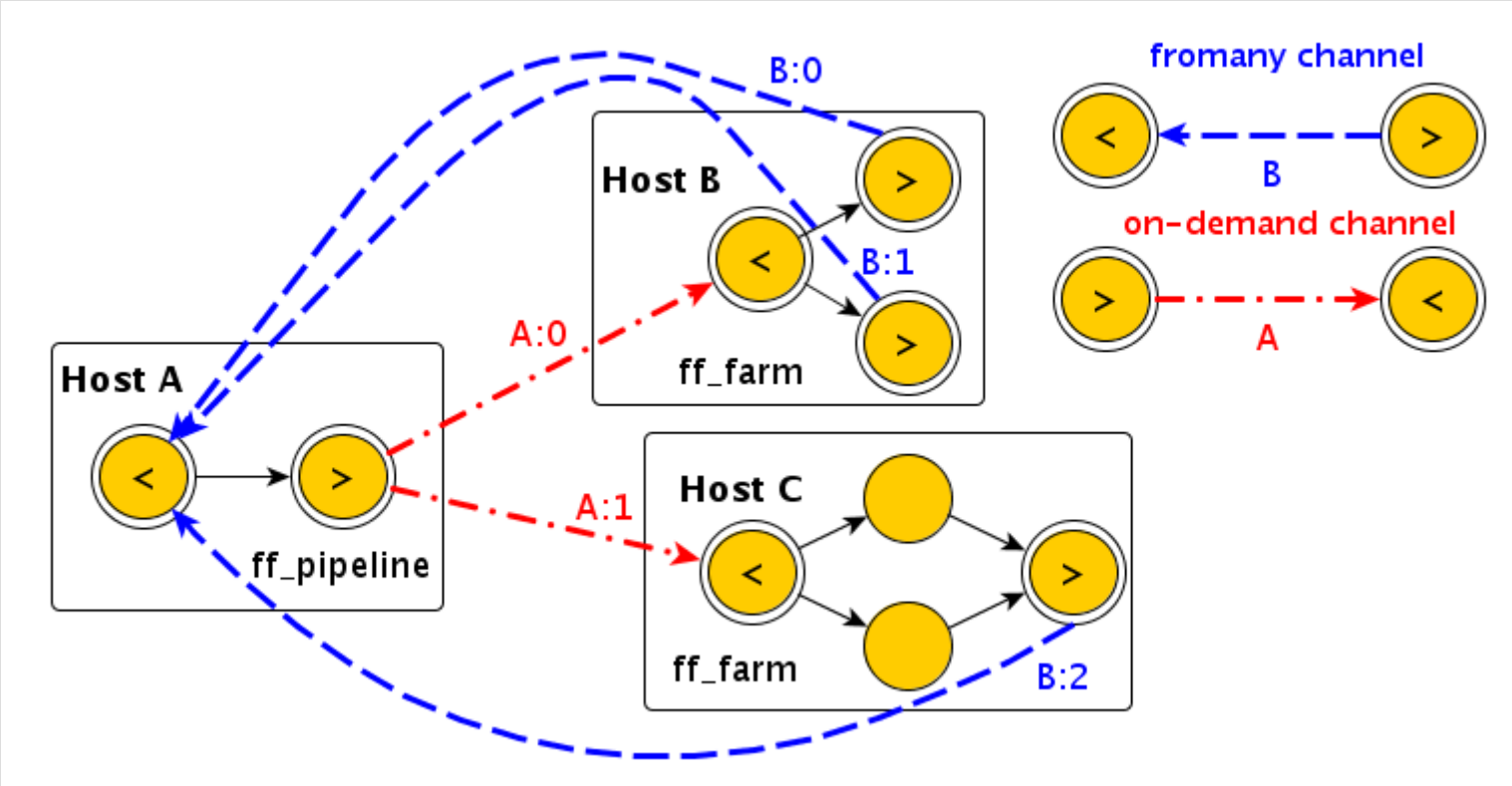


test11_pipe A 1 hostB:port **test11_pipe A 0 hostB:port**



test11_pipe A 1 hostB:port hostA:port **test11_pipe A 0 hostB:port hostA:port**

Distributed farm example



- Take a look at the farm.cpp and farm_farm.cpp examples in the tests/d directory

Marshalling/Unmarshalling

- Consider the case where two or more objects have to be sent in a single message
- If the two objects are non contiguous in memory we have to *memcpy* one of the two
  - but can be quite costly in term of performance
- A classical solution to this problem is to use *readv*/*writev*-like primitives, i.e. multi-part messages (see man writev):

```
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

```
struct iovec {
  void  *iov_base; // starting address
  size_t  iov_len;   // number of bytes to transfer
};
```

Marshalling/Unmarshalling (2)

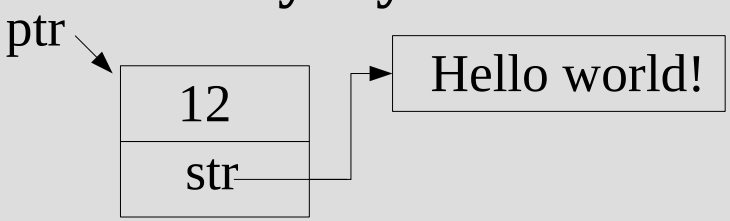
- The ff_dnode class provides 3 methods that can be (have to be) **overloaded**: 2 ***prepare*** methods (1 for the sender and 1 for the receiver), and 1 ***unmarshall*** method only for the receiver
- sender-side: the ***prepare*** method is called by the run-time before sending data into the channel
- receiver-side: the ***unmarshall*** method is called before passing the received data to the svc()

Marshalling/Unmarshalling (3)

Object definition:

```
struct mystring_t {
  int      length;
  char* str;
}; mystring_t* ptr;
```

Memory layout:



- ***prepare*** *(top one) creates 2 iovec for the 2 parts of memory*
  - *those pointed by ptr and str*
- ***unmarshall*** *arranges things to have a single pointer to the object*

emacs@pirotecnia

File Edit Options Buffers Tools C++ Help

```
// Called by the PRODUCER before sending data
// Used to prepare (non contiguous) output messages
virtual void prepare(svector<iovec>& v, void* ptr, const int sender=-1) {
    mystring_t* p = static_cast<mystring_t*>(ptr);
    struct iovec iov={ptr,sizeof(mystring_t)};
    v.push_back(iov);
    iov.iov_base = p->str;
    iov.iov_len  = p->length+1;
    v.push_back(iov);
}
// Called by the CONSUMER before receiving data
// Gives a pool of messages on which input data can be received
virtual void prepare(svector<msg_t*>*& v, size_t len, const int sender=-1) {
    svector<msg_t*> * v2 = new svector<msg_t*>(len);
    assert(v2);
    for(size_t i=0;i<len;++i) {
        msg_t * m = new msg_t;
        assert(m);
        v2->push_back(m);
    }
    v = v2;
}
// Called by the CONSUMER before calling the svc() method
virtual void unmarshalling(svector<msg_t*>* const v[],const int vlen,
                           void *& task) {
    assert(vlen==1 && v[0]->size()==2);
    mystring_t* p =static_cast<mystring_t*>(v[0]->operator[](0)->getData());
    p->str = static_cast<char*>(v[0]->operator[](1)->getData());
    assert(strlen(p->str)== p->length());
    task=p;
}
```

How to play with it

- You have to install ZeroMQ
  - Package distribution (.rpm, .deb, ….)
  - Or download the tarball and compile it
    - for the 2.2.0 versione of ZMQ you must have installed the uuid-dev package
- The distributed version of FastFlow is available since version 2.0.0
  - Better to use the sourceforge svn version of FF because it's updated more frequently