FastFlow: targeting distributed systems

Massimo Torquati

ParaPhrase project meeting, Pisa Italy

11th July, 2012

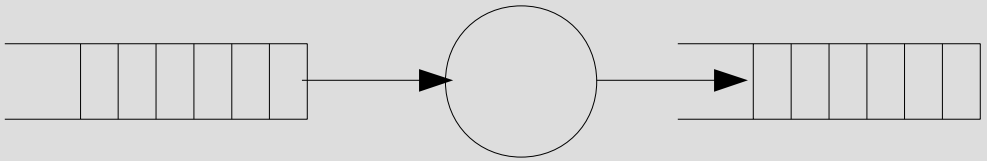
torquati@di.unipi.it

Talk outline

- FastFlow basic concepts
- two-tier parallel model
- From single to many multi-core workstations
  - Definition of the *dnode* in FastFlow
- Brief introduction to ZeroMQ
- *dnode* usage demonstration
- Marshaling/unmarshaling of messages
- Preliminar results

FastFlow node

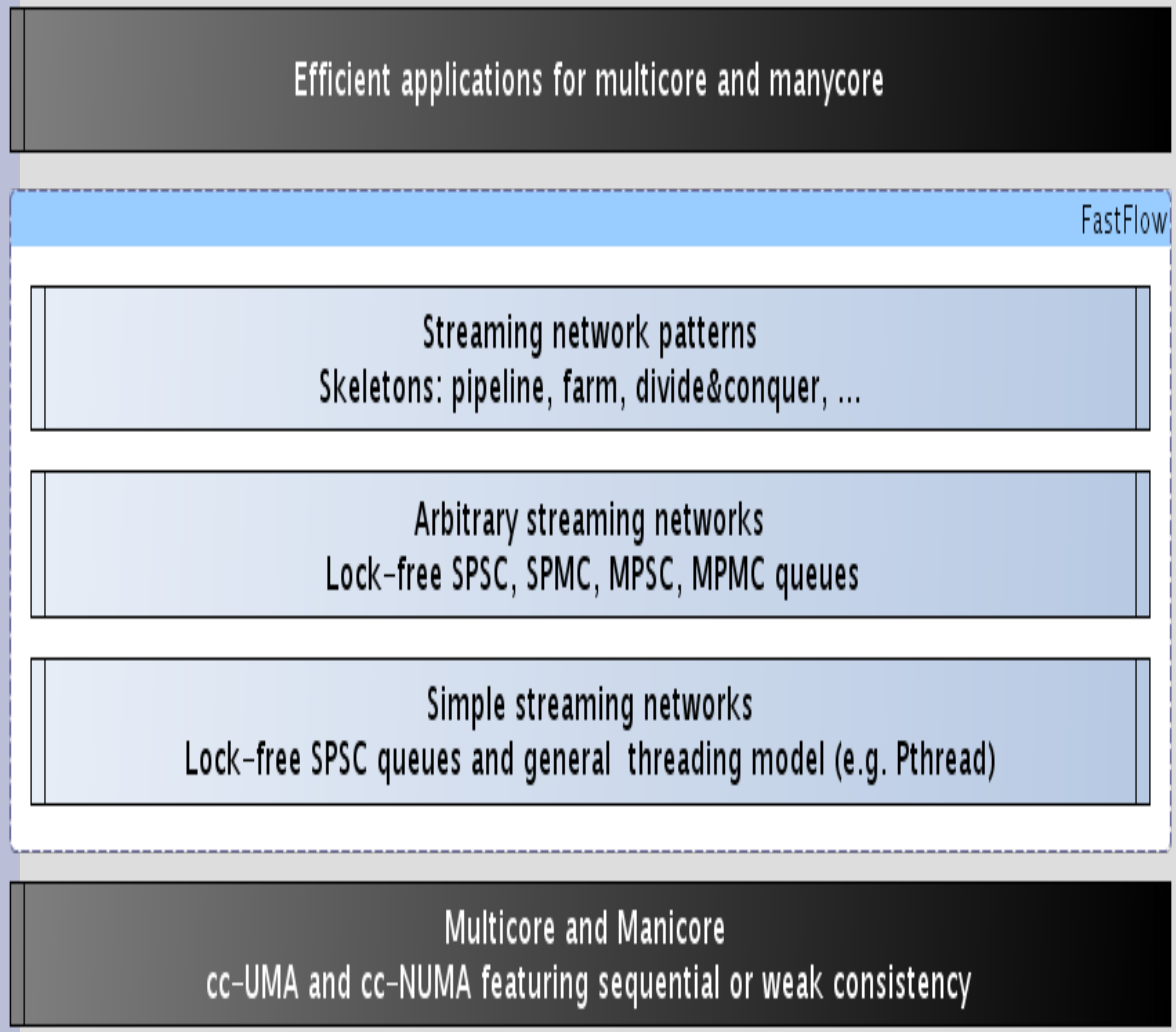
- FastFlow's implementation is based on the concept of *node* (ff_node class)
- A *node* is an abstraction which has an input and an output SPSC queue. The queues can be bounded or unbounded. Nodes are connected one each other by queues.



generic node

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

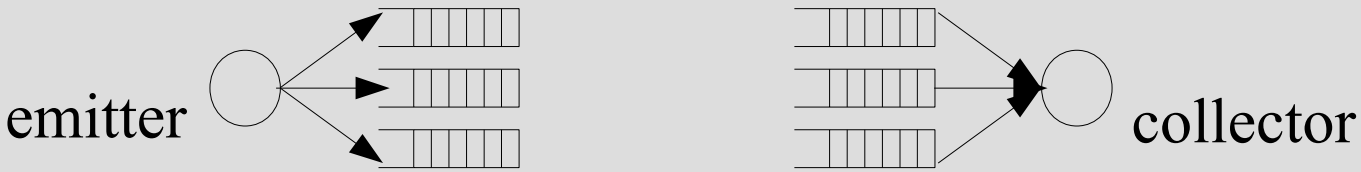
FastFlow node (2)




```
class ff_node {   // class sketch
protected:
   virtuall bool push(void* data) {
         return qout->push(data);
   }
   virtual bool pop(void** data) {
         return qin->pop(data);
   }
public:
   virtual void* svc(void* task)=0;
   virual int svc_init() { return 0;}
   virtual void svc_end() {}
private:
   SPSC* qin;
   SPSC* qout;
} ;
```

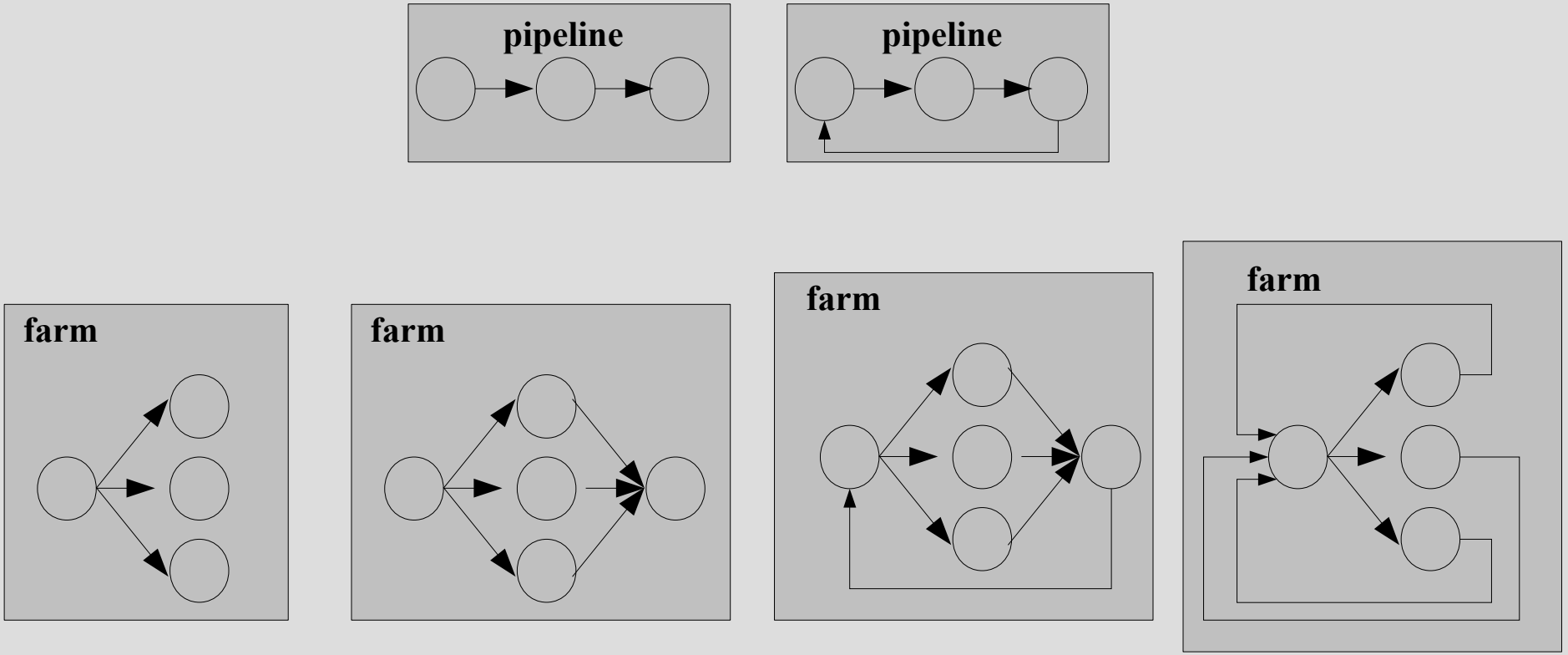
FastFlow node (3)

- 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



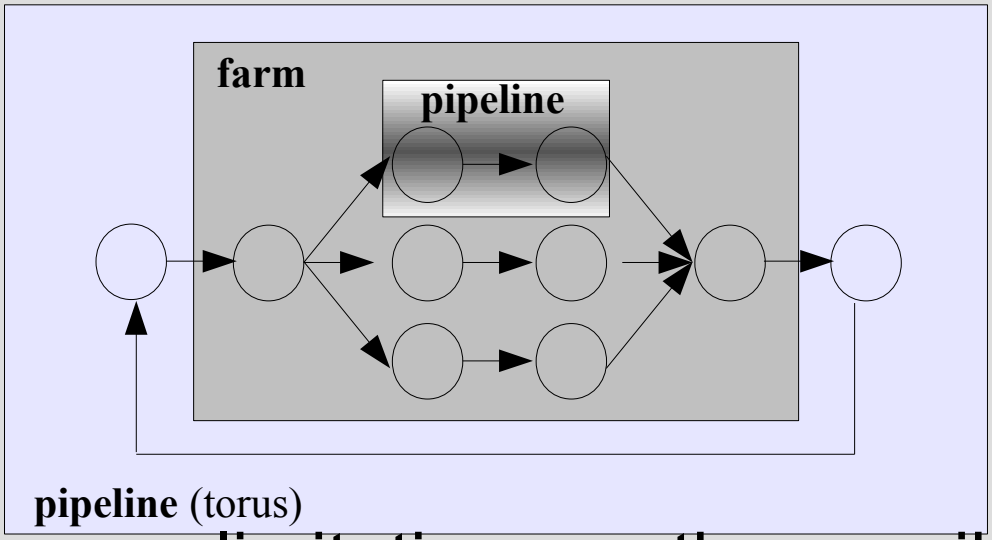
- An ongoing activity is trying to implement the SPMC and MCSP queues as a lock-free CDS in order to remove (in some particular cases) the *emitter* and the *collector* threads

Basic schemas



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 heterogeneous SMP workstations

- We need to scale to hundreds/thousands of cores
  - ⇒ We have to exploit GPU devices and HW accelerators present on the single workstation
  - ⇒ We have to use more than one single multi-core workstation
- The streaming network model provided by FastFlow, can be easily extended to work outside the single workstation

Two-tier parallel model

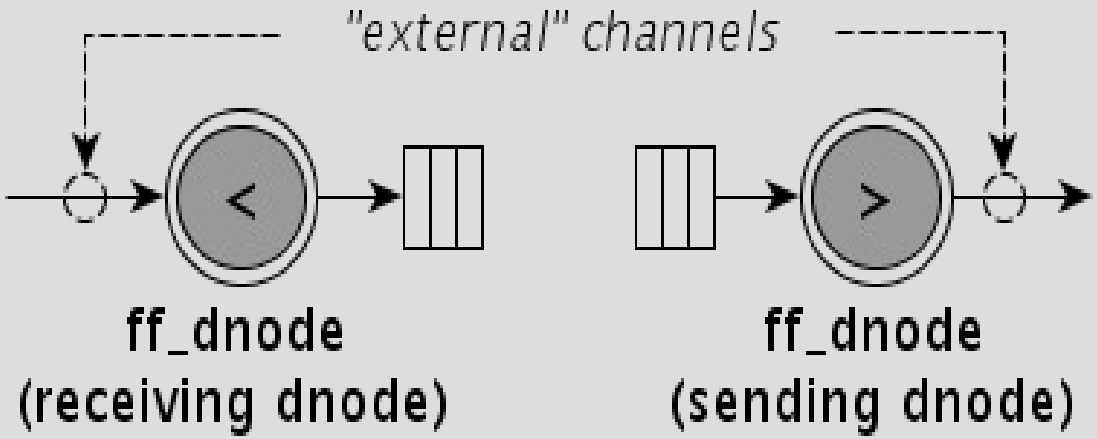
- We propose a *two-tier model*:
  - **Lower layer**: supports fine grain parallelism on single multi/many core workstation
  - **Upper layer**: supports structured coordination, across a number of internetworked workstations, of medium/coarse parallel activities.

The lower layer

- The Lower layer is basically the FastFlow framework extended with:
  - Mechanisms and interfaces for exploiting GPUs and HW coprocessors present on the single multi-core workstation.
  - Mechanisms which allow to connect togheter multiple multi-core workstations
- In the following we describe some of the mechanisms which will allow us to build the upper layer.

From *node* to *dnode*

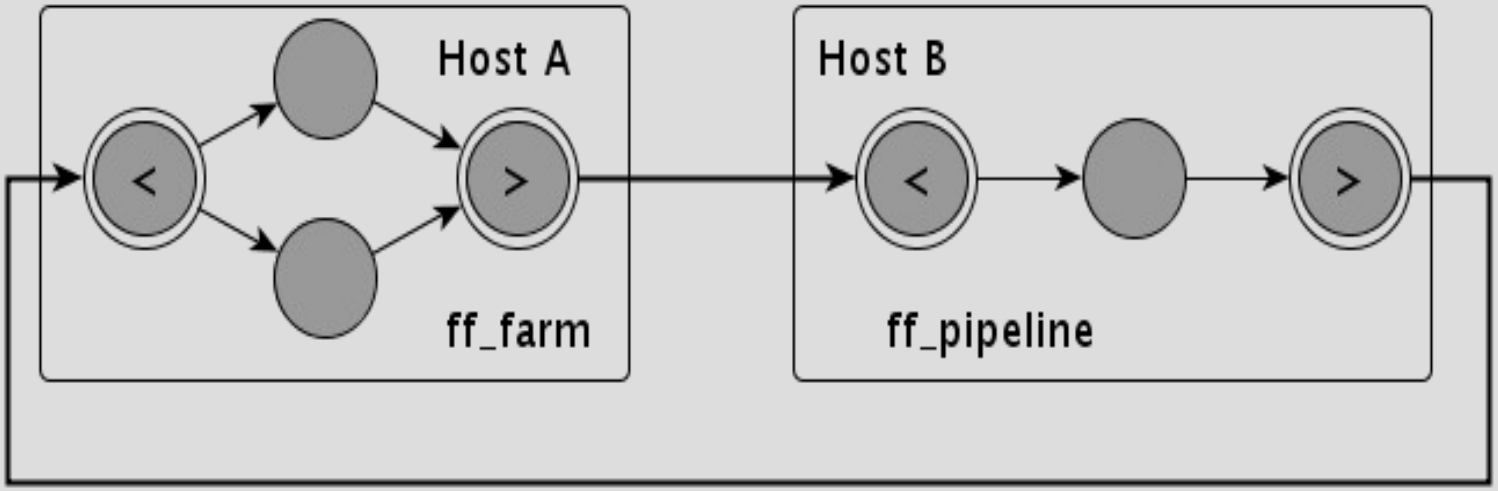
- A *dnode* (class ff_dnode) is a *node* (i.e. extends the ff_node class) with an external communication channel



- The **external channels** are specialized to be an input or an output channel (not both)

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



- In the above scenario we have 2 FastFlow applications whose edge-nodes are 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

FastFlow's dnode

File Edit Options Buffers Tools C++ Help

```
template <typename CommImpl>
class ff_dnode: public ff_node {
protected:
    bool push(void* data) {... return com.push(data);}
    bool pop(void** data) {... return com.get(data); }
....

public:
    int init(const std::string& name, const std::string& address,
             const int peers, CommImpl::TransportImpl* const transp,
             const bool p, const int nodeId=-1, dnode_cbk_t cbk=0) {
        ....
        return com.init(address,nodeId);
    }

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

protected:
    CommImpl com;
};
```

Available communication patterns

- Unicast
- Broadcast
- Scatter
- One-To-Many
- On-demand
- All Gather
- Collect from Any

TODO:

- Many-To-One

Communication Pattenrs

File Edit Options Buffers Tools C++ Help

```
/* ---------------- Pattern macro definitions -------------------- */

#define UNICAST                         commPattern<zmq1_1>
#define UNICAST_DESC(name,trasp,P)      UNICAST::descriptor(name,1,trasp,P)
#define BROADCAST                       commPattern<zmqBcast>
#define BROADCAST_DESC(name,n,trasp,P)  BROADCAST::descriptor(name,n,trasp,P)
#define ALLGATHER                       commPattern<zmqAllGather>
#define ALLGATHER_DESC(name,n,trasp,P)  ALLGATHER::descriptor(name,n,trasp,P)
#define FROMANY                         commPattern<zmqFromAny>
#define FROMANY_DESC(name,n,trasp,P)    FROMANY::descriptor(name,n,trasp,P)
#define SCATTER                         commPattern<zmqScatter>
#define SCATTER_DESC(name,n,trasp,P)    SCATTER::descriptor(name,n,trasp,P)
#define ONETOMANY                       commPattern<zmq1_N>
#define ONETOMANY_DESC(name,n,trasp,P)  ONETOMANY::descriptor(name,n,trasp,P)
#define ONDEMAND                        commPattern<zmqOnDemand>
#define ONDEMAND_DESC(name,n,trasp,P)   ONDEMAND::descriptor(name,n,trasp,P)

/* TODO */
#define MANYTOONE                       commPattern<zmqN_1>
#define MANYTOONE_DESC(name,n,trasp,P)  MANYTOONE::descriptor(name,n,trasp,P)
```

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)

Comm. Pattern Interface

File Edit Options Buffers Tools C++ Help

```
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;
    typedef typename Impl::TransportImpl   TransportImpl;
    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) {}
    // initializes communication pattern
    // nodeId is the unique identifier of the calling in the range [0..inf[
    inline bool init(const std::string& address,const int nodeId=-1);
    // Specifies that the message being sent is just a part of the entire
    // message. Further message parts are to follow.
    inline bool putmore(const tosend_t& msg);
    // sends one message
    inline bool put(const tosend_t& msg);
    // receives the message header (should be called before get)
    inline bool gethdr(torecv_t& msg, int& peer);
    // receives one message part
    inline bool get(torecv_t& msg) { return impl.get(msg); }
    // received all messages
    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

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

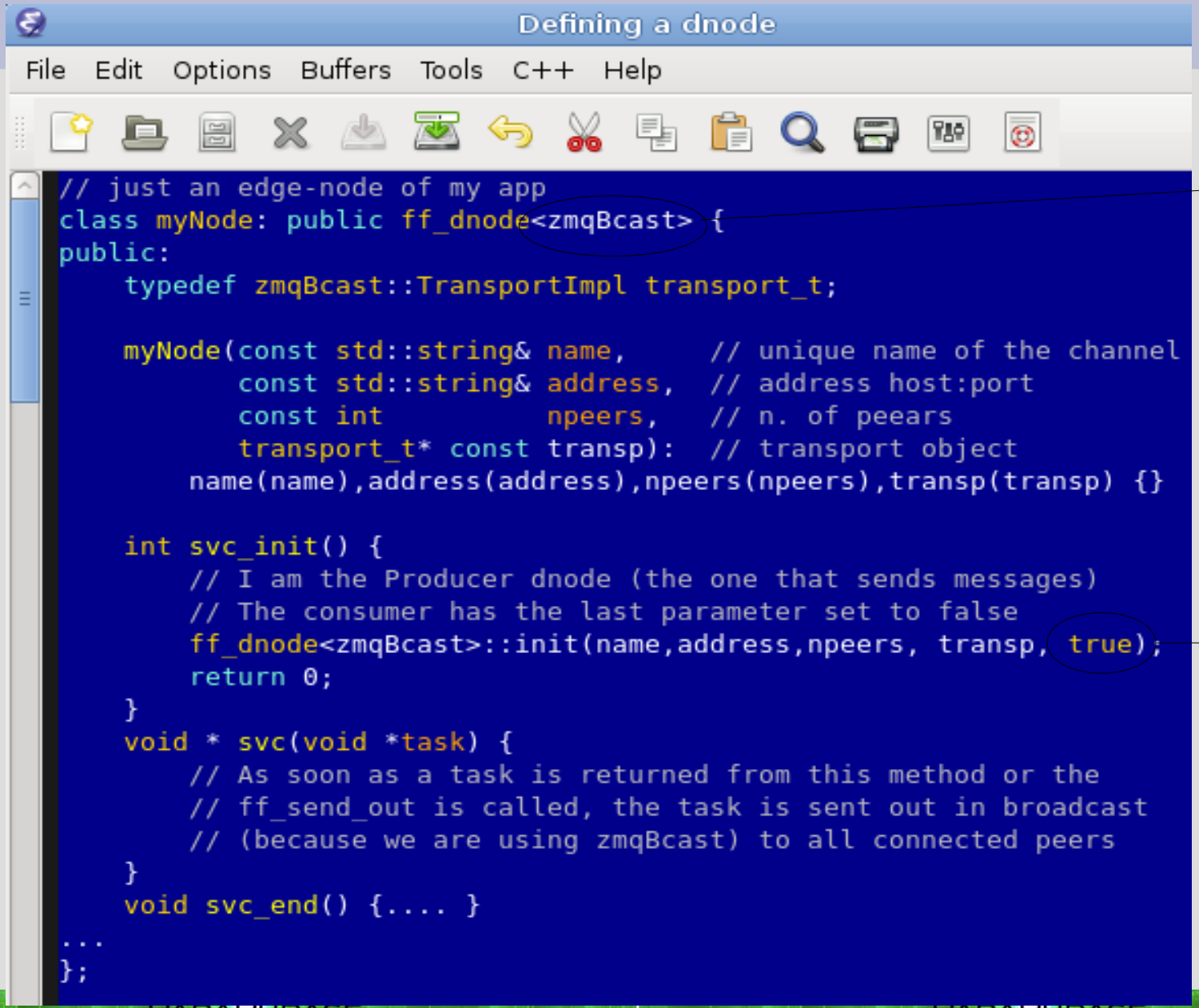
```c
int main (void)
{
    void *context = zmq_init (1);

    //  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 message 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.

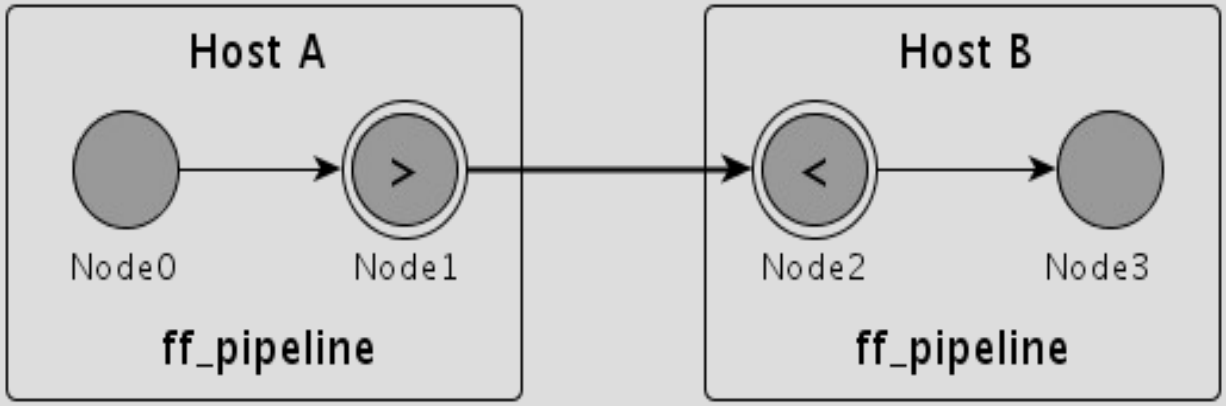
How to define a dnode



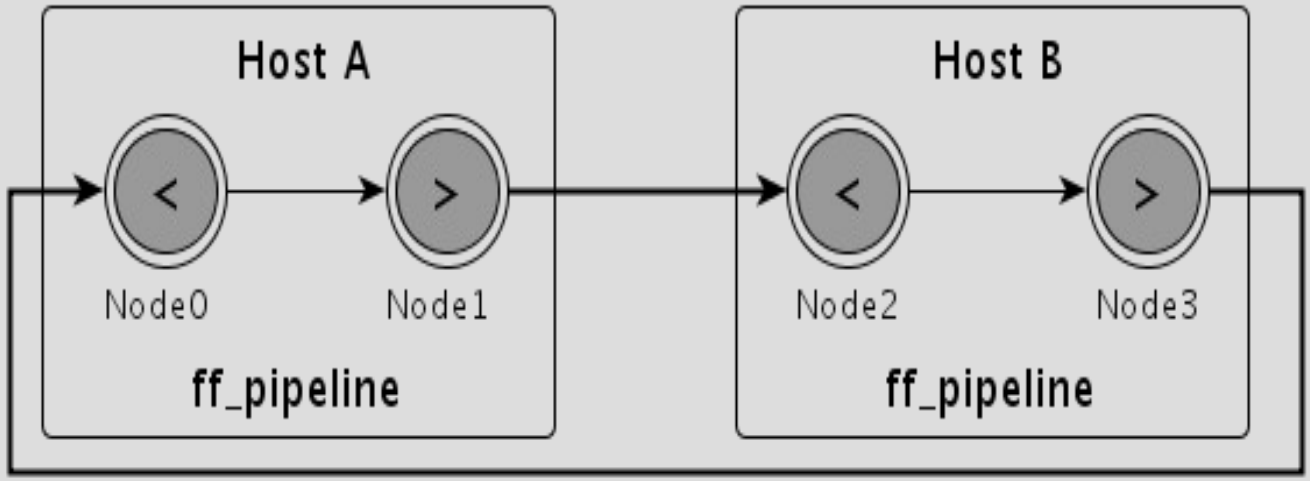
Implementation of the comm. pattern we want to use: broadcast inplemented on top of ZeroMQ

true identifies a producer, false a consumer node

Simple distributed example: pipeline

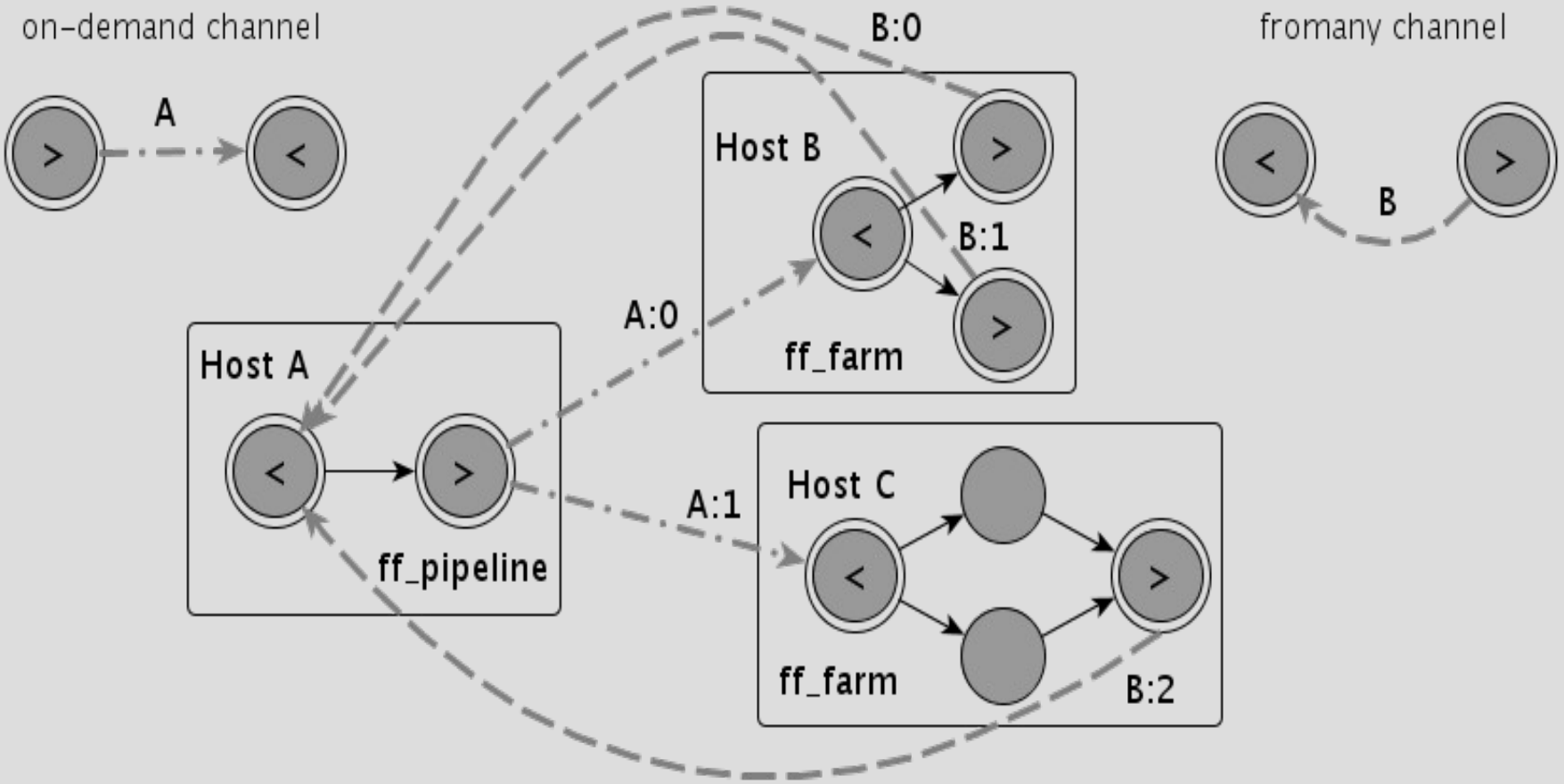


test11_pipe A 1 hostA:port **test11_pipe A 0 hostA:port**



test11_torus A B 1 hostA:port hostB:port **test11_torus A B 0 hostA:port hostB:port**

A more complex scenario



Usage demonstration

- test11_pipe
- test11_torus

Marshalling/Unmarshalling

- Consider the case where two or more objects have to be sent as 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 POSIX *readv*/*writev*-like primitives, i.e. multi-part messages.

Marshalling/Unmarshalling (2)

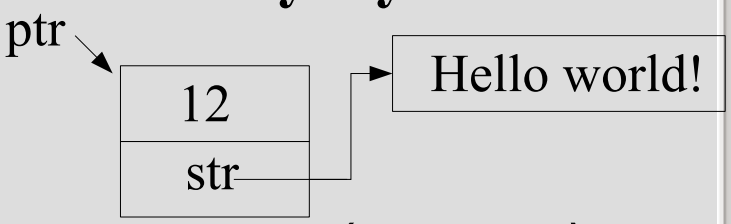
- The ff_dnode class provides 3 methods that can be (have to be) **overloaded**: 2 ***prepare*** methods (1 for the sender and 1 for the receiver), and 1 ***unmarshall*** method only for the receiver
- sender-side: the ***prepare*** method is called by the run-time before sending data into the channel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

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*

Defining a dnode

File Edit Options Buffers Tools C++ Help

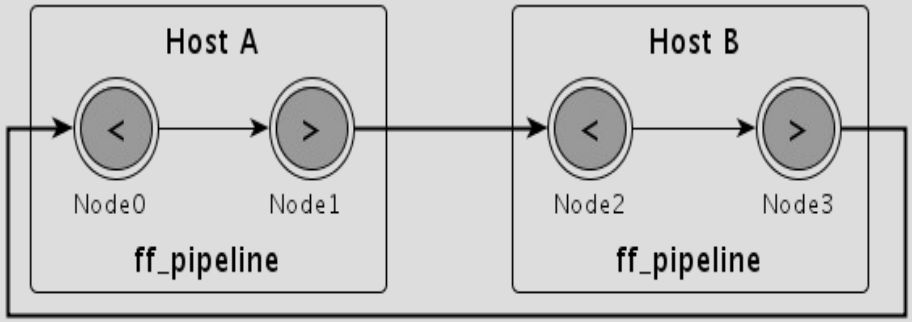
```
// Called at the PRODUCER side before sending data
void prepare(svector<iovec>& v, void* ptr, const int sender=-1) {
    mystring_t* p = static_cast<mystring_t*>(ptr);
    struct iovec iov={ptr,sizeof(mystring_t)};
    v.push_back(iov);
    iov.iov_base = p->str;
    iov.iov_len  = p->length+1;
    v.push_back(iov);
}
// The following 2 methods are called at the CONSUMER side.
// The prepare is called before receiving all partial messages
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;
}
// The unmarshalling method is called before calling the svc() method
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;
}
```

Preliminar results

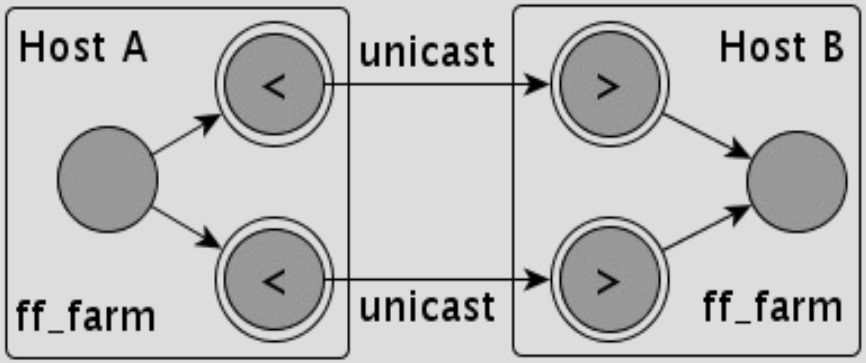
- Raw performance (Unicast)

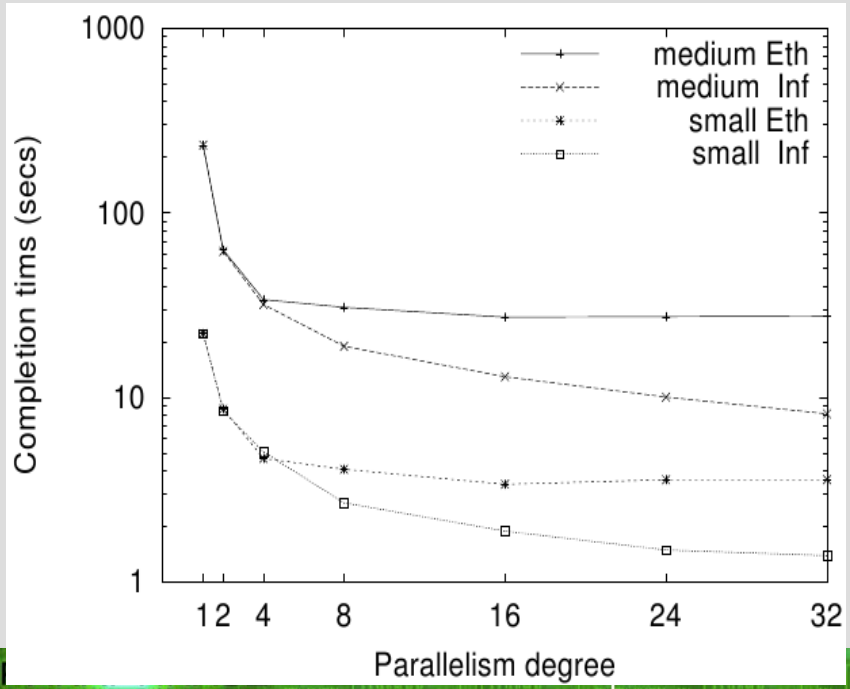
	Latency (us)	Bandwidth Gb/s	
size	8B	8KB	1MB
Eth	69	0.93	0.95
Inf	27	5.1	14.7

The latency has been measured when considering a torus of 2 FastFlow pipeline (2 stages each) whose edge-nodes are connected with a unicast channel.



- Simple image filtering app.





How to use it

- You have to install ZeroMQ
  - Package distribution (.rpm, .deb, ….)
  - Or download the tarball and compile it
    - You have to have installed the uuid-dev package
- The distributed version of FastFlow is now available on sourceforge SVN

  svn co https://mc-fastflow.svn.sourceforge.net/svnroot/mc-fastflow

  (current version 2.0.0 has yet to be fully tested on OSX and Window OSs)
- Drop us an e-mail if you find bugs or problems.