THE BIOROBOTICS INSTITUTE

Sant'Anna
School of Advanced Studies – Pisa

Introductio to YARP

Egidio Falotico

Outline

- Introduction to YARP
- Introduction to Communication mechanisms in YARP
- Programming with YARP

What is YARP

YARP is a set of libraries, protocols, and tools to keep modules and devices cleanly decoupled. It is a middleware, YARP as ROS is a meta operating system.

YARP is free and open software that, as ROS, is developed to speed software development

YARP Components

The components of YARP can be broken down into:

- **YARP_os** - interfacing with the operating system(s) to support easy streaming of data across many threads across many machines. YARP is written to be **OS neutral**, and has been used on Linux, Microsoft Windows, Apple macOS and iOS, Solaris, and Android. YARP uses the open-source ACE (ADAPTIVE Communication Environment) library, which is portable across a very broad range of environments, and YARP inherits that portability. YARP is written almost entirely in C++.
- **YARP_sig** - performing common signal processing tasks (visual, auditory) in an open manner easily interfaced with other commonly used libraries, for example OpenCV.
- **YARP_dev** - interfacing with common devices used in robotics: framegrabbers, digital cameras, motor control boards, etc.

YARP Communication system: nameserver

The name server is a YARP program that maintains a list of all YARP Ports and how to connect to them (as ROS master).

The name server itself has a YARP Port, usually named **"/root".** All other YARP programs communicate with the name server through this port. This communication is usually hidden within YARP library calls, but we document it here in order to allow communication with the name server by clients not using the YARP libraries.

YARP name server

- Connecting to the name server is just like connecting to any other YARP Port (see **Port Protocol**). The one problem is that you have to find out where the name server is (what machine, what socket port number) somehow. For other YARP Ports, you can solve that problem by asking the name server, but that option isn't available for the name server itself.
- One option is simply to make sure the name server is started on a particular known machine (e.g. **192.168.0.1**) on a known socket port number (say **10000**, the default).
- The name server itself, once started, records its contact information in a configuration file (you can type "yarp conf" to find out where that file is). Other YARP programs will check this file to see how to reach the name server. If that doesn't work, there is a multicast protocol for discovering the server.

YARP Port

- A **Port** is an object that can read and write values to peer objects spread throughout a network of computers. It is possible tocreate, add and remove connections either from that program, from the command line, or from another program.
- **Ports are specialized for streaming communication**, such as camera images or motor commands. You can switch network protocols for any or all your connections without changing a line of code.
- The YARP library supports transmission of a stream of user data across various protocols – **TCP, UDP, MCAST** (multi-cast), shared memory – insulating a user of the library from the idiosyncratic details of the network technology used.

YARP PORT PROPERTIES

- For the purposes of YARP, communication takes place **through Connections**'' between named entities called Ports''. These form a directed graph, the ``YARP Network'', where Ports are the nodes, and Connections are the edges.

- The purpose of Ports is to **move ``Content**'' (sequences of bytes representing user data) from one thread to another (or several others) across process and machine boundaries. The flow of data can be manipulated and monitored externally (e.g. from the command-line) at run-time.

- **A Port can send Content to any number of other Ports**. A Port can receive Content from any number of other Ports. If one Port is configured to send Content to another Port, they are said to have a **Connection**. Connections can be freely added or removed.

- The **YARP name server tracks information about ports**. It indexes this information by name, playing a role analogous to DNS on the internet. To communicate with a port, the properties of that port need to be known (the machine it is running on, the socket it is listening on, the carriers it supports). The YARP name server offers a convenient place to store these properties, so that only the name of the port is needed to recover them.

Properties of a YARP network

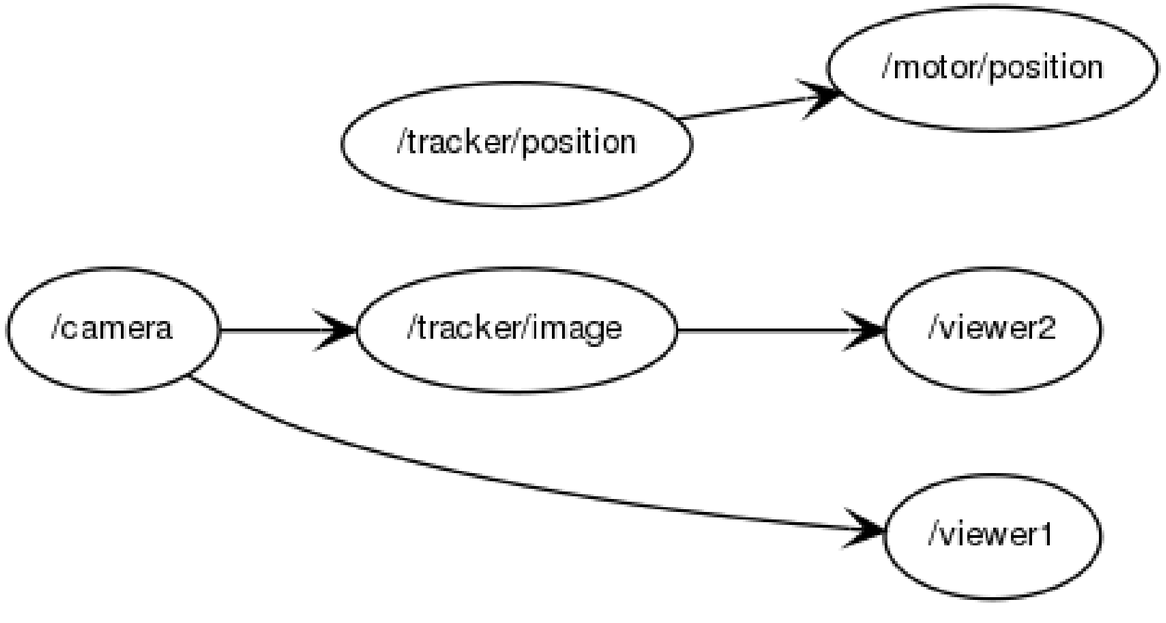
A YARP network consists of the following entities:
a set of ports, a set of connections, a set of names, a name server, and a set of registrations.

- Every port has a unique name.
- Every connection has a source port and a target port.
- Each port maintains a list of all connections for which it is the target port.
- Each port maintains a list of all connections for which it is the source port.
- There is a single name server in a YARP network.
- The name server maintains a list of registrations. Each registration contains information about a single port, identified by name.

Communication in a YARP Network

- Communication within a YARP network can occur between two ports, between a port and the name server, between a port and an external entity, and between the name server and an external entity.
  - Communication between two ports occurs if and only if there is a connection between them.
  - Connections involving a port can be created, destroyed, or queried by communication between an external entity and that port. This is done by sending ``port commands'' using the YARP connection protocol.
  - Ports communicate with the name server using the ``YARP name server protocol''. Such communication is needed to create, remove, and query registrations.

An example of a YARP network

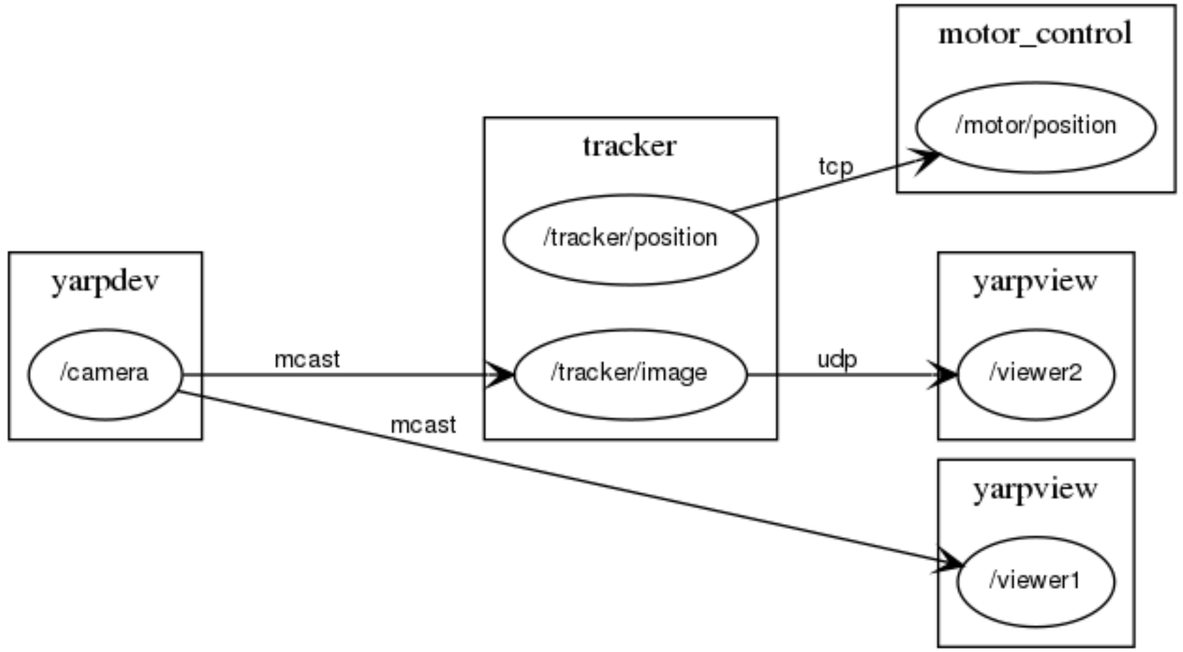


An example of a YARP network (more detailed)

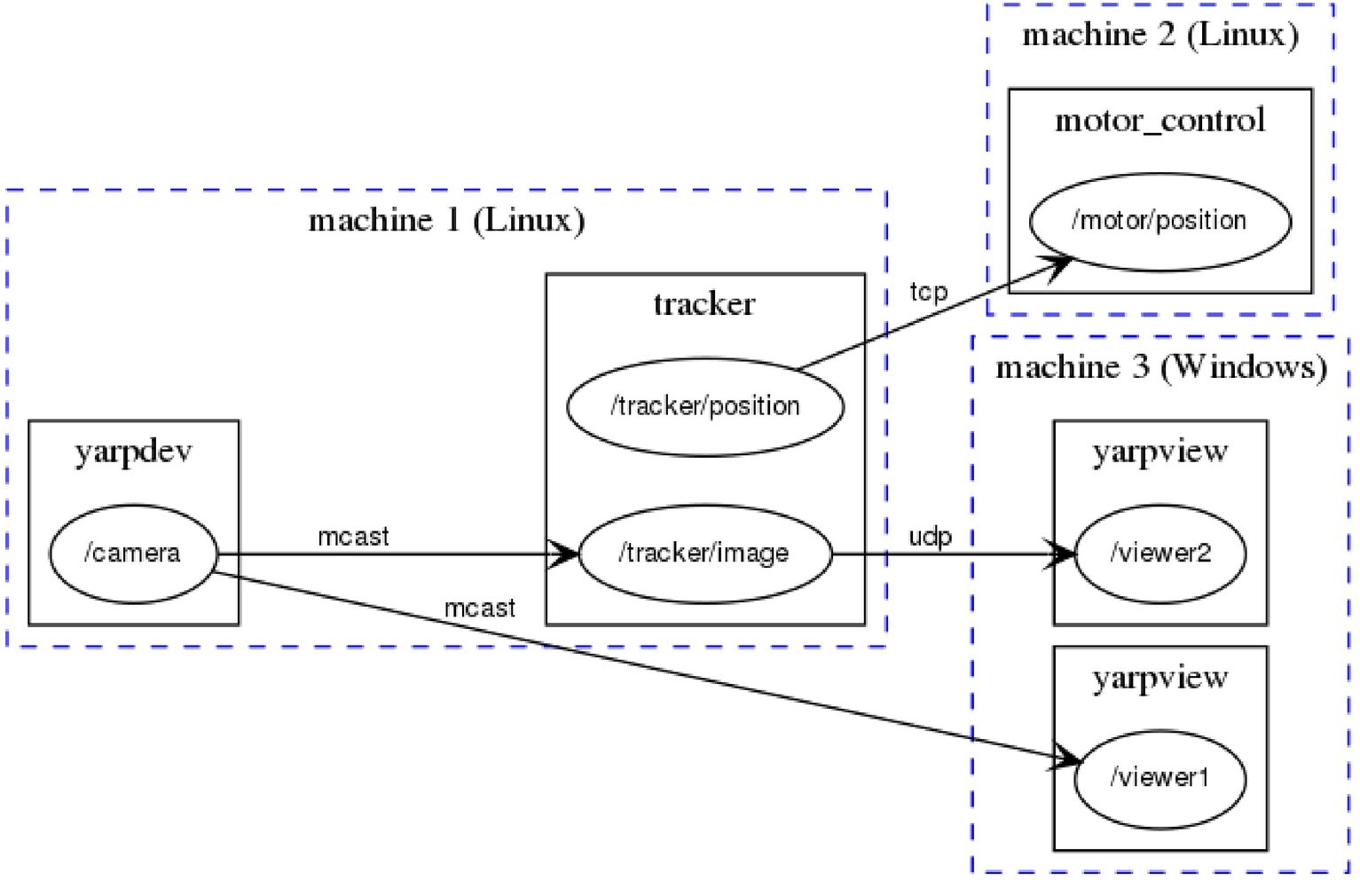
TCP: reliable, it can be used to guarantee the reception of a message;

UDP: faster than TCP, but without guarantees;

multicast: efficient for distributing the same information to large numbers of targets;



An example of a YARP network (more detailed)



Crating ports with the console

Run yarpserver

```
roscourse@cloud-pc:~$ yarpserver

     ==========================
    |  __ __ ___  ____   ____  |
    |  \ \/ //   ||  _ \ |  _ \ |
    |   \  // /| || |/ / | |/ / |
    |   / // ___ ||  _ \ |  _/  |
    |  /_//_/  |_||_| \_\|_|    |
    |                           |
     ==========================

Call with --help for information on available options
Using port database: :memory:
Using subscription database: :memory:
IP address: default
Port number: 10000
[INFO] |yarp.os.Port|/root| Port /root active at tcp://10.0.2.15:10000/
Registering name server with itself
 * register "/root" tcp "10.0.2.15" 10000
```

Make a reading port */read*

```
roscourse@cloud-pc:~$ yarp read /read
[INFO] |yarp.os.Port|/read| Port /read active at tcp://10.0.2.15:10002/
```

Make a writing port sending messages to */read*

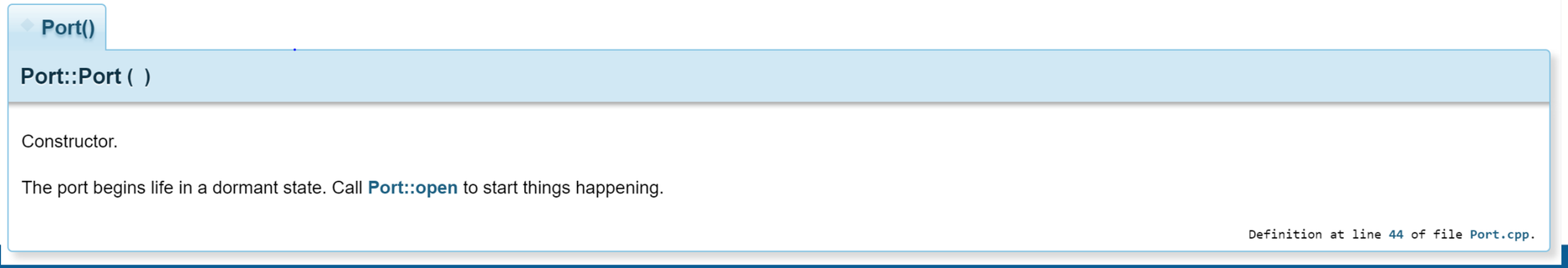
```
roscourse@cloud-pc:~$ yarp write /write /read
[INFO] |yarp.os.Port|/write| Port /write active at tcp://10.0.2.15:10003/
[INFO] |yarp.os.impl.PortCoreOutputUnit|/write| Sending output from /write to /r
ead using tcp
```

Yarp Port

- Ports is a communication channel in the YARP network.

- Data coming from any incoming connection can be received by calling **Port::read**. Calls to **Port::write** result in data being sent to all the outgoing connections.

- Communication with **Port** objects is **blocking by default**, this means that YARP will not drop messages and timing between readers and senders will be coupled. It implements a **synchronous** communication mechanism.



Yarp Port

open() [2/2]

bool Port::open (const std::string & name)

Start port operation, with a specific name, with automatically-chosen network parameters.

The port is registered with the given name, and allocated network resources, by communicating with the YARP name server.

Returns

true iff the port started operation successfully and is now visible on the YARP network

read()

bool Port::read (PortReader & reader,
bool willReply = `false`
)

override virtual

Read an object from the port.

Parameters

reader any object that knows how to read itself from a network connection - see for example **Bottle**

willReply you must set this to true if you intend to call **reply()**

Returns

Argument is *PortReader &*

Yarp Port

write() [1/2]

```
bool Port::write ( const PortWriter &  writer,
                   const PortWriter *  callback = nullptr
                 )                     const
```

Write an object to the port.

Parameters

writer any object that knows how to write itself to a network connection - see for example **Bottle**

callback object on which to call onCompletion() after write is done (otherwise writer.onCompletion() is called)

Returns

true iff the object is successfully written

Argument is *PortWriter &*

Class Network

Public Member Functions

	Network () Constructor.
	Network (**yarp::os::yarpClockType clockType**, **yarp::os::Clock** *custom=nullptr) Initialize the YARP network using the specified clock.
virtual	**~Network** () Destructor.

Static Public Member Functions

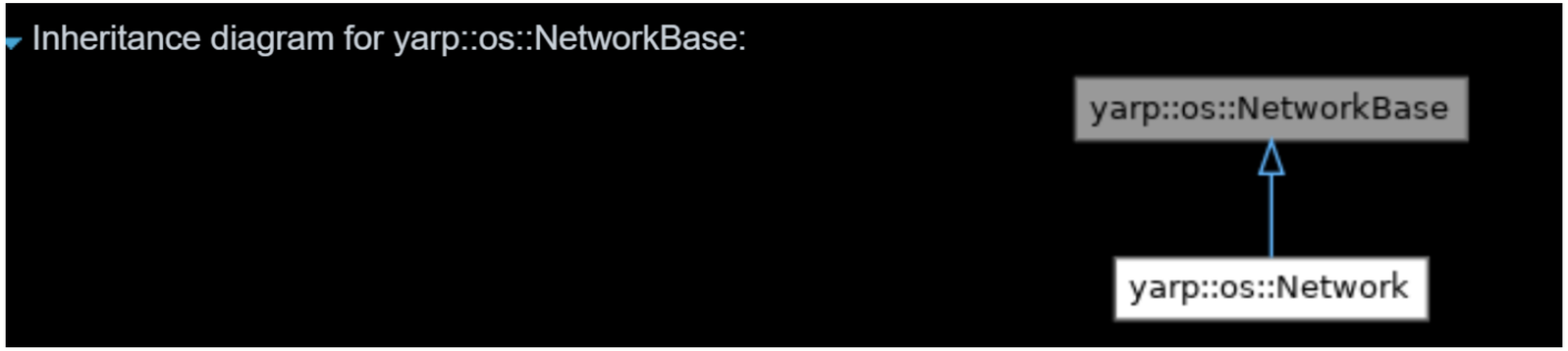
static void	**init** () Initialization.
static void	**init** (**yarp::os::yarpClockType clockType**, **Clock** *custom=nullptr) Initialization.
static void	**fini** () Deinitialization.

▸ **Static Public Member Functions inherited from yarp::os::NetworkBase**

Class used to initialize and shutdown of the network

https://www.yarp.it/latest/classyarp_1_1os_1_1Network.html

Class NetworkBase



Network is a derived class

connect() [3/3]

```
bool NetworkBase::connect ( const std::string &  src,
                            const std::string &  dest,
                            const std::string &  carrier = "",
                            bool                 quiet = true
                          )
```

static

Request that an output port connect to an input port.

Parameters

src	the name of an output port
dest	the name of an input port
carrier	the name of the protocol to use (tcp/udp/mcast)
quiet	suppress messages displayed upon success/failure

Returns

true on success, false on failure

Examples

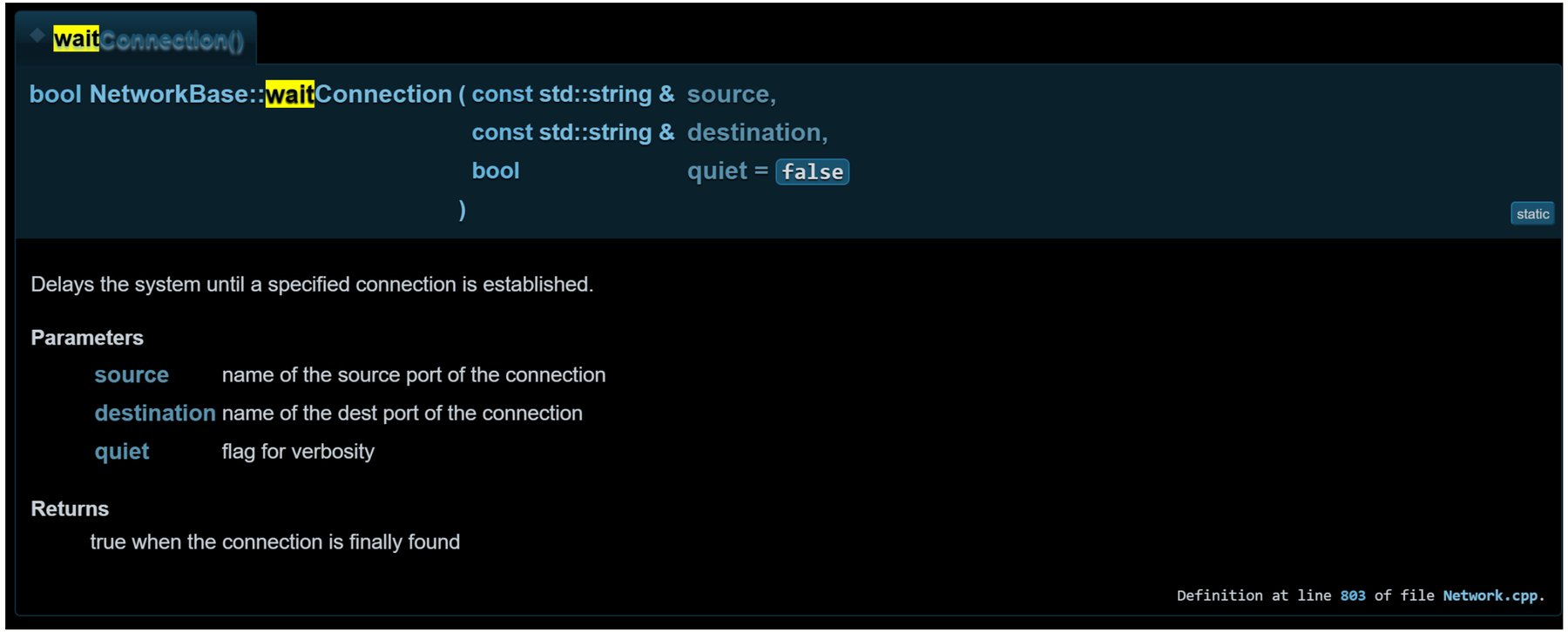
carrier/carrier_stub.cpp.

Definition at line 682 of file Network.cpp.

connect() is a member function of *NetworkBase*, that we can use with *Network* objects

https://www.yarp.it/latest/classyarp_1_1os_1_1NetworkBase.html

Class NetworkBase



waitConnection()

```
bool NetworkBase::waitConnection ( const std::string &  source,
                                   const std::string &  destination,
                                   bool                 quiet = false
                                 )
```

static

Delays the system until a specified connection is established.

Parameters

source	name of the source port of the connection
destination	name of the dest port of the connection
quiet	flag for verbosity

Returns

true when the connection is finally found

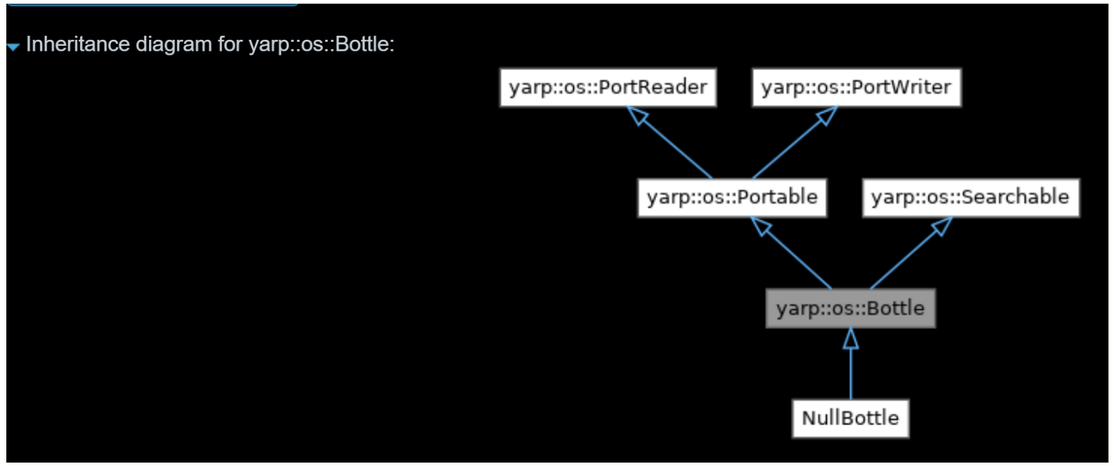
Definition at line 803 of file Network.cpp.

We use it to be sure that the connection is in place before sending data

https://www.yarp.it/latest/classyarp_1_1os_1_1NetworkBase.html

Yarp Bottle

- A Bottle is a collection of objects that can be transmitted
- Objects are stored in a list. It is possible to access this list and add objects.



Bottle is a *PortReader*

https://www.yarp.it/latest/classyarp_1_1os_1_1Bottle.html

Bottle: Member function get

get()

Value & Bottle::get (size_type index) const

Reads a **Value** v from a certain part of the list.

Methods like v.isInt32() or v.isString() can be used to check the type of the result. Methods like v.asInt32() or v.asString() can be used to access the result as a particular type.

Parameters

index the part of the list to read from.

Returns

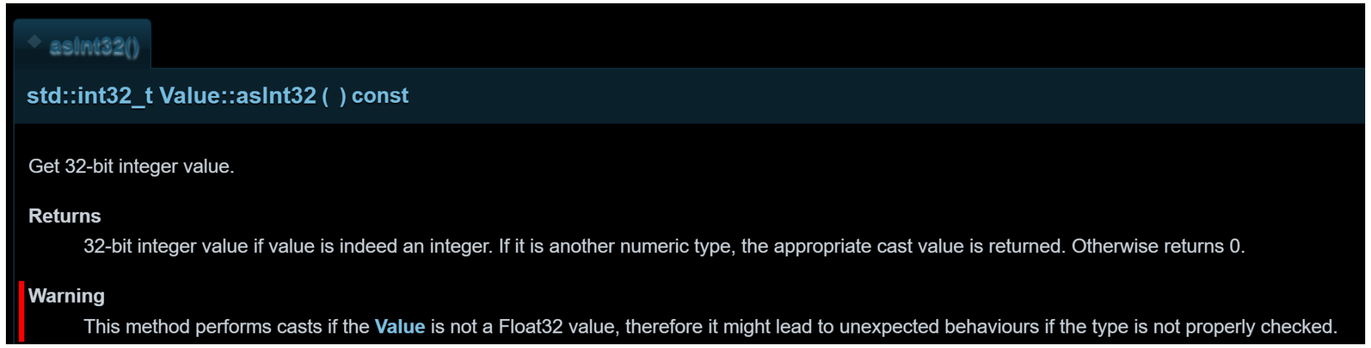
the **Value** v; if the index lies outside the range of elements present, then v.isNull() will be true.

Definition at line 246 of file Bottle.cpp.

It returns a *Value*

Value

Values can be integers, strings, doubles (floating-point numbers), lists, vocabulary… This set is carefully chosen to have good text and binary representations for network transmission.



asInt32()

std::int32_t Value::asInt32 ( ) const

Get 32-bit integer value.

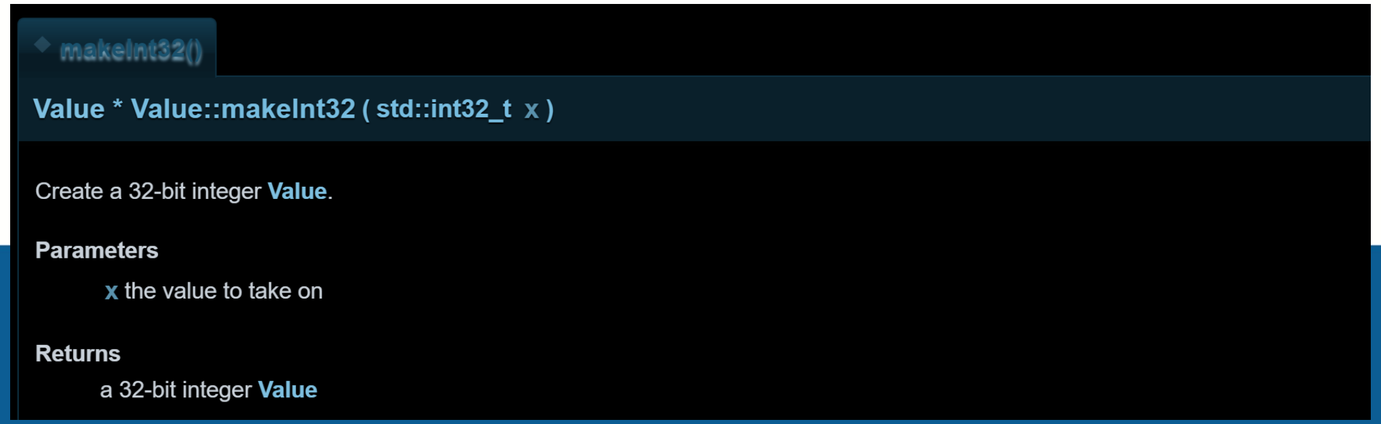
Returns

32-bit integer value if value is indeed an integer. If it is another numeric type, the appropriate cast value is returned. Otherwise returns 0.

Warning

This method performs casts if the **Value** is not a Float32 value, therefore it might lead to unexpected behaviours if the type is not properly checked.

Example to read as a *Value* as an *integer32*



makeInt32()

Value * Value::makeInt32 (std::int32_t x)

Create a 32-bit integer **Value**.

Parameters

x the value to take on

Returns

a 32-bit integer **Value**

Example to make a *Value* object with an *integer32*

Bottle: Member function addString

addString() [1/2]

void Bottle::addString (const char * str)

Places a string in the bottle, at the end of the list.

Parameters

`str` the string to add.

Examples

port_power/ex0001_sender.cpp, port_power/ex0101_sender.cpp, port_power/ex0400_expect_reply.cpp, port_power/ex0401_give_reply.cpp, port_power/ex0402_port_callback_reply.cpp, and port_power/ex0403_bufferedport_callback_reply.cpp.

Definition at line 170 of file Bottle.cpp.

void	**addInt** (int x) Places an integer in the bottle, at the end of the list. More...
void	**addInt8** (std::int8_t x) Places a 8-bit integer in the bottle, at the end of the list. More...
void	**addInt16** (std::int16_t x) Places a 16-bit integer in the bottle, at the end of the list. More...
void	**addInt32** (std::int32_t x) Places a 32-bit integer in the bottle, at the end of the list. More...
void	**addInt64** (std::int64_t x) Places a 64-bit integer in the bottle, at the end of the list. More...
void	**addVocab** (int x) Places a vocabulary item in the bottle, at the end of the list. More...
void	**addDouble** (double x) Places a floating point number in the bottle, at the end of the list. More...
void	**addFloat32** (**yarp::conf::float32_t** x) Places a 32-bit floating point number in the bottle, at the end of the list. M
void	**addFloat64** (**yarp::conf::float64_t** x) Places a 64-bit floating point number in the bottle, at the end of the list. M
void	**addString** (const char *str) Places a string in the bottle, at the end of the list. More...
void	**addString** (const std::string &str) Places a string in the bottle, at the end of the list. More...

Bottle: Member function clear

clear()

void Bottle::clear ( )

Empties the bottle of any objects it contains.

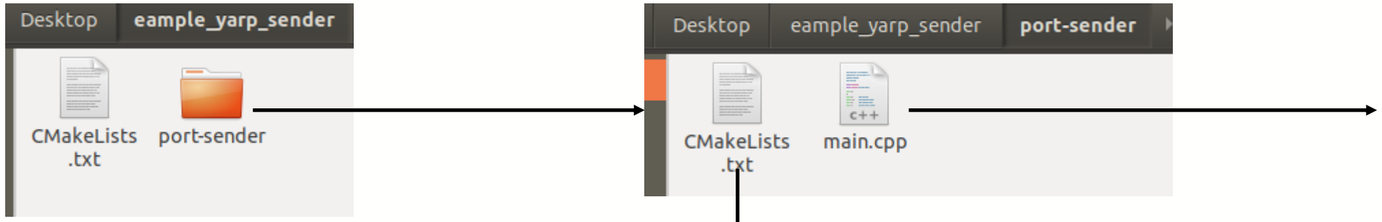
Examples

port_power/ex0001_sender.cpp, port_power/ex0101_sender.cpp, port_power/ex0400_expect_reply.cpp, port_power/ex0401_give_reply.cpp, port_power/ex0402_port_callback_reply.cpp, and port_power/ex0403_bufferedport_callback_reply.cpp.

Definition at line 121 of file Bottle.cpp.

It is convenient to use clear() to avoid a continuos appending of new values inside the Bottle

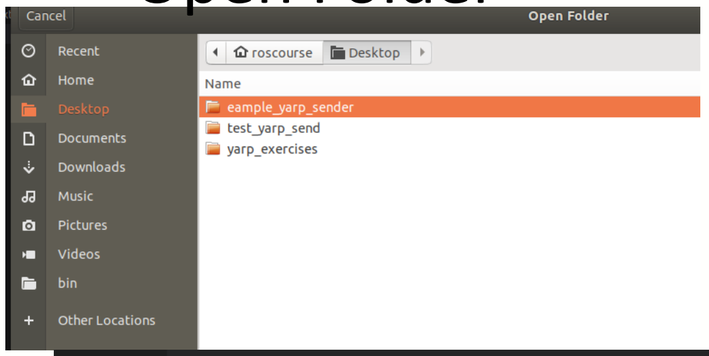
Example: Yarp sender using Ports



Open Visual Studio Code

```
roscourse@roscourse-VirtualBox:~/Desktop/yarp_exercises/build/bin$ code
```

Open Folder



Open ▾ CMakeLists.txt
~/Desktop/eample_yarp_sender/port-sender

```
# set up our program
add_executable(port-sender)

# declare our source files
target_sources(port-sender PRIVATE main.cpp)

# link with YARP libraries
target_link_libraries(port-sender PRIVATE YARP::YARP_os
                                          YARP::YARP_init)
```

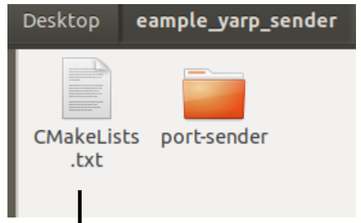
main.cpp ×

port-sender > main.cpp > ...

```
#include <yarp/os/all.h>
#include <iostream>

using namespace yarp::os;


int main(int argc, char *argv[]) {

    // Set up YARP
    Network yarp;

    // Create port
    Port p;
    bool ok  = p.open("/examples/port/sender");
    if (!ok) {
        std::cerr << "Failed to open port" << std::endl;
        return 1;
    }

    // waiting for the receiver before transmitting
    yarp.waitConnection("/examples/port/sender", "/examples/port/receiver");

    // send data in a loop


        // prepare the message using a bottle
        Bottle b;
        b.addString("Hello");

        // send the message through the port
        std::cout << "Sending Hello message " << std::endl;
        p.write(b);



    // close port
    p.close();

    return 0;


}
```

The program creates a port and sends a “Hello” message

Example: Yarp sender using Ports (compile)



mkdir build
cd build
cmake ../

CMake has to be executed on this CMakeLists

Setting up configuration for building

```
roscourse@roscourse-VirtualBox:~/Desktop/example_yarp_sender$ mkdir build
roscourse@roscourse-VirtualBox:~/Desktop/example_yarp_sender$ cd build
roscourse@roscourse-VirtualBox:~/Desktop/example_yarp_sender/build$ cmake ../
-- The C compiler identification is GNU 7.5.0
-- The CXX compiler identification is GNU 7.5.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc - works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ - works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found YARP: /usr/lib/x86_64-linux-gnu/cmake/YARP (found version "3.3.2")
-- Configuring done
-- Generating done
-- Build files have been written to: /home/roscourse/Desktop/example_yarp_sender/build
roscourse@roscourse-VirtualBox:~/Desktop/example_yarp_sender/build$
```

make

building

```
roscourse@roscourse-VirtualBox:~/Desktop/example_yarp_sender/build$ make
Scanning dependencies of target port-sender
[ 50%] Building CXX object port-sender/CMakeFiles/port-sender.dir/main.cpp.o
[100%] Linking CXX executable ../bin/port-sender
[100%] Built target port-sender
roscourse@roscourse-VirtualBox:~/Desktop/example_yarp_sender/build$
```

Example: Yarp sender using Ports (execute)

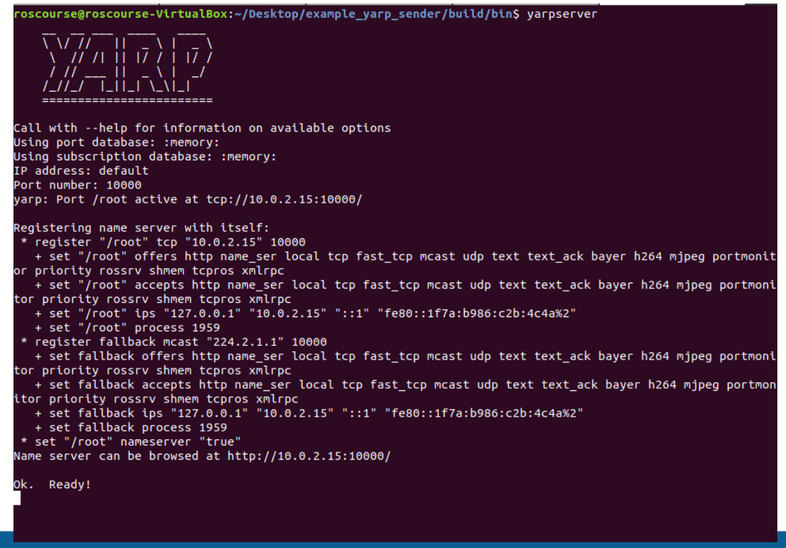
```
roscourse@roscourse-VirtualBox:~/Desktop/example_yarp_sender/build$ cd bin
roscourse@roscourse-VirtualBox:~/Desktop/example_yarp_sender/build/bin$ ls
port-sender
```

Run the executable file

```
roscourse@roscourse-VirtualBox:~/Desktop/example_yarp_sender/build/bin$ ./port-sender
```

Optional:
yarpserver &
For running in background

If you want it to send data in a YARP network do not forget to run the nameserver before



Receiver : What is needed to do…

Network yarp

Create a port -> ***Port p;***

Open the port

Create a connection: *Network::connect (name sender name receiver)*

Network::connect("/port/sender","/port/receiver");

Create a Bottle (like in the sender) : ***Bottle b;***

Read from the port: ***p.read(b);***

Access the Bottle (example with string): **std::string str =**

b.get(0).asString();

Sender/Receiver communication

SENDER

Create object Network: *Network yarp;*

Create a port: *Port p;*

Open the port: *p.open();*

Wait for a connection: *yarp.waitConnection(portsender, port receiver)*

Create a Bottle for communication: *Bottle b;*

Fill in the Bottle: b.addString()/b.addInt()…

Write message in the port: *p.write(b);*

Empty the bottle if used for more writings: *b.clear();*

Close the port: *p.close();*

RECEIVER

Create object Network: *Network yarp;*

Create a port: *Port p;*

Open the port: *p.open();*

Make a connection: *yarp.connect(portsender, port receiver)*

Create a Bottle for communication: *Bottle b;*

Read message in the port: *p.read(b);*

Access the Bottle: b.get(index).asString()/b.get(index).asInt()…

Empty the bottle if used for more readings: *b.clear();*

Close the port: *p.close();*

Sender is stuck until the receiver reads the message