YARP Network Protocol Specification 2.0
YARP Companion Utility Specification 2.0
(*** draft ***) [1]

Paul Fitzpatrick (paulfitz@liralab.it)

March 2, 2006

[1] This document specifies YARP network protocol version 2.0, and a companion utility. The protocol and utility is based on code by Giorgio Metta, Paul Fitzpatrick, Lorenzo Natale, and many others.

Contents

Chapter 1

Introduction

1.1 Definition of terms

The YARP library supports transmission of a stream of user data across various protocols – TCP, UDP, MCAST (multi-cast), shared memory, QNX message passing – insulating a user of the library from the idiosyncratic details of the network technology used. We call these these low-level protocols the "Carriers", to distinguish them from the higher-level protocols we will be concerned with here.

For the purposes of YARP, communication takes place throogh "Connections" between named entities called "Ports". These form a directed graph, the "YARP Network", where Ports are the nodes, and Connections are the edges.

Each Port is assigned a unique name, such as "/motor/wheels/left". Every Port is registered by name with a "name server". The goal is to ensure that if you know the name of a Port, that is all you need in order to be able to communicate with it from any machine.

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. In other words, the edges in the YARP Network are entirely mutable.

Ports are specialized into InputPorts and OutputPorts. Any OutputPort can send Content to any number of InputPorts. Any InputPort can receive Content from any number of OutputPorts. If an OutputPort is configured to send Content to an InputPort, they are said to have a Connection. Connections can be freely added or removed, and may use different Carriers.

The YARP name server is a server that 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. The protocol for communicating with the name server and its operation is specified in section 5.

Here are the specifications available in this document:

- Properties of a YARP network.
- Properties of the YARP utility.
- Properties of the YARP name server and the protocol used for communicating with it.
- Properties of ports and the protocol used for communicating with them.

1.2 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 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. That communication uses the "connection protocol".
- ▷ 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.
- ▷ External entities can also use the YARP name server protocol to query the name server.

The standard YARP companion utility can be used to create a name server, and also to act as an external entity for querying and modifying the YARP network.

Chapter 2

The standard YARP companion utility

We specify a standard command-line utility called "yarp" for performing a set of useful operations for a YARP network. The functionality described here can be provided in other ways also, but at a minimum this utility should be present on a YARP 2.0 system. We specify the utility by the user-facing functionality it provides. For any of the examples below, the word "verbose" inserted as the first argument should increase the level of detail at which the operation of the utility and problems encountered is described.

▷ *Command:* `yarp`

This lists a human-readable summary of the ways the utility can be used. Example output:

```
here are ways to use this program:
<this program> read /name
<this program> write /name [/target]
<this program> connect /source /target [carrier]
carrier can be: tcp udp mcast text
<this program> disconnect /source /target
<this program> server
<this program> where
<this program> version
<this program> name arguments
```

▷ *Command:* `yarp server`

▷ *Command:* `yarp server SOCKETPORT`

▷ *Command:* `yarp server IP SOCKETPORT`

This starts a name server running on the current machine, optionally specifying the socket-port to listen to (default whatever was used in the previous invocation, as recorded in a configuration file, or 10000 if this is the first time to run). Also, the IP by which the name server should be identified can optionally be specified (default is a fairly random choice of the IPs associated with the current machine).

▷ *Command:* `yarp where`

This will report where the name server is believed to be running, and the location of the configuration file used to determine that. Example output:

```
Name server is available at ip 5.255.112.225 port 10000
This is configured in file /home/paulfitz/.yarp/conf/namer.conf
You can change the directory where this configuration file is stored
with the YARP_ROOT environment variable.
```

▷ *Command:* `yarp version`

This will report on the yarp version available. Example:

```
YARP network version 2.0
```

▷ *Command:* `yarp name COMMAND` ARG_1 ARG_2 `...`

This will send the given command and arguments to the name server using the YARP name server protocol, and report the results. See Section 5.

▷ *Command:* `yarp read PORT`

This will create an input port of the specified name. It will then loop, reading "yarp bottles" (a simple serialized list) and prints their content to standard output. This simple utility is intended for use in testing, or getting familiar with using YARP.

▷ *Command:* `yarp write PORT`

This will create an output port of the specified name. It will then loop, reading from standard input and writing yarp bottles.. Optionally, a list of input ports to connect to automatically can be appended to the command. This simple utility is intended for use in testing, or getting familiar with using YARP.

▷ *Command:* `yarp connect OUTPUT_PORT INPUT_PORT`

▷ *Command:* `yarp connect OUTPUT_PORT INPUT_PORT CARRIER`

This will request the specified output port to send its output in future to the specified input port. Optionally, the carrier to be used can be added as an extra argument (e.g. tcp, udp, mcast, ...).

▷ *Command:* `yarp disconnect OUTPUT_PORT INPUT_PORT`

This will request the specified output port to cease sending its output to the specified input port.

Appendix B has a user guide to getting started with the YARP network companion utility.

Chapter 3

The connection protocol

This is the protocol used for a single connection from an output port to an input port. We discuss how this process gets initiated in the next section. At the point of creation of a connection, the following information is needed:

- An address – the machine name and socket-port at which the input port is listening.
- The name of the input port.
- The name of the output port associated with the connection. This name needs to be retained for proper disconnection in some cases. If the connection is not actually associated with a port, but is initiated by an external entity, then the name is not important and should be set to "external" (or any name without a leading slash characher).

3.1 Basic phases

The connection protocol has several phases – header, index, and body.

- Initiation phase
  - We begin once the sender has successfully opened a tcp socket connection to the receiver (assuming that is the carrier it is registered with).
- Header phase
  - This phase follows immediately after the initiation phase.
  - Transmission of protocol specifier
    - Sender transmits 8 bytes that identify the carrier that will be used. The header may be used to pass a few flags also.
    - Receiver expects 8 bytes, and attempts to find a carrier that is consistent with them.
  - Transmission of sender name
    - Sender transmits the name of the output port it is associated with, in a carrier specific way.
    - Receiver expects the name of the output port, transmitted in a carrier specific way.
  - Transmission of extra header material
    - Sender may transmit extra information, depending on the carrier.
    - Receiver may expect extra information, depending on the carrier.
- Header reply phase

  - This phase follows immediately after the header phase, and concludes the preamble to actual data transmission. After this phase, the two ports are considered connected.
  - Receiver may transmit some data, depending on the carrier. Receiver then may switch from the initial network protocol used to something else (udp, mcast, etc), again depending on the carrier.
  - Sender may expect some data, depending on the carrier. Sender then may switch from the initial network protocol used to something else (udp, mcast, etc), again depending on the carrier.

- Index phase
  - Sender sends carrier-dependent data describing properties of the payload data to come.
  - Receiver expects carrier-dependent data describing properties of the payload data to come.

- Payload data phase
  - Sender sends carrier-dependent expression of user data (maybe none).
  - Receiver expects carrier-dependent expression of user data.

- Acknowledgement phase
  - Receiver sends carrier-dependent acknowledgement of receipt of payload data (maybe none).
  - Sender expects carrier-dependent acknowledgement of receipt of payload data (maybe none).

This is the basic pattern of YARP communication between ports. Clearly different carriers have a lot of freedom in how they operate.

3.2 The "tcp" carrier

- Header and header reply
  - The 8-byte protocol specifier for tcp is: 'Y' 'A' 0xE4 0x1E 0 0 'R' 'P'. Another possible variant is: 'Y' 'A' 0x64 0x1E 0 0 'R' 'P'. The first version identifies a connection that sends acknowledgements; the second is for connections that omit acknowledgements.
  - The sender name is transmitted and expected to be in the following format: a 4 byte integer (little endian) giving the length of port, followed by the port name in characters, followed by the null character.
  - There is no extra header material for this carrier.
  - The header reply is as 8 bytes long: 'Y' 'A' B1 B2 0 0 'R' 'P', where (B1,B2) is a (little-endian) two-byte integer specifying a socket-port number (unused).
  - After the header reply, there is no switch in network protocol – the initial tcp connection continues to be used.

- Index, payload, and acknowledgement
  - The sender transmits 8 bytes: 'Y' 'A' 10 0 0 0 'R' 'P'. This identifies the length of the "index header" as 10.
  - The sender transmits 10 bytes: LEN 1 255 255 255 255 255 255 255 255. LEN is the number of blocks of user data need to be sent. This byte-sequence says there are LEN send blocks, 1 reply block expected, and that the sizes will be listed individually next (this odd format is for backward compatability with older YARP versions).

- The sender transmits LEN 4-byte little-endian integers, one for each of the LEN blocks of user data, giving the length of each block.
- The sender transmits 4 bytes: 0 0 0 0. This asks for a reply length of 0.
- If this is the variant of the tcp carrier that requires acknowledgments, then the receiver sends 8 bytes: 'Y' 'A' B1 B2 B3 B4 'R' 'P', where B1-4 is a little-endian integer giving a length (could be 0). It then sends that number of extra bytes.

3.3 The "udp" carrier

- Header and header reply
  - The 8-byte protocol specifier for udp is: 'Y' 'A' 0x61 0x1E 0 0 'R' 'P'. The following variant of this should also be accepted: 'Y' 'A' 0xE1 0x1E 0 0 'R' 'P' (it is the same thing).
  - Otherwise header and header reply are identical to the tcp case.
  - After the header reply, both sides switch to a udp connection to the socket-port specified in the header reply.
- Index, payload, and acknowledgement
  - Identical to tcp. Data is split arbitrarily to fit into datagrams.
  - Acknowledgments are not a possibility.

3.4 The "mcast" carrier

- Header and header reply
  - The 8-byte protocol specifier for mcast is: 'Y' 'A' 0x62 0x1E 0 0 'R' 'P'. The following variant of this should also be accepted: 'Y' 'A' 0xE2 0x1E 0 0 'R' 'P' (it is the same thing).
  - The sender name is sent as for tcp.
  - Extra header material is send – 6 bytes. The first 4 bytes specify a multicast IP address. Next 2 bytes are a (bigendian) integer giving a socket-port number. Note that producing these numbers can be helped by side communication with the name server.
  - There is no header reply for mcast.
  - Both sides switch to a multi-cast group on the specified IP and socket-port.
- Index, payload, and acknowledgement
  - Identical to udp.
  - But at most one connection from a given port with an mcast carrier should actually write to the multi-cast group.

3.5 The "text" carrier

- This carrier is carefully designed to make it easy to type into a terminal.
- Header and header reply
  - The 8-byte protocol specifier for text is: 'C' 'O' 'N' 'N' 'E' 'C' 'T' '␣'.

  - The sender name is sent as plain text followed by the newline character '\n'.
  - There is no extra material
  - There is no header reply expected for text.
  - There is no network protocol switch.

- Index, payload, and acknowledgement
  - There is no index.
  - The payload is expected to be a series of lines of text terminated by the newline character '\n'.
  - There is no acknowledgement expected for text.

3.6 The "shmem" carrier

This is essentially the same as the tcp carrier, except that there is no header reply, and there is a shift in protocol after header transmission on both sides to an ACE shared memory stream. This carrier is currently being reworked to make its specification independent of ACE, and to further improve efficiency in an existing implementation.

The advantage of this carrier is that it is fast – the best way to send messages between processes on a single machine. Of course, it doesn't work for processes on different machines.

3.7 The "local" carrier

This is a new carrier designed specifically for communication between threads in a single process. Giving a specification for the protocol it uses has low priority, since two such threads are unlikely to be using different YARP implementations.

3.8 Known protocol specifiers

Here are the currently known protocol specifiers. The "shmem" carrier is not yet documented, but is implemented in the C++ version of YARP.

8-byte magic number								**protocol**	**variant**
'Y'	'A'	0x61	0x1E	0	0	'R'	'P'	udp	
'Y'	'A'	0xE1	0x1E	0	0	'R'	'P'	udp	
'Y'	'A'	0x62	0x1E	0	0	'R'	'P'	mcast	
'Y'	'A'	0xE2	0x1E	0	0	'R'	'P'	mcast	
'Y'	'A'	0x63	0x1E	0	0	'R'	'P'	shmem	
'Y'	'A'	0xE3	0x1E	0	0	'R'	'P'	shmem	
'Y'	'A'	0x64	0x1E	0	0	'R'	'P'	tcp	without acks
'Y'	'A'	0xE4	0x1E	0	0	'R'	'P'	tcp	with acks
'C'	'O'	'N'	'N'	'E'	'C'	'T'	'␣'	text	
'L'	'O'	'C'	'A'	'L'	'I'	'T'	Y	local	

Chapter 4

Port commands

Every port is always available for new connections from external entities – to request that new connections between ports be created, old connections be removed, to inquire after status, etc. The protocol used for communicating with a port is layered on top of the protocol described in the previous section. Any carrier can be used. The "payload data" is as follows:

- We send an 8 byte header: 0,0,0,0, '~', CHAR, 0, 1.
- CHAR is a character that identifies what the message is about.
  - CHAR = 'd': this header is used to signal that user data is arriving next, as opposed to a port command
  - CHAR = anything else: this signals that a port command follows.
- for the port command case (CHAR = 0) the remainder of the message is interpreted as a string S.
  - S begins with '/', e.g. '/read': this is a request to add a Connection to the named InputPort.
  - S begins with '!', e.g. '!/read': this is a request to remove a Connection to the named InputPort.
  - S begins with '~', e.g. '~/read': this is a request to remove a Connection from the named OutputPort.
  - S is '*': this is a request for the port to dump information about what it is connected to.
  - S is 'q': the specific connection that the command is received on should now shut down.

Alternatively, with the "text" carrier, we send a string terminated in '\n'. This is the string S. The first letter is copied to be CHAR.

4.1 YARP URIs

Port names in YARP can contain multiple special elements. We've seen names such as "/write". We can also have names such as "udp://write" which means "connect to the port named /write using the udp carrier".

We can also prepend a network selector of the form "/net=NETNAME/". For example, a name such as "udp://net=196/write" means "connect to the port named /write using the udp carrier, and make the connection on the network with ip addresses beginning with 196". This is useful in scenarios with multiple networks, where it may be desirable to route connections through particular networks (for example, to devote a network to time-critical traffic). This functionality is supported primarily with the help of the name server. The ip it reports for a machine is usually a reasonable default, but the user can choose using "net=" to request a name on a particular network.

Symbolic network names can be configured. This process is not yet specified. You can do it right now by setting properties of a fake port called "networks" (no leading slash), where the properties are symbolic names and their values are the numeric network IP prefix. But this process will change.

4.2 Carriers supported

An implementation of YARP2 must support at least the "tcp" carrier. Other carriers that may be supported: "text", "udp", "mcast", "shmem", "qnx", "local".

As a place to start an implementation, the "text" carrier is very simple to implement, and can masquerade as "tcp" for the purposes of initial handshaking.

To see this, get the "netcat" program (available as debian package of the same name). In one terminal, run:

```
nc -l -p 9000
```

This starts a tcp listener on socket-port 9000, and prints out any data that arrives there. Then tell the name server to create an entry for this listener, and tag it as accepting text:

```
yarp name register /nc tcp ... 9000
yarp name set /nc accepts text
```

Now lets write some data to that port.

```
yarp write /write text://nc
```

Type something in, such as "hello world", and hit return. On the terminal running `nc` (netcat) you should see:

```
CONNECT /write
d
0 "hello world"
```

This is what text mode looks like, for the particular data type used by yarp read and write ("bottles"). As we saw in an earlier section, we can also write to ports in text mode. And if we were to restart nc and then try the following:

```
yarp connect text://nc /foo
```

```
CONNECT external
/foo
```

This is what a command to connect looks like in YARP2. If we omit the "text:/" then the tcp carrier may be used, which is compatible with YARP1 but is a bit less trivial to work with. Once our YARP implementations are up to date, the default command carrier will be switched to text.

Chapter 5

The name server protocol

The name server is a program that listens on a known socket-port[1] on a known machine. It tracks information about Ports in the YARP Network. If you know the name of a Port, you can query the name server to learn how to communicate with that Port.

The name server maintains a set of records, whose key is a text string. The contents are at least hostname, socket number, and protocol name. This describes how to contact the port. There is also a description of what kinds of connections the port can or is willing to participate in. The set of protocols the port can accept an incoming data connection for are named - this is the "accept" set. The set of protocols the port can create an outgoing data connection for are also named - this is the "offer" set.

For example, suppose you want to communicate with a Port called "/write". The first step is to ask the name server about this Port. The name server runs on a known socket-port of a known machine, listening for tcp connections. It is usually queried through a library call, but for illustration purposes we describe querying it using telnet. Suppose the name server is running on machine 192.168.0.3 and listening on socket-port 10000 (we will discuss a procedure for discovering this information later). Then we can query the name server about the Port /write as follows:

```
telnet 192.168.0.3 10000
```

The name server should start listening – if the connection is refused, something is wrong. Once the connection is made, type:

```
NAME_SERVER query /write
```

The server will respond with something of the form:

```
registration name /write ip 5.255.112.227 port 10001 type tcp
*** end of message
```

So the Port named /write is listening on the machine with IP address 5.255.112.227, on port 10001, and it expects TCP connections.

How do Ports get registered in the same place? Here's how to create a (fake) registration manually (usually it is of course done through a library call). Telnet to the name server as before, and type:

```
NAME_SERVER register /write
```

The server will respond with something of the form:

```
registration name /write ip 5.255.112.227 port 10001 type tcp
*** end of message
```

[1] We write socket-ports to distinuish tcp/ip port numbers for sockets, from higher-level YARP Ports.

The name server takes responsibility for allocating socket-ports and identifying the machine the Port runs on.

Note that the protocol described here for communicating with the name server is a YARP2 feature. YARP1 used a different, binary protocol. The human-readable protocol has been introduced to make the system more transparent and easier to step through.

For yarp utilities to correctly discover how to contact the name server, there should be a file namer.conf in the directory $HOME/.yarp/conf/ (or in the directory specified by an environment variable $YARP_ROOT) that looks like this:

```
192.168.0.3 10000
```

This gives the machine and socket-port that the name server is assumed to be running on.

If this file does not exist, or is incorrect, yarp utilities will attempt to contact the nameserver using multi-cast broadcasts to 224.2.1.1 port 10001 (this is a YARP2 feature, not available in YARP1). If the nameserver is running a machine reachable from multi-cast, it will respond with its "true" tcp address, which will then be used by the utility. The configuration file will be updated automatically for future reference. The multi-cast protocol is identical to the normal tcp protocol. Clients can broadcast "NAME_SERVER query root" to trigger the name server to send a record of the form "registration name root ip ADDRESS port NUMBER type CARRIER". The "root" record is a special record corresponding to the name server. Multi-cast broadcasts should not generally be used by clients to communicate with the name server, since the output of the name server is not tagged with the recipient, so there is the potential for cross-talk.

▷ *Command:* `NAME_SERVER query PORT`

Requests registration information for the named port. Response is of the following form:

```
registration name PORT ip ADDRESS port NUMBER type CARRIER
*** end of message
```

For example:

```
registration name /write ip 5.255.112.227 port 10001 type tcp
*** end of message
```

If there is no registration for the port, the registration line is omitted, and instead the response is simply:

```
*** end of message
```

▷ *Command:* `NAME_SERVER register PORT`

Requests creation of registration information for the named port. Response is of the following form:

```
registration name PORT ip ADDRESS port NUMBER type CARRIER
*** end of message
```

For example:

```
registration name /write ip 5.255.112.227 port 10001 type tcp
*** end of message
```

Optionally, the user can take responsibility for more, and issue commands in one of the following forms:

```
NAME_SERVER register PORT CARRIER
NAME_SERVER register PORT CARRIER IP
NAME_SERVER register PORT CARRIER IP NUMBER
```

Any value (including the port name) can be replaced by "..." to leave it up to the name-server to choose it. For example:

```
NAME_SERVER register ... tcp 127.0.0.1 8080
```

Gives something of the form:

```
registration name /tmp/port/1 ip 127.0.0.1 port 8080 type tcp
*** end of message
```

If you choose to set the ip yourself, be careful – there is the possibility of problems with multiple ways to identify the same machine. It is best to let the name server choose a name, which it should do in a consistent way. If a machine has multiple ip addresses on multiple networks, that can be handled – see the discussion of the `ips` property in the section on `set`. That is important for the purposes of controlling which network is used for connections from one port to another.

▷ *Command:* `NAME_SERVER unregister PORT`

Removes registration information for the named port. Response is of the following form:

```
*** end of message
```

▷ *Command:* `NAME_SERVER list`

Gives registration information of all known ports. Response is of the following form:

```
registration name /write ip 130.251.4.159 port 10021 type tcp
registration name /read ip 130.251.4.159 port 10031 type tcp
registration name /tmp/port/4 ip 130.251.4.159 port 10011 type tcp
registration name /tmp/port/3 ip 130.251.4.52 port 10021 type tcp
registration name /tmp/port/2 ip 130.251.4.52 port 10011 type tcp
registration name /tmp/port/1 ip 130.251.4.159 port 10001 type tcp
*** end of message
```

▷ *Command:* `NAME_SERVER set PORT PROPERTY VALUE1 VALUE2 ...`

The name server can store extra properties of a port, beyond the bare details associated with registration. The `set` command is used to do this. For example, the command:

```
NAME_SERVER set /write offers tcp udp mcast
```

Gets the following response:

```
port /write property offers = tcp udp mcast
```

The `get` and `check` commands can then be used to query such properties.

There are some special properties used by YARP. Property "ips" can list multiple identifiers of a machine. Property "offers" lists carriers that an output port can support. Propery "accepts" lists carriers that an input port can support.

▷ *Command:* `NAME_SERVER get PORT PROPERTY`

Gets the values of a stored property. For example, after the `set` command example shown earlier, the command:

```
NAME_SERVER get /write offers
```

Returns the following response:

```
port /write property offers = tcp udp mcast
```

▷ *Command:* `NAME_SERVER check PORT PROPERTY VALUE`

Checks if a stored property can take the given value. For example, after the `set` command example shown earlier, the command:

```
NAME_SERVER check /write offers tcp
```

Returns the following response:

```
port /write property offers value tcp present true
```

▷ *Command:* `NAME_SERVER route PORT1 PORT2`

Finds a good way to connect an output port to an input port, based on the carriers they have in common (preferred carriers can optionally be added to this command in decreasing order of preference) and which carriers are physically possible (for example, 'shmem' requires ports to be on the same machine, and 'local' requires ports to belong to threads with a shared memory space). For example, the command:

```
NAME_SERVER route /write /read
```

Returns the following response:

```
port /write route /read = shmem://read
```

Suggesting that shmem is the best carrier to use.

Bibliography

[] Richard T. Vaughan, Brian P. Gerkey, and Andrew Howard. On device abstractions for portable, reusable robot code. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2121–2427, Las Vegas, Nevada, USA, October 2003.

Appendix A

Status of implementations

A.1 C++

There is a partially-conforming implementation of YARP2 written in C++. It is available in CVS from:

```
CVSROOT = cvs.sourceforge.net:/cvsroot/yarp0
Modules = yarp/src/libYARP_OS yarp/src/tools yarp/conf
```

See yarp0.sourceforge.net for instructions on how to access this. This code is mature, well-tested, but does not yet conform fully with the protocols described in this document.

There is a drop-in replacement implementation of YARP2 being developed in:

```
Modules = yarp/src/libYARP_OS2
```

That implementation follows this document carefully, but at the time of writing is not yet tested much in the real world.

A.2 Matlab

There is a Matlab wrapper around the C++ implementation – see `yarp/src/libYARP_matlab` in the above repository.

A.3 Python

Python wrappers around the C++ implementation exist.

A.4 Java

There is a conforming implementation of YARP2 written in Java. It is available in CVS from in the `yarp/src/java` directory. See yarp0.sourceforge.net for instructions on how to access this.

Appendix B

Using the YARP companion utility

This is informal, tutorial-style section meant to give the flavor of using YARP.

First, please open two terminal windows; we'll refer to these as A and B. Before doing any communication, is is necessary to start the YARP name service – this is a program which keeps track of what YARP resources are currently available and how to access them. In terminal A, type:

```
yarp server
```

If all goes well, you will see the message "yarp: name server starting at ..." – if not, check any configuration file mentioned in the output of "yarp where". Next, in terminal B, type:

```
yarp check
```

This will try to communicate with the name server, send a message, receive a message, and basically make sure that everything is working well. Here is what you should see:

```
================================================================
=== Trying to register some ports
yarp: Registered port ... as tcp://5.255.112.225:10011
yarp: Registered port ... as tcp://5.255.112.225:10001
================================================================
=== Trying to connect some ports
yarp: /tmp/port/2: Receiving input from /tmp/port/1 to /tmp/port/2 using tcp
yarp: Sending output from /tmp/port/1 to /tmp/port/2 using tcp
================================================================
=== Trying to write some data
================================================================
=== Trying to read some data
*** Read number 42
================================================================
=== Trying to close some ports
yarp: Shutting down port /tmp/port/2
yarp: /tmp/port/2: Stopping input from /tmp/port/1 to /tmp/port/2
yarp: stopping output from /tmp/port/1 to /tmp/port/2
yarp: Shutting down port /tmp/port/1
*** YARP seems okay!
```

Here is an example of things going wrong:

```
================================================================
=== Trying to register some ports
yarp: Couldn't connect to 5.255.112.225
yarp: Name server missing
```

If networking is broken on your machine, then YARP will have trouble. If you are running the YARP name server on a machine that does not have a static IP address or a name registered with DNS, then you should check the configuration file mentioned by "yarp where", and make sure that the first line of the configuration file is of the form:

current.valid.ip.address 10000

The number "10000" is the default port number that the YARP name service uses. This is usually fine; if you have a conflict with this, then please change it in the configuration file.

If all is well, we can try exercising YARP a little more.

In terminal B, type:

```
yarp read /port1
```

This will report `yarp: Registered port /port1 ...` in terminal B and in terminal A some corresponding message will also appear. Next, in terminal C, type:

```
yarp write /port2
```

If all goes well, this will report `yarp: Registered port /port2 ...` in terminal C and in terminal A some corresponding message will also appear. Now, in terminal D, type:

```
yarp connect /port2 /port1
```

Terminal C will report `Connecting /port2 to /port1`. Now if you type "hello world" into terminal C, and hit enter, that text will appear on terminal B.

If everything is going okay so far, type on terminal E:

```
yarp read /port3
```

and on terminal D, type:

```
yarp connect /port2 /port3
```

Now any line you enter in terminal C should appear in both terminal B and E.

Appendix C

Manually interacting with ports

Suppose we have created ports as follows by typing the following in different terminals:

```
yarp server
yarp write /write
yarp read /read
yarp read /read2
```

We could connect and disconnect ports using the YARP companion utility, but here's how we could do the same thing "manually":

```
command: yarp where
response: Name server is available at ip 192.168.0.3 port 10000

command: telnet 192.168.0.3 10000
type: NAME_SERVER query /write
response: registration name /write ip 192.168.0.3 port 10001 type tcp
*** end of message
[connection closes]

command: telnet 192.168.0.3 10001
type: CONNECT anonymous
response: Welcome anonymous
type: *
response: This is /write
There are no outgoing connections
There is this connection from anonymous to /write using protocol tcp
*** end of message
type: /read
response: Connected to /read
type: *
response: This is /write
There is a connection from /write to /read using protocol tcp
There is this connection from anonymous to /write using protocol tcp
*** end of message
type: !/read
response: Removing connection from /write to /read
type: /mcast://read
response: Connected to /read
type: /read2
response: Connected to /read2
type: *
response: This is /write
```

```
There is a connection from /write to /read using protocol mcast
There is a connection from /write to /read2 using protocol tcp
There is this connection from anonymous to /write using protocol tcp
*** end of message
type: q
response: Bye bye
[connection closes]

command: telnet 192.168.0.3 10000
type: NAME_SERVER query /read
response: registration name /write ip 192.168.0.3 port 10002 type tcp
*** end of message
[connection closes]

command: telnet 192.168.0.3 10002
type: CONNECT anonymous
response: Welcome anonymous
type: *
response: This is /read
There are no outgoing connections
There is a connection from /write to /read using protocol mcast
There is this connection from anonymous to /read using protocol tcp
*** end of message
type: q
response: Bye bye
[connection closes]
```