

# The NARP protocol specification

## A Generic Recursive Communication Protocol for Networked Applications

In this document we explain the purpose and provide a draft specification for the NARP protocol, a general-purpose networking protocol destined to be used in many layers of a new operating system and networking system.

## 1 Introduction

We begin by remarking that a basic operation in all computer operation processes consists in naming objects and providing acces to these named objects. Here are a few examples of naming in real use cases:

- Naming of files on a local or distant file system
- Naming of devices in the `/dev` virtual filesystem on Unix machines
- Naming of networked machines (with IP adresses and DNS records)
- Naming of internet ressources over protocols such as HTTP, IMAP, ...

We propose here a novel architecture with the purpose of unifying all the naming happening at all levels of the system, with two base concepts : *objects* and *service*.

- *objects* are ressources that may implement different semantics : bidirectionnal communication (such as sockets) ; unidirectionnal communication (FIFO-like) ; file semantics ; etc.
- *services* are a way of naming objects, querying the interfaces they implement, and multiplexing communications with them

We suggest that a NARP service may be provided on any bidirectionnal channel of communication supporting the (reliable) sending and recieving of messages. In addition, NARP objects may implement such a send/recieve interface ; therefore a NARP service can be channeled into an object. Such a construction of using a NARP object to access a NARP service is a fundamental operation that we call *recursive multiplexing*, or just *multiplexing*.

The NARP protocol is meant to include a variety of different operations that may or may not be implemented by a specific NARP server.

## 2 High-level overview

### 2.1 The basic operations on services and objects

A NARP service is basically any object that implements the following operations:

- *query* : get information on a ressource identified by name
- *list* : know the names of ressources presented by the service (eventually in a specific sub-path)
- *attach* : get an object interface for accessing a ressource, identified by name

A NARP object is basically any object that implements the following operations :

- *send* : send a message (an arbitrary byte string) to the object

- *recieve* : recieve a message from the object (this may be done asynchronously with handler functions)
- *detach* : delete object connection

## 2.2 The basics of the NARP protocol

Given any interface with send/recieve capabilities considered as an assymetric (client/server) configuration, the following client messages consitute the basics of the NARP protocol for providing a NARP service on the interface:

- **hello** : initialize a connection, check version information, ...
- **authenticate** and appropriate response messages : use credentials (user/password or access token) to gain acces to some ressources provided by the server (the protocol is thus statefull)
- **walk**, **list** and appropriate response messages : get information about the available ressources
- **attach** and appropriate response messages : give an identifier (a descriptor) to a ressource in order to communicate with it
- **send** and appropriate response messages : send a message to an attached ressource, identified by its descriptor
- **detach** : close a descriptor and detach from a ressource
- **create** : requests the creation of a ressource in the namespace
- **delete** : requests the deletion of a ressource in the namespace
- **link** : create a link for an object under another name (these are always symbolic links, and are independent of the pointed ressource)

The server may also at any moment send a message, including:

- a response to a query
- **recieve** : a notification of a message sent from the object to the client
- **detached** : the connection to the object has been terminated by the object server

## 2.3 Recursion

If an object is a NARP server, the messages sent to it and recieved from it are messages of the NARP protocol. Otherwise, they are arbitrary.

## 2.4 Reverse object

Some NARP servers may support reverse object serving: the client creates an object on the server and handles all the requests arriving to this object (therefore the initial NARP server only serves as a relay between the new server and its clients<sup>1</sup>). A client wishing to act as a reverse object server may use the following commands:

- **serve** : listen for attach requests on a servable (empty) object created in the server namespace (if authorized)

---

1. Research is to be done on shortcutting mechanisms

- **accept** and **reject** : accept (or reject) an attach request to the object
- **detach** : close connection between object and client (this is the same detach message as in standard communications)
- **unserve** : stop serving for the object. Attached clients continue to be attached.

The server may in turn send the following messages concerning the server object:

- **attach\_request** : a client is willing to attach to the object. A descriptor is already associated to the connection to be established, but the server may reject it.

Once a client is attached to the object, a classical send/recieved interface is provided.

Typically, the protocol exchanged over the object is NARP protocol, therefore enabling the reverse server to provide its own namespace and other fonctionnality.

## 2.5 Specific object types and associated messages

### 2.5.1 Objects are sockets

Sockets are the basis of the NARP protocol : attaching to an objects opens a socket connection to the process serving the object, and when the connection is accepted, basic send/recieve functionality is provided. See also the reverse object protocol described in section 2.4.

### 2.5.2 File objects

Small files may implement the following interface:

- **put** : erase the whole file and put the transmitted content
- **get** : retrieve the whole file content

Big files may implement the following interface:

- **write** : write a portion of the file at a given offset
- **read** : read a portion of the file at a given offset

### 2.5.3 User IO (terminals...)

Virtual terminals can be seen as objects implementing a simple send/recieve semantic, where the data transmitted is unstructured (or structured given a specific terminal data structure). More specific interfaces can be defined for advanced terminals and GUIs.

### 2.5.4 Specific applications

Specific applications may define custom messages. Examples include:

- e-mail
- instant messaging
- collaborative editing of text-based documents

and many other applications yet to be invented.

## 2.6 Big messages

The message size in the NARP protocol is limited to 64kb, and recommended not to exceed 4kb+header (4kb is the size of a memory page on many machines). Therefore a possibility would be for the NARP protocol to include a way to transmit big messages by fragmenting them into small messages. Optional error correction may be included. This can be useful for example when using `put` or `get` on large files, or `reads` and `writes` of big file portions. The receiving of a large fragmented message may have a specific implementation allowing the receiver to work with the partial data as soon as it starts arriving and not having to wait for the whole message to be transmitted and buffered.

## 2.7 Permissions

For each attached client the server may keep track of associated permissions, and accept or reject requests according to those permissions. The client may use an authentication command to gain supplementary privileges on the server's resources. The client may request a token to delegate its privileges on a given object to another client. Advanced right management functionalities are to be discussed.

## 2.8 Example NARP servers

### 2.8.1 Virtual NARP server (i.e. NARP router)

This server implements a namespace where any client may create an empty object and serve connections to it. Additionally, the server may implement the possibility to create virtual files, virtual directories, FIFO queues, etc.

This server may be connected to other virtual NARP servers in order to provide a global namespace accessible to all. Each virtual NARP server acts as an endpoint into the network and may have functionality for routing the communications to objects to the clients that serve them.

### 2.8.2 NARP file server

This server simply implements access to a filesystem : listed resources are the same as the files present in a served directory, each of these implements the filing protocol (served directly by the file server), and the creation of files/directories may also be implemented.

### 2.8.3 NARP terminal/GUI server

Clients may create objects on the server ; each of these objects corresponds to a GUI window. Two interfaces may be implemented : text IO (terminal) and graphical interaction. Advanced terminal interaction features may be implemented at the protocol level, such as auto-completion of commands or of text being edited...

Suggestion for a third kind of window : the data sent by the client corresponds to a description of the scene in a given markup language and the server does the rendering. The client can also subscribe to events such as clicking on an item or entering text. This possibility is to be explored.

### 2.8.4 NARP e-mail and newsgroup server

Several features to be implemented:

- user login and private user mailboxes
- bridge to standard SMTP/POP3/IMAP services
- private threads of conversation with access rights (the users don't each have a copy of the thread)

- synchronization between many servers
- public discussion forums

### 2.8.5 NARP chat server

- user login and status notification
- online and offline private messaging
- public chat rooms, chat room logging independently of user being online or offline
- bridging and synchronization between many servers

### 2.8.6 NARP applicative server

TODO...

## 3 Specifics of the NARP protocol

### 3.1 Protocol description format

A protocol message is given in the following form:

element type	element type	...	element type
element description	element description	....	element description

The following element types apply:

- int16, int32, int64 : 16-bit, 32-bit or 64-bit little-endian integers
- str : a string, prefixed by a 16-bit length header
- arr( $T$ ) : an array of  $T$ 's (where  $T$  is another element type), prefixed by a 16-bit length header
- \* (for the last element) : consider all the rest of the message as a byte string

### 3.2 Basic message format

The basic format of a message is :

int16	int16	*
message size	message type	payload

We will abbreviate by “header” the first 32 bits (4 bytes) of the message. The list of message types is given in section 3.10.1.

Messages for communication with an attached resource will have the following format :

int16	int16	int32	*
message size	message type	resource descriptor (handle)	payload

Many client messages awaiting a response will have a message ID included ; this message ID is an arbitrary number generated by the client and used by the server when giving its response. The header then looks like this:

int16	int16	int32	*
message size	message type	message ID	payload

### 3.3 Message list for core NARP protocol

Client messages have an up arrow (↑) next to their name, while server messages have a down arrow (↓).

The core NARP protocol is meant for small size and rapidity (so that many layers can be encapsulated with minimal overhead), therefore no acknowledgment is to be sent for most messages. Specific interfaces for objects are responsible for making sure messages have been transmitted correctly.

#### Hello↑↓

	int32	arr(int32)
header	version	list of needed/provided interfaces

When a NARP connection is established, the client is always the first to send a **Hello** message. The object may then respond either with a **Hello** message indicating that the requested interfaces can be provided, or with an **Error** message. The two common error causes are *interface not implemented* and *incompatible versions*.

For interface numbers : see table in section 3.10.3.

#### Error↓

Generic error response message for any operation.

	int32	int32	str
header	request ID	error ID	error string

Common error IDs are specified in section 3.10.2.

#### Ack↓

	int32
header	request ID

Generic acknowledgment message for commands that require it. An acknowledge implies the command has been successfully executed (otherwise an error message is sent).

#### Stat↑

	int32	str
header	request ID	filename

The request ID is an ID decided by the client so that it can identify the answer.

#### StatR↓

Response to the **Stat** message.

	int32	arr(int32)
header	request ID	implemented interface

Common interface numbers are to be found in section 3.10.3.

If a **Stat** query on an object gives a certain list of interfaces, then when connecting to the object at least all these interfaces must be included in the server's **Hello** message as supported interfaces.

Note that some interface numbers correspond to actions that can be done on the object from the connection where the object exists (e.g. : symbolic link, directory), and others correspond to actions that can be performed after attaching to the object (e.g. file, terminal, ...)

#### List↑

	int32	int32	int32	str
header	request ID	first entry number	number of entries requested	base path string

### ListR↓

One message is passed for each entry in the requested range:

	int32	int32	str
header	request ID	entry number	entry name

After the directory has finished being enumerated, a supplementary entry is given with entry number the last valid entry number plus one and an empty entry name. This supplementary entry is only given if its (fictious) entry number is included in the range requested by the client.

Possible extension : combine List and Stat so that when the answer to List is given, information is also given on the object's implemented interfaces.

### Attach↑

	int32	str
header	request ID	filename

### Attached↓

	int32	int32
header	request ID	handle

(the handle, ie the ressource descriptor, is attributed by the server)

### Send↑

	int32	*
header	handle	payload

### Recieve↓

Spontaneous server message indicating some data is sent by object.

	int32	*
header	handle	payload

### Detach↑

	int32
header	handle

### Detached↓

Spontaneous server message indicating the object has been detached.

	int32
header	handle

### Create↑

	int32	arr(int32)	str
header	request ID	needed interfaces	path

A create request is accompanied with a list of needed interfaces that direct the server into creating the corresponding type of object (e.g. an empty object to be served, a directory, a file, ...)

**Created**↓

	int32	arr(int32)
header	request ID	implemented interfaces

Signals that the object has been created, and has corresponding interfaces associated to it.

**Delete**↑

	int32	str
header	request ID	path

This message expects a standard **Ack** response message.

**Link**↑

	int32	str	str
header	request ID	destination path	link path

This message expects a standard **Ack** response message.

Semantics of the link object:

- attaching or serving on this objects corresponds to resolving the linked path and attaching/serving on the linked object
- stating the link will stat the linked object and add as an implemented interface the “this is a symlink” information
- directory listings follow links
- deleting the link will not delete the original file but only the link

**ReadLink**↑

	int32	str
header	request ID	path

**ReadLinkR**↓

	int32	str
header	request ID	link description

This will only return the first level of linking, ie the link data directly associated to the link object.

**Rename**↑

	int32	str	str
header	request ID	original path	new path

This message expects a standard **Ack** response message.

**Serve**↑

	int32	str
header	request ID	path

This message is a request for the client to be a reverse server to an object. The response message to this message is an **Attached** message. The handle attributed to the served object is known as the *server handle* and is used in the following few messages.



To stop serving an object, the client simply sends a **Detach** command on the server handle. The semantics is that all connections that have been opened through the reverse-served object are preserved when the object stops being served, and an individual **Detach** message must be sent to all of them if we want to close them.

#### Incoming↓

	int32	int32
header	server handle	client handle

This message is sent by the server when another client wishes to attach to an object reverse-served by this client. The server handle is the one given as a response to the **Serve** message. The client handle is a handle associated to the connection. The reverse server may reject the connection by issuing a **Detach** command on the client handle, or may accept it using the **Accept** message given below.

#### Accept↑

	int32
header	client handle

Once a connection has been accepted, the reverse server may at any moment close it by sending a **Detach** command on the corresponding client handle.

### 3.4 Big message protocol

To be defined. Is it really useful? What role exactly does it have? Can it implement repetition in the case where the message hasn't been acknowledged? ...

Reserved message IDs : [20, 30) and [10020, 10030).

### 3.5 Authentication and rights management commands

#### Authenticate↑

	int32	int32	*
header	message ID	authentication method	authentication data

Used to gain access using credentials (user/password, token, ...). Response messages are standard **Ack** on success or **Error** on failure. Authentication methods include :

- 1 : user + password
- 2 : token

#### NewToken↑

	int32	str
header	message ID	path

Requests the server to create an authentication token for accessing a given object with the privileges of the connected client. Once the token has been returned, it may be transmitted to another client so that that client will use it to gain same access to the object.

#### NewTokenR↓

	int32	str
header	message ID	token

More TODO : request account creation, manage user groups and ACLs, ...

## 3.6 File protocol

Client messages [50,100) ; server messages [10050,10100).

## 3.7 UI protocols

Client messages [100, 200) and server messages [10100, 10200).

### 3.7.1 Terminal protocol

TODO

### 3.7.2 Graphical user interface protocol

TODO

## 3.8 Communication protocols

Client messages [200, 300) and server messages [10200, 10300).

### 3.8.1 Email and newsgroups protocol

Client messages [200, 220), server messages [10200, 10220).

### 3.8.2 Instant messaging protocol

Client messages [220, 250), server messages [10220, 10250).

## 3.9 Other protocols

Protocols not discussed in this specification may use client messages with type IDs [1000, 10000) and server messages [11000, 20000). Overlaps between several protocols are allowed : the information about implemented interfaces for an object is meant to disambiguate such situations.

## 3.10 Table of IDs

The tables presented in this section give the number associated to the message types. These tables are the reference on the subject ; any information found somewhere else is wrong if it is not the same as found here. This is for protocol version 1.

### 3.10.1 Message types

#### Base protocol

message	↑ id	↓ id	message	↑ id	↓ id
<b>Hello</b>	0	10000	<b>Stat / StatR</b>	10	10010
<b>Error</b>		10001	<b>List / ListR</b>	11	10011
<b>Ack</b>		10002	<b>Create / Created</b>	12	10012
			<b>Delete</b>	13	
<b>Attach / Attached</b>	5	10005	<b>Rename</b>	14	
<b>Send / Recieve</b>	6	10006	<b>Link</b>	15	
<b>Detach / Detached</b>	7	10007	<b>ReadLink / ReadLinkR</b>	16	10016
<b>Serve</b>	8				
<b>Incoming</b>		10008			
<b>Accept</b>	9				

### Authentication & privileges

message	↑ id	↓ id
<b>Authenticate</b>	30	
<b>NewToken / NewTokenR</b>	31	10031

#### 3.10.2 Error messages

id	cause
1	Incompatible versions
2	Command/interface not implemented
3	Invalid request (e.g. : out of bounds)
4	Invalid handle
5	Attach request rejected
6	Action impossible because object is in use (cannot delete, ...)
7	No such object (invalid path)
8	Could not resolve link
9	Incorrect credentials
10	Unauthorized

#### 3.10.3 Object interfaces

id	name	must implement messages
0	servable	<b>Serve, Accept, Incoming</b>
1	NARP service	once attached, inside data is also NARP protocol
2	non-NARP socket	once attached, inside data is arbitrary
3	enumerable	<b>List, ListR</b>
4	is symlink	<b>ReadLink, ReadLinkR</b>
10	file	once attached, file semantics
11	terminal	once attached, terminal semantics
12	graphics window	once attached, graphics semantics

## 4 Architecture of a NARP implementation in OCaml or Haskell