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, IRC, specfic web services, ...

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 a client/server protocol 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 (possibly 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`, `delete`, `rename`, `link` : requests the creation or modification of a ressource in the namespace

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[1]). 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 in specific situations where too many levels of recursion cause a performance issue.

- `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 functionnality.

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 functionnality 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. Optionnal error correction may be included. This can be useful for example when using `put` or `get` on large files, or `read`s and `write`s of big file portions. The recieving of a large fragmented message may have a specific implementation allowing the reciever 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. Research is yet to be done on this specific subject.

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 ressources. The client may request a token to delegate it's privileges on a given object to another client. Advanced right management functionnalities are to be discussed.

2.8 Reliability concerns

The NARP protocol relies on the fact that when transmitting a message, the other end will recieve it. It is nevertheless recommended that NARP implementations support the repeating of messages if an expected acknowlegment has not arrived after a given delay.

2.9 Example NARP servers

2.9.1 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. Additionnaly, 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 functionnality for routing the communications to objects to the clients that serve them.

2.9.2 NARP file server

This server simply implements access to a filesystem : listed ressources 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.

2.9.3 NARP terminal/GUI server

Clients may create objects on the server ; each of these objects correspond 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 subscribed to events such as clicking on an item or entering text. This possibility is to be explored.

2.9.4 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

2.9.5 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

2.9.6 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 3.7.1.

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

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

Many client messages awating 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 acknowlegment is to be sent for recursive send/recieve messages. Other messages usually imply some kind of action or getting of information, therefore an acknowlegment or an error is usually sent as a response.

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 3.7.3.

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 3.7.2.

Ack↓

	int32
header	request ID

Generic acknowlegment message for commands that require it. An acknowlege implies the command has been sucessfully 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 3.7.3.

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↓ Response to the **List** message.

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 (ficious) 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↓ Response to the **Attach** command.

	int32	int32
header	request ID	handle

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

Send↑

	int32	*
header	handle	payload

This message does not expect a response.

Recieve↓

Spontaneous server message indicating some data is sent by an attached ressource. This message does not expect a response.

	int32	*
header	handle	payload

Detach↑

	int32
header	handle

This message does not expect a response.

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↓ Response to the **Create** command.

	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↓ Response to the **ReadLink** message.

	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	array(int32)
header	request ID	path	announced interfaces

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 **Incoming** and **Detach** 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 openned 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.

The *announced interfaces* serves to answer **Stat** messages on the object while we are serving it.

Incoming↓

	int32	int32
header	server handle	client hande

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.

Unbox↑

	int32	int32	int32
header	request ID	outer handle	inner handle

Consider the handle *outer handle* as a NARP protocol service, and associate a handle in the outer layer to the handle of the inner layer with handle *inner handle*.

Example : in connection A we have a connection open on handle 5 which contains NARP data that we will call B, and in connection B we have another connection open on handle 7. Issuing a Unbox(id, 5, 7) request on A will lead to the server creating a handle (say 12) where sending corresponds to sending a message to handle 7 on connection B, and such that all messages recieved on handle 5 (ie on connection B) are filtered and messages whose destination is handle 7 on connection B are removed from the stream and issued on handle 12 of connection A instead.

The answer to such a request is an **Attached** response giving a handle to the unboxed connection.

Systematically unboxing open connections may lead in some cases to the network infrastructure being able to do simplifications in the interconnections. In other cases it may result to useless overhead on the server side : in such a case the server may refuse an unbox request.

Plug↑

	int32	int32	int32
header	request ID	handle A	handle B

Ask the server to redirect all messages recieved on handle A to handle B and all mesages recieved on handle B to handle A. The messages recieved on either handle are not sent to the client anymore.

The answer messages are standard **Ack**/**Error** messages.

Unplug↑

	int32	int32	int32
header	request ID	handle A	handle B

Undoes a plugging.

3.4 Big message protocol

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

3.5 Authentification and rights managment commands

Authenticate↑

	int32	int32	*
header	message ID	authentification method	authentification data

Used to gain access using credentials (user/password, token, ...). Response messages are standard **Ack** on success or **Error** on failure. Autentification 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↓ Response to the **NewToken** message.

	int32	str
header	message ID	token

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

3.6 TODO

- file protocol
- system protocols (see section on OS design using NARP)
- UI protocols (terminal, GUI)
- communication protocols (mail, IM)

3.7 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.

3.7.1 Message types

Base protocol

message	↑ id	↓ id
Hello	0	10000
Error		10001
Ack		10002
Attach / **Attached**	5	10005
Send / **Recieve**	6	10006
Detach / **Detached**	7	10007
Serve	8	
Incoming		10008
Accept	9	

message	↑ id	↓ id
Stat / **StatR**	10	10010
List / **ListR**	11	10011
Create / **Created**	12	10012
Delete	13	
Rename	14	
Link	15	
ReadLink / **ReadLinkR**	16	10016
Unbox	20	
Plug	21	
Unplug	22	

Authentication & privileges

message	↑ id	↓ id
Authenticate	30	
NewToken / **NewTokenR**	31	10031

3.7.2 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

3.7.3 Object interfaces

id	name	must implement messages
0	servable	**Serve**, **Accept**, **Incoming**
1	enumerable	**List**, **ListR**
2	is symlink	**ReadLink**, **ReadLinkR**
9	non-NARP inside	once attached, inside data is arbitrary
10	NARP service	once attached, inside data is a NARP service (ie has objects, ...)
11	NARP unbox	once attached, **Unbox** command supported
12	NARP plug	once attached, **Plug** and **Unplug** commands supported
20	file	once attached, file semantics
21	terminal	once attached, terminal semantics
22	graphics window	once attached, GUI semantics

Servable This interface specifies that the object is currently an empty object waiting for someone to issue a **Serve** command on it, providing it with an implementation of some interfaces.

non-NARP inside This interfaces indicates that once attached to the object, the messages sent/recieved to it are not supposed to be NARP format but any arbitrary format. If this interface is not specified, then it is expected that the messages transmitted will follow the general NARP protocol (message format, standard hello/ack/error messages).

NARP service This interface indicates that once attached to the object, one can have access to a new NARP namespace where at least the following operations are supported : **Stat**, **Attach**, **Send**, **Recieve**, **Detach**. Additionnal messages may or may not be supported.

4 Architecture of a NARP implementation in OCaml or Haskell

An asynchronous implementation can be easily programmed in functionnal languages such as OCaml or Haskell, using closures as continuations for *what to do when a (response) message arrives*.

TODO

5 Using NARP to design an Operating System

When designing the NARP protocol, we had in mind that it would be possible to use it in a new operating system design at many levels : access to devices, process management, memory management, filesystems, IPC, GUI, ...

Kernel helpers could be developped so that a part of the NARP multiplexing and demultiplexing takes place in kernel land, before messages are passed to userspace. For instance, this would allow the simplification of useless mux-demux chains taking place on the same machine. The mux-demux helper can be implemented via the **Unbox** protocol message, handled at the level of the root stream of NARP communication with the kernel. Another possible helper would be to map a virtual memory region to a NARP ressource implementing a standard filing protocol, much like memory mapped files in standard OSes (only this would work with arbitrary ressources).

In this section we will develop on a concrete proposal for a NARP-based operating system.

5.1 Architecture of the OS

The basic primitive of the system being message-passing, the system looks a lot like a micro-kernel. Only the message format has a complex semantic and the communication layer is not really "simple". Furthermore, the system has device drivers, file system and networking running as kernel-mode processes, making the kernel more monolithic (but still having a micro-kernel spirit). It should be easy to make any user mode process run as a kernel mode process instead, for the sake of performance (eg : graphical server & compositor).

The kernel land is divided in three major parts, with strict dependency order:

- Level 0 : System ressource managment : physical memory, virtual memory, hardware interaction (IRQ, v86), debug output
- Level 1 : Scheduler, IPC & NARP core server : builds on top of level 0, adds support for processes and communication between them restricted to NARP protocol data.
- Level 2 : System processes : hardware, file systems, network, ... (may access level 0 and level 1 features)

User processes are restricted to syscalls that call level 1 primitives.

Here are a few basic principles for the design of these three levels :

- Level 2 processes may not communicate directly nor share memory : they must go through level 0 and level 1 primitives to achieve such a goal. Each level 2 process has a separate heap, which is completely freed when the process dies. Level 2 processes do not use separate virtual memory spaces : since the kernel memory space is mapped in all page directories, a level 2 process may run with any page directory.

  Benefits : critical system parts are restricted to level 0 and level 1. Level 2 components may leak or crash with less consequences.

- All synchronization & locking is handle by level 1, except for level 0 that must implement its own locking devices (since it cannot rely on level 1).

  Benefits : no complex synchronization in most of the code (which is either level 2 or user-land), only simple message passing and waiting for stuff to happen

- No concept of "threads" : system processes are actually kernel threads, but we call them processes since they use separate parts of memory. Userlands processes cannot spawn multiple threads of execution either : they must fork and communicate through NARP if they want to do so (eg: launching an expensive communication in the background).

  (since fork is a complicated system call, and features such as copy-on-write depend on processes using different paging directories, the fork system call is accessible only to userland processes : level 2 processes may not fork, but only create new processes)

- Level 1 also has a memory heap ; it is used with `core_malloc/core_free`. Level 2 proceses use standard `malloc/free`, which are modified to act on the heap of the current process.

- Each process (system or user) has a *mailbox*, ie a queue of incoming NARP messages waiting to be transferred. The mailbox has a maximum *size* (buffer size), and a *send* call may fail with a *no space left in queue* error. This is the only possible failure for a *send* call.

  System processes (level 2) spend most of their time in *waiting mode* ; they may be waked up by either recieving a NARP messsage or by a hardware event. Therefore the *wait_for_event* function that composes the main loop may return either : *a message was recieved* or *a system event happenned*. If the reason is *a message was recieved*, the process is free not to read the message immediately.

  On the other hand, user processes can wait for only one thing : recieving a NARP message. Each user process has a *message zone* in its memory space, and the *wait for message* function just copies the first message of the mailbox into this zone (overwriting whatever was there before) and returns control to the process (returning the length of the message).

- Handling of IRQs : some hardware stuff requires action as soon as the interrupt is fired, therefore a specifi IRQ handler may be used. Such a handler must do as little as possible, and when it is done signal level 1 that an IRQ has happenned (it may add specific data to the "IRQ happenned" message). Level 1 adds a message to the queue of the recipient process (if there is one) and returns immediately : the IRQ handler must leave as soon as possible. An IRQ is handled on whatever stack is currently used, and the IF flag is constantly off while the IRQ handler is running. The timer IRQ is the only one that behaves differently, since it has to trigger a task switch.

5.2 Steps of the developpment of the OS

1. Develop level 0 completely and with cleanest possible design

2. Develop level 1 with only basic funcionnality

3. Develop some basic applications in level 2 : display, keyboard, mini kernel shell, mini file system, ...

4. Improve level 1 with more complex stuff ; try to quickly attain a complete level 1

5. Work on the rest of the stuff