

UniK olsrd plugin implementation HOWTO

Version 0.3
for olsrd 0.4.3 and up
by [Andreas Tønnesen](#)

|Abstract

The [Optimized Link-State Routing protocol](#) (RFC3626) is a table-driven pro-active routing protocol for mobile multi-hop ad-hoc networks(MANETs). OLSR uses a optimization called *Multipoint Relaying* to flood packets throughout the network. All control traffic in OLSR is transmitted by broadcast(or multicast) on port 698 using the *User Datagram Protocol(UDP)*.

OLSR provides a default forwarding algorithm that allows for forwarding of OLSR messages of unknown types. A user may want to use the optimized flooding technique in OLSR to flood certain information, routing related or not, to all nodes that knows how to handle this message. This document does not try to explain the MPR functioning of the OLSR protocol. If you are not familiar with the MPR scheme you should read up on it!

The [UniK OLSR implementation](#) allows for use of dynamically linked plugins which are able to access the necessary functionality in olsrd to process and generate packets and to maintain communication sessions of its own. Plugins can be used for close to everything as the plugin interface also offers a multi-purpose call to perform specialized tasks.

This document describes the plugin-interface of UniK olsrd and walks the user trough the implementation of a simple plugin under a GNU/Linux system.

Introduction

|The UniK OLSR daemon

The UniK OLSR daemon is a implementation of the OLSR protocol for GNU/Linux systems by Andreas Tønnesen.

UniK OLSRd has a homepage at www.olsr.org. Form this page the latest source code can be downloaded.

Throughout this HOWTO it is assumed that the reader has access to the source-code of UniK olsrd >0.4.0 and is familiar with the C programming language.

Cross-referenced and commented code created with [doxygen](#) is available at the UniK olsrd website.

|Why add custom packages?

OLSR provides a *default forwarding algorithm* that allows for forwarding of OLSR messages of unknown types. This is really neat - because it means that even if only a subset of the nodes in the network actually known how to interpret a certain message-type - all nodes will forward them according to the MPR pragma.

A user may want to use the optimized flooding technique in OLSR to broadcast certain information, routing related or not, to all nodes that knows how to handle this message. Services that needs to broadcast/multicast data can encapsulate data in a private OLSR message-type using a olsrd plug-in.

|Why add custom functionality?

The design of the various entities of OLSR allows one to easily add special functionality into most aspects of OLSR. One can both register functions and unregister them with the socket parser, packet parser, scheduler and HNA set. This opens up for possibilities like intercepting current operation and replacing it with custom actions.

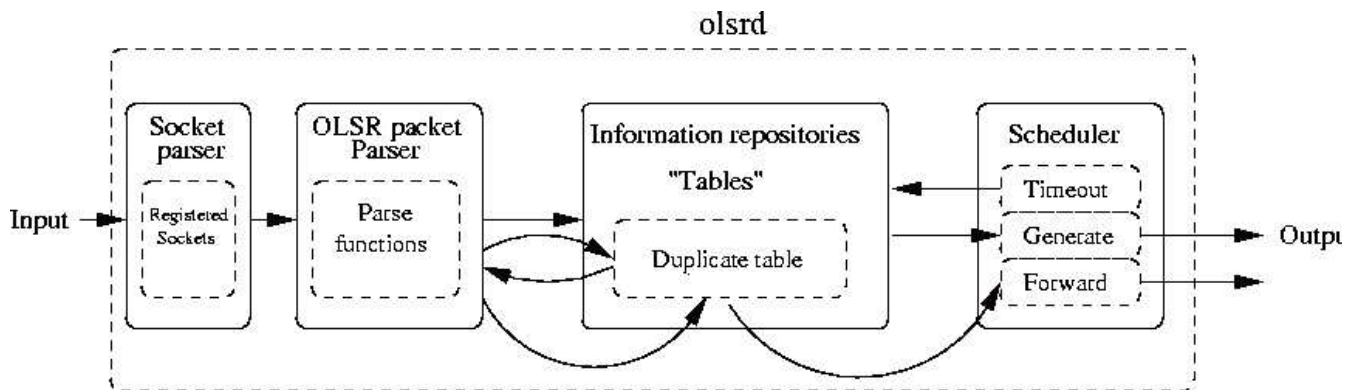
|Why plug-ins?

As of version 0.4.3 UniK olsrd will supports dynamic loading of plug-ins(dynamically linked libraries) for generation and processing of private package-types. This design is chosen for, amongst others, the following reasons:

- No need to change any code in the OLSR daemon to add custom packages or functionality.
- Users are free to implement olsrd plugins and licence them under whatever terms they like.
- If you, unlike yours truly, don't love C, the plug-ins can be written in any language that can be compiled as a dynamic library. Linux even allow scripts!
- No need for people using extended OLSR functioning to rely on heavy patching to maintain functionality when new olsrd versions are released.

|How does it work?

The basic parts of olsrd can be outlined as:



A very short description of the different components:

- **Socket parser** - The socket-parser checks for incoming traffic using a typical select(2) loop. It then calls the function associated with the sockets that has incoming data.
- **Packet parser** - The packet parser receives all incoming OLSR traffic. Even though the parser has several responsibilities it basically has three options when it comes to treating a message.
 - Discard the message. This is done if the message is found to be invalid.
 - Pass the message on to a message parsing function. A message parser for the message type must be registered.
 - Forward the message according to the default forwarding algorithm. This is done if the message is valid but the parser has no message parser registered for this message-type.

The message parsers are responsible for updating the information repositories needed if the information is considered fresh enough and has not already been processed.

- **Information repositories** - The "tables" are the heart of a table driven routing protocol. Here "fresh" information is kept and all calculations of routes and packets are done based on these repositories.

OLSR keeps the information needed to at minimum describe the current state of the network and this nodes immediate links in these tables.

The various packet parsing functions both updates these tables and relies on information in these tables to be able to process the messages.

The forwarding functioning in particular relies on the duplicate table which is a cache of all recent processed and/or forwarded packets.

All these tables are regularly "timed out". That means that entries no longer considered valid are removed.

Whenever these databases are updated in a way that changes the understanding of the network topology the routes are recalculated.

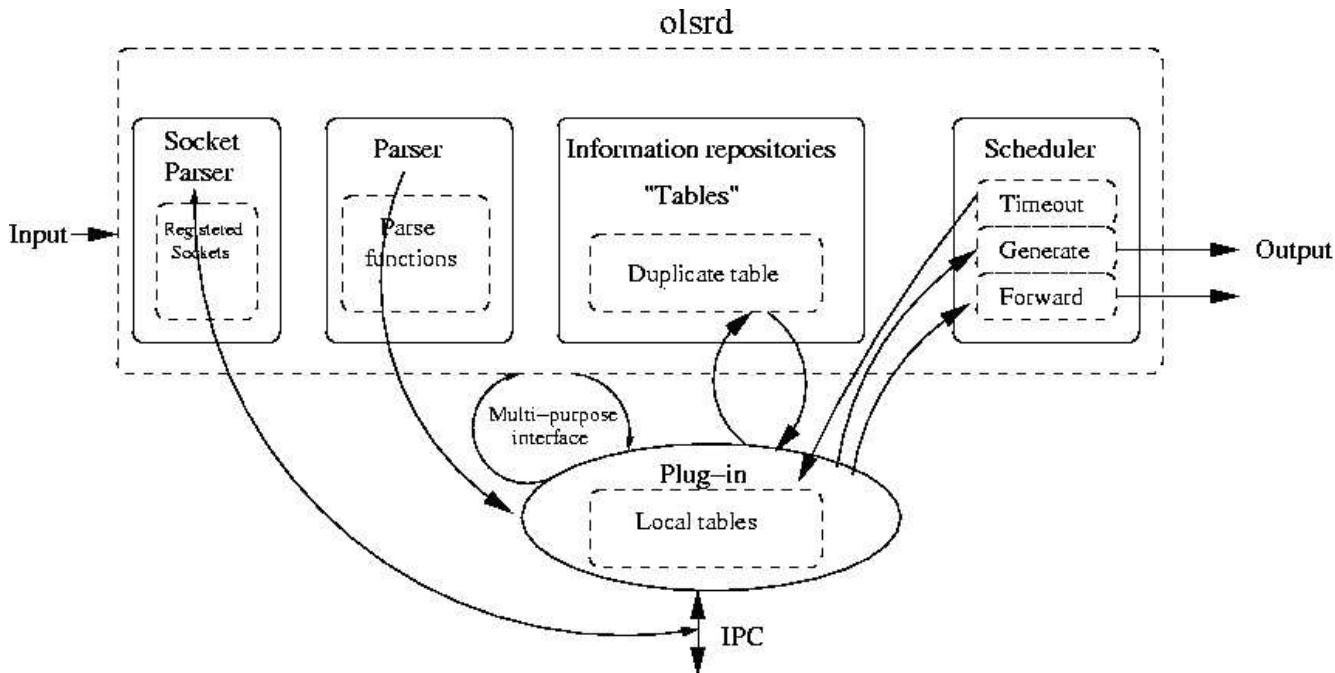
- **Event scheduler** - The event scheduler runs different events at different intervals. When one

wishes to transmit a message at a given interval one must register a packet generation entity with the scheduler.

Timing out of tables is also triggered by the scheduler. If one wishes to maintain a information repository that should be timed out on a regular basis then one must register a timeout entity with the scheduler.

The scheduler runs in a thread of its own and share memory is protected using a *mutex* so that the packet creation entity does not have to consider synchronisation and memory-protection.

The role of a plug-in will be:



Any of the relations on the figure can be omitted. This is just to show all possibilities. A brief explanation:

- The plugin can register and remove sockets and their corresponding functions with the socket parser. Useful for IPC.
- The plugin can register a parse function for a message-type with the parser
- The plugin can register a timeout(for table maintenance) with the scheduler.
- The plugin can register a timed event(like packet generation) with the scheduler
- The plugin has access to the OLSR duplicate table to check if messages have been processed and/or forwarded before.
- The plugin has access to the transmit function in OLSR

All these functions are made available to the plugin through a *multi purpose interface* which olsrd makes available to plugins. Through this interface virtually all

olsrd functions and data can be made available to plugins.

Example case

Source code

The source code for the example used in this HOWTO is contained in the UniK olsrd source package available at www.olsr.org.

After downloading the tarball do:

```
#tar jxvf uolsrd-0.x.x.tar.bz2
```

And the source will be unpacked into `./unik-olsrd-0.x.x`.

Then you will find the source for the example plugin in `./unik-olsrd-0.x.x/lib/power/`

(All x'es are of cause replaced with the actual version number)

As of now two plugins are included in the olsrd source tarball. The first one is the power-status plugin which is the example we will look into in this howto. This plugin uses olsrd to flood information. The second plugin is a **dynamic gateway discovery** plugin. This plugin does not use olsrd to transmit any traffic - but it updates the local HNA repository of the olsr daemon dynamically according to local Internet routes. This plugin should also be interesting reading to anyone planning on implementing a olsrd plugin.

|The powerstatus plugin

Notice that the powerplugin code is likely to be updated from time to time. These updates will not necessary be reflected here!

We'll consider this simple example case throughout the rest of this HOWTO.

We are interested in a solution where the powerstatus of nodes in the MANET can be registered. This information should be available through IPC using a TCP socket on port 8888 only allowing local connections (127.0.0.1). This solution should not affect nodes that have not enabled this functionality. We'll assume having access to a MANET where a subset of nodes are power-status enabled.

A node is to periodically flood the network with a custom packet containing the following information:

- Whether or not the node is battery powered
- Estimated usage time left on the battery if using a battery
- percentage of power left on the battery, if battery-powered

So what we need to implement is:

- A message format for the power-info.
- A information repository in which we store the info.
- Periodic generation of this message based on our own powerstatus.
- Parsing of this message and updating of the repository
- Timeout of the repository
- Forwarding of the message (nodes without support for this message-type will forward it automatically)
- IPC functionality

We decide that this message is to be transmitted every 2.5 seconds and that the information should be considered valid for 7.5 seconds.

We also decide that the IPC interface should just be a stream of characters if a connection is established. This way an application like *telnet* can be used to read the information.

This example-case is in no way intended to be a real-life solution! We'll consider it as an example for the sake of the example!

|Example source code files

We'll take a quick step through the files in the powerstatus plugin:

Makefile

This is the Makefile used by the **make** command. Take a look at it while reading the *Dynamically linked libraries* section.

src/olsrd_plugin_io.h

This file contains all the definitions of the commands that are available to interface with olsrd from the plugin.

src/olsrd_plugin.h

This file contains all needed declarations to create a olsrd plugin. It should be altered where marked to fit your needs.

src/olsrd_plugin.c

This file contains source code for initializing the plugin and some common used helper functions. You should not have to alter much code in this file.

src/olsrd_power.h

This file contains definitions for our power plugin.

src/olsrd_power.c

This file contains the implementation of the powerstatus case.

Dynamically linked libraries

|What are they?

A dynamically loadable library(DLL) is a piece of executable data that contain functions and data. But unlike normal executables DLLs are not "fully" linked after compilation. They are set up in a way that allows the actual linking to take place at runtime. A application can load and run functions from a DLL - ergo, the library is linked at runtime.

DLLs can be used simoultaneously by multiple processes. Still only one instance of the library is maintained in memory.

This is however not something we'll take advantage of - we'll consider a DLL as a **plugin**. A plugin provides new functions to an existing application without altering the original application. DLLs exist for all common operating system. In Linux they are known as `so` files while in Microsoft Windows they are known as `DLL` files.

Linux allows for DLLs to be written even as scripts! But in this HOWTO we'll implement a plugin in good ol' C.

|How do I create them?

Like many other things in life - creating a dynamically linked library is a rather trivial task *as long as one knows how to do it ;-)*

The main difference form writing you normal application is:

- There is no `main` function
- Linking of object-files need to be handled different
- One should add special `constructor` and `destructor` methods

Let's start with the first and last points. Instead of the normal `main(int, char **)` entry point in you code plugins uses a constructor function called when the plugin is loaded, and a destructor called when the plugin is unloaded.

Remember that a plugin should not necessaryli have a program flow like normal applications - so don't think of the constructor as a `main` function but rather as a easy way to set up the necessary elements for operation and communication with the caller.

In older days these functions were declared different(`_init` and `_fini`) - but as we don't live in the past we'll only consider the modern way of doing this. The functions are prototyped as:

```
void __attribute__((constructor))
my_init(void);
```

```
void __attribute__((destructor))
my_fini(void);
```

The `__attribute__((constructor))` and `__attribute__((destructor))` tells gcc that these functions should be used as constructor and destructor.

Object-files should be compiled using the **-fPIC** switch to create position independent code. e.g:

```
gcc -fPIC -c -o object.o source.c
```

Linking of object-files should be done on the following form:

```
gcc -shared -Wl,-soname,mylibrary.so -o mylibrary.so \
```

obj1.o obj2.o -lc

`-shared` tells gcc that this should be a shared library, `-Wl,-soname,mylibrary.so` sets the (internal) name of the library and `-lc` links the library with the standard C library.

After you have installed a new dynamically linked library in one of the predefined paths you should run **ldconfig** for the system to update the registered libraries! In the example code this is done in the makefile.

Olsrd plugins

| Making olsrd load a plugin

To make olsrd load a plugin add a entry of the type:

```
LOAD_PLUGIN    libname
```

to the `/etc/olsrd.config` file (or whatever config file you use).

If you don't provide a absolute path to the library the loader will try the following locations:

- A colon-separated list of directories in the users `LD_LIBRARY_PATH` environment variable.
- The list of libraries cached in `/etc/ld.so.cache`.
- `/lib`, followed by `/usr/lib`.

In our example the plugin is installed into `/usr/lib` so there is no need to add a complete path.

| Functions and datatypes

We'll start out with a look at the various olsrd specific functions and datatypes used in olsrd which you should also use in your plugins.

All the datatypes are defined in `src/uolsr_plugin.h`. The functions are either implemented in `src/uolsr_power.c` or pointers to the functions in olsrd are imported.

olsrd specific datatypes

Olsr uses the following type-definitions:

```
typedef u_int8_t    olsr_u8_t;
typedef u_int16_t   olsr_u16_t;
typedef u_int32_t   olsr_u32_t;
typedef int8_t      olsr_8_t;
typedef int16_t     olsr_16_t;
typedef int32_t     olsr_32_t;
```

These datatypes will be used throughout this document.

olsrd IP addresses

To be able to handle both 32-bit IPv4 addresses and 128-bit IPv6 addresses olsrd uses a union defined as:

```
union olsr_ip_addr
{
    olsr_u32_t v4;
    struct in6_addr v6;
};
```

This datatype should be used whenever dealing with IP addresses in olsrd plugins. The variable **ipsize** contains the size of the addresses used. So whenever copying or comparing IP addresses in olsrd you should do:

```
/* compare */
memcmp(first_addr, second_addr, ipsize);
/* copy */
memcpy(first_addr, second_addr, ipsize);
```

Where `first_addr` and `second_addr` are pointers to a `olsr_ip_addr` struct.

olsrd hashing

Olsrd provides some hashing functions that will create a 4 bit hash (represented as a 32 bit datatype) from a union `olsr_ip_address`. The function is defined as:

```
olsr_u32_t
olsr_hashing(union olsr_ip_addr *);
```

It returns a 32 bit value but only the lower 4 bits are actually used. The maximal hash-value is defined in the symbol `HASHSIZE`. It is currently 16. This function is implemented in `src/olsrd_plugin.c`.

olsr_ip_to_string

This function converts a IP address of type union `olsr_ip_addr` to a human readable string. Very useful for debugging output.

```
char *
olsr_ip_to_string(union olsr_ip_addr *);
```

OBS: it returns a pointer to a statically allocated buffer - and is not reentrant! If you are to print two ip addresses you **MUST** do it in two separate instructions(eg. not in the same `printf(3)` call)! This function is implemented in `src/olsrd_plugin.c`.

Mantissa exponent functions

The `vtime` and `htime` fields of a OLSR message is to be calculated using a exponent/mantissa scheme described in the RFC. These calculations can be done using two functions:

```
olsr_u8_t
double_to_me(double);

double
me_to_double(olsr_u8_t);
```

The first one converts a double to a mantissa/exponent value(8 bits). The second one does the opposite. these functions are imported from olsrd in `olsrd_plugin.c`.

timer functions

Most entries in all tables in olsrd keep a value representing for how long this entry is to be considered valid. The datatype used is struct `timeval`(defined in `sys/time.h`) and olsrd offers some functions for working on `timvals`:

```
int
olsr_timed_out(struct timeval *);
```

Function that checks if the time-data contained in a struct `timeval` is in the future. It returns positive if the `timeval` you passed is not in the future(that means it is "timed out"). This function is implemented in `src/olsrd_plugin.c`.

```
void
olsr_init_timer(olsr_u32_t, struct timeval *);
```

function that fills a provided structure of type `timeval` with data representing the amount of milliseconds provided in the first argument. **OBS** note that this function does NOT create a timestamp for the future. This function is implemented in `src/olsrd_plugin.c`.

```
void
olsr_get_timestamp(olsr_u32_t, struct timeval *);
```

Function fills a provided `timeval` structure with a timestamp of current time + the number of milliseconds provided by the first argument. This function is implemented in `src/olsrd_plugin.c`.

Sequencenumber function

All OLSR messages carries a "unique"(using wraparound) sequence number. To obtain such a `seqno` olsrd offers the following function:

```
olsr_u16_t
get_msg_seqno();
```

This returns a 16-bit `seqno` incremented by 1 from the last delivered `seqno`.

This function **MUST** be imported from olsrd and one has to use one global Sequencenumber for all OLSR traffic!

| Interfacing with olsrd

From the plugins point of view

We have the following requirements for what functionality our example plugin must be able to access in olsrd:

- Register a socket with the socket parser
- Register a message parsing function with the parser
- Register a message generation function with the scheduler
- Register a table timeout function with the scheduler
- Send packets
- Access the duplicate table
- Get a valid Sequencenumber
- Access the default forwarding function
- Access the link status query function

Your plugin might need other functions.

Retrieving data from olsrd

Two pointers to olsrd data is set up automatically by the olsrd plugin loader:

union olsr_ip_addr *main_addr

A struct `olsr_ip_addr` describing this nodes main address.

int ipversion

The *Internet Protocol* version used by olsrd. Either `AF_INET` or `AF_INET6`.

The plugin-loader will also automatically set up this function in the plugin:

int (*olsr_plugin_io)(int, void *, size_t)

This is the function from which nearly all other data and functions are retrieved from olsrd! The function takes a "command" argument. These commands are defined in `olsr_plugin_io.h`. The second argument is used for data exchange while the third argument describes the size of the second argument.

In `src/olsrd_plugin.c` we find all the retrieving of variables and functions in the function `int fetch_olsrd_data()` which is called from the `int register_olsr_data(struct olsr_plugin_data *)` function in the same file, which again is automatically invoked by the olsrd plugin loader.

Here are some samples from the `int fetch_olsrd_data()` function:

```
int
fetch_olsrd_data()
{
    int retval = 1;

    /* Packet buffer */
    if(!olsr_plugin_io(GETD__PACKET, &buffer, sizeof(buffer)))
    {
        buffer = NULL;
        retval = 0;
    }

    /* Packet buffer size */
    if(!olsr_plugin_io(GETD__OUTPUTSIZE, &outputsize, sizeof(outputsize)))
    {
        outputsize = NULL;
        retval = 0;
    }

    .....

    /* Parser registration */
    if(!olsr_plugin_io(GETF__OLSR_PARSER_ADD_FUNCTION,
        &olsr_parser_add_function,
        sizeof(olsr_parser_add_function)))
    {
        olsr_parser_add_function = NULL;
    }
}
```

```

    retval = 0;
}

/* Scheduler timeout registration */
if(!olsr_plugin_io(GETF__OLSR_REGISTER_TIMEOUT_FUNCTION,
    &olsr_register_timeout_function,
    sizeof(olsr_register_timeout_function)))
{
    olsr_register_timeout_function = NULL;
    retval = 0;
}

/* Scheduler event registration */
if(!olsr_plugin_io(GETF__OLSR_REGISTER_SCHEDULER_EVENT,
    &olsr_register_scheduler_event,
    sizeof(olsr_register_scheduler_event)))
{
    olsr_register_scheduler_event = NULL;
    retval = 0;
}

.....

return retval;
}

```

Here we can see examples of both how to fetch pointers to variables and pointers to functions.

Here are the functions imported from olsrd in our example plugin:

void (*olsr_parser_add_function)(void (*)(union olsr_message *, struct interface *, union olsr_ip_addr *), int, int)

This function adds a parser function to the olsrd packet parser. This function should be on the form: *void function(union olsr_message *, struct interface *, union olsr_ip_addr *)*. This function is called every time a message of type *type* is received. If *type* is set to PROMISCIIOUS then all messages are passed to this function. The last argument says if this function is responsible for forwarding the message or not. If set to 1 the packet parser takes no more action on this message after passing it to the parser function. If set to 0 the packet parser will forward this message (if no other parser function claims to forward it) using the default forwarding algorithm.

int (*olsr_register_timeout_function)(void (*)())

This function registers a function with the olsrd scheduler. The registered function is called *every time the scheduler polls*. That means 10 times every second by default. This is useful for timing out information repositories.

int (*olsr_register_scheduler_event)(void (*)(), float, float, olsr_u8_t *)

This function registers a function with the olsrd scheduler to be called on a given interval. The interval is given as the second argument. The third argument sets the initial time value - that is if one wishes the first occurrence of the event to happen before the interval has passed. The fourth argument sets a "trigger" if this trigger is set to 1 the event-function is called immediately (at the next scheduler poll).

int(*net_output)(struct interface *)

Function that sends Note that in addition to functions - different data is also shared between the plugin and olsrd.

int(*check_dup_proc)(union olsr_ip_addr *, olsr_u16_t)

Function that checks if a given OLSR message has been processed by this node before. This implies checking and in necessary, updating the duplicate table. Does not check if the message has been forwarded.

int(*default_fwd)(union olsr_message *, union olsr_ip_addr *, olsr_u16_t, struct interface *, union olsr_ip_addr *)

Function that adds a message to the forward buffer IF this message has not been forwarded before. This implies checking and in necessary, updating the duplicate table. Once a message has been added to the forward buffer you need not take any further action for it to be forwarded.

void(*add_olsr_socket)(int, void(*) (int))

Function that adds a socket-filedescriptor to the socket parser in uolsrd. The function given as the second argument is called every time data is available at the socket(with the socket as argument).

int(*remove_olsr_socket)(int, void(*) (int))

Function that removes a socket-filedescriptor from the socket parser in uolsrd.

olsr_u16_t get_msg_seqno()

Function that returns a valid Sequencenumber for this node.

int (*chk_neigh_link)(union olsr_ip_addr *)

Returns the (best)link status of a link to a given neighbor. In general messages should only be processed if this function returns SYM which indicates a symmetric(bi-directional) link.

In addition to these offered functions pointers to the following data is set up:

char *packet_buffer

The global buffer for packets to be transmitted by olsrd.

int *bufferize

The size of the package contained in the packet_buffer **including the OLSR packet header**

struct interface *ifs

A pointer to the linked list of interfaces olsrd runs on.

struct timeval *sched_now

This is olsrds current idea of time as set by the scheduler.

Be ware that these are pointers to the actual data used by olsrd. NEVER alter them(except packet_buffer and bufferize) unless you are 100% certain of what you are doing!!

From olsrds point of view

As olsrd plugins are meant to be as independent as possible from the olsrd code - not many requirements are laid out from olsrd except the initialization functions. The following functions MUST be present in all olsrd plugins:

void register_olsr_data(struct olsr_plugin_data *)

This function MUST BE PRESENT in all olsrd plugin. It is implemented in *src/uolsr_plugin.c* in the example code and should not need to be altered.
This functions sets up the *olsr_plugin_io* function from olsrd.

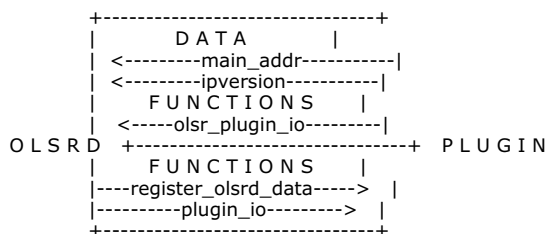
int *plugin_io(int, void *, size_t)

Like the *olsr_plugin_io* function available to the plugin, this is a multiple-purpose function available to olsrd if other functionality than the predefined functions is desired.
It has the same *ioctl(2)*-like syntax.

olsrd plugin interface summary

To be able to communicate with olsrd a standard interface for communication has been defined as seen in the previous section. ALL olsrd plugins must respect this interface.

This interface can be depicted as:



All other access can be set up through this interface.

A plugin framework

The example code provided with UniK olsrd can be considered not just an example - but a framework for writing olsrd plugins.

|What should not be altered?

The file `src/uolsr_plugin.c` implements all helper functions and initialization functions. You should alter the `int fetch_olsrd_data()` to set up the pointers you need. Besides from that this code should not need to be altered any further.

All powerstatus-plugin-specific code resides in `src/uolsr_power.c`.

The header-file `src/uolsr_power.h` contains powerstatus-plugin-specific declarations while `src/uolsr_plugin.h` contains plugin-general definitions. `src/uolsr_plugin.h` should however be modified by the developer. In `src/uolsr_plugin.h` one should possibly add the packet definitions, the version and name of the plugin, data like message type and function and data pointers to olsrd data.

|Constructor and destructor

The constructor and destructor of the code resides in `src/uolsr_plugin.c`. These functions call the functions `void olsr_plugin_init()` and `void olsr_plugin_exit()` that you MUST implement in your plugin code. They are already defined in `src/uolsr_plugin.h`. All initialization specific to your plugin should be done in these functions.

All IPC initialization should be done in the function `void plugin_ipc_init()` that you must provide in your plugin.

So you must implement the following functions:

- `void olsr_plugin_init()`
- `void olsr_plugin_exit()`
- `void plugin_ipc_init()`

Implementing the powerstatus case

The rest of this document will be a step-by-step walk-through on implementation of the powerstatus case leading up to the code example included with UniK olsrd.

The previous sections should be used as reference material.

Defining a message

|OLSR message types

OLSR uses a 8 bit integer to identify packet types which gives

256 possible different packet types

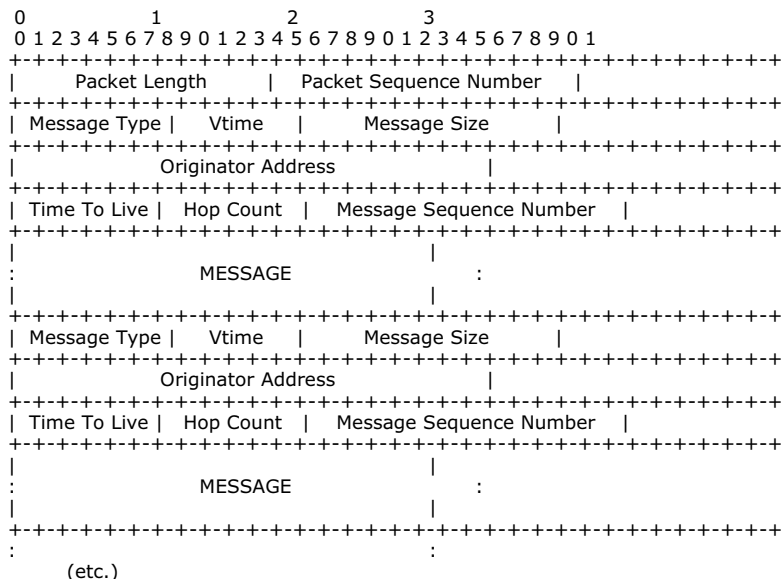
totally. Types 0-127 are reserved by OLSR while 128-255 are available for custom packet types. OLSR uses a *default forwarding algorithm* on unknown packet-types (and possibly on known packet-types as well), which makes nodes transmitting custom packets not breaking compability - and nodes not configured to process these packets forward them according to the MPR scheme.

Default forwarding of unknown message-types using MPR flooding makes the design of custom messages containing various information to be diffused into the network rather easy.

Designing a custom message

OLSR packages and messages

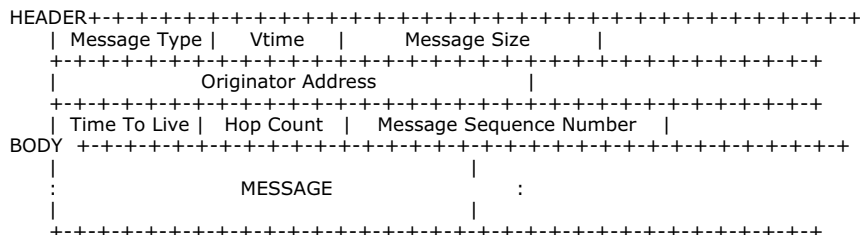
All OLSR traffic is transmitted in OLSR packets. They all use the same generic OLSR header. A OLSR packet can contain several OLSR messages. The basic for of the OLSR packet is:



OLSR defines 4 types of packages as of yet:

- **TYPE 1 - HELLO** A message used for neighbor and link detection, MPR calculation and other things. This message is NOT flooded.
- **TYPE 2 - TC** The Topology Control is flooded throughout the network. In the Tc message a node announces a subset of its link set.
- **TYPE 3 - MID** The Multiple Interface Declaration message is used to announce that a node runs OLSR on multiple interfaces. This message is flooded.
- **TYPE 4 - HNA** The Host and Network Association message is used to announce connectivity to external networks. This message is flooded.

A OLSR message has the following form



Header fields:

- **Message Type** -Type of the message. For custom messages the space 127-225 can be used.
- **Vtime** - The time the information in this message should be considered valid. This value is computed using a exponent/mantissa function as described later.
- **Message Size** -Size of this message, including message header, in bytes.
- **Originator Address** -IP address of the originator of this message.
- **Time To Live** -A value describing how many hop this message should be forwarded. This field is decremented by 1 every time the message is forwarded.

- **Hop Count** - Incremented by 1 every time this message is forwarded.
- **Message Sequence Number** - The sequence-number of this message. Should be incremented by 1 every time a node generates a message.

The body part is defined by the designer.

In our plugin framework this data-structure(the OLSR message) is defined in the header-file *uolsr_plugin.h* as(IPv4 version):

```
struct olsrmsg
{
  olsr_u8_t   olsr_msgtype;
  olsr_u8_t   olsr_vtime;
  olsr_u16_t  olsr_msgsize;
  olsr_u32_t  originator;
  olsr_u8_t   ttl;
  olsr_u8_t   hopcnt;
  olsr_u16_t  seqno;

  union
  {
    struct hellormsg hello;
    struct tcmsg tc;
    struct hnamsg hna;
    struct midmsg mid;
  } message;
};
```

The *uolsr_plugin.h* header-file should be studied as it contains all OLSR standard packet definitions!

Looking at the *olsrmsg* struct we recognize the fields from the packet-format we looked at earlier. The actual message body is contained in the *message* union. If you are not familiar with the union keyword you should consult a C programming manual.

The powerinfo message

What we need to do first is to design the format of the message body of our powerstatus message. We go for the following:

```
BODY +-----+
      | Powersource | percentage | Remaining Usage Time |
      +-----+-----+-----+
```

The fields are:

- **Powersource** - set to 1 if the originator-node is running on battery power and 0 if it's running on AC power.
- **Percentage** - A integer in the 1-100 space if Powersource == 1, set to 0 if powersource == 0.
- **Remaining Usage Time**- the calculated time left of operation on this node based on the powersource status in minutes. Set to 0 if Powersource == 0.

To make this as easy as possible we'll just add our datatypes to the *uolsr_plugin.h* header file.

First we add our message-type to the header-file. We decide that the powerstatus message is of **type 128**. So we add the following to the *uolsr_plugin* header-file:

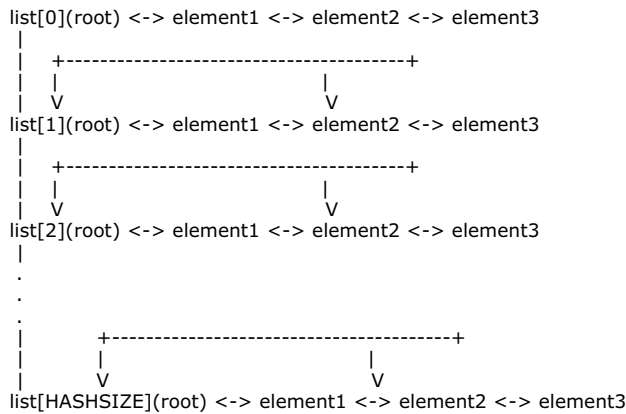
```
/* The type of message you will use */
#define MESSAGE_TYPE 128
```

While we're at it we add the name and version of the plugin as well:

```
#define PLUGIN_NAME "OLSRD Powerstatus plugin"
#define PLUGIN_VERSION "0.1"
#define PLUGIN_AUTHOR "Andreas Tønnesen"
#define MOD_DESC PLUGIN_NAME " by " PLUGIN_AUTHOR
#define PLUGIN_INTERFACE_VERSION 1
```

Next we add the struct describing the message body of the powerstatus message. We add the following to the " *DEFINE YOUR CUSTOM PACKET HERE* " section in the header-file:

```
struct powermsg
{
  olsr_u8_t   source_type;
  olsr_u8_t   percentage;
  olsr_u16_t  time_left;
};
```

We can see that to find a entry we must first find the hash to be able to access the correct list-array element. Than we need to traverse that list.

In our example we have created two files called `uolsr_power.c` and `uolsr_power.h`. In the header-file we define the list entries needed and in the c file we implement the functions needed.

Lets start with the header-file:

```

#include "uolsrd_plugin.h" /* All plugins must include this header-file */

/* Database entry */

struct pwentry
{
    union olsr_ip_addr originator; /* IP address of the node this entry describes */
    olsr_u8_t      source_type; /* Powersource of the node */
    olsr_u8_t      percentage; /* Percentage of battery power left */
    olsr_u16_t     time_left; /* Operation time left on battery */
    struct timeval timer; /* Validity time */
    struct pwentry *next; /* Next element in line */
    struct pwentry *prev; /* Previous element in line */
};

/* The database root elements - (using hashing) */
struct pwentry list[HASHSIZE];

```

Before starting to use the hashed list one MUST initialize the pointers. In `uolsr_power.c` we add the following to the `olsr_plugin_init` function(with `i` declared as a integer):

```

/* Init list */
for(i = 0; i < HASHSIZE; i++)
{
    list[i].next = &list[i];
    list[i].prev = &list[i];
}

```

What is done here is setting all the root entries to point to them selves - both ways. Thus the lists now only contains the root elements.

Working on the lists

To insert a entry into a list you can do:

```

/* You have some pointer to a struct powerset_entry
called new_entry */

int hash;

/* Get the hash */
hash = olsr_hashing(&new_entry->main_address);

/* Insert the entry into the correct hashlist */

list[hash].next->prev = new_entry;
new_entry->next = list[hash].next;
list[hash].next = new_entry;
new_entry->prev = &list[hash];

```

OBS!! Notice that the root element(`list[hash]`) is NOT a pointer so this has to be dereferenced!

To remove an entry you can do:

```
/* You have some pointer to a struct powerset_entry
   called del_entry */

/* Dequeue */
del_entry->next->prev = del_entry->prev;
del_entry->prev->next = del_entry->next;

/* Delete */
free(del_entry);
```

Later we'll see how these techniques are used in the parser and timeout code.

Registering a timeout function with the scheduler

To keep information in our database "fresh" we create a function that traverses the lists and removes timed out entries. We need to register this function with the scheduler and to do this our function has to be of a type that the scheduler accepts which are functions of type `void` taking no arguments. We create a function `void olsr_timeout()`.

The function-pointer `olsr_register_timeout_function` has been set up in `olsrd_plugin.c` and we can use this to register our timeout function:

```
olsr_register_timeout_function(&olsr_timeout);
```

We do this in the init function `olsr_plugin_init`

Ok - now it's time to implement the actual function. We add the following to `uolsr_power.c`

```
/**
 *A timeoutfunction called every time
 *the scheduler is polled
 */
void
olsr_timeout()
{
    struct pwentry *tmp_list;
    struct pwentry *entry_to_delete;
    int index;

    for(index=0;index<HASHSIZE;index++)
    {
        tmp_list = list[index].next;
        /*Traverse MID list*/
        while(tmp_list != &list[index])
        {
            /*Check if the entry is timed out*/
            if(olsr_timed_out(&tmp_list->timer))
            {
                entry_to_delete = tmp_list;
                tmp_list = tmp_list->next;
                printf("POWER info for %s timed out.. deleting it\n",
                    olsr_ip_to_string(&entry_to_delete->originator));
                /* Dequeue */
                entry_to_delete->prev->next = entry_to_delete->next;
                entry_to_delete->next->prev = entry_to_delete->prev;

                /* Delete */
                free(entry_to_delete);
            }
            else
                tmp_list = tmp_list->next;
        }
    }
    return;
}
```

This should not need to much explanation - but check out the calls to the handy helper functions `olsr_timed_out` and `olsr_ip_to_string`. Read up on what they do and get familiar with them!

Parsing messages

Overview

We now need to create functionality to parse a received powerstatus message. We need to:

- Register the parsing function and message type with the parser
- Update the powerstatus database based on the information in the message if necessary
- If necessary forward the message

Implementing a parser function

To be able to register a function with the olsrd packet parser it must be of the type void and take the following arguments: taking the following arguments: *union olsr_message **, *struct interface *in_if*, *union olsr_ip_addr *in_addr* .

To register the function to be called every time a message of a given type is received we do:

```
olsr_parser_add_function(&olsr_parser, PARSE_TYPE, 1);
```

We do this in the *olsr_plugin_init* function.

Now for the actual parsing function - I've chosen to comment it in-line:

```
void
olsr_parser(union olsr_message *m, struct interface *in_if, union olsr_ip_addr *in_addr)
{
    struct powermsg *message;
    union olsr_ip_addr originator;
    double vtime;

    /* Fetch the originator of the message */
    memcpy(&originator, &m->v4.originator, ipsize);

    /* Fetch the message based on IP version */
    if(ipversion == AF_INET)
    {
        message = &m->v4.msg;
        vtime = me_to_double(m->v4.olsr_vtime);
    }
    else
    {
        message = &m->v6.msg;
        vtime = me_to_double(m->v6.olsr_vtime);
    }

    /* Check if message originated from this node */
    if(memcmp(&originator, main_addr, ipsize) == 0)
        /* If so - back off */
        return;

    /* Check that the neighbor this message was received
       from is symmetric */
    if(check_neighbor_link(in_addr) != SYM_LINK)
    {
        /* If not symmetric - back off */
        printf("Received POWER from NON SYM neighbor %s\n", olsr_ip_to_string(in_addr));
        return;
    }

    /* Check if this message has been processed before
       * Remember that this also registers the message as
       * processed if necessary
       */
    if(!check_dup_proc(&originator,
                       ntohs(m->v4.seqno)) /* REMEMBER NTOHS!! */)
    {
        /* If so - do not process */
        goto forward;
    }

    /* Process */

    printf("POWER PLUG-IN: Processing PWR from %s seqno: %d\n",
           olsr_ip_to_string(&originator),
           ntohs(m->v4.seqno));
}
```

```

/* Call a function that updates the database entry */
update_power_entry(&originator, message, vtime);

forward:
/* Forward the message if necessary
 * default_fwd does all the work for us!
 */
default_fwd(m,
            &originator,
            ntohs(m->v4.seqno), /* IMPORTANT!!! */
            in_if,
            in_addr);
}

```

Ok - first of all pay attention to the *network byte order* converting! All data MUST be converted to network byte order when generating a packet(to ensure platform compability) and they have to be converted back when parsing the message!

The parser delivers data representing the message, the interface on which it arrived and the address of the last-hop sender(not the originator!). We use this information to do some checks:

- Check if the message originates from us. If so we discard it.
- Check if the message came from a asymmetric neighbor. If so we discard it.
- Then we check the duplicate set in olsrd to see if we have processed this message before. Note that this also updates the duplicate table if necessary.
- Last of all we forward the message by passing it to the `default_fwd` function provided by olsrd. This function does all the work for us. If the message has been forwarded before - no forwarding is done.

If you wish not to use the default forwarding algorithm you must define some forwarding functionality yourself. Before you get into that reconsider it for about 99 times ;-)

Also note the way the validity time is processed and the 8bit message vtime is converted into a double.

The `update_power_entry` function updates, if necessary, the entry in the database. Consult the `uolsr_power.c` file for the source.

And that's all there is to it!

Generating messages

Now we need to create functionality to generate powerstatus messages on a regular basis.

|Sequencenumbers

All handling of packet and message sequencenumbers SHOULD be handled by olsrd. The olsr-packet header seqno is set automagically by the transmission function while the message seqno should ALWAYS be set using the `get_msg_seqno` function available from olsrd.

|The outputbuffer and outputsize

UniK olsrd uses a global buffer for data that is to be sent. This buffer can be accessed trough the `union olsr_packet *msg` pointer declared in the `uolsr_plugin` header-file.

The size of the data in this buffer must be set in the variable `outputsize`. The size set here(in bytes) is the amount of bytes that will be transmitted from this node when calling `net_output` which transmits data. This means that this size must include ALL messages in the packet(we will only send one message pr packet) in addition to the OLSR packet header(4 bytes).

|Creating a message build function

We need to implement a function `olsr_event` (or whatever you want to call it) of type `void` taking no arguments that is to be registered with the olsrd scheduler.

The function `olsr_register_scheduler_event` has been made available for us from olsrd so all we need to do to register our generation function is:

```
olsr_register_scheduler_event(&olsr_event, 2.5, 0, NULL);
```

Now this function will be called every 2.5 seconds by the olsrd scheduler.

Now to implement the actual function in `src/uolsr_power.c` :

```
void
olsr_event()
{
    union olsr_packet *packet;
    union olsr_message *message;
    struct interface *ifn;

    /* If we can't produce power info we do nothing */
    if(!has_apm)
        return;

    printf("PLUG-IN: Generating package - ");

    /* Cast the char* buffer to the packetformat */
    packet = (union olsr_packet*)buffer;

    /* Fetch the message based on IPversion */
    if(ipversion == AF_INET)
        message = (union olsr_message *)packet->v4.olsr_msg;
    else
        message = (union olsr_message *)packet->v6.olsr_msg;

    /* looping trough interfaces */
    for (ifn = ifs; ifn ; ifn = ifn->int_next)
    {
        printf("[%s] ", ifn->int_name);
        /* Fill message */
        if(ipversion == AF_INET)
        {
            /* IPv4 */
            message->v4.olsr_msgtype = MESSAGE_TYPE;
            message->v4.olsr_vtime = double_to_me(7.5);
            message->v4.olsr_msgsize = htons(sizeof(struct olsrmsg));
            memcpy(&message->v4.originator, main_addr, ipsize);
            message->v4.ttl = MAX_TTL;
            message->v4.hopcnt = 0;
            message->v4.seqno = htons(get_msg_seqno());

            get_powerstatus(&message->v4.msg);

            *outputsize = sizeof(struct olsrmsg) + sizeof(olsr_u32_t);
            packet->v4.olsr_packlen = htons(*outputsize);
        }
        else
        {
            /* IPv6 */
            message->v6.olsr_msgtype = MESSAGE_TYPE;
            message->v6.olsr_vtime = double_to_me(7.5);
            message->v6.olsr_msgsize = htons(sizeof(struct olsrmsg));
            memcpy(&message->v6.originator, main_addr, ipsize);
            message->v6.ttl = MAX_TTL;
            message->v6.hopcnt = 0;
            message->v6.seqno = htons(get_msg_seqno());

            get_powerstatus(&message->v6.msg);

            *outputsize = sizeof(struct olsrmsg6) + sizeof(olsr_u32_t);
            packet->v6.olsr_packlen = htons(*outputsize);
        }

        /* Send data */
        net_output(ifn);
    }
    printf("\n");
    print_power_table();
    return;
}
```

We'll go trough this step-by-step:

- First we check if we are able to fetch and power information(if has_apm is set). has_apm is set

- at initialization.
- Then we make sure we can access the outputbuffer in a feasible way by casting a pointer to it to a *union olsr_packet**
- then we fetch a pointer to the message data area. This is actually the same address for IPv4 and IPv6, but as our code is poetry we do a check on the ipversion ;-)
- Now we loop trough all OLSR interfaces. For every interface we:
 - Fill a message based on the IPversion in use
 - Set the correct message type
 - Generate the 8bit validity time
 - Set the size of the message - remember network byte order!
 - Set the originator which MUST be our main address
 - Set time to live and hop-count
 - Get and set a sequence number. Again network byte order!
 - Call a function that reads powerinfo and fills it into the powerstatus message
 - Set the size of the complete package(including the olsr packet header)
 - Set the packetize in the olsr header
 - Send the data
- Finally we call a function that prints out some information.

Be sure you understand what happens in this function! Consult the previous sections and the *uolsr_plugin.** files for references. Thats it!!!

Interprocess communication - IPC

|Background

The ability to communicate with other processes from your plugin is a very important one! This makes flooding information throughout the MANET possible for applications. A application could pass data to the plugin, the plugin can then encapsulate the data in a OLSR message and flood it. When a plugin receives such a message it decapsulates it and delivers it to a application on the local machine in addition to forward it. This opens for a whole lot of new possibilities!

|How can a plugin listen for data?

In a typical socket-based communication scheme applications wait for incoming data using system calls like `select(2)` or `poll(2)`. If a plugin is to do this it needs a "life of its own" to prevent it from interfering with olsrd operation. This means that it must run in a thread of its own. But this again creates a lot of trouble for us concerning synchronization and protection of shared data.

To solve this we allow plugins to register their sockets with the socket parser in olsrd. This way olsrd checks for available data for us and calls the specified function when data is available. This is done using the `void(*add_olsr_socket)(int, void(*)())` call available in the plugin.

|IPC in the powerstatus-plugin

In our example we will uses one-way communication with some application. All we want to do is to send all powerstatus information we have stored in clean text. First we set up a typical server-side socket by using the `socket(2)` and `listen(2)` system calls. Then we pass this socket along with our "action" function to the olsrd socket parser. This is all done in the function `plugin_ipc_init` in `uolsr_power.c`:

```
void
plugin_ipc_init()
{
    struct sockaddr_in sin;

    /* Init ipc socket */
    if ((ipc_socket = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("IPC socket");
    }
    else
    {
        /* Bind the socket */

        /* complete the socket structure */
    }
}
```

```

memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = htons(9999);

/* bind the socket to the port number */
if (bind(ipc_socket, (struct sockaddr *) &sin, sizeof(sin)) == -1)
{
    perror("IPC bind");
    return;
}

/* show that we are willing to listen */
if (listen(ipc_socket, 1) == -1)
{
    perror("IPC listen");
    return;
}

/* Register with olsrd */
add_olsr_socket(ipc_socket, &ipc_action);
}
}

```

Next we must implement the "action" function `ipc_action` that will be called every time data is available on the socket.

In our example this function should set up communication and set the variable `ipc_open` to 1 if everything is ok.

As we are not to listen for data from the socket we do not need to add the new socket to the socket parser in olsrd.

Our `ipc_action` function is implemented like:

```

void
ipc_action(int fd)
{
    struct sockaddr_in pin;
    size_t addrlen;
    char *addr;

    if ((ipc_connection = accept(ipc_socket, (struct sockaddr *) &pin, &addrlen)) == -1)
    {
        perror("IPC accept");
        exit(1);
    }
    else
    {
        addr = inet_ntoa(pin.sin_addr);
        if(ntohl(pin.sin_addr.s_addr) != INADDR_LOOPBACK)
        {
            printf("Front end-connection from foreign host(%s) not allowed!\n", addr);
            close(ipc_connection);
            return;
        }
        else
        {
            ipc_open = 1;
            printf("POWER: Connection from %s\n",addr);
        }
    }
}
}

```

Here we check that the incoming connection is initiated from our host. If so we set up the connection and set `ipc_open` to 1.

Now we implement a function that writes data to the socket:

```

int
ipc_send(char *data, int size)
{
    if(!ipc_open)
        return 0;
    if (send(ipc_connection, data, size, MSG_NOSIGNAL) < 0)
    {
        printf("(OUTPUT)IPC connection lost!\n");
        close(ipc_connection);
        ipc_open = 0;
        return -1;
    }
}

return 1;

```

```
}

```

Notice that `ipc_open` is reset to 0 if the connection is lost.

From now all output can be done using this function. Take a look at the `print_power_table` function for examples of usage.

Bi-directional communication

If you are to read from a connected application as well you should register the newly opened socket (in `ipc_action`) with the socket parser and create a "action" function for incoming data as well.

When a connection is lost you should use the `remove_olsr_socket` call to remove the socket from the olsrd select check.

Testing

Now compile, install and run olsrd and the plugin (as root) by:

```
#tar jxvf uolsrd-0.x.x.tar.bz2
#cd unik-olsrd-0.x.x
#make
#make install
(edit /etc/olsrd.conf)
#cd lib
#make
#make install
#olsrd
```

Then in another shell do:

```
#telnet 127.0.0.1 9999
```

This will connect to your plugin and information will be displayed on a regular basis.

Your node needs to have APM support for the powerplugin to work. if there exists no file /proc/apm on your system then the powerplugin will not send any powerinfo from this system!

Do run it on a couple of machines, and try a multihop scenario where some nodes does not load the plugin to see that the default forwarding works.

Try to unplug the powersource on one node and check that its powerstatus is updated on the other nodes.

Beyond the example

A olsr plugin can really do just about anything. If you want to do more complex operations than the offered functions from olsrd allows you can define your own commands in the multi-purpose call.

You can also imagine a plugin using IPC to communicate with another process to enable this process to use OLSRs flooding technique.

The possibilities are - ehr... not endless - but pretty damn close :-D

(C) [Andreas Tønnesen](#) 2004