SIP Express Router v0.8.8 - Developer's Guide

Jan Janak

Jiri Kuthan

Bogdan Iancu

SIP Express Router v0.8.8 - Developer's Guide
by Jan Janak, Jiri Kuthan, and Bogdan Iancu

The document describes the SIP Express Router internals and algorithms. It describes overall server architecture, request processing, configuration, memory management, interprocess locking, module interface and selected modules in detail.

The document is intended mainly for module developers wishing to implement a new module for the server. Other people like developers of SIP related software or students might be interested too.

Table of Contents

List of Examples

Chapter 1. The Server Startup

The `main` function in file `main.c` is the first function called upon server startup. It's purpose is to initialize the server and enter main loop. The server initialization will be described in the following sections.

Particular initialization steps are described in order in which they appear in `main` function.

1.1. Installation Of New Signal Handlers

The first step in the initialization process is the installation of new signal handlers. We need our own signal handlers to be able to do graceful shutdown, print server statistics and so on. There is only one signal handler function which is function `sig_usr` in file `main.c`.

The following signals are handled by the function: SIGINT, SIGPIPE, SIGUSR1, SIGCHLD, SIGTERM, SIGHUP and SIGUSR2.

1.2. Processing Command Line Parameters

SER utilizes the `getopt` function to parse command line parameters. The function is extensively described in the man pages.

1.3. Parser Initialization

SER contains a fast 32-bit parser. The parser uses precalculated hash table that needs to be filled in upon startup. The initialization is done here, there are two functions that do the job. Function `init_hfname_parser` initializes hash table in header field name parser and function `init_digest_parser` initializes hash table in digest authentication parser. The parser's internals will be described later.

1.4. Malloc Initialization

To make SER even faster we decided to re-implement memory allocation routines. The new `malloc` better fits our needs and speeds up the server a lot. The memory management subsystem needs to be initialized upon server startup. The initialization mainly creates internal data structures and allocates memory region to be partitioned.

> **Important:** The memory allocation code must be initialized *BEFORE* any of its function is called !

1.5. Timer Initialization

Various subsystems of the server must be called periodically regardless of the incoming requests. That's what timer is for. Function `init_timer` initializes the timer subsystem. The function is called from `main.c` and can be found in `timer.c` The timer subsystem will be described later.

> **Warning**
>
> Timer subsystem must be initialized before config file is parsed !

1.6. FIFO Initialization

SER has built-in support for FIFO control. It means that the running server can accept commands over a FIFO special file (a named pipe). Function `register_core_fifo` initializes FIFO subsystem and registers basic commands, that are processed by the core itself. The function can be found in file `fifo_server.c`.

The FIFO server will be described in another chapter.

1.7. Built-in Module Initialization

Modules can be either loaded dynamically at runtime or compiled in statically. When a module is loaded at runtime, it is registered [1] immediately with the core. When the module is compiled in statically, the registration[1] must be performed during the server startup. Function `register_builtin_modules` does the job.

1.8. Server Configuration

The server is configured through a configuration file. The configuration file is C-Shell like script which defines how incoming requests should be processed. The file cannot be interpreted directly because that would be very slow. Instead of that the file is translated into an internal binary representation. The process is called compilation and will be described in the following sections.

> **Note:** The following sections only describe how the internal binary representation is being constructed from the config file. The way how the binary representation is used upon a request arrival will be described later.

The compilation can be divided in several steps:

1.8.1. Lexical Analysis

Lexical analysis is process of converting the input (the configuration file in this case) into a stream of tokens. A token is a set of characters that 'belong' together. A program that can turn the input into stream of tokens is called scanner. For example, when scanner encounters a number in the config file, it will produce token NUMBER.

There is no need to implement the scanner from scratch, it can be done automatically. There is a utility called flex. Flex accepts a configuration file and generates scanner according to the configuration file. The configuration file for flex consists of several lines - each line describing one token. The tokens are described using regular expressions. For more details, see flex manual page or info documentation.

Flex input file for the SER config file is in file `cfg.lex`. The file is processed by flex when the server is being compiled and the result is written in file `lex.yy.c`. The output file contains the scanner implemented in the C language.

1.8.2. Syntactical Analysis

The second stage of configuration file processing is called syntactical analysis. Purpose of syntactical analysis is to check if the configuration file has been well formed, doesn't contain syntactical errors and perform various actions at various stages of the analysis. Program performing syntactical analysis is called parser.

Structure of the configuration file is described using grammar. Grammar is a set of rules describing valid 'order' or 'combination' of tokens. If the file isn't conformable with it's grammar, it is syntactically invalid and cannot be further processed. In that case an error will be issued and the server will be aborted.

There is a utility called yacc. Input of the utility is a file containing the grammar of the configuration file, in addition to the grammar, you can describe what action the parser should do at various stages of parsing. For example, you can instruct the parser to create a structure describing an IP address every time it finds an IP address in the configuration file and convert the address to its binary representation.

For more information see yacc documentation.

yacc creates the parser when the server is being compiled from the sources. Input file for yacc is `cfg.y`. The file contains grammar of the config file along with actions that create the binary representation of the file. Yacc will write its result into file `cfg.tab.c`. The file contains function `yyparse` which will parse the whole configuration file and construct the binary representation. For more information about the bison input file syntax see bison documentation.

1.8.3. Config File Structure

The configuration file consist of three sections, each of the sections will be described separately.

- *Route Statement* - The statement describes how incoming requests will be processed. When a request is received, commands in one or more "route" sections will be executed step by step. The config file must always contain one main "route" statement and may contain several additional "route" statements. Request processing always starts at the beginning of the main “route” statement. Additional “route” statements can be called from the main one or another additional “route” statements (It it similar to function calling).

- *Assign Statement* - There are many configuration variables across the server and this statement makes it possible to change their value. Generally it is a list of assignments, each assignment on a separate line.
- *Module Statement* - Additional functionality of the server is available through separate modules. Each module is a shared object that can be loaded at runtime. Modules can export functions, that can be called from the configuration file and variables, that can be configured from the config file. The module statement makes it possible to load modules and configure them. There are two commands in the statement - `loadmodule` and `modparam`. The first can load a module. The second one can configure module’s internal variables.

In the following sections we will describe in detail how the three sections are being processed upon server startup.

1.8.3.1. Route Statement

The following grammar snippet describes how the route statement is constructed

```
route_stm = "route" "{" actions "}"
{
    $$ = push($3, &rlist[DEFAULT_RT]);
}

actions = actions action { $$ = append_action($1, $2}; }
| action { $$ = $1; }

action = cmd SEMICOLON { $$ = $1; }
       | SEMICOLON { $$ = 0; }

cmd = "forward" "(" host ")" { $$ = mk_action(FORWARD_T, STRING_ST, NUMBER_ST, $3, 0)
    | ...
```

A config file can contain one or more "route" statements. "route" statement without number will be executed first and is called the main route statement. There can be additional route statements identified by number, these additional route statements can be called from the main route statement or another additional route statements.

Each route statement consists of a set of actions. Actions in the route statement are executed step by step in the same order in which they appear in the config file. Actions in the route statement are delimited by semicolon.

Each action consists of one and only one command (cmd in the grammar). There are many types of commands defined. We don’t list all of them here because the list would be too long and all the commands are processed in the same way. Therefore we show only one example (forward) and interested readers might look in `cfg.y` file for full list of available commands.

Each rule in the grammar contains a section enclosed in curly braces. The section is the C code snippet that will be executed every time the parser recognizes that rule in the config file.

For example, when the parser finds `forward` command, `mk_action` function (as specified in the grammar snippet above) will be called. The function creates a new structure with *`type`* field set to FORWARD_T representing the command. Pointer to the structure will be returned as the return value of the rule.

The pointer propagates through `action` rule to `actions` rule. `Actions` rule will create linked list of all commands. The linked list will be then inserted into *`rlist`* table. (Function `push` in rule `route_stm`). Each element of the table represents one “route” statement of the config file.

Each route statement of the configuration file will be represented by a linked list of all actions in the statement. Pointers to all the lists will be stored in rlist array. Additional route statements are identified by number. The number also serves as index to the array.

When the core is about to execute route statement with number n, it will look in the array at position n. If the element at position n is not null then there is a linked list of commands and the commands will be executed step by step.

Reply-Route statement is compiled in the same way. Main differences are:

- Reply-Route statement is executed when a SIP *REPLY* comes (not ,SIP *REQUEST*).
- Only subset of commands is allowed in the reply-route statement. (See file `cfg.y` for more details).
- Reply-route statement has it’s own array of linked-lists.

1.8.3.2. Assign Statement

The server contains many configuration variables. There is a section of the config file in which the variables can be assigned new value. The section is called The Assign Statement. The following grammar snippet describes how the section is constructed (only one example will be shown):

```
assign_stm = "children" ’=’ NUMBER { children_no=$3; }
    | "children" ’=’ error  { yyerror("number expected"); }
...
```

The number in the config file is assigned to `children_no` variable. The second statement will be executed if the parameter is not number or is in invalid format and will issue an error and abort the server.

1.8.3.3. Module Statement

The module statement allows module loading and configuration. There are two commands:

- *loadmodule* - Load the specified module in form of a shared object. The shared object will be loaded using `dlopen`.
- *modparam* - It is possible to configure a module using this command. The command accepts 3 parameters: *module name*, *variable name* and *variable value*.

The following grammar snippet describes the module statement:

```
module_stm = "loadmodule" STRING
{
    DBG("loading module %s\n", $2);
    if (load_module($2)!=0) {
        yyerror("failed to load module");
    }
}
   | "loadmodule" error  { yyerror("string expected");  }
   | "modparam" "(" STRING "," STRING "," STRING ")"
{
    if (set_mod_param($3, $5, STR_PARAM, $7) != 0) {
        yyerror("Can't set module parameter");
    }
}
   | "modparam" "(" STRING "," STRING "," NUMBER ")"
{
    if (set_mod_param($3, $5, INT_PARAM, (void*)$7) != 0) {
        yyerror("Can't set module parameter");
    }
}
   | MODPARAM error { yyerror("Invalid arguments"); }
```

When the parser finds `loadmodule` command, it will execute statement in curly braces. The statement will call `load_module` function. The function will load the specified filename using `dlopen`. If `dlopen` was successful, the server will look for exports structure describing the module's interface and register the module. For more details see module section.

If the parser finds `modparam` command, it will try to configure the specified variable in the specified module. The module must be loaded using `loadmodule` before `modparam` for the module can be used ! Function `set_mod_param` will be called and will configure the variable in the specified module.

1.9. Interface Configuration

The server will try to obtain list of all configured interfaces of the host it is running on. If it fails the server tries to convert hostname to IP address and will use interface with the IP address only.

Function `add_interfaces` will add all configured interfaces to the array.

Try to convert all interface names to IP addresses, remove duplicates...

1.10. Turning into a Daemon

When configured so, SER becomes a daemon during startup. A process is called daemon when it hasn't associated controlling terminal. See function `daemonize` in file `main.c` for more details. The function does the following:

- *chroot* is performed if necessary. That ensures that the server will have access to a particular directory and its subdirectories only.
- Server's working directory is changed if the new working directory was specified (usually it is /).
- If command line parameter -g was used, the server's group ID is changed to that value.
- If command line parameter -u was used, the server's user ID is changed to that value.
- Perform *fork*, let the parent process exit. This ensures that we are not a group leader.
- Perform *setsid* to become a session leader and drop the controlling terminal.
- Fork again to drop group leadership.
- Create a pid file.
- Close all opened file descriptors.

1.11. Module Initialization

The whole config file was parsed, all modules were loaded already and can be initialized now. A module can tell the core that it needs to be initialized by exporting `mod_init` function. `mod_init` function of all loaded modules will be called now.

1.12. Routing List Fixing

After the whole routing list was parsed, there might be still places that can be further processed to speed-up the server. For example, several commands accept regular expression as one of their parameters. The regular expression can be compiled too and processing of compiled expression will be much faster.

Another example might be string as parameter of a function. For example if you call `append_hf("Server: SIP Express Router\r\n")` from the routing script, the function will append a new header field after the last one. In this case, the function needs to know length of the string parameter. It could call `strlen` every time it is called, but that is not a very good idea because `strlen` would be called every time a message is processed and that is not necessary.

Instead of that the length of the string parameter could be precalculated upon server startup, saved and reused later. The processing of the request will be faster because `append_hf` doesn't need to call `strlen` every time, I can just reuse the saved value.

This can be used also for string to int conversions, hostname lookups, expression evaluation and so on.

This process is called Routing List Fixing and will be done as one of last steps of the server startup.

Every loaded module can export one or more functions. Each such function can have associated a fixup function, which should do fixing as described in this section. All such fixups of all loaded modules will be called here. That makes it possible for module functions to fix their parameters too if necessary.

1.13. Statistics Initialization

If compiled-in, the core can produce some statistics about itself and traffic processed. The statistics subsystem gets initialized here, see function `init_stats`.

1.14. Socket Initialization

UDP socket initialization depends on `dont_fork` variable. If this variable is set (only one process will be processing incoming requests) and there are multiple listen interfaces, only the first one will be used. This mode is mainly for debugging.

If the variable is not set, then sockets for all configured interfaces will be created and initialized. See function `udp_init` in file `udp_server.c` for more details.

1.15. Forking

The rest of the initialization process depends on value of `dont_fork` variable. `dont_fork` is a global variable defined in `main.c`. We will describe both variants separately.

1.15.1. `dont_fork` variable is set (not zero)

If `dont_fork` variable is set, the server will be operating in special mode. There will be only one process processing incoming requests. This is very slow and was intended mainly for debugging purposes. The main process will be processing all incoming requests itself.

The server still needs additional children:

- One child is for the timer subsystem, the child will be processing timers independently of the main process.
- FIFO server will spawn another child if enabled. The child will be processing all commands coming through the fifo interface.
- If SNMP support was enabled, another child will be created.

The following initialization will be performed in dont fork mode. (look into function `main_loop` in file `main.c`.

- Another child will be forked for the timer subsystem.
- Initialize the FIFO server if enabled, this will fork another child. For more info about the FIFO server, see section The FIFO server.

- Call `init_child(0).` The function performs per-child specific initialization of all loaded modules. A module can be initialized though `mod_init` function. The function is called *BEFORE* the server forks and thus is common for all children.

  If there is anything, that needs to be initialized in every child separately (for example if each child needs to open its own file descriptor), it cannot be done in `mod_init`. To make such initialization possible, a module can export another initialization function called `init_child`. The function will be called in all children *AFTER* fork of the server.

  And since we are in "dont fork" mode and there will no children processing requests (remember the main process will be processing all requests), the `init_child` wouldn't be called.

  That would be bad, because `child_init` might do some initialization that must be done otherwise modules might not work properly.

  To make sure that module initialization is complete we will call `init_child` here for the main process even if we are not going to fork.

That's it. Everything has been initialized properly and as the last step we will call `udp_rcv_loop` which is the main loop function. The function will be described later.

1.15.2. `dont_fork` is not set (zero)

`dont_fork` is not set. That means that the server will fork children and the children will be processing incoming requests. How many children will be created depends on the configuration (`children` variable). The main process will be sleeping and handling signals only.

The main process will then initialize the FIFO server. The FIFO server needs another child to handle communication over FIFO and thus another child will be created. The FIFO server will be described in more detail later.

Then the main process will perform another fork for the timer attendant. The child will take care of timer lists and execute specified function when a timer hits.

The main process is now completely initialized, it will sleep in `pause` function until a signal comes and call `handle_sigs` when such condition occurs.

The following initialization will be performed by each child separately:

Each child executes `init_child` function. The function will sequentially call `child_init` functions of all loaded modules.

Because the function is called in each child separately, it can initialize per-child specific data. For example if a module needs to communicate with database, it must open a database connection. If the connection would be opened in `mod_init` function, all the children would share the same connection and locking would be necessary to avoid conflicts. On the other hand if the connection was opened in `child_init` function, each child will have its own connection and concurrency conflicts will be handled by the database server.

And last, but not least, each child executes `udp_rcv_loop` function which contains the main loop logic.

Notes

1. Module registration is a process when the core tries to find what functions and parameters are offered by the module.

Chapter 2. Main Loop

Upon startup, all children execute `recvfrom` function. The process will enter the kernel mode. When there is no data to be processed at the moment, the kernel will put the process on list of processes waiting for data and the process will be put asleep.

When data to be processed was received, the first process on the list will be removed from the list and woken up. After the process finished processing of the data, it will call `recvfrom` again and will be put by the kernel at the end of the list.

When next data arrives, the first process on the list will be removed, processes the data and will be put on the end of the list again. And so on...

The main loop logic can be found in function `udp_rcv_loop` in file `udp_server.c`.

The message is received using `recvfrom` function. The received data is stored in buffer and zero terminated.

If configured so, basic sanity checks over the received message will be performed.

The message is then processed by `receive_msg` function and `recvfrom` is called again.

2.1. `receive_msg` Function

The function can be found in `receive.c` file.

- In the server, a request or response is represented by sip_msg structure. The structure is allocated in this function. The original message is stored in *`buf`* attribute of the structure and is zero terminated. Then, another copy of the received message will be created and the copy will be stored in *`orig`* field. The original copy will be not modified during the server operation. All changes will be made to the copy in *`buf`* field. The second copy of the message will be removed in the future.
- The message will be parsed (function `parse_msg`). We don't need the whole message header to be parsed at this stage. Only the first line and first Via header need to be parsed. The server needs to know if the message is request or response - hence the first line. The server also needs the first Via to be able to add its own Via - hence the first Via. Nothing else will be parsed at the moment - this saves time. (Message parser as well as sip_msg structure will be described later).
- A module may register callbacks. Each callback have associated an event, that will trigger the callback. One such callback is *pre-script* callback. Such callback will be called immediately before the routing part of the config file will be executed. If there are such callbacks registered, they will be executed now.
- As the next step we will determine type of the message. If the message being processed is a REQUEST then basic sanity checks will be performed (make sure that there is the first Via and parsing was successful) and the message will be passed to routing engine. The routing engine is one of the most complicated parts of the server and will be in detail described in chapter The Routing Engine.
- If the message is a RESPONSE, it will be simply forwarded to its destination.

- After all, *post-script* callbacks will be executed if any and the structure representing the message will be released.
- Processing of the message is done now and the process is ready for another SIP message.

Chapter 3. The Server Shutdown

The server shutdown can be triggered by sending a signal to the server. The server will behave differently upon receiving various types of signals, here is a brief summary:

- *SIGINT, SIGPIPE, SIGTERM, SIGCHLD* will terminate the server.
- *SIGUSR1* will print statistics and let the server continue.
- *SIGHUP, SIGUSR2* will be ignored.

There is only one common signal handler for all signals - function `sig_usr` in file `main.c`.

In normal mode of operation (`dont_fork` variable is not set), the main server is not processing any requests, it calls `pause` function and will be waiting for signals only. What happens when a signal arrives is shown in the previous paragraph.

When in normal mode (`dont_fork` is not set), the signal handler of the main process will only store number of the signal received. All the processing logic will be executed by the main process outside the signal handler (function `handle_sigs`) The function will be called immediately after the signal handler finish. The main process usually does some cleanup and running such things outside the signal handler is much more safe than doing it from the handler itself. Children only print statistics and exit or ignore the signal completely, that is quite safe and can be done directly from the signal handler of children.

When `dont_fork` is set, all the cleanup will be done directly from the signal handler, because there is only one process - the main process. This is not so safe as the previous case, but this mode should be used for debugging only and such shortcoming doesn't harm in that case.

Upon receipt of SIGINT, SIGPIPE or SIGTERM `destroy_modules` will be called. Each module may register so-called `destroy` function if it needs to do some cleanup when the server is terminating (flush of cache to disk for example). `destroy_modules` will call destroy function of all loaded modules.

If you need to terminate the server and all of its children, the best way how to do it is to send SIGTERM to the main process, the main process will in turn send the same signal to its children.

The main process and its children are in the same process group. Therefore the main process can kill all its children simply by sending a signal to pid 0, sending to pid 0 will send the signal to all processes in the same process group as the sending process. This is how the main process will terminate all children when it is going to shut down.

If one child exited during normal operation, the whole server will be shut down. This is better than let the server continue - a dead child might hold a lock and that could block the whole server, such situation cannot be avoided easily. Instead of that it is better to shutdown the whole server and let it restart.

Chapter 4. Internal Data Structures

There are some data structures that are important and widely used in the server. We will describe them in detail in this section.

> **Note:** There are many more structures and types defined across the server and modules. We will describe only the most important and common data structure here. The rest will be described in other sections if needed.

4.1. Type str

One of our main goals was to make SER really fast. There are many functions across the server that need to work with strings. Usually these functions need to know string length. We wanted to avoid using of `strlen` because the function is relatively slow. It must scan the whole string and find the first occurrence of zero character. To avoid this, we created str type. The type has 2 fields, field *`s`* is pointer to the beginning of the string and field *`len`* is length of the string. We then calculate length of the string only once and later reuse saved value.

> **Important:** str structure is quite important because it is widely used in SER (most functions accept str instead of char*).

str Type Declaration

```
struct _str{
    char* s;
    int len;
};

typedef struct _str str;
```

The declaration can be found in header file `str.h`.

> ### Warning
>
> Because we store string lengths, there is no need to zero terminate them. Some strings in the server are still zero terminated, some are not. Be careful when using functions like `snprintf` that rely on the ending zero. You can print variable of type str this way:
>
> ```
> printf("%.*s", mystring->len, mystring->s);
> ```
>
> That ensures that the string will be printed correctly even if there is no zero character at the end.

4.2. Structure hdr_field

The structure represents a header field of a SIP message. A header field consist of *name* and *body* separated by a double colon. For example: "Server: SIP Express Router\r\n" is one header field. "Server" is header field name and "SIP Express Router\r\n" is header field body.

The structure is defined in file `hf.h` under `parser` subdirectory.

Structure Declaration

```
struct hdr_field {
    int type;                /* Header field type */
    str name;                /* Header field name */
    str body;                /* Header field body */
    void* parsed;            /* Parsed data structures */
    struct hdr_field* next;  /* Next header field in the list */
};
```

Field Description:

- *`type`* - Type of the header field, the following header field types are defined (and recognized by the parser):

  HDR_VIA1, HDR_VIA2, HDR_TO, HDR_FROM, HDR_CSEQ, HDR_CALLID, HDR_CONTACT, HDR_MAXFORWARDS, HDR_ROUTE, HDR_RECORDROUTE, HDR_CONTENTTYPE, HDR_CONTENTLENGTH, HDR_AUTHORIZATION, HDR_EXPIRES, HDR_PROXYAUTH, HDR_WWWAUTH, HDR_SUPPORTED, HDR_REQUIRE, HDR_PROXYREQUIRE, HDR_UNSUPPORTED, HDR_ALLOW, HDR_EVENT, HDR_OTHER.

  Their meaning is self explanatory. HDR_OTHER marks header field not recognized by the parser.

- *`name`* - Name of the header field (the part before colon)
- *`body`* - body of the header field (the part after colon)
- *`parsed`* - Each header field body can be further parsed. The field contains pointer to parsed structure if the header field was parsed already. The pointer is of type void* because it can point to different types of structure depending on the header field type.
- *`next`* - Pointer to the next header field in linked list.

4.3. Structure sip_uri

This structure represents parsed SIP URI.

```
struct sip_uri {
    str user;     /* Username */
    str passwd;   /* Password */
    str host;     /* Host name */
    str port;     /* Port number */
```

```
    str params;   /* Parameters */
    str headers;
};
```

Field Description:

- *user* - Username if found in the URI.
- *passwd* - Password if found in the URI.
- *host* - Hostname of the URI.
- *params* - Parameters of the URI if any.
- *headers* - See the SIP RFC.

4.4. Structure via_body

The structure represents parsed Via header field. See file `parse_via.h` under `parser` subdirectory for more details.

```
struct via_body {
    int error;
    str hdr;                      /* Contains "Via" or "v" */
    str name;
    str version;
    str transport;
    str host;
    int port;
    str port_str;
    str params;
    str comment;
    int bsize;                    /* body size, not including hdr */
    struct via_param* param_lst;  /* list of parameters*/
    struct via_param* last_param; /*last via parameter, internal use*/

    /* shortcuts to "important" params*/
    struct via_param* branch;
    struct via_param* received;

    struct via_body* next;        /* pointer to next via body string if
    compact via or null */
};
```

Field Description:

- *error* - The field contains error code when the parser was unable to parse the header field.
- *hdr*- Header field name, it can be “Via” or “v” in this case.
- *name* - Protocol name (“SIP” in this case).
- *version* - Protocol version (for example “2.0”).
- *transport* - Transport protocol name (“TCP”, “UDP” and so on).
- *host* - Hostname or IP address contained in the Via header field.

- *port* - Port number as integer.
- *port_str* - Port number as string.
- *params* - Unparsed parameters (as one string containing all the parameters).
- *comment* - Comment.
- *bsize* - Size of the body (not including hdr).
- *param_lst* - Linked list of all parameters.
- *last_param* - Last parameter in the list.
- *branch* - Branch parameter.
- *received* - Received parameter.
- *next* - If the Via is in compact form (more Vias in the same header field), this field contains pointer to the next Via.

4.5. Structure ip_addr

The structure represents IPv4 or IPv6 address. It is defined in `ip_addr.h`.

```
struct ip_addr{
    unsigned int af;      /* address family: AF_INET6 or AF_INET */
    unsigned int len;     /* address len, 16 or 4 */

    /* 64 bits aligned address */
    union {
        unsigned int   addr32[4];
        unsigned short addr16[8];
        unsigned char  addr[16];
    }u;
};
```

4.6. Structure lump

The structure describes modifications that should be made to the message before the message will be sent.

The structure will be described in more detail later in chapter SIP Message Modifications.

4.7. Structure lump_rpl

The structure represents text that should be added to reply. List of such data is kept in the request and processed when the request is being turned into reply.

The structure will be described in more detail later in chapter SIP Message Modifications.

4.8. Structure msg_start

The structure represents the first line of a SIP request or response.

The structure is defined in file `parse_fline.h` under `parser` subdirectory.

Structure Declaration

```
struct msg_start {
    int type;                        /* Type of the Message - Request/Response */
    union {
        struct {
            str method;              /* Method string */
            str uri;                 /* Request URI */
            str version;             /* SIP version */
            int method_value;        /* Parsed method */
        } request;
struct {
            str version;             /* SIP version */
            str status;              /* Reply status */
            str reason;              /* Reply reason phrase */
            unsigned int statuscode; /* Status code */
        } reply;
    }u;
};
```

Description of Request Related Fields:

- *`type`* - Type of the message - REQUEST or RESPONSE.
- *`method`* - Name of method (same as in the message).
- *`uri`* - Request URI.
- *`version`* - Version string.
- *`method_value`* - Parsed method. Field method which is of type str will be converted to integer and stored here. This is good for comparison, integer comparison is much faster then string comparison.

Description of Response Related Fields:

- *`version`* - Version string.
- *`status`* - Response status code as string.
- *`reason`* - Response reason string as in the message.
- *`statuscode`* - Response status code converted to integer.

4.9. Structure sip_msg

This is the most important structure in the whole server. This structure represents a SIP message. When a message is received, it is immediately converted into this structure and all operations are performed over the structure. After the server finished processing, this structure is converted back to character array buffer and the buffer is sent out.

Structure Declaration:

```
struct sip_msg {
    unsigned int id;               /* message id, unique/process*/
    struct msg_start first_line;   /* Message first line */
    struct via_body* via1;         /* The first via */
    struct via_body* via2;         /* The second via */
    struct hdr_field* headers;     /* All the parsed headers*/
    struct hdr_field* last_header; /* Pointer to the last parsed header*/
    int parsed_flag;               /* Already parsed header field types */

    /* Via, To, CSeq, Call-Id, From, end of header*/
    /* first occurrence of it; subsequent occurrences
     * saved in 'headers'
     */

    struct hdr_field* h_via1;
    struct hdr_field* h_via2;
    struct hdr_field* callid;
    struct hdr_field* to;
    struct hdr_field* cseq;
    struct hdr_field* from;
    struct hdr_field* contact;
    struct hdr_field* maxforwards;
    struct hdr_field* route;
    struct hdr_field* record_route;
    struct hdr_field* content_type;
    struct hdr_field* content_length;
    struct hdr_field* authorization;
    struct hdr_field* expires;
    struct hdr_field* proxy_auth;
    struct hdr_field* www_auth;
    struct hdr_field* supported;
    struct hdr_field* require;
    struct hdr_field* proxy_require;
    struct hdr_field* unsupported;
    struct hdr_field* allow;
    struct hdr_field* event;

    char* eoh;        /* pointer to the end of header (if found) or null */
    char* unparsed;   /* here we stopped parsing*/

    struct ip_addr src_ip;
    struct ip_addr dst_ip;

    char* orig;       /* original message copy */
    char* buf;        /* scratch pad, holds a modified message,
                       *  via, etc. point into it
       */
    unsigned int len; /* message len (orig) */

    /* modifications */
```

```
    str new_uri;               /* changed first line uri*/
    int parsed_uri_ok;         /* 1 if parsed_uri is valid, 0 if not */
    struct sip_uri parsed_uri; /* speed-up > keep here the parsed uri*/

    struct lump* add_rm;         /* used for all the forwarded
                                  * requests */
    struct lump* repl_add_rm;    /* used for all the forwarded replies */
    struct lump_rpl *reply_lump; /* only for locally generated replies !!!*/

    char add_to_branch_s[MAX_BRANCH_PARAM_LEN];
    int add_to_branch_len;

    /* index to TM hash table; stored in core to avoid unnecessary calcs */
    unsigned int  hash_index;

    /* allows to set various flags on the message; may be used for
     * simple inter-module communication or remembering processing state
     * reached
     */
    flag_t flags;
};
```

Field Description:

- `id` - Unique ID of the message within a process context.
- `first_line` - Parsed first line of the message.
- `via1` - The first Via - parsed.
- `via2` - The second Via - parsed.
- `headers` - Linked list of all parsed headers.
- `last_header` - Pointer to the last parsed header (parsing is incremental, that means that the parser will stop if all requested headers were found and next time it will continue at the place where it stopped previously. Therefore this field will not point to the last header of the message if the whole message hasn't been parsed yet).
- `parsed_flag` - Already parsed header field types (bitwise OR).

The following fields are set to zero if the corresponding header field was not found in the message or hasn't been parsed yet. (These fields are called hooks - they always point to the first occurrence if there is more than one header field of the same type).

- `h_via1` - Pointer to the first Via header field.
- `h_via2` - Pointer to the second Via header field.
- `callid` - Pointer to the first Call-ID header field.
- `to` - Pointer to the first To header field.
- `cseq` - Pointer to the first CSeq header field.
- `from` - Pointer to the first From header field.
- `contact` - Pointer to the first Contact header field.
- `maxforwards` - Pointer to the first Max-Forwards header field.
- `route` - Pointer to the first Route header field.

- *record_route* - Pointer to the first Record-Route header field.
- *content_type* - Pointer to the first Content-Type header field.
- *content_length* - Pointer to the first Content-Length header field.
- *authorization* - Pointer to the first Authorization header field.
- *expires* - Pointer to the first Expires header field.
- *proxy_auth* - Pointer to the first Proxy-Authorize header field.
- *www_auth* - Pointer to the first WWW-Authorize header field.
- *supported* - Pointer to the first Supported header field.
- *require* - Pointer to the first Require header field.
- *proxy_require* - Pointer to the first Proxy-Require header field.
- *unsupported* - Pointer to the first Unsupported header field.
- *allow* - Pointer to the first Allow header field.
- *event* - Pointer to the first Event header field.

The following fields are mostly used internally by the server and should be modified through dedicated functions only.

- *eoh* - Pointer to the End of Header or null if not found yet (the field will be set if and only if the whole message was parsed already).
- *unparsed* - Pointer to the first unparsed character in the message.
- *src_ip* - Sender's IP address.
- *dst_ip* - Destination's IP address.
- *orig* - Original (unmodified) message copy, this field will hold unmodified copy of the message during the whole message lifetime.
- *buf* - Message scratch-pad (modified copy of the message) - All modifications made to the message will be done here.
- *len* - Length of the message (unmodified).

- *new_uri* - New Request-URI to be used when forwarding the message.
- *parsed_uri_ok* - 1 if *parsed_uri* is valid, 0 if not.
- *parsed_uri* - The original parsed Request URI, sometimes it might be necessary to revert changes made to the Request URI and therefore we store the original URI here.

- *add_rm* - Linked list describing all modifications that will be made to *REQUEST* before it will be forwarded. The list will be processed when the request is being converted to character array (i.e. immediately before the request will be send out).
- *repl_add_rm* - Linked list describing all modifications that will be made to *REPLY* before it will be forwarded. the list will be processed when the reply is being converted to character array (i.e. immediately before the request will be send out).
- *reply_lump* - This is list of data chunks that should be appended to locally generated reply, i.e. when the server is generating local reply out of the request. A local reply is reply generated by the server. For example, when processing of a request fails for some reason, the server might generate an error reply and send it back to sender.

- *add_to_branch_s* - String to be appended to branch parameter.
- *add_to_branch_len* - Length of the string.

- *hash_index* - Index to a hash table in TM module.
- *flags* - Allows to set various flags on the message. May be used for simple inter-module communication or remembering processing state reached.

Chapter 5. The Routing Engine

In a previous section we discussed how routing part of a config file gets translated into binary representation. In this section, we will discuss how the binary representation is used during message processing.

Upon a SIP message receipt, the server performs some basic sanity checks and converts the message into sip_msg structure. After that the Routing Engine will start processing the message.

The routing engine can be found in file `action.c`.

The main function is `run_actions.` The function accepts two parameters. The first parameter is list of actions to be processed (Remember, the config file gets translated into array of linked lists. Each linked list in the array represents one "route" part of the config file). The second parameter is sip_msg structure representing the message to be processed.

Upon a receipt of a request, the linked list representing the main route part will be processed so the first parameter will be `rlist[0].` (The linked list of main route part is always at index 0).

The function will then sequentially call `do_action` function for each element of the linked list. Return value of the function is important. If the function returns 0, processing of the list will be stopped. By returning 0 a command can indicate that processing of the message should be stopped and the message will be dropped.

Modules may export so-called *on_break handlers*. *on_break* handler is a function, that will be called when processing of the linked-list is interrupted (ret == 0). All such handlers will be called when processing of the linked-list is finished and ret == 0.

5.1. `do_action` Function

`do_action` function is core of the routing engine. There is a big `switch` statement. Each case of the statements is one command handled by the server core itself.

The following commands are handled by the SER core itself: `drop`, `forward`, `send`, `log`, `append_branch`, `len_gt`, `setflag`, `resetflag`, `isflagset`, `error`, `route`, `exec`, `revert_uri`, `set_host`, `set_hostport`, `set_user`, `set_userpass`, `set_port`, `set_uri`, `prefix`, `strip`, `if`, `module`.

Each of the commands is represented by a *case* statement in the switch. (For example, if you are interested in implementation of `drop` command, look at “case DROP_T:” statement in the function.

The respective commands will be described now.

- `drop` - This command is very simple, it simply returns 0 which will result in abortion of processing of the request. No other commands after `drop` will be executed.

- `forward` - The function will forward the message further. The message will be either forwarded to the Request URI of the message or to IP or host given as parameter.

  In the first case, host in the Request URI must be converted into corresponding IP address. Function `mk_proxy` converts hostname to corresponding IP address. The message is then sent out using `forward_request` function.

  In the second case, hostname was converted to IP address in fixup i.e. immediately after the config file was compiled into its binary representation. The first parameter is pointer to proxy structure created in the fixup and therefore we only need to call `forward_request` here to forward the message further.

- `send` - This functions sends the message to a third-party host. The message will be sent out as is - i.e. without Request URI and Via altering.

  Hostname or IP address of the third-party host is specified as a parameter of the function.

  The message will be sent out using `udp_send` directly.

- `log` - The message given as a parameter will be logged using system logger. It can be either **syslog** or `stderr` (depends on configuration). The message is logged using `LOG` which is a macro defined in `dprint.h` header file.
- `append_branch` - Append a new URI for forking.

  More than one destinations may be associated with a single SIP request. If the server was configured so, it will use all the destinations and fork the request.

  The server keeps an array of all destinations, that should be used when forking. The array and related functions can be found in file `dset.c`. There is function `append_branch` which adds a new destination to the set.

  This command simply calls `append_branch` function and adds a new destination to the destination set.

- `len_gt` - The command accepts one number as a parameter. It then compares the number with length of the message. If the message length is greater or equal then the number then 1 will be returned otherwise the function returns -1.
- `setflag` - Sets a flag in the message. The command simply calls `setflags` function that will set the flag. Fore more information see file `flag.c`.
- `resetflag` - Same as command `setflag` - only resetflag will be called instead of setflag.
- `isflagset` - Test if the flag is set or not.
- `error` - Log a message with NOTICE log level.
- `route` - Execute another route statement.

  As we have mentioned already, there can be more that one route statement in the config file. One of them is main (without number), the other are additional. This command makes it possible to execute an additional route statement.

The command accepts one parameter which is route statement number. First sanity checks over the parameter will be performed. If the checks passed, function `run_actions` will be called. The function accepts two parameters. The first one is linked list to execute, the second one is sip_msg structure representing the message to be processed.

As you might remember, each route statement was compiled into linked list of commands to be executed and head of the linked list was stored in `rlist` array. For example, head of linked list representing route statement with number 4 will be stored at position 4 in the array (position 0 is reserved for the main route statement).

So the command will simply call `run_actions(rlist[a->p1.number], msg)` and that will execute route statement with number given as parameter.

- `exec` - Execute a shell command.

  The command accepts one parameter of type char*. The string given as parameter will be passed to `system` function which will in turn execute `/bin/sh -c <string>`.

- `revert_uri` - Revert changes made to the Request URI.

  If there is a new URI stored in *`new_uri`* of sip_msg structure, it will be freed. The original Request URI will be used when forwarding the message.

  If there is a valid URI in *`parsed_uri`* field of sip_msg structure (indicated by *`parsed_uri_ok`* field), it will be freed too.

- `set_host` - Change hostname of Request URI to value given as parameter.

  If there is a URI in *`new_uri`* field, it will be modified, otherwise the original Request URI will be modified.

- `set_hostport` - change hostname and port of Request URI to value given as string parameter.

  If there is a URI in *`new_uri`* field, it will be modified, otherwise the original Request URI will be modified.

- `set_user` - Set username part of Request URI to string given as parameter.

  If there is a URI in *`new_uri`* field, it will be modified, otherwise the original Request URI will be modified.

- `set_userpass` - Set username and password part of Request URI to string given as parameter.

  If there is a URI in *`new_uri`* field, it will be modified, otherwise the original Request URI will be modified.

- `set_port` - Set port of Request URI to value given as parameter.

If there is a URI in *new_uri* field, it will be modified, otherwise the original Request URI will be modified.

- set_uri - Set a new Request URI.

  If there is a URI in *new_uri* field, it will be freed. If there is a valid URI in *parsed_uri* field, it will be freed too.

  Then URI given as parameter will be stored in *new_uri* field. (If *new_uri* contains a URI it will be used instead of Request URI when forwarding the message).

- prefix - Set the parameter as username prefix.

  The string will be put immediately after "sip:" part of the Request URI.

  If there is a URI in *new_uri* field, it will be modified, otherwise the original Request URI will be modified.

- strip - Remove first n characters of username in Request URI.

  If there is a URI in *new_uri* field, it will be modified, otherwise the original Request URI will be modified.

- if - if Statement.

  There is an expression associated with the command and one or two linked lists of commands. The expression is a regular expression compiled into binary form in the fixup when the config file was compiled.

  The expression will be evaluated now. If the result is > 0, the first linked list will be executed using run_action function. The linked list represents command enclosed in curly braces of if command.

  Otherwise, if there is the second list, it will be executed in the same way. The second list represents commands of else statement.

- module - Execute a function exported by a module.

  When a command in a route statement is not recognized by the core itself (i.e. it is not one of commands handled by the core itself), list of exported functions of all loaded modules will be searched for a function with corresponding name and number of parameters.

  If the function was found, module command (this one) will be created and pointer to the function will be stored in *p1.data* field.

  So, this command will simply call function whose pointer is in *p1.data* field and will pass 2 parameters to the function. If one or both of the parameters were not used, 0 will be passed instead.

Return value of the function will be returned as return value of `module` command.

This command makes SER pretty extensible while the core itself is still reasonably small and clean. Additional functionality is put in modules and loaded only when needed.

Chapter 6. The Message Parser

In this section we will discuss internals of the SIP message header parser implemented in the server. Message parsing is very important and one of the most time consuming operations of a SIP server. We have been trying to make the parser as fast as possible.

A header field parser can be either in the server core or in a module. By convention, parser that is needed by the core itself or is needed by at least two modules will be in the core. Parsers contained in modules will be not described in this section.

There is a `parser` subdirectory that contains all the parsers and related stuff.

The following parsers can be found under `parser` subdirectory:

6.1. Structure of a SIP Message

A SIP message consists of message header and optional message body. The header is separated from the body with a empty line (containing CRLF only).

Message header consists of the first line and one or more header fields. The first line determines type of the message. Header fields provide additional information that is needed by clients and servers to be able to process the message.

Each header field consists of header field name and header field body. Header field name is delimited from header field body by a colon (":"). For example, "Server: SIP Express Router" - in this case "Server" is header field name and "SIP Express Router" is header field body.

6.2. The Parser Organization

- First Line Parser - Parses the first line of a SIP message.
- Header Name Parser- Parsers Name part of a header field (part before colon).
- To Header Parser - Parses body of To header field.
- From Header Parser - Parses body of From header field.
- CSeq Header Parser - Parses body of CSeq header field.
- Event Header Parser - Parses body of Event header field.
- Expires Header Parser - Parses body of Expires header field.
- Via Header Parser - Parses body of Via header field.
- Contact Header Parser - Parses body of Contact header field.
- Digest Parser - Parses digest response.

The server implements what we call *incremental parsing*. It means that a header field will be not parsed unless it is really needed. There is a minimal set of header that will be parsed every time. The set includes:

- The first line - the server must know if the message is request or response
- Via header field - Via will be needed for sure. We must add ourself to Via list when forwarding the message.

6.2.1. The First Line Parser

Purpose of the parser is to parse the first line of a SIP message. The first line is represented by msg_start structure define in file `parse_fline.h` under `parser` subdirectory.

The main function of the first line parser is `parse_first_line`, the function will fill in msg_start structure.

Follow inline comments in the function if you want to add support for a new message type.

6.2.2. The Header Field Name Parser

The purpose of the header field type parser is to recognize type of a header field. The following types of header field will be recognized:

Via, To, From, CSeq, Call-ID, Contact, Max-Forwards, Route, Record-Route, Content-Type, Content-Length, Authorization, Expires, Proxy-Authorization, WWW-Authorization, supported, Require, Proxy-Require, Unsupported, Allow, Event.

All other header field types will be marked as HDR_OTHER.

Main function of header name parser is `parse_hname2`. The function can be found in file `parse_hname.c`. The function accepts pointers to begin and end of a header field and fills in hdf_field structure. *`name`* field will point to the header field name, *`body`* field will point to the header field body and *`type`* field will contain type of the header field if known and HDR_OTHER if unknown.

The parser is 32-bit, it means, that it processes 4 characters of header field name at time. 4 characters of a header field name are converted to an integer and the integer is then compared. This is much faster than comparing byte by byte. Because the server is compiled on at least 32-bit architectures, such comparison will be compiled into one instruction instead of 4 instructions.

We did some performance measurement and 32-bit parsing is about 3 times faster for a typical SIP message than corresponding automaton comparing byte by byte. Performance may vary depending on the message size, parsed header fields and header fields type. Test showed that it was always as fast as corresponding 1-byte comparing automaton.

Since comparison must be case insensitive in case of header field names, it is necessary to convert it to lower case first and then compare. Since converting byte by byte would slow down the parser a lot, we have implemented a hash table, that can again convert 4 bytes at once. Since set of keys that need to be converted to lowercase is known (the set consists of all possible 4-byte parts of all recognized header field names) we can precalculate size of the hash table to be synonym-less. That will simplify (and speed up) the lookup a lot. The hash table must be initialized upon the server startup (function `init_hfname_parser`).

The header name parser consists of several files, all of them are under `parser` subdirectory. Main file is `parse_hname2.c` - this files contains the parser itself and functions used to initialize and lookup the hash table. File `keys.h` contains automatically generated set of macros. Each macro is a group of 4 bytes converted to integer. The macros are used for comparison and the hash table initialization. For example, for Max-Forwards header field name, the following macros are defined in the file:

```
#define _max__ 0x2d78616d   /* "max-" */
#define _maX__ 0x2d58616d   /* "maX-" */
#define _mAx__ 0x2d78416d   /* "mAx-" */
#define _mAX__ 0x2d58416d   /* "mAX-" */
#define _Max__ 0x2d78614d   /* "Max-" */
#define _MaX__ 0x2d58614d   /* "MaX-" */
#define _MAx__ 0x2d78414d   /* "MAx-" */
#define _MAX__ 0x2d58414d   /* "MAX-" */

#define _forw_ 0x77726f66   /* "forw" */
#define _forW_ 0x57726f66   /* "forW" */
#define _foRw_ 0x77526f66   /* "foRw" */
#define _foRW_ 0x57526f66   /* "foRW" */
#define _fOrw_ 0x77724f66   /* "fOrw" */
#define _fOrW_ 0x57724f66   /* "fOrW" */
#define _fORw_ 0x77524f66   /* "fORw" */
#define _fORW_ 0x57524f66   /* "fORW" */
#define _Forw_ 0x77726f46   /* "Forw" */
#define _ForW_ 0x57726f46   /* "ForW" */
#define _FoRw_ 0x77526f46   /* "FoRw" */
#define _FoRW_ 0x57526f46   /* "FoRW" */
#define _FOrw_ 0x77724f46   /* "FOrw" */
#define _FOrW_ 0x57724f46   /* "FOrW" */
#define _FORw_ 0x77524f46   /* "FORw" */
#define _FORW_ 0x57524f46   /* "FORW" */

#define _ards_ 0x73647261   /* "ards" */
#define _ardS_ 0x53647261   /* "ardS" */
#define _arDs_ 0x73447261   /* "arDs" */
#define _arDS_ 0x53447261   /* "arDS" */
#define _aRds_ 0x73645261   /* "aRds" */
#define _aRdS_ 0x53645261   /* "aRdS" */
#define _aRDs_ 0x73445261   /* "aRDs" */
#define _aRDS_ 0x53445261   /* "aRDS" */
#define _Ards_ 0x73647241   /* "Ards" */
#define _ArdS_ 0x53647241   /* "ArdS" */
#define _ArDs_ 0x73447241   /* "ArDs" */
#define _ArDS_ 0x53447241   /* "ArDS" */
#define _ARds_ 0x73645241   /* "ARds" */
#define _ARdS_ 0x53645241   /* "ARdS" */
#define _ARDs_ 0x73445241   /* "ARDs" */
#define _ARDS_ 0x53445241   /* "ARDS" */
```

As you can see, Max-Forwards name was divided into three 4-byte chunks: Max-, Forw, ards. The file contains macros for every possible lower and upper case character combination of the chunks. Because the name (and therefore chunks) can contain colon (":"), minus or space and these characters are not allowed in macro name, they must be substituted. Colon is substituted by "1", minus is substituted by underscore ("_") and space is substituted by "2".

When initializing the hash table, all these macros will be used as keys to the hash table. One of each upper and lower case combinations will be used as value. Which one ?

There is a convention that each word of a header field name starts with a upper case character. For example, most of user agents will send "Max-Forwards", messages containing some other combination of upper and lower case characters (for example: "max-forwards", "MAX-FORWARDS", "mAX-fORWARDS") are very rare (but it is possible).

Considering the previous paragraph, we optimized the parser for the most common case. When all header fields have upper and lower case characters according to the convention, there is no need to do hash table lookups, which is another speed up.

For example suppose we are trying to figure out if the header field name is Max-Forwards and the header field name is formed according to the convention (i.e. "Max-Forwards"):

- Get the first 4 bytes of the header field name ("Max-"), convert it to an integer and compare to "_Max__" macro. Comparison succeeded, continue with the next step.
- Get next 4 bytes of the header field name ("Forw"), convert it to an integer and compare to "_Forw_" macro. Comparison succeeded, continue with the next step.
- Get next 4 bytes of the header field name ("ards"), convert it to an integer and compare to "_ards_" macro. Comparison succeeded, continue with the next step.
- If the following characters are spaces and tabs followed by a colon (or colon directly without spaces and tabs), we found Max-Forwards header field name and can set `type` field to HDR_MAXFORWARDS. Otherwise (other characters than colon, spaces and tabs) it is some other header field and set `type` field to HDR_OTHER.

As you can see, there is no need to do hash table lookups if the header field was formed according to the convention and the comparison was very fast (only 3 comparisons needed !).

Now lets consider another example, the header field was not formed according to the convention, for example "MAX-forwards":

- Get the first 4 bytes of the header field name ("MAX-"), convert it to an integer and compare to "_Max__" macro.

  Comparison failed, try to lookup "MAX-" converted to integer in the hash table. It was found, result is "Max-" converted to integer.

  Try to compare the result from the hash table to "_Max__" macro. Comparison succeeded, continue with the next step.

- Compare next 4 bytes of the header field name (“forw”), convert it to an integer and compare to “_Max__” macro.

  Comparison failed, try to lookup “forw” converted to integer in the hash table. It was found, result is “Forw” converted to integer.

  Try to compare the result from the hash table to “Forw” macro. Comparison succeeded, continue with the next step.

- Compare next 4 bytes of the header field name (“ards”), convert it to integer and compare to “ards” macro. Comparison succeeded, continue with the next step.
- If the following characters are spaces and tabs followed by a colon (or colon directly without spaces and tabs), we found Max-Forwards header field name and can set *type* field to HDR_MAXFORWARDS. Otherwise (other characters than colon, spaces and tabs) it is some other header field and set *type* field to HDR_OTHER.

In this example, we had to do 2 hash table lookups and 2 more comparisons. Even this variant is still very fast, because the hash table lookup is synonym-less, lookups are very fast.

6.2.3. The Header Field Body Parsers

6.2.3.1. To HF Body Parser

Purpose of this parser is to parse body of To header field. The parser can be found in file `parse_to.c` under `parser` subdirectory.

Main function is `parse_to` but there is no need to call the function explicitly. Every time the parser finds a To header field, this function will be called automatically. Result of the parser is to_body structure. Pointer to the structure will be stored in *parsed* field of hdr_field structure. Since the pointer is void*, there is a convenience macro `get_to` in file `parse_to.h` that will do the necessary type-casting and will return pointer to to_body structure.

The parser itself is a finite state machine that will parse To body according to the grammar defined in RFC3261 and store result in to_body structure.

The parser gets called automatically from function `get_hdr_field` in file `msg_parser.c`. The function first creates and initializes an instance of to_body structure, then calls `parse_to` function with the structure as a parameter and if everything went OK, puts the pointer to the structure in *parsed* field of hdr_field structure representing the parsed To header field.

The newly created structure will be freed when the message is being destroyed, see function `clean_hdr_field` in file `hf.c` for more details.

6.2.3.1.1. Structure to_body

The structure represents parsed To body. The structure is declared in `parse_to.h` file.

Structure Declaration:

```
struct to_param{
    int type;              /* Type of parameter */
    str name;              /* Name of parameter */
    str value;             /* Parameter value */
    struct to_param* next; /* Next parameter in the list */
};


struct to_body{
    int error;                    /* Error code */
    str body;                     /* The whole header field body */
    str uri;                      /* URI */
    str tag_value;                /* Value of tag */
    struct to_param *param_lst;   /* Linked list of parameters */
    struct to_param *last_param;  /* Last parameter in the list */
};
```

Structure to_param is a temporary structure representing a To URI parameter. Right now only TAG parameter will be marked in *`type`* field. All other parameters will have the same type.

Field Description:

- *`error`* - Error code will be put here when parsing of To body fails.
- *`body`* - The whole header field body.
- *`uri`* - URI of the To header field.
- *`tag_value`* - Value of tag parameter if present.
- *`param_lst`* - Linked list of all parameters.
- *`last_param`* - Pointer to the last parameter in the linked list.

6.2.3.2. From HF Body Parser

This parser is only a wrapper to the To header field parser. Since bodies of both header fields are identical, From parser only calls To parser.

The wrapper can be found in file `parse_from.c` under `parser` subdirectory. There is only one function called `parse_from_header`. The function accepts one parameter which is pointer to structure representing the From header field to be parsed. The function creates an instance of to_body structure and initializes it. It then calls `parse_to` function and if everything went OK, the pointer to the newly created structure will be put in *`parsed`* field of the structure representing the parsed header field.

The newly created structure will be freed when the whole message is being destroyed. (See To header field parser description for more details).

From parser *must be called explicitly* !

If the main parser finds a From header field, it will not parse the header field body automatically. It is up to you to call the `parse_from_header` when you want to parse a From header field body.

6.2.3.3. CSeq HF Body Parser

Purpose of this parser is to parse body of CSeq header field. The parser can be found in file `parse_cseq.c` under `parser` subdirectory.

Main function is `parse_cseq` but there is no need to call the function explicitly. Every time the parser finds a CSeq header field, this function will be called automatically. Result of the parser is cseq_body structure. Pointer to the structure will be stored in *`parsed`* field of hdr_field structure. Since the pointer is void*, there is a convenience macro `get_cseq` in file `parse_cseq.h` that will do the necessary type-casting and will return pointer to cseq_body structure.

The parser will parse CSeq body according to the grammar defined in RFC3261 and store result in cseq_body structure.

The parser gets called automatically from function `get_hdr_field` in file `msg_parser.c`. The function first creates and initializes an instance of cseq_body structure, then calls `parse_cseq` function with the structure as a parameter and if everything went OK, puts the pointer to the structure in *`parsed`* field of hdr_field structure representing the parsed CSeq header field.

The newly created structure will be freed when the message is being destroyed, see function `clean_hdr_field` in file `hf.c` for more details.

6.2.3.3.1. Structure cseq_body

The structure represents parsed CSeq body. The structure is declared in `parse_cseq.h` file.

Structure Declaration:

```
struct cseq_body{
    int error;  /* Error code */
    str number; /* CSeq number */
    str method; /* Associated method */
};
```

Field Description:

- *`error`* - Error code will be put here when parsing of CSeq body fails.
- *`number`* - CSeq number as string.

- *method* - CSeq method.

6.2.3.4. Event HF Body Parser

Purpose of this parser is to parse body of an Event Header field. The parser can be found in file `parse_event.c` under `parser` subdirectory.

> **Note:** This is *NOT* fully featured Event body parser ! The parser was written for Presence Agent module only and thus can recognize Presence package only. No subpackages will be recognized. All other packages will be marked as "OTHER".
>
> The parser should be replace by a more generic parser if subpackages or parameters should be parsed too.

Main function is `parse_event` in file `parse_event.c`. The function will create an instance of event_t structure and call the parser. If everything went OK, pointer to the newly created structure will be stored in *parsed* field of hdr_field structure representing the parsed header field.

As usually, the newly created structure will be freed when the whole message is being destroyed. See function `clean_hdr_field` in file `hf.c`.

The parser will be not called automatically when the main parser finds an Event header field. It is up to you to call the parser when you really need the body of the header field to be parsed (call `parse_event` function).

6.2.3.4.1. Structure event_t

The structure represents parsed Event body. The structure is declared in `parse_event.h` file.

Structure Declaration:

```
#define EVENT_OTHER    0
#define EVENT_PRESENCE 1

typedef struct event {
    str text;   /* Original string representation */
    int parsed; /* Parsed variant */
} event_t;
```

Field Description:

- *text* - Package name as text.
- *parsed* - Package name as integer. It will be EVENT_PRESENCE for presence package and EVENT_OTHER for rest.

6.2.3.5. Expires HF Body Parser

The parser parses body of Expires header field. The body is very simple, it consists of number only. so the parser only removes any leading tabs and spaces and converts the number from string to integer. That's it.

The parser can be found in file `parse_expires.c` under `parser` subdirectory. Main function is `parse_expires`. The function is not called automatically when an Expires header field was found. It is up to you to call the function if you need the body to be parsed.

The function creates a new instance of exp_body_t structure and calls the parser. If everything went OK, pointer to the newly created structure will be saved in *`parsed`* field of the hdr_field structure representing the parsed header field.

6.2.3.5.1. Structure exp_body_t

The structure represents parsed Expires body. The structure is declared in `parse_expires.h` file.

Structure Declaration:

```
typedef struct exp_body {
    str text;   /* Original text representation */
    int val;    /* Parsed value */
} exp_body_t;
```

Field Description:

- *`text`* - Expires value as text.
- *`val`* - Expires value as integer.

6.2.3.6. Via HF Body Parser

Purpose of this parser is to parse body of Via header field. The parser can be found in file `parse_via.c` under `parser` subdirectory.

Main function is `parse_via` but there is no need to call the function explicitly. Every time the parser finds a Via header field, this function will be called automatically. Result of the parser is via_body structure. Pointer to the structure will be stored in *`parsed`* field of hdr_field structure representing the parsed header field.

The parser itself is a finite state machine that will parse Via body according to the grammar defined in RFC3261 and store result in via_body structure.

The parser gets called automatically from function `get_hdr_field` in file `msg_parser.c`. The function first creates and initializes an instance of via_body structure, then calls `parse_via` function with the structure as a parameter and if everything went OK, puts the pointer to the structure in *`parsed`* field of hdr_field structure representing the parsed Via header field.

The newly created structure will be freed when the message is being destroyed, see function `clean_hdr_field` in file `hf.c` for more details.

Structure via_body is described in section Structure via_body.

6.2.3.7. Contact HF Body Parser

The parser is located under `parser/contact` subdirectory. The parser is not called automatically when the main parser finds a Contact header field. It is your responsibility to call the parser if you want a Contact header field body to be parsed.

Main function is `parse_contact` in file `parse_contact.c`. The function accepts one parameter which is structure hdr_field representing the header field to be parsed. A single Contact header field may contain multiple contacts, the parser will parse all of them and will create linked list of all such contacts.

The function creates and initializes an instance of contact_body structure. Then function `contact_parser` will be called. If everything went OK, pointer to the newly created structure will be stored in *`parsed`* field of the hdr_field structure representing the parsed header field.

Function `contact_parser` will then check if the contact is star, if not it will call `parse_contacts` function that will parse all contacts of the header field.

Function `parse_contacts` can be found in file `contact.c`. It extracts URI and parses all contact parameters.

The Contact parameter parser can be found in file `cparam.c`.

The following structures will be created during parsing:

> **Note:** Mind that none of string in the following structures is zero terminated ! Be very careful when processing the strings with functions that require zero termination (printf for example) !

```
typedef struct contact_body {
    unsigned char star;    /* Star contact */
    contact_t* contacts;   /* List of contacts */
} contact_body_t;
```

This is the main structure. Pointer to instance of this structure will be stored in *`parsed`* field of structure representing the header field to be parsed. The structure contains two field:

- *`star`* field - This field will contain 1 if the Contact was star (see RFC3261 for more details).
- *`contacts`* field - This field contains pointer to linked list of all contacts found in the Contact header field.

```
typedef struct contact {
    str uri;              /* contact uri */
    cparam_t* q;          /* q parameter hook */
    cparam_t* expires;    /* expires parameter hook */
    cparam_t* method;     /* method parameter hook */
    cparam_t* params;     /* List of all parameters */
    struct contact* next; /* Next contact in the list */
} contact_t;
```

This structure represents one Contact (Mind that there might be several contacts in one Contact header field delimited by a comma). Its fields have the following meaning:

- *uri* - This field contains pointer to begin of URI and its length.
- *q* - This is a hook to structure representing q parameter. If there is no such parameter, the hook contains 0.
- *expires* - This is a hook to structure representing expires parameter. If there is no such parameter, the hook contains 0.
- *method* - This is a hook to structure representing method parameter. If there is no such parameter, the hook contains 0.
- *params* - Linked list of all parameters.
- *next* - Pointer to the next contact that was in the same header field.

```
typedef enum cptype {
    CP_OTHER = 0,  /* Unknown parameter */
    CP_Q,          /* Q parameter */
    CP_EXPIRES,    /* Expires parameter */
    CP_METHOD      /* Method parameter */
} cptype_t;
```

This is an enum of recognized types of contact parameters. Q parameter will have type set to CP_Q, Expires parameter will have type set to CP_EXPIRES and Method parameter will have type set to CP_METHOD. All other parameters will have type set to CP_OTHER.

```
/*
 * Structure representing a contact
 */
typedef struct cparam {
    cptype_t type;       /* Type of the parameter */
    str name;            /* Parameter name */
    str body;            /* Parameter body */
    struct cparam* next; /* Next parameter in the list */
} cparam_t;
```

This structure represents a contact parameter. Field description follows:

- *type* - Type of the parameter, see cptype enum for more details.
- *name* - Name of the parameter (i.e. the part before “=”).

- *body* - Body of the parameter (i.e. the part after “=”).
- *next* - Next parameter in the linked list.

6.2.3.8. Digest Body Parser

Purpose of this parser is to parse digest response. The parser can be found under `parser/digest` subdirectory. There might be several header fields containing digest response, for example Proxy-Authorization or WWW-Authorization. The parser can be used for all of them.

The parser is not called automatically when by the main parser. It is your responsibility to call the parser when you want a digest response to be parsed.

Main function is `parse_credentials` defined in `digest.c`. The function accepts one parameter which is header field to be parsed. As result the function will create an instance of auth_body_t structure which will represent the parsed digest credentials. Pointer to the structure will be put in *parsed* field of the hdr_field structure representing the parsed header field. It will be freed when the whole message is being destroyed.

The digest parser contains 32-bit digest parameter parser. The parser was in detail described in section Header Field Name Parser. See that section for more details about the digest parameter parser algorithm, they work in the same way.

Description of digest related structures follows:

```
typedef struct auth_body {
    /* This is pointer to header field containing
     * parsed authorized digest credentials. This
     * pointer is set in sip_msg->{authorization,proxy_auth}
     * hooks.
     *
     * This is necessary for functions called after
     * {www,proxy}_authorize, these functions need to know
     * which credentials are authorized and they will simply
     * look into
     * sip_msg->{authorization,proxy_auth}->parsed->authorized
     */
    struct hdr_field* authorized;
    dig_cred_t digest;           /* Parsed digest credentials */
    unsigned char stale;         /* Flag is set if nonce is stale */
    int nonce_retries;           /* How many times the nonce was used */
} auth_body_t;
```

This is the “main” structure. Pointer to the structure will be stored in *parsed* field of hdr_field structure. Detailed description of its fields follows:

- *authorized* - This is a hook to header field containing authorized credentials.

A SIP message may contain several credentials. They are distinguished using realm parameter. When the server is trying to authorize the message, it must first find credentials with corresponding realm and than authorize the credentials. To authorize credentials server calculates response string and if the string matches to response string contained in the credentials, credentials are authorized (in fact it means that the user specified in the credentials knows password, nothing more, nothing less).

It would be good idea to remember which credentials contained in the message are authorized, there might be other functions interested in knowing which credentials are authorized.

That is what is this field for. A function that successfully authorized credentials (currently there is only one such function in the server, it is function `authorize` in auth module) will put pointer to header field containing the authorized credentials in this field. Because there might be several header field containing credentials, the pointer will be put in *`authorized`* field in the first header field in the message containing credentials. That means that it will be either header field whose pointer is in *`www_auth`* or *`proxy_auth`* field of sip_msg structure representing the message.

When a function wants to find authorized credentials, it will simply look in *`msg->www_auth->parsed->authorized`* or *`msg->proxy_auth->parsed->authorized`*, where *`msg`* is variable containing pointer to sip_msg structure.

To simplify the task of saving and retrieving pointer to authorized credentials, there are two convenience functions defined in `digest.c` file. They will be described later.

- *`digest`* - Structure containing parsed digest credentials. The structure will be described in detail later.
- *`stale`* - This field will be set to 1 if the server received a stale nonce. Next time when the server will be sending another challenge, it will use “stale=true” parameter. “stale=true” indicates to the client that username and password used to calculate response were correct, but nonce was stale. The client should recalculate response with the same username and password (without disturbing user) and new nonce. For more details see RFC2617.
- *`nonce_retries`* - This fields indicates number of authorization attempts with same nonce.

```
/*
 * Errors returned by check_dig_cred
 */
typedef enum dig_err {
    E_DIG_OK = 0,        /* Everything is OK */
    E_DIG_USERNAME  = 1, /* Username missing */
    E_DIG_REALM = 2,     /* Realm missing */
    E_DIG_NONCE = 4,     /* Nonce value missing */
    E_DIG_URI = 8,       /* URI missing */
    E_DIG_RESPONSE = 16, /* Response missing */
    E_DIG_CNONCE = 32,   /* CNONCE missing */
    E_DIG_NC = 64,       /* Nonce-count missing */
} dig_err_t;
```

This is enum of all possible errors returned by `check_dig_cred` function.

- *E_DIG_OK* - No error found.
- *E_DIG_USERNAME* - Username parameter missing in digest response.
- *E_DIG_REALM* - Realm parameter missing in digest response.
- *E_DIG_NONCE* - Nonce parameter missing in digest response.
- *E_DIG_URI* - Uri parameter missing in digest response.
- *E_DIG_RESPONSE* - Response parameter missing in digest response.
- *E_DIG_CNONCE* - Cnonce parameter missing in digest response.
- *E_DIG_NC* - Nc parameter missing in digest response.

```
/* Type of algorithm used */
typedef enum alg {
    ALG_UNSPEC = 0,   /* Algorithm parameter not specified */
    ALG_MD5 = 1,      /* MD5 - default value*/
    ALG_MD5SESS = 2,  /* MD5-Session */
    ALG_OTHER = 4     /* Unknown */
} alg_t;
```

This is enum of recognized algorithm types. (See description of algorithm structure for more details).

- *ALG_UNSPEC* - Algorithm was not specified in digest response.
- *ALG_MD5* - “algorithm=MD5” was found in digest response.
- *ALG_MD5SESS* - “algorithm=MD5-Session” was found in digest response.
- *ALG_OTHER* - Unknown algorithm parameter value was found in digest response.

```
/* Quality Of Protection used */
typedef enum qop_type {
    QOP_UNSPEC = 0,   /* QOP parameter not present in response */
    QOP_AUTH = 1,     /* Authentication only */
    QOP_AUTHINT = 2,  /* Authentication with integrity checks */
    QOP_OTHER = 4     /* Unknown */
} qop_type_t;
```

This enum lists all recognized qop parameter values.

- *QOP_UNSPEC* - qop parameter was not found in digest response.
- *QOP_AUTH* - “qop=auth” was found in digest response.
- *QOP_AUTHINT* - “qop=auth-int” was found in digest response.
- *QOP_OTHER* - Unknown qop parameter value was found in digest response.

```
/* Algorithm structure */
struct algorithm {
```

```
    str alg_str;       /* The original string representation */
    alg_t alg_parsed;  /* Parsed value */
};
```

The structure represents “algorithm” parameter of digest response. Description of fields follows:

- *alg_str* - Algorithm parameter value as string.
- *alg_parsed* - Parsed algorithm parameter value.

```
/* QOP structure */
struct qp {
    str qop_str;           /* The original string representation */
    qop_type_t qop_parsed; /* Parsed value */
};
```

This structure represents “qop” parameter of digest response. Description of fields follows:

- *qop_str* - Qop parameter value as string.
- *qop_parsed* - Parsed “qop” parameter value.

```
/*
 * Parsed digest credentials
 */
typedef struct dig_cred {
    str username;         /* Username */
    str realm;            /* Realm */
    str nonce;            /* Nonce value */
    str uri;              /* URI */
    str response;         /* Response string */
    str algorithm;        /* Algorithm in string representation */
    struct algorithm alg; /* Type of algorithm used */
    str cnonce;           /* Cnonce value */
    str opaque;           /* Opaque data string */
    struct qp qop;        /* Quality Of Protection */
    str nc;               /* Nonce count parameter */
} dig_cred_t;
```

This structure represents set of digest credentials parameters. Description of field follows:

- *username* - Value of “username” parameter.
- *realm* - Value of “realm” parameter.
- *nonce* - Value of “nonce” parameter.
- *uri* - Value of “uri” parameter.
- *response* - Value of “response” parameter.

- *algorithm* - Value of “algorithm” parameter as string.
- *alg* - Parsed value of “algorithm” parameter.
- *cnonce* - Value of “cnonce” parameter.
- *opaque* - Value of “opaque” parameter.
- *qop* - Value of “qop” parameter.
- *nc* - Value of “nc” parameter.

6.2.3.8.1. Other Functions Of the Digest Body Parser

There are some other mainly convenience functions defined in the parser. The function will be in detail described in this section. All the functions are defined in `digest.c` file.

```
dig_err_t check_dig_cred(dig_cred_t* _c);
```

This function performs some basic sanity check over parsed digest credentials. The following conditions must be met for the checks to be successful:

- There must be non-empty “username” parameter in the credentials.
- There must be non-empty “realm” parameter in the credentials.
- There must be non-empty “nonce” parameter in the credentials.
- There must be non-empty “uri” parameter in the credentials.
- There must be non-empty “response” parameter in the credentials.
- If qop parameter is set to QOP_AUTH or QOP_AUTHINT, then there must be also non-empty “cnonce” and “nc” parameters in the digest.

> **Note:** It is recommended to call `check_dig_cred` before you try to authorize the credentials. If the function fails, there is no need to try to authorize the credentials because the authorization will fail for sure.

```
int mark_authorized_cred(struct sip_msg* _m, struct hdr_field* _h);
```

This is convenience function. The function saves pointer to the authorized credentials. For more info see description of *authorized* field in auth_body structure.

```
int get_authorized_cred(struct sip_msg* _m, struct hdr_field** _h);
```

This is convenience function. The function will retrieve pointer to authorized credentials previously saved using `mark_authorized_cred` function. If there is no such credentials, 0 will be stored in variable pointed to by the second parameter. The function returns always zero. For more information see description of *`authorized`* field in auth_body structure.

Chapter 7. The Module Interface

The server can load additional functionality through modules. Module loading related functions and module interface will be described in this section.

All the data structures and functions mentioned in this section can be found in files `sr_module.h` and `sr_module.c`.

7.1. Structure sr_module

Each loaded module is represented by an instance of sr_module structure. All the instances are linked. There is a global variable `modules` defined in file `sr_module.c` which is head of linked-list of all loaded modules.

Detailed description of the structure follows:

```
struct sr_module{
    char* path;
    void* handle;
    struct module_exports* exports;
    struct sr_module* next;
};
```

Fields and their description:

- *`path`* - Path to the module. This is the path you pass as parameter to `loadmodule` function in the config file.
- *`handle`* - Handle returned by `dlopen`.
- *`exports`* - Pointer to structure describing interface of the module (will be described later).
- *`next`* - Pointer to the next sr_module structure in the linked list.

7.2. Structure module_exports

This structure describes interface that must be exported by each module. Every module must have a global variable named `exports` which is of type struct module_exports.

Immediately after `dlopen` the server will try to find symbol named `exports` in the module to be loaded. This symbol is a structure describing interface of the module. Pointer to the symbol will be then put in *`exports`* field of sr_module structure representing the module in the server.

Detailed description of the structure follows:

```
struct module_exports{
```

```
    char* name;                      /* null terminated module name */
    char** cmd_names;                /* cmd names registered
                                      * by this modules */
    cmd_function* cmd_pointers;      /* pointers to the
                                      * corresponding functions */
    int* param_no;                   /* number of parameters used by
                                      * the function */
    fixup_function* fixup_pointers; /* pointers to functions
                                      * called to "fix"
                                      * the params, e.g: precompile
                                      * a re */
    int cmd_no;                      /* number of registered commands
                                      * (size of cmd_{names,pointers}
                                      */

    char** param_names;       /* parameter names registered
                               * by this modules */
    modparam_t* param_types; /* Type of parameters */
    void** param_pointers;    /* Pointers to the corresponding
                               * memory locations */
    int par_no;               /* number of registered parameters */

    init_function init_f;         /* Initialization function */
    response_function response_f; /* function used for responses,
                                   * returns yes or no; can be null
                                   */
    destroy_function destroy_f;   /* function called when the module
                                   * should be "destroyed", e.g: on
                                   * ser exit;
                                   * can be null */
    onbreak_function onbreak_f;
    child_init_function init_child_f;  /* function called by all
                                        * processes after the fork */
};
```

Fields and their description:

- *name* - Name of the module.
- *cmd_names* - Array of names of exported commands.
- *cmd_pointers* - Array of pointers to functions implementing commands specified in *cmd_names* array.

  Function Prototype:

  ```
  int cmd_function(struct sip_msg* msg, char* param1, char* param2);
  ```

  The first parameter is sip_msg currently being processed. Remaining parameters are parameters from the config file. If the function accepts only one parameter, *param2* will be set to zero, if the function accepts no parameters, *param1* and *param2* will be set to zero.

The function should return number > 0 if everything went OK and processing of the message should continue. The function should return 0 if processing of the message should be stopped. The function should return number < 0 on an error.

- `param_no` - Array of number of parameters of exported commands.
- `fixup_pointer` - Array of pointers to fixup functions, each fixup function for one exported command. If there is no fixup function for a particular exported function, corresponding field in the array will contain zero.

  Function Prototype:

  ```
  int fixup_function(void** param, int param_no);
  ```

  The first parameter is pointing to variable to be fixed. The second parameter is order of the variable.

  The function should return 0 if everything went OK and number < 0 on an error.

- `cmd_no` - Number of exported commands.

  > **Important:** `cmd_names`, `cmd_pointers`, `param_no` and `fixup_pointer` arrays must have at least `cmd_no` elements ! (It might even kill your cat if you fail to fulfill this condition).

- `param_names` - Array of names of exported parameters.
- `param_types` - Array of types of parameters, each field of the array can be either STR_PARAM or INT_PARAM (currently only two parameter types are defined).
- `param_pointers` - Array of pointers to variables, that hold values of the parameters.
- `param_no` - Number of exported parameters.

  > **Important:** `param_names`, `param_types` and `param_pointers` arrays must have at least `param_no` elements ! (Remember the previous note about your cat ? The same might happen to your dog if you fail to fulfill the condition second time !).

- `init_f` - Pointer to module's initialization function, 0 if the module doesn't need initialization function.

  Function Prototype:

  ```
  int init_function(void);
  ```

  The function should return 0 if everything went OK and number < 0 on an error;

- *response_f* - If a module is interested in seeing responses, it will provide pointer to a function here. The function will be called when a response comes. The field will contain 0 if the module doesn't want to see responses.

  Function Prototype:

  ```
  int response_function(struct sip_msg* msg);
  ```

  The function accepts one parameter which is structure representing the response currently being processed.

  The function should return 0 if the response should be dropped.

- *destroy_f* - Destroy function. The function will be called when the server is shutting down. Can be 0 if the module doesn't need destroy function.

  Function Prototype:

  ```
  void destroy_function(void);
  ```

- *onbreak_f* - On break function. The function will be called when processing of a route statement was aborted. Can be 0 if module doesn't need this function.

  Function Prototype:

  ```
  void onbreak_function(struct sip_msg* msg);
  ```

  The function accepts one parameter which is message currently being processed.

- *init_child_f* - Child initialization function. This is an additional initialization function. *init_f* will be called from the main process *BEFORE* the main process forks children. *init_child_f* will be called from all children *AFTER* the fork.

  Per-child specific initialization can be done here. For example, each child can open its own database connection in the function, and so on.

  Function Prototype:

  ```
  int child_init_function(int rank);
  ```

The function accepts one parameter, which is rank (starting from 0) of child executing the function.

The function should return 0 if everything went OK and number < 0 on an error.

7.3. Module Loading

Modules are compiled and stored as shared objects. Shared objects have usually appendix ".so". Shared objects can be loaded at runtime.

When you instruct the server to load a module using `loadmodule` command in the config file, it will call function `load_module`. The function will do the following:

- It will try to open specified file using `dlopen`. For example if you write loadmodule "/usr/lib/ser/modules/auth.so" in the config file, the server will try to open file "/usr/lib/ser/modules/auth.so" using `dlopen` function.

  If `dlopen` failed, the server will issue an error and abort.

- As the next step, list of all previously loaded modules will be searched for the same module. If such module is found, it means, that user is trying to load the same module twice. In such case an warning will be issued and server will abort.
- The server will try to find pointer to "exports" symbol using `dlsym` in the module. If that fails, server will issue an error and abort.
- And as the last step, function `register_module` will register the module with the server core and loading of the module is complete.

Function `register_module` registers a module with the server core. By registration we mean the following set of steps (see function `register_module` in file `sr_module.c` for more details):

- The function creates and initializes new instance of sr_module structure.
- *`path`* field will be set to path of the module.
- *`handle`* field will be set to handle previously returned by `dlopen`.
- *`exports`* field will be set to pointer to module's `exports` structure previously obtained through `dlsym` in `load_module` function.
- As the last step, the newly created structure will be inserted into linked list of all loaded modules and registration is complete.

7.4. Module Configuration

In addition to set of functions each module can export set of configuration variables. Value of a module's configuration variable can be changed in the config file using `modparam` function. Module configuration will be described in this section.

7.4.1. Function `modparam`

`modparam` function accepts three parameters:

- *module name* - Name of module as exported in *`name`* field of `exports` global variable.
- *variable name* - Name of variable to be set - it must be one of names specified in *`param_names`* field of `exports` variable of the module.
- *value* - New value of the variable. There are two types of variables: string and integer. If the last parameter (value) of `modparam` function is enclosed in quotes, it is string parameter and server will try to find the corresponding variable among string parameters only.

  Otherwise it is integer parameter and server will try to find corresponding variable among integer parameters only.

7.4.2. Function `set_mod_param`

When the server finds `modparam` function in the config file, it will call `set_mod_param` function. The function can be found in `modparam.c` file. The function will do the following:

- It tries to find corresponding variable using `find_param_export` function.
- If it is string parameter, a new copy of the string will be obtained using `strdup` function and pointer to the copy will be stored in the variable.

  If it is integer parameter, its value will be simply copied in the variable.

7.4.3. Function `find_param_export`

This function accepts 3 parameters:

- *module* - Name of module.
- *parameter* - Name of parameter to be found.
- *type* - Type of the parameter.

The function will search list of all modules until it finds module with given name. Then it will search through all module's exported parameters until it finds parameter with corresponding name and type. If such parameter was found, pointer to variable holding the parameter's value will be returned. If the function failed to find either module or parameter with given name and type then zero will be returned.

7.5. Finding an Exported Function

This section describes how to find an exported function.

If you need to find exported function with given name and number of parameters, `find_export` function is what you need. The function is defined in `sr_module.c` file. The function accepts two parameters:

- *name* - Name of function to be found.
- *param_no* - Number of parameters of the function.

The function will search through list of all loaded modules and in each module through array of all exported functions until it finds function with given name and number of parameters. If such exported function was found, `find_exported` will return pointer to the function, otherwise zero will be returned.

7.6. Additional Functions

There are several additional functions defined in file `sr_module.c`. There functions are mostly internal and shouldn't be used directly by user. We will shortly describe them here.

- `register_builtin_modules` - Some modules might be linked statically with main executable, this is handy for debugging. This function will register all such modules upon server startup.
- `init_child` - This function will call child-initialization function of all loaded modules. The function will be called by the server core immediately after the fork.
- `find_module` - The function accepts pointer to an exported function and number of parameters as parameters and returns pointer to corresponding module that exported the function.
- `destroy_modules` - The function will call destroy function of all loaded modules. This function will be called by the server core upon shut down.
- `init_modules` - The function will call initialization function of all loaded modules. The function will be called by the server before the fork.

Chapter 8. The Database Interface

This is a generic database interface for modules that need to utilize a database. The interface should be used by all modules that access database. The interface will be independent of the underlying database server.

> **Note:** If possible, use predefined macros if you need to access any structure attributes.
>
> For additional description, see comments in sources of mysql module.
>
> If you want to see more complicated examples of how the API could be used, see sources of dbexample, usrloc or auth modules.

8.1. Data types

There are several data types. All of them are defined in header files under `db` subdirectory, a client must include `db.h` header file to be able to use them.

8.1.1. Type db_con_t

This type represents a database connection, all database functions (described below) use a variable of this type as one argument. In other words, variable of db_con_t type serves as a handle for a particular database connection.

```
typedef struct db_con {
    char* table;     /* Default table to use */
    void* con;       /* Database connection */
    void* res;       /* Result of previous operation */
    void* row;       /* Internal, not for public use */
    int connected;   /* 1 if connection is established */
} db_con_t;
```

There are no macros defined for db_con_t type.

8.1.2. Type db_key_t

This type represents a database key. Every time you need to specify a key value, this type should be used. In fact, this type is identical to const char*.

```
    typedef const char* db_key_t;
```

There are no macros defined (they are not needed).

8.1.3. Type db_type_t

Each cell in a database table can be of a different type. To distinguish among these types, the db_type_t enumeration is used. Every value of the enumeration represents one datatype that is recognized by the database API. This enumeration is used in conjunction with db_type_t. For more information, see the next section.

```
typedef enum {
    DB_INT,       /* Integer number */
    DB_DOUBLE,    /* Decimal number */
    DB_STRING,    /* String */
    DB_STR,       /* str structure */
    DB_DATETIME   /* Date and time */
    DB_BLOB       /* Binary large object */
} db_type_t;
```

There are no macros defined.

8.1.4. Type db_val_t

This structure represents a value in the database. Several datatypes are recognized and converted by the database API:

- *DB_INT* - Value in the database represents an integer number.
- *DB_DOUBLE* - Value in the database represents a decimal number.
- *DB_STRING* - Value in the database represents a string.
- *DB_STR* - Value in the database represents a string.
- *DB_DATETIME* - Value in the database represents date and time.
- *DB_BLOB* - Value in the database represents binary large object.

These datatypes are automatically recognized, converted from internal database representation and stored in a variable of corresponding type.

```
typedef struct db_val {
    db_type_t type;              /* Type of the value */
    int nul;                     /* NULL flag */
    union {
        int int_val;             /* Integer value */
        double double_val;       /* Double value */
        time_t time_val;         /* Unix time_t value */
        const char* string_val;  /* Zero terminated string */
        str str_val;             /* str structure */
        str blob_val;            /* Structure describing blob */
    } val;
} db_val_t;
```

> **Note:** All macros expect pinter to db_val_t variable as a parameter.

- `VAL_TYPE(value)` Macro.

  Use this macro if you need to set/get the type of the value

  Example 8-1. VAL_TYPE Macro

  ```
  ...
  VAL_TYPE(val) = DB_INT;
  if (VAL_TYPE(val) == DB_FLOAT)
  ...
  ```

- `VAL_NULL(value)` Macro.

  Use this macro if you need to set/get the null flag. Non-zero flag means that the corresponding cell in the database contained no data (NULL value in MySQL terminology).

  Example 8-2. VAL_NULL Macro

  ```
  ...
  if (VAL_NULL(val) == 1) {
      printf("The cell is NULL");
  }
  ...
  ```

- `VAL_INT(value)` Macro.

  Use this macro if you need to access integer value in db_val_t structure.

  Example 8-3. VAL_INT Macro

  ```
  ...
  if (VAL_TYPE(val) == DB_INT) {
      printf("%d", VAL_INT(val));
  }
  ...
  ```

- `VAL_DOUBLE(value)` Macro.

  Use this macro if you need to access double value in the db_val_t structure.

  Example 8-4. VAL_DOUBLE Macro

  ```
  ...
  if (VAL_TYPE(val) == DB_DOUBLE) {
      printf("%f", VAL_DOUBLE(val));
  }
  ...
  ```

- `VAL_TIME(value)` Macro.

  Use this macro if you need to access time_t value in db_val_t structure.

  Example 8-5. VAL_TIME Macro

```
...
time_t tim;
if (VAL_TYPE(val) == DB_DATETIME) {
    tim = VAL_TIME(val);
}
...
```

- `VAL_STRING(value)` Macro.

  Use this macro if you need to access string value in db_val_t structure.

  Example 8-6. VAL_STRING Macro

```
...
if (VAL_TYPE(val) == DB_STRING) {
    printf("%s", VAL_STRING(val));
}
...
```

- `VAL_STR(value)` Macro.

  Use this macro if you need to access str structure in db_val_t structure.

  Example 8-7. VAL_STR Macro

```
...
if (VAL_TYPE(val) == DB_STR) {
    printf("%.*s", VAL_STR(val).len, VAL_STR(val).s);
}
...
```

- `VAL_BLOB(value)` Macro.

  Use this macro if you need to access blob value in db_val_t structure.

  Example 8-8. VAL_STR Macro

```
...
if (VAL_TYPE(val) == DB_BLOB) {
    printf("%.*s", VAL_BLOB(val).len, VAL_BLOB(val).s);
}
...
```

8.1.5. Type db_row_t

This type represents one row in a database table. In other words, the row is an array of db_val_t variables, where each db_val_t variable represents exactly one cell in the table.

```
typedef struct db_row {
    db_val_t* values;    /* Array of values in the row */
    int n;               /* Number of values in the row */
} db_val_t;
```

- `ROW_VALUES(row)` Macro.

  Use this macro to get pointer to array of db_val_t structures.

  Example 8-9. ROW_VALUES Macro

  ```
  ...
  db_val_t* v = ROW_VALUES(row);
  if (VAL_TYPE(v) == DB_INT)
  ...
  ```

- `ROW_N(row)` Macro.

  Use this macro to get number of cells in a row.

  Example 8-10. ROW_N Macro

  ```
  ...
  db_val_t* val = ROW_VALUES(row);
  for(i = 0; i < ROW_N(row); i++) {
      switch(VAL_TYPE(val + i)) {
          case DB_INT: ...; break;
          case DB_DOUBLE: ...; break;
          ...
      }
  }
  ...
  ```

8.1.6. Type db_res_t

This type represents a result returned by `db_query` function (see below). The result can consist of zero or more rows (see db_row_t description).

> **Note:** A variable of type db_res_t returned by `db_query` function uses dynamically allocated memory, don't forget to call `db_free_query` if you don't need the variable anymore. You will encounter memory leaks if you fail to do this !

In addition to zero or more rows, each db_res_t object contains also an array of db_key_t objects. The objects represent keys (names of columns).

```
typedef struct db_res {
    struct {
        db_key_t* keys;     /* Array of column names */
        db_type_t* types;   /* Array of column types */
        int n;              /* Number of columns */
    } col;
    struct db_row* rows;    /* Array of rows */
    int n;                  /* Number of rows */
} db_res_t;
```

- `RES_NAMES(res)` Macro.

  Use this macro if you want to obtain pointer to an array of cell names.

  Example 8-11. RES_NAMES Macro

  ```
  ...
  db_key_t* column_names = ROW_NAMES(row);
  ...
  ```

- `RES_COL_N(res)` Macro.

  Use this macro if you want to get the number of columns in the result.

  Example 8-12. RES_COL_N Macro

  ```
  ...
  int ncol = RES_COL_N(res);
  for(i = 0; i < ncol; i++) {
      /* do something with the column */
  }
  ...
  ```

- `RES_ROWS(res)` Macro.

  Use this macro if you need to obtain pointer to array of rows.

  Example 8-13. RES_ROWS Macro

  ```
  ...
  db_row_t* rows = RES_ROWS(res);
  ...
  ```

- `RES_ROW_N(res)` Macro.

  Use this macro if you need to obtain the number of rows in the result.

  Example 8-14. RES_ROW_N Macro

  ```
  ...
  int n = RES_ROW_N(res);
  ...
  ```

8.2. Functions

There are several functions that implement the database API logic. All function names start with db_ prefix, except `bind_dbmod`. `bind_dbmod` is implemented in `db.c` file, all other functions are implemented in a standalone module. Detailed description of functions follows.

8.2.1. `bind_dbmod`

This function is special, it's only purpose is to call `find_export` function in the SER core and find addresses of all other functions (starting with db_ prefix). This function *MUST* be called *FIRST* !

```
int bind_dbmod(void);
```

The function takes no parameters.

The function returns 0 if it was able to find addresses of all other functions, otherwise value < 0 is returned.

8.2.2. `db_init`

Use this function to initialize the database API and open a new database connection. This function must be called after `bind_dbmod` but before any other function is called.

```
db_con_t* db_init(const char* _sql_url);
```

The function takes one parameter, the parameter must contain database connection URL. The URL is of the form sql://username:password@host:port/database where:

- *username* - Username to use when logging into database (optional).
- *password* - Password if it was set (optional).

- *host* - Hostname or IP address of the host where database server lives (mandatory).
- *port* - Port number of the server if the port differs from default value (optional).
- *database* - If the database server supports multiple databases, you must specify name of the database (optional).

The function returns pointer to db_con_t* representing the connection if it was successful, otherwise 0 is returned.

8.2.3. db_close

The function closes previously open connection and frees all previously allocated memory. The function `db_close` must be the very last function called.

```
void db_close(db_con_t* _h);
```

The function takes one parameter, this parameter is a pointer to db_con_t structure representing database connection that should be closed.

Function doesn't return anything.

8.2.4. db_query

This function implements SELECT SQL directive.

```
int db_query(db_con_t* _h, db_key_t* _k, db_val_t* _v, db_key_t* _c, int _n, int
_nc, db_key_t* _o, db_res_t** _r);
```

The function takes 8 parameters:

- *_h* - Database connection handle.
- *_k* - Array of column names that will be compared and their values must match.
- *_v* - Array of values, columns specified in _k parameter must match these values.
- *_c* - Array of column names that you are interested in.
- *_n* - Number of key-value pairs to match in _k and _v parameters.
- *_nc* - Number of columns in _c parameter.
- *_o* - Order by.
- *_r* - Address of variable where pointer to the result will be stored.

If _k and _v parameters are NULL and _n is zero, you will get the whole table. If _c is NULL and _nc is zero, you will get all table columns in the result

_r will point to a dynamically allocated structure, it is necessary to call db_free_query function once you are finished with the result.

Strings in the result are not duplicated, they will be discarded if you call db_free_query, make a copy yourself if you need to keep it after db_free_query.

You must call db_free_query *BEFORE* you can call db_query again !

The function returns 0 if everything is OK, otherwise value < 0 is returned.

8.2.5. db_free_query

This function frees all memory allocated previously in `db_query`, it is necessary to call this function for a db_res_t structure if you don't need the structure anymore. You must call this function *BEFORE* you call `db_query` again !

```
int db_free_query(db_con_t* _h, db_res_t* _r);
```

The function takes 2 parameters:

- *_h* - Database connection handle.
- *_r* - Pointer to db_res_t structure to destroy.

The function returns 0 if everything is OK, otherwise the function returns value < 0.

8.2.6. db_insert

This function implements INSERT SQL directive, you can insert one or more rows in a table using this function.

```
int db_insert(db_con_t* _h, db_key_t* _k, db_val_t* _v, int _n);
```

The function takes 4 parameters:

- *_h* - Database connection handle.
- *_k* - Array of keys (column names).
- *_v* - Array of values for keys specified in _k parameter.

- *_n* - Number of keys-value pairs int _k and _v parameters.

The function returns 0 if everything is OK, otherwise the function returns value < 0.

8.2.7. `db_delete`

This function implements DELETE SQL directive, it is possible to delete one or more rows from a table.

```
int db_delete(db_con_t* _h, db_key_t* _k, db_val_t* _v, int _n);
```

The function takes 4 parameters:

- *_h* - Database connection handle.
- *_k* - Array of keys (column names) that will be matched.
- *_v* - Array of values that the row must match to be deleted.
- *_n* - Number of keys-value parameters in _k and _v parameters.

If _k is NULL and _v is NULL and _n is zero, all rows are deleted (table will be empty).

The function returns 0 if everything is OK, otherwise the function returns value < 0.

8.2.8. `db_update`

The function implements UPDATE SQL directive. It is possible to modify one or more rows in a table using this function.

```
int db_update(db_con_t* _h, db_key_t* _k, db_val_t* _v, db_key_t* _uk, db_val_t*
_uv, int _n, int _un);
```

The function takes 7 parameters:

- *_h* - Database connection handle.
- *_k* - Array of keys (column names) that will be matched.
- *_v* - Array of values that the row must match to be modified.
- *_uk* - Array of keys (column names) that will be modified.
- *_uv* - New values for keys specified in _k parameter.
- *_n* - Number of key-value pairs in _k and _v parameters.
- *_un* - Number of key-value pairs in _uk and _uv parameters.

The function returns 0 if everything is OK, otherwise the function returns value < 0.

8.2.9. `db_use_table`

The function db_use_table takes a table name and stores it in db_con_t structure. All subsequent operations (insert, delete, update, query) are performed on that table.

```
int db_use-table(db_con_t* _h, cons char* _t);
```

The function takes 2 parameters:

- *_h* - Database connection handle.
- *_t* - Table name.

The function returns 0 if everything is OK, otherwise the function returns value < 0.

Chapter 9. Basic Modules

9.1. Digest Authentication

The module exports functions needed for digest authentication.

The module depends on:

- *mysql* - Used as interface to database.
- *sl* - Used for stateless replies.

9.1.1. Exported Parameters

- `db_url` - Database url string in form “sql://<user>:<pass>@host/database”.

  Type: string

  Default: “sql://serro:47serro11@localhost/ser”

- `user_column` - Name of column containing usernames in subscriber table.

  Type: string

  Default: “user”

- `realm_column` - Name of column containing realm in subscriber table.

  Type: string

  Default: “realm”

- `password_column` - Name of column containing (plaintext passwords)/(ha1 strings) if calculate_ha1 parameter is set to true/false.

  Type: string

  Default: “ha1”

- `password_column_2` - The parameter can be used if and only if USER_DOMAIN_HACK macro is set in defs.h header file. The column of this name contains alternate ha1 strings calculated from username containing also domain, for example username="jan@iptel.org". This hack is necessary for some broken user agents. The parameter has no meaning if "calculate_ha1" is set to true.

Type: string

Default: “ha1b”

- `secret` - Nonce secret phrase.

  Type: string

  Default: Randomly generated string.

- `group_table` - Name of table containing group definitions.

  Type: string

  Default: “grp”

- `group_user_column` - Name of column containing usernames in group table.

  Type: string

  Default: “user”

- `group_group_column` - Name of column containing groups in group table.

  Type: string

  Default: “grp”

- `calculate_ha1` - If set to true, auth module assumes that “password_column” contains plaintext passwords and ha1 string will be calculated at runtime. If set to false, "password_column" must contain precalculated ha1 strings.

  Type: integer

  Default: false

- `nonce_expire` - Every nonce is valid only for a limited amount of time. This parameter specifies nonce validity interval in seconds.

Type: integer

Default: 300

- `retry_count` - This parameter specifies how many times a user is allowed to retry authentication with incorrect credentials. After that the user will receive 403 Forbidden and must retry with different credentials. This should prevent DoS attacks from misconfigured user agents which try to authenticate with incorrect password again and again and again.

  Type: integer

  Default: 5

9.1.2. Exported Functions

- `int **www_authorize**(struct sip_msg* *msg*, char* *realm*, char* *table*);`

  The function checks credentials in Authorization header field.

  `realm` - Realm string

  `table` - Subscriber table name

  Example 9-1. www_authorize

```
if (!www_authorize( "iptel.org", "subscriber" )) {
    www_challenge(  "iptel.org", "0");
    break;
}
```

- `int **proxy_authorize**(struct sip_msg* *msg*, char* *realm*, char* *table*);`

  The function checks credentials in Proxy-Authorization header field.

  `realm` - Realm string

  `table` - Subscriber table name

Example 9-2. proxy_authorize

```
if (!proxy_authorize( "iptel.org", "subscriber" )) {
    proxy_challenge(  "iptel.org", "0");
    break;
}
```

- `int` **`www_challenge`**`(struct sip_msg*` *`msg`*`, char*` *`realm`*`, char*` *`qop`*`);`

  Challenges a user agent using WWW-Authenticate header field. The second parameter specifies if qop parameter (according to rfc2617) should be used or not. (Turning off is useful primarily to make UAC happy, which have a broken qop implementation, particularly M$ Messenger 4.6).

  `realm` - Realm string

  `qop` - Qop string, “1” means use qop parameter “0” means do not use qop parameter.

- `int` **`proxy_challenge`**`(struct sip_msg*` *`msg`*`, char*` *`realm`*`, char*` *`qop`*`);`

  Challenges a user agent using Proxy-Authenticate header field. The second parameter specifies if qop parameter (according to rfc2617) should be used or not. (Turning off is useful primarily to make UAC happy, which have a broken qop implementation, particularly M$ Messenger 4.6).

  `realm` - Realm string

  `qop` - Qop string, “1” means use qop parameter “0” means do not use qop parameter.

- `int` **`is_user`**`(struct sip_msg*` *`msg`*`, char*` *`username`*`, char*` *`s`*`);`

  Checks if the specified username and matches the username in credentials. Call after *_authorize, otherwise an error will be issued.

  `username` - Username string.

  `s` - Not used.

- `int` **`is_in_group`**`(struct sip_msg*` *`msg`*`, char*` *`group`*`, char*` *`s`*`);`

  Checks if the user specified in credentials is member of given group Call after *_authorize, otherwise an error will be issued.

`group` - Group name.

`s` - Not used.

- `int check_to(struct sip_msg* msg, char* s1, char* s2);`

  Checks if the username given in credentials and username in To header field are equal Call after *_authorize, otherwise an error will be issued.

  `s1` - Not used.

  `s2` - Not used.

  Example 9-3. check_to

```
if (method=="REGISTER" & proxy_authorize("iptel.org", "subscriber" ) {
    if (!check_to) {
        sl_send_reply("403", "cheating: user!=to");
        break;
    }
}
```

- `int check_from(struct sip_msg* msg, char* s1, char* s2);`

  Checks if the username given in credentials and username in From header field are equal. Call after *_authorize, otherwise an error will be issued.

  `s1` - Not used.

  `s2` - Not used.

- `int consume_credentials(struct sip_msg* msg, char* s1, char* s2);`

  Removes previously authorized credentials from the message. The function must be called after {www,proxy}_authorize.

  `s1` - Not used.

  `s2` - Not used.

- `int is_user_in(struct sip_msg* msg, char* hf, char* group);`

Checks, if the user is in specified table.

`hf` - Use username in this header field, the following values are recognized:

- “From” - Extract username from From URI.
- “To” - Extract username from To URI.
- “Request-URI” - Extract username from Request-URI.
- “credentials” - Use username digest parameter.

`group` - Group name.

9.2. Max Forwards

Implements all the operations regarding MAX-Forward header, like adding it (if not present) or decrementing and checking the value of the existent one.

Module dependencies: none.

9.2.1. Exported Parameters

None.

9.2.2. Exported Functions

```
int mf_process_maxfwd_header(struct sip_msg* msg, char* max_value, char* s);
```

If no Max-Forwards header is present in the received request, a header will be added having the original value equal with "max_value". An OK code is returned by the function.

If a Max-Forwards header is already present, its value will be decremented. If after this operation its value will be positive non-zero, an OK code will be returned. Otherwise (for a zero value) an error code will be returned. Note that an error code will be also returned if the SIP message couldn’t be parsed or if the Max-Forwards header’s body invalid (non numerical string or negative numerical value)

Parameter s is not used.

9.3. Registrar

The module contains REGISTER processing logic.

The module depends on:

- *usrloc* - User location module.
- *sl* - Used for stateless replies.

9.3.1. Exported Parameters

- `default_expires` - If the processed message contains neither Expires HFs nor expires contact parameters, this value will be used for newly created usrloc records. The parameter contains number of second to expire (for example use 3600 for one hour).

  Type: integer

  Default: 3600

- `default_q` - The parameter represents default q value for new contacts. Because ser doesn't support float parameter types, the value in the parameter is divided by 100 and stored as float. For example, if you want default_q to be 0.38, use value 38 here.

  Type: integer

  Default: 0

- `append_branches` - The parameter controls how lookup function processes multiple contacts. If there are multiple contacts for the given username in usrloc and this parameter is set to 1, Request-URI will be overwritten with the highest-q rated contact and the rest will be appended to sip_msg structure and can be later used by tm for forking. If the parameter is set to 0, only Request-URI will be overwritten with the highest-q rated contact and the rest will be left unprocessed.

  Type: integer

  Default: 1

9.3.2. Exported Functions

-

```
int save(struct sip_msg* msg, char* table, char* s);
```

The function processes a REGISTER message. It can add, remove or modify usrloc records depending on Contact and Expires HFs in the REGISTER message. On success, 200 OK will be returned listing all contacts that are currently in usrloc. On an error, error message will be send with a short description in reason phrase.

`table` - Table name where contacts should be stored.

`s` - Not used.

- `int` **`lookup`**`(struct sip_msg*` *`msg`*`, char*` *`table`*`, char*` *`s`*`);`

  The functions extracts username from Request-URI and tries to find all contacts for the username in usrloc. If there are no such contacts, -1 will be returned. If there are such contacts, Request-URI will be overwritten with the contact that has the highest q value and optionally the rest will be appended to the message (depending on append_branches parameter value).

  `table` - Name of table that should be used for the lookup.

  `s` - Not used.

9.4. Record-Routing

Record Routing module.

The module depends on: None.

9.4.1. Exported Parameters

None.

9.4.2. Exported Functions

- `int` **`rewriteFromRoute`**`(struct sip_msg*` *`msg`*`, char*` *`s1`*`, char*` *`s2`*`);`

  If there are any Route HFs in the message, the function takes the first one, rewrites Request-URI with it's value and removes the first URI from Route HFs.

  `s1` - Not used.

`s2` - Not used.

- `int addRecordRoute(struct sip_msg* msg, char* s1, char* s2);`

  The function adds a new Record-Route header field. The header field will be inserted in the message before any other Record-Route header fields.

  `s1` - Not used.

  `s2` - Not used.

9.5. Stateless Replies

The SL module provide possibilities to send stateless replies for the current request. Additional, a filter function is available for checking the ACK requests generated by sending a SL reply to a INVITE req. Also, offers the possibility to send explanatory SL replies in case of an internal error.

For the SL module to be able to recognize the ACKs generated by INVITE's replies sent by itself, all the sent replies will have attached to TO header a "tag" param. having a unique value (that was random generated at ser startup).When an ACK is received, the TO tag param tag is checked if corresponds - all UAS MUST copy the TO tag from replies into ACK requests! TO speed up the filtering process, the module uses a timeout mechanism. When a reply is sent, a timer us set. As time as the timer is valid, The incoming ACK requests will be checked using TO tag value Once the timer expires, all the ACK are let through - a long time passed till it sent a reply, so it does not expect any ACK that have to be blocked.

9.5.1. Exported Parameters

None.

9.5.2. Exported Functions

- `int sl_send_reply(struct sip_msg* msg, char* code, char* text_reason);`

  For the current request, a reply is sent back having the given code and text reason. The reply is sent stateless, totally independent of the Transaction module and with no retransmission for the INVITE's replies.

  `code` - Reply code to be used.

`text_reason` - Reason phrase to be used.

- `int sl_filter_ACK(struct sip_msg* msg, char* s1, char* s2);`

  Identifies and blocks the ACK requests generated by INVITE's replies sent using the sl_send_reply() or sl_send_error() functions.

  `s1` - Not used.

  `s2` - Not used.

- `int sl_reply_error(struct sip_msg* msg, char* s1, char* s2);`

  Identifies and blocks the ACK requests generated by INVITE's replies sent using the sl_send_reply() or sl_send_error() functions.

  `s1` - Not used.

  `s2` - Not used.

9.6. Transaction Module

The module implements logic necessary to manage SIP transactions.

The module depends on: None.

TM Module enables stateful processing of SIP transactions. The main use of stateful logic, which is costly in terms of memory and CPU, is some services inherently need state. For example, transaction-based accounting (module acc) needs to process transaction state as opposed to individual messages, and any kinds of forking must be implemented statefully. Other use of stateful processing is it trading CPU caused by retransmission processing for memory. That makes however only sense if CPU consumption per request is huge. For example, if you want to avoid costly DNS resolution for every retransmission of a request to an unresolvable destination, use stateful mode. Then, only the initial message burdens server by DNS queries, subsequent retransmissions will be dropped and will not result in more processes blocked by DNS resolution. The price is more memory consumption and higher processing latency.

From user's perspective, there are two major functions : t_relay and t_relay_to. Both setup transaction state, absorb retransmissions from upstream, generate downstream retransmissions and correlate replies to requests.

t_relay forwards to current uri (be it original request's uri or a uri changed by some of uri-modifying functions, such as sethost). t_relay_to forwards to a specific address.

In general, if TM is used, it copies clones of received SIP messages in shared memory. That costs the memory and also CPU time (memcpys, lookups, shmem locks, etc.) Note that non-TM functions operate over the received message in private memory, that means that any core operations will have no effect on statefully processed messages after creating the transactional state. For example, calling addRecordRoute *after* t_relay is pretty useless, as the RR is added to privately held message whereas its TM clone is being forwarded.

TM is quite big and uneasy to program -- lot of mutexes, shared memory access, malloc & free, timers -- you really need to be careful when you do anything. To simplify TM programming, there is the instrument of callbacks. The callback mechanisms allow programmers to register their functions to specific event. See t_hooks.h for a list of possible events.

Other things programmers may want to know is UAC -- it is a very simplistic code which allows you to generate your own transactions. Particularly useful for things like NOTIFYs or IM gateways. The UAC takes care of all the transaction machinery: retransmissions , FR timeouts, forking, etc. See t_uac prototype in uac.h for more details. Who wants to see the transaction result may register for a callback.

9.6.1. External Usage of TM

There are applications which would like to generate SIP transactions without too big involvement in SIP stack, transaction management, etc. An example of such an application is sending instant messages from a website. To address needs of such apps, SER accepts requests for new transactions via fifo pipes too. If you want to enable this feature, start FIFO server by configuration option fifo="/tmp/filename" Then, an application can easily launch a new transaction by writing a transaction request to this named pipe. The request must follow very simple format, which is

```
:t_uac:[<file_name>]\n
<method>\n
<dst uri>\n
<CR_separated_headers>\n
<body>\n
\n
\n
```

(Filename is to where a report will be dumped. ser assumes /tmp as file's directory.)

A convenience library fifo_uac.c implements this simple functionality. Note the the request write must be atomic, otherwise the request might get intermixes with writes from other writers. You can easily use it via Unix command-line tools, see the following example:

Example 9-4. UAC

```
[jiri@bat jiri]$ cat > /tmp/fifo
:t_uac:xxx
MESSAGE
sip:mrx@iptel.org
header:value
foo:bar
```

```
bznk:hjhjk
p_header: p_value

body body body
yet body
end of body
```

Or use an example file and call cat test/transaction.fifo > /tmp/fifo

9.6.2. Exported Parameters

- `fr_timer` - Timer which hits if no final reply for a request or ACK for a negative INVITE reply arrives.

  Type: integer (seconds)

  Default: 30

- `fr_inv_timer` - Timer which hits if no final reply for an INVITE arrives after a provisional message was received.

  Type: integer (seconds)

  Default: 120

- `wt_timer` - Time for which a transaction stays in memory to absorb delayed messages after it completed; also, when this timer hits, retransmission of local cancels is stopped (a puristic but complex behaviour would be not to enter wait state until local branches are finished by a final reply or FR timer -- we simplified)

  Type: integer (seconds)

  Default: 5

- `delete_timer` - Time after which a to-be-deleted transaction currently ref-ed by a process will be tried to be deleted again.

  Type: integer (seconds)

  Default: 2

- `retr_timer1p1, 2, 3` - Retransmission period.

  Type: integer (seconds)

  Default: RETR_T1=1, 2*RETR_T1, 4*RETR_T1

- `retr_timer2` - Maximum retransmission period.

  Type: integer (seconds)

  Default: RETR_T2=4

- `noisy_ctimer` - If set, on FR timer INVITE transactions will be explicitly cancelled if possible, silently dropped otherwise; preferably, it is turned off to allow very long ringing; this behaviour is overridden if a request is forked, or some functionality explicitly turned it off for a transaction (like acc does to avoid unaccounted transactions due to expired timer).

  Type: integer (boolean)

  Default: 0 (false)

9.6.3. Exported Functions

-

```
int t_relay_to(struct sip_msg* msg, char* ip_address, char* port_number);
```

  Relay a message statefully to a fixed destination; this along with t_relay is the function most users want to use -- all other are mostly for programming; programmers interested in writing TM logic should review how t_relay is implemented in tm.c and how TM callbacks work.

  `ip_address` - IP address of the destination.

  `port_number` - Port of the destination.

  Example 9-5. t_relay_to

```
if (!t_relay_to("1.2.3.4", "5060")) {
    sl_reply_error();
    break;
};
```

-

```
int t_relay(struct sip_msg* msg, char* s1, char* s2);
```

  Relay a message statefully to destination indicated in current URI; (if the original URI was rewritten by UsrLoc, RR, strip/prefix, etc., the new URI will be taken); returns a negative value on failure -- you may still want to send a negative reply upstream statelessly not to leave upstream UAC in lurch.

`s1` - Not used.

`s2` - Not used.

Example 9-6. t_relay

```
if (!t_relay()) {
    sl_reply_error();
    break;
};
```

- `int t_on_negative(struct sip_msg* msg, char* reply_route, char* s);`

  Sets reply routing block, to which control is passed after a transaction completed with a negative result but before sending a final reply; In the referred block, you can either start a new branch (good for services such as forward_on_no_reply) or send a final reply on your own (good for example for message silo, which received a negative reply from upstream and wants to tell upstream "202 I will take care of it"); Note that the set of command which are usable within reply_routes is strictly limited to rewriting URI, initiating new branches, logging, and sending 'unsafe' replies (t_reply_unsafe). Any other commands may result in unpredictable behaviour and possible server failure. Note that whenever reply_route is entered, uri is reset to value which it had on relaying. If it temporarily changed during a reply_route processing, subsequent reply_route will ignore the changed value and use again the original one.

  `reply_route` - Reply routing block.

  `s` - Not used.

  Example 9-7. t_on_negative

```
route {
    t_on_negative("1");
    t_relay();
} reply_route[1] {
    revert_uri();
    setuser("voicemail");
    append_branch();
}
```

- `int t_newtran(struct sip_msg* msg, char* s1, char* s2);`

  Creates a new transaction, returns a negative value on error; this is the only way a script can add a new transaction in an atomic way; typically, it is used to deploy a UAS

  `s1` - Not used.

`s2` - Not used.

Example 9-8. t_newtran

```
if (t_newtran()) {
    log("UAS logic");
    t_reply("999","hello");
} else sl_reply_error();
```

- `int t_reply(struct sip_msg* msg, char* code, char* reason_phrase);`

  Sends a stateful reply after a transaction has been established; see t_newtran for usage; note: never use t_reply from within reply_route ... always use t_reply_unsafe.

  `code` - Code of the response.

  `reason_phrase` - Reason phrase of the response.

- `int t_lookup_request(struct sip_msg* msg, char* s1, char* s2);`

  Checks if a transaction exists; returns a positive value if so, negative otherwise; most likely you will not want to use it, as a typical application of a look-up is to introduce a new transaction if none was found; however this is safely (atomically) done using t_newtran.

  `s1` - Not used.

  `s2` - Not used.

- `int t_retransmit_reply(struct sip_msg* msg, char* s1, char* s2);`

  Retransmits a reply sent previously by UAS transaction.

  `s1` - Not used.

  `s2` - Not used.

- `int t_release(struct sip_msg* msg, char* s1, char* s2);`

  Remove transaction from memory (it will be first put on a wait timer to absorb delayed messages).

`s1` - Not used.

`s2` - Not used.

- `int **t_forward_noack**(struct sip_msg* *msg*, char* *ip*, char* *port*);`

  Mainly for internal -- forward a non-ACK request statefully.

  `ip` - IP address.

  `port` - Port number.

- `int **register_tmcb**(struct sip_msg* *msg*, char* *cb_type*, char* *cb_func*);`

  For programmatic use only -- register a function to be called back on an event; see t_hooks.h for more details.

  `cb_type` - Callback type.

  `cb_func` - Callback function.

- `int **load_tm**(struct sip_msg* *msg*, char* *import_structure*, char* *s*);`

  For programmatic use only -- import exported TM functions; see the acc module for an example of use.

  `import_structure` - Structure where pointers to tm functions will be stored.

  `s` - Not used.

- `int **t_reply_unsafe**(struct sip_msg* *msg*, char* *code*, char* *reason_phrase*);`

  Sends a stateful reply after a transaction has been established; it can only be used from reply processing; using it from regular processing will introduce erroneous conditions; using t_reply from reply_processing will introduce a deadlock.

  `code` - Code of the reply.

  `reason_phrase` - Reason phrase of the reply.

9.6.4. Known Issues

- Need to revisit profiling again.
- Review whether there is not potential for to-tag rewriting and ACK matching.
- We don't have authentication merging on forking.
- Branch tid is not used yet.
- local ACK/CANCELs copy'n'pastes Route and ignores deleted Routes
- 6xx should be delayed.
- Possibly, performance could be improved by not parsing non-INVITEs, as they do not be replied with 100, and do not result in ACK/CANCELs, and other things which take parsing. However, we need to rethink whether we don't need parsed headers later for something else. Remember, when we now conserver a request in sh_mem, we can't apply any pkg_mem operations to it any more. (that might be redesigned too).
- t_replicate should be done more cleanly -- Vias, Routes, etc. should be removed from a message prior to replicating it.
- SNMP support.
- Lookup fails to recognize subsequent requests which have additional leading spaces in header field values.
- Make UAC session-aware (as opposed to just transaction aware) -- needed for keeping SUB-NOT dialog state, etc. Currently, there are only place-holders for in in TM.
- Places labeled with "HACK" strongly deserve beautification.

9.7. User Location Module

Support for location of users.

Module depends on: Optionally mysql (if configured for persistence)

> **Important:** Usrloc is convenient module only. It's functions cannot be called from scripts directly, but are used by registrar module internally. This module will be utilized by more modules in the future and therefore it is standalone. Use registrar module functions if you need usrloc support in your scripts.

9.7.1. Exported Parameters

- `user_col` - Name of column containing usernames.

  Type: string

  Default: "user"

- `contact_col` - Name of column containing contacts.

Type: string

Default: “contact”

- `expires_col` - Name of column containing expires.

  Type: string

  Default: “expires”

- `q_col` - Name of column containing q values.

  Type: string

  Default: “q”

- `callid_col` - Name of column containing callids.

  Type: string

  Default: “callid”

- `cseq_col` - Name of column containing cseq numbers.

  Type: string

  Default: “cseq”

- `method_col` - Name of column containing supported methods.

  Type: string

  Default: “method”

- `db_url` - URL of the database that should be used.

  Type: string

  Default: “sql://ser:heslo@localhost/ser”

- `timer_interval` - Number of seconds between two timer runs. The module uses timer to delete expired contacts, synchronize with database and other tasks, that need to be run periodically.

  Type: integer

  Default: 60 seconds

- `db_mode` - The usrloc module can utilize database for persistent contact storage. If you use database, your contacts will survive machine restarts or sw crashes. The disadvantage is that accessing database can be very time consuming. Therefore, usrloc module implements three database accessing modes:
  - 0 - This disables database completely. Only memory will be used. Contacts will not survive restart. Use this value if you need a really fast usrloc and contact persistence is not necessary or is provided by other means.
  - 1 - *Write-Through* scheme. All changes to usrloc are immediately reflected in database too. This is very slow, but very reliable. Use this scheme if speed is not your priority but need to make sure that no registered contacts will be lost during crash or reboot.
  - 2 - *Write-Back* scheme. This is a combination of previous two schemes. All changes are made to memory and database synchronization is done in the timer. The timer deletes all expired contacts and flushes all modified or new contacts to database. Use this scheme if you encounter high-load peaks and want them to process as fast as possible. The mode will not help at all if the load is high all the time. Also, latency of this mode is much lower than latency of mode 1, but slightly higher than latency of mode 0.

  Type: integer

  Default: 0

  > **Warning**
  >
  > In case of crash or restart contacts that are in memory only and haven't been flushed yet will get lost. If you want minimize the risk, use shorter timer interval.

9.7.2. Exported Functions

-

```
int ul_register_domain(const char* name);
```

  The function registers a new domain. Domain is just another name for table used in registrar. The function is called from fixups in registrar. It gets name of the domain as a parameter and returns pointer to a new domain structure. The fixup than 'fixes' the parameter in registrar so that it will pass the pointer instead of the name every time save() or lookup() is called. Some usrloc functions get the pointer as parameter when called. For more details see implementation of save function in registrar.

  `name` - Name of the domain (also called table) to be registered.

- `int` **`ul_insert_urecord`**`(udomain_t*` *`domain`*`, str*` *`aor`*`, urecord_t**` *`rec`*`);`

  The function creates a new record structure and inserts it in the specified domain. The record is structure that contains all the contacts for belonging to the specified username.

  `domain` - Pointer to domain returned by ul_register_udomain.

  `aor` - Address of Record (aka username) of the new record (at this time the record will contain no contacts yet).

  `rec` - The newly created record structure.

- `int` **`ul_delete_urecord`**`(udomain_t*` *`domain`*`, str*` *`aor`*`);`

  The function deletes all the contacts bound with the given Address Of Record.

  `domain` - Pointer to domain returned by ul_register_udomain.

  `aor` - Address of record (aka username) of the record, that should be deleted.

- `int` **`ul_get_urecord`**`(udomain_t*` *`domain`*`, str*` *`aor`*`);`

  The function returns pointer to record with given Address of Record.

  `domain` - Pointer to domain returned by ul_register_udomain.

  `aor` - Address of Record of request record.

- `int` **`ul_lock_udomain`**`(udomain_t*` *`domain`*`);`

  The function lock the specified domain, it means, that no other processes will be able to access during the time. This prevents race conditions. Scope of the lock is the specified domain, that means, that multiple domain can be accessed simultaneously, they don't block each other.

  `domain` - Domain to be locked.

- `int` **`ul_unlock_udomain`**`(udomain_t*` *`domain`*`);`

Unlock the specified domain previously locked by ul_lock_udomain.

`domain` - Domain to be unlocked.

- `int` **`ul_release_urecord`**`(urecord_t*` *`record`*`);`

  Do some sanity checks - if all contacts have been removed, delete the entire record structure.

  `record` - Record to be released.

- `int` **`ul_insert_ucontact`**`(urecord_t*` *`record`*`, str*` *`contact`*`, time_t` *`expires`*`, float` *`q`*`, str*` *`callid`*`, int` *`cseq`*`, ucontact_t*` *`cont`*`);`

  The function inserts a new contact in the given record with specified parameters.

  `record` - Record in which the contact should be inserted.

  `contact` - Contact URL.

  `expires` - Expires of the contact in absolute value.

  `q` - q value of the contact.

  `callid` - Call-ID of the REGISTER message that contained the contact.

  `cseq` - CSeq of the REGISTER message that contained the contact.

  `cont` - Pointer to newly created structure.

- `int` **`ul_delete_ucontact`**`(urecord_t*` *`record`*`, ucontact_t*` *`contact`*`);`

  The function deletes given contact from record.

  `record` - Record from which the contact should be removed.

  `contact` - Contact to be deleted.

- `int` **`ul_get_ucontact`**`(urecord_t*` *`record`*`, str*` *`contact`*`);`

The function tries to find contact with given Contact URI and returns pointer to structure representing the contact.

`record` - Record to be searched for the contact.

`contact` - URI of the request contact.

- `int` **`ul_update_ucontact`**`(ucontact_t*` *`contact`*`, time_t` *`expires`*`, float` *`q`*`, str*` *`callid`*`, int` *`cseq`*`);`

  The function updates contact with new values.

  `contact` - Contact to be updated.

  `expires` - New expires value.

  `q` - New q value.

  `callid` - New Call-ID.

  `cseq` - New CSeq.