cookbooks:5.3.x:core

−Table of Contents

- auto_bind_ipv6
- bind_ipv6_link_local
- check_via
- children
- chroot
- corelog
- debug
- description
- disable_core_dump
- disable_tls
- enable_tls
- exit_timeout
- flags
- force_rport
- fork
- fork_delay
- group
- http_reply_parse
- ip_free_bind
- kemi.onsend_route_callback
- kemi.received_route_callback
- kemi.reply_route_callback
- latency_cfg_log
- latency_limit_action
- latency_limit_db
- latency_log
- listen
- loadmodule
- loadpath
- log_engine_data
- log_engine_type
- log_facility
- log_name
- log_prefix
- log_prefix_mode
- log_stderror
- cfgengine
- maxbuffer
- max_branches
- max_recursive_level
- max_while_loops
- mcast
- mcast_loopback
- mcast_ttl
- memdbg
- memlog
- mem_join
- mem_safety

Core Cookbook

Version: Kamailio SIP Server v5.3.x (stable)

Overview

This tutorial collects the functions and parameters exported by Kamailio core to configuration file.

Note: The parameters on this page are **NOT** in alphabetical order.

Structure

The structure of the kamailio.cfg can be seen as thee parts:

- global parameters
- modules settings
- routing blocks

For clarity and making it easy to maintain, it is recommended to keep them in this order, although some of them can be mixed.

Global Parameters Section

This is the first part of the configuration file, containing the parameters for the core of kamailio and custom global parameters.

Typically this is formed by directives of the form:

```
name=value
```

The name corresponds to a core parameter as listed in one of the next sections of this document. If a name is not matching a core parameter, then Kamailio will not start, rising an error during startup.

The value is typically an integer, boolean or a string.

Several parameters can get a complex value which is formed from a group of integer, strings or identifiers. For example, such parameter is **listen**, which can be assigned a value like **proto:ipaddress:port**.

Example of content:

```
log_facility=LOG_LOCAL0
```

```
children=4

disable_tcp=yes

alias="sip.mydomain.com"

listen=udp:10.0.0.10:5060
```

Usually setting a parameter is ended by end of line, but it can be also ended with **;** (semicolon). This should be used when the grammar of a parameter allows values on multiple lines (like **listen** or **alias**) and the next line creates a conflict by being swallowed as part of value for previous parameter.

```
alias="sip.mydomain.com";
```

If you want to use a reserved config keyword as part of a parameter, you need to enclose it in quotes. See the example below for the keyword “dns”.

```
listen=tcp:127.0.0.1:5060 advertise "sip.dns.example.com":5060
```

Modules Settings Section

This is the second section of the configuration file, containing the directives to load modules and set their parameters.

It contains the directives **loadmodule** and **modparam**. In the default configuration file starts with the line setting the path to modules (the assignment to **mpath** core parameter.

Example of content:

```
loadmodule "debugger.so"
...
modparam("debugger","cfgtrace",1)
```

Routing Blocks Section

This is the last section of the configuration file, typically the biggest one, containing the routing blocks with the routing logic for SIP traffic handled by Kamailio.

The only mandatory routing block is **request_route**, which contains the actions for deciding the routing for SIP requests.

See the chapter **Routing Blocks** in this document for more details about what types of routing blocks can be used in the configuration file and their role in routing SIP traffic and Kamailio behaviour.

Example of content:

```
request_route {

    # per request initial checks
    route(REQINIT);

    ...
}

branch_route[MANAGE_BRANCH]{
    xdbg("new branch [$T_branch_idx] to $ru\n");
    route(NATMANAGE);}
```

Generic Elements

Comments

Line comments start with # (hash/pound character - like in shell) or / / (double forward slash - like in C++/Java).

Block comments start with /* (forward slash and asterisk) and are ended by */ (sterisk and forward slash) (like in C, C++, Java).

Example:

```
# this is a line comment

// this is another line comment

/* this
   is
   a
   block
   comment */
```

Important: be aware of preprocessor directives that start with **#!** (hash/pound and exclamation) - those are no longer line comments.

Values

There are three types of values:

- integer - numbers of 32bit size
- boolean - aliases to 1 (true, on, yes) or 0 (false, off, no)
- string - tokens enclosed in between double or single quotes

Example:

```
// next two are strings

  "this is a string value"'this is another string value'

// next is a boolean

  yes

// next is an integer

  64
```

Identifiers

Identifiers are tokens which are not enclosed in single or double quotes and to match the rules for integer or boolean values.

For example, the identifiers are the core parameters and functions, module functions, core keywords and statements.

Example:

```
return
```

Variables

The variables start with **$** (dollar character).

You can see the list with available variables in the Pseudo-Variables Cookbook.

Example:

```
$var(x)= $rU +"@"+ $fd;
```

Actions

An action is an element used inside routing blocks ended by **;** (semicolon). It can be an execution of a function from core or a module, a conditional or loop statement, an assignment expression.

Example:

```
  sl_send_reply("404","Not found");exit;
```

Expressions

An expression is an association group of statements, variables, functions and operators.

Example:

```
if(!t_relay())

if($var(x)>10)

"sip:"+ $var(prefix)+ $rU +"@"+ $rd
```

Config Pre-Processor Directives

include_file

```
   include_file "path_to_file"
```

Include the content of the file in config before parsing. path_to_file must be a static string. Including file operation is done at startup. If you change the content of included file, you have to restart the SIP server to become effective.

The path_to_file can be relative or absolute. If it is not absolute path, first attempt is to locate it relative to current directory, and if fails, relative to directory of the file that includes it. There is no restriction where include can be used or what can contain - any part of config file is ok. There is a limit of maximum 10 includes in depth, otherwise you can use as many includes as you want. Reporting of the cfg file syntax errors prints now the file name for easier troubleshooting.

If the included file is not found, the config file parser throws error. You can find this error message at the logging destination, usually in the system logging (file).

You can use also the syntax **#!include_file** or **!!include_file**.

Example of usage:

```
route {
    ...
    include_file"/sr/checks.cfg"
    ...
}

---/sr/checks.cfg---

   if(!mf_process_maxfwd_header("10")){
       sl_send_reply("483","Too Many Hops");exit;}

---
```

import_file

```
import_file "path_to_file"
```

Similar to **include_file**, but does not throw error if the included file is not found.

define

Control in C-style what parts of the config file are executed. The parts in non-defined zones are not loaded, ensuring lower memory usage and faster execution.

Available directives:

- **#!define NAME** - define a keyword
- **#!define NAME VALUE** - define a keyword with value
- **#!ifdef NAME** - check if a keyword is defined
- **#!ifndef** - check if a keyword is not defined
- **#!else** - switch to false branch of ifdef/ifndef region
- **#!endif** - end ifdef/ifndef region
- **#!trydef** - add a define if not already defined
- **#!redefine** - force redefinition even if already defined

Among benefits:

- easy way to enable/disable features (e.g., see default cfg – controlling support of nat traversal, presence, etc…)
- switch control for parts where conditional statements were not possible (e.g., global parameters, module settings)
- faster by not using conditional statements inside routing blocks when switching between running environments

Example: how to make config to be used in two environments, say testbed and production, controlled just by one define to switch between the two modes:

```
...

#!define TESTBED_MODE

#!ifdef TESTBED_MODE
  debug=5
  log_stderror=yes
  listen=192.168.1.1
#!else
  debug=2
  log_stderror=no
  listen=10.0.0.1
#!endif

...

#!ifdef TESTBED_MODE
modparam("acc|auth_db|usrloc","db_url","mysql://kamailio:kamailiorw@localhost/kamailio_testbed")#!else
modparam("acc|auth_db|usrloc","db_url","mysql://kamailio:kamailiorw@10.0.0.2/kamailio_production")#!endif

...

#!ifdef TESTBED_MODE
route[DEBUG]{
  xlog("SCRIPT: SIP $rm from: $fu to: $ru - srcip: $si"\n);}#!endif

...

route{#!ifdef TESTBED_MODE
  route(DEBUG);#!endif

  ...
}

...
```

- you can define values for IDs

```
#!define MYINT 123#!define MYSTR "xyz"
```

- defined IDs are replaced at startup, during config parsing, e.g.,:

```
$var(x)=100+ MYINT;
```

- is interpreted as:

```
$var(x)=100+123;
```

- you can have multi-line defined IDs

```
#!define IDLOOP $var(i) = 0; \
```

```
                while($var(i)<5) { \
                    xlog("++++ $var(i)\n"); \
                    $var(i) = $var(i) + 1; \
                }
```

- then in routing block

```
route {
    ...
    IDLOOP
    ...
}
```

- number of allowed defines is now set to 256

- notes:
  - multilines defines are reduced to single line, so line counter should be fine
  - column counter goes inside the define value, but you have to omit the '\' and CR for the accurate inside-define position
  - text on the same line as the directive will cause problems. Keep the directive lines clean and only comment on a line before or after.

subst

- perform substitutions inside the strings of config (note that define is replacing only IDs - alphanumeric tokens not enclosed in quotes)
- #!subst offers an easy way to search and replace inside strings before cfg parsing. E.g.,:

```
#!subst "/regexp/subst/flags"
```

- flags is optional and can be: 'i' - ignore case; 'g' - global replacement

Example:

```
#!subst "/DBPASSWD/xyz/"
modparam("acc","db_url","mysql://user:DBPASSWD@localhost/db")
```

- will do the substitution of db password in db_url parameter value

substdef

```
#!substdef "/ID/subst/"
```

Similar to **subst**, but in addition it adds a **#!define ID subst**.

substdefs

```
#!substdefs "/ID/subst/"
```

Similar to **subst**, but in addition it adds a **#!define ID "subst"** (note the difference from #!substdef that the value for define is enclosed in double quotes, useful when the define is used in a place for a string value).

Core Keywords

Keywords specific to SIP messages which can be used mainly in `if` expressions.

af

The address family of the received SIP message. It is INET if the message was received over IPv4 or INET6 if the message was received over IPv6.

Exampe of usage:

```
if(af==INET6){log("Message received over IPv6 link\n");}
```

dst_ip

The IP of the local interface where the SIP message was received. When the proxy listens on many network interfaces, makes possible to detect which was the one that received the packet.

Example of usage:

```
if(dst_ip==127.0.0.1){log("message received on loopback interface\n");};
```

dst_port

The local port where the SIP packet was received. When Kamailio is listening on many ports, it is useful to learn which was the one that received the SIP packet.

Example of usage:

```
if(dst_port==5061){log("message was received on port 5061\n");};
```

from_uri

This script variable is a reference to the URI of 'From' header. It can be used to test 'From'- header URI value.

Example of usage:

```
if(is_method("INVITE")&& from_uri=~".*@kamailio.org"){log("the caller is from kamailio.org\n");};
```

method

The variable is a reference to the SIP method of the message.

Example of usage:

```
if(method=="REGISTER"){log("this SIP request is a REGISTER message\n");};
```

msg:len

The variable is a reference to the size of the message. It can be used in 'if' constructs to test message's size.

Example of usage:

```
if(msg:len>2048){
        sl_send_reply("413","message too large");exit;};
```

.

proto

This variable can be used to test the transport protocol of the SIP message.

Example of usage:

```
if(proto==UDP){log("SIP message received over UDP\n");};
```

status

If used in onreply_route, this variable is a referece to the status code of the reply. If it used in a standard route block, the variable is a reference to the status of the last reply sent out for the current request.

Example of usage:

```
if(status=="200"){log("this is a 200 OK reply\n");};
```

snd_af

snd_ip

snd_port

snd_proto

src_ip

Reference to source IP address of the SIP message.

Example of usage:

```
if(src_ip==127.0.0.1){log("the message was sent from localhost!\n");};
```

src_port

Reference to source port of the SIP message (from which port the message was sent by previous hop).

Example of usage:

```
if(src_port==5061){log("message sent from port 5061\n");}
```

to_ip

to_port

to_uri

This variable can be used to test the value of URI from To header.

Example of usage:

```
if(to_uri=~"sip:.+@kamailio.org"){log("this is a request for kamailio.org users\n");};
```

uri

This variable can be used to test the value of the request URI.

Example of usage:

```
if(uri=~"sip:.+@kamailio.org"){log("this is a request for kamailio.org users\n");};
```

Core Values

Values that can be used in '`if`' expressions to check against Core Keywords

INET

This keyword can be used to test whether the SIP packet was received over an IPv4 connection.

Example of usage:

```
if(af==INET){log("the SIP message was received over IPv4\n");}
```

INET6

This keyword can be used to test whether the SIP packet was received over an IPv6 connection.

Example of usage:

```
if(af==INET6){log("the SIP message was received over IPv6\n");};
```

SCTP

This keyword can be used to test the value of 'proto' and check whether the SIP packet was received over SCTP or not.

Example of usage:

```
if(proto==SCTP){log("the SIP message was received over SCTP\n");};
```

TCP

This keyword can be used to test the value of 'proto' and check whether the SIP packet was received over TCP or not.

Example of usage:

```
if(proto==TCP){log("the SIP message was received over TCP\n");};
```

TLS

This keyword can be used to test the value of 'proto' and check whether the SIP packet was received over TLS or not.

Example of usage:

```
if(proto==TLS){log("the SIP message was received over TLS\n");};
```

UDP

This keyword can be used to test the value of 'proto' and check whether the SIP packet was received over UDP or not.

Example of usage:

```
if(proto==UDP){log("the SIP message was received over UDP\n");};
```

WS

This keyword can be used to test the value of 'proto' and check whether the SIP packet was received over WS or not.

Example of usage:

```
if(proto==WS){log("the SIP message was received over WS\n");};
```

WSS

This keyword can be used to test the value of 'proto' and check whether the SIP packet was received over WSS or not.

Example of usage:

```
if(proto==WSS){log("the SIP message was received over WSS\n");};
```

max_len

Note: This command was removed.

myself

This is a reference to the list of local IP addresses, hostnames and aliases that has been set in the Kamailio configuration file. This lists contain the domains served by Kamailio.

The variable can be used to test if the host part of an URI is in the list. The usefulness of this test is to select the messages that has to be processed locally or has to be forwarded to another server.

See “alias” to add hostnames,IP addresses and aliases to the list.

Example of usage:

```
if(uri==myself){log("the request is for local processing\n");};
```

Note: You can also use the is_myself() function.

Core parameters

advertised_address

It can be an IP address or string and represents the address advertised in Via header. If empty or not set (default value) the socket address from where the request will be sent is used.

```
WARNING:
- don't set it unless you know what you are doing (e.g. nat traversal)
- you can set anything here, no check is made (e.g. foo.bar will be accepted even if foo.bar doesn't exist)
```

Example of usage:

```
  advertised_address="1.2.3.4"
  advertised_address="kamailio.org"
```

Note: this option may be deprecated and removed in the near future, it is recommended to set **advertise** option for **listen** parameter.

advertised_port

The port advertised in Via header. If empty or not set (default value) the port from where the message will be sent is used. Same warnings as for 'advertised_address'.

Example of usage:

```
  advertised_port=5080
```

Note: this option may be deprecated and removed in the near future, it is recommended to set **advertise** option for **listen** parameter.

alias

Parameter to set alias hostnames for the server. It can be set many times, each value being added in a list to match the hostname when 'myself' is checked.

It is necessary to include the port (the port value used in the “port=” or “listen=” defintions) in the alias definition otherwise the loose_route() function will not work as expected for local forwards. Even if you do not use 'myself' explicitly (for example if you use the domain module), it is often necessary to set the alias as these aliases are used by the loose_routing function and might be needed to handle requests with pre-loaded route set correctly.

Example of usage:

```
alias=other.domain.com:5060
alias=another.domain.com:5060
```

Note: the hostname has to be enclosed in between quotes if it has reserved tokens such as **forward**, **drop** … or operators such as **-** (minus) …

async_workers

Specify how many child processes to create for asynchronous execution. These are processes that can receive tasks from various components and execute them locally, which is different process than the task sender.

Default: 0 (asynchronous framework is disabled).

Example:

```
async_workers=4
```

auto_aliases

Kamailio by default discovers all IPv4 addresses on all interfaces and does a reverse DNS lookup on these addresses to find host names. Discovered host names are added to aliases list, matching the **myself** condition. To disable host names auto-discovery, turn off auto_aliases.

Example:

```
auto_aliases=no
```

auto_bind_ipv6

When turned on, Kamailio will automatically bind to all IPv6 addresses (much like the default behaviour for IPv4). Default is 0.

Example:

```
auto_bind_ipv6=1
```

bind_ipv6_link_local

If set to 1, try to bind also IPv6 link local addresses by discovering the scope of the interface. This apply for UDP socket for now, to be added for the other protocols. Default is 0.

Example:

```
bind_ipv6_link_local=1
```

check_via

Check if the address in top most via of replies is local. Default value is 0 (check disabled).

Example of usage:

```
check_via=1
```

children

Number of children to fork for the UDP interfaces (one set for each interface - ip:port). Default value is 8. For example if you configure the proxy to listen on 3 UDP ports, it will create 3xchildren processes which handle the incoming UDP messages.

For configuration of the TCP/TLS worker threads see the option “tcp_children”.

Example of usage:

```
children=16
```

chroot

The value must be a valid path in the system. If set, Kamailio will chroot (change root directory) to its value.

Example of usage:

```
chroot=/other/fakeroot
```

corelog

Set the debug level used to print some log messages from core, which might become annoying and don't represent critical errors. For example, such case is failure to parse incoming traffic from the network as SIP message, due to someone sending invalid content.

Default value is -1 (L_ERR).

Example of usage:

```
corelog=1
```

debug

Set the debug level. Higher values make Kamailio to print more debug messages. Log messages are usually sent to syslog, except if logging to stderr was activated (see log_stderror parameter).

The following log levels are defined:

```
 L_ALERT     -5
 L_BUG       -4
 L_CRIT2     -3
 L_CRIT      -2
 L_ERR       -1
 L_WARN       0
 L_NOTICE     1
 L_INFO       2
```

```
L_DBG          3
```

A log message will be logged if its log-level is lower than the defined debug level. Log messages are either produced by the the code, or manually in the configuration script using log() or xlog() functions. For a production server you usually use a log value between -1 and 2.

Default value: L_WARN (debug=0)

Examples of usage:

- debug=3: print all log messages. This is only useful for debugging of problems. Note: this produces a lot of data and therefore should not be used on production servers (on a busy server this can easily fill up your hard disk with log messages)
- debug=0: This will only log warning, errors and more critical messages.
- debug=-6: This will disable all log messages.

Value of 'debug' parameter can also be get and set dynamically using the 'debug' Core MI function or the RPC function, e.g.:

```
kamcmd cfg.get core debug
kamcmd cfg.set_now_int core debug 2
kamcmd cfg.set_now_int core debug -- -1
```

Note: There is a difference in log-levels between Kamailio 3.x and Kamailio<=1.5: Up to Kamailio 1.5 the log level started with 4, whereas in Kamailio>=3 the log level starts with 3. Thus, if you were using debug=3 in older Kamailio, now use debug=2.

For configuration of logging of the memory manager see the parameters memlog and memdbg.

Further information can also be found at: https://www.kamailio.org/wiki/tutorials/3.2.x/syslog

description

Alias name: descr desc

disable_core_dump

Can be 'yes' or 'no'. By default core dump limits are set to unlimited or a high enough value. Set this config variable to 'yes' to disable core dump-ing (will set core limits to 0).

Default value is 'no'.

Example of usage:

```
disable_core_dump=yes
```

disable_tls

Alias name: tls_disable

Global parameter to disable TLS support in the SIP server. Default value is 'no'.

Note: Make sure to load the “tls” module to get tls functionality.

Example of usage:

```
  disable_tls=yes
```

In Kamailio TLS is implemented as a module. Thus, the TLS configuration is done as module configuration. For more details see the README of the TLS module: http://kamailio.org/docs/modules/devel/modules/tls.html

enable_tls

Alias name: tls_enable

Reverse Meaning of the disable_tls parameter. See disable_tls parameter.

```
enable_tls=yes # enable tls support in core
```

exit_timeout

Alias name: ser_kill_timeout

How much time Kamailio will wait for all the shutdown procedures to complete. If this time is exceeded, all the remaining processes are immediately killed and Kamailio exits immediately (it might also generate a core dump if the cleanup part takes too long).

Default: 60 s. Use 0 to disable.

```
 exit_timeout = seconds
```

flags

SIP message (transaction) flags can have string names. The *name* for flags cannot be used for **branch** or **script flags**(*)

```
...
flags
  FLAG_ONE   :1,
  FLAG_TWO   :2;
...
```

(*) The named flags feature was propagated from the source code merge back in 2008 and is not extensively tested. The recommended way of defining flags is using #!define (which is also valid for branch/script flags):

```
#!define FLAG_NAME FLAG_BIT
```

force_rport

yes/no: Similar to the force_rport() function, but activates symmetric response routing globally.

fork

If set to 'yes' the proxy will fork and run in daemon mode - one process will be created for each network interface the proxy listens to and for each protocol (TCP/UDP), multiplied with the value of 'children' parameter.

When set to 'no', the proxy will stay bound to the terminal and runs as single process. First interface is used for listening to. This is equivalent to setting the server option “-F”.

Default value is 'yes'.

Example of usage:

```
fork=no
```

fork_delay

Number of usecs to wait before forking a process.

Default is 0 (don't wait).

Example of usage:

```
fork_delay=5000
```

group

Alias name: gid

The group id to run Kamailio.

Example of usage:

```
group="siprouter"
```

http_reply_parse

Alias: http_reply_hack

When enabled, Kamailio can parse HTTP replies, but does so by treating them as SIP replies. When not enabled HTTP replies cannot be parsed. This was previously a compile-time option, now it is run-time.

Default value is 'no'.

Example of usage:

```
http_reply_parse=yes
```

ip_free_bind

Alias: ipfreebind, ip_nonlocal_bind

Control if Kamailio should attempt to bind to non local ip. This option is the per-socket equivalent of the system **ip_nonlocal_bind**.

Default is 0 (do not bind to non local ip).

Example of usage:

```
ip_free_bind =1
```

kemi.onsend_route_callback

Set the name of callback function in the KEMI script to be executed as the equivalent of `onsend_route` block (from the native configuration file).

Default value: ksr_onsend_route

Set it to empty string or “none” to skip execution of this callback function.

Example:

```
kemi.onsend_route_callback="ksr_my_onsend_route"
```

kemi.received_route_callback

Set the name of callback function in the KEMI script to be executed as the equivalent of `event_route[core:msg-received]` block (from the native configuration file). For execution, it also require to have the received_route_mode global parameter set to 1.

Default value: none

Set it to empty string or “none” to skip execution of this callback function.

Example:

```
kemi.received_route_callback="ksr_my_receieved_route"
```

kemi.reply_route_callback

Set the name of callback function in the KEMI script to be executed as the equivalent of `reply_route` block (from the native configuration file).

Default value: ksr_reply_route

Set it to empty string or “none” to skip execution of this callback function.

Example:

```
kemi.onsend_route_callback="ksr_my_reply_route"
```

latency_cfg_log

If set to a log level less or equal than debug parameter, a log message with the duration in microseconds of executing request route or reply route is printed to syslog.

Default value is 3 (L_DBG).

Example:

```
latency_cfg_log=2
```

latency_limit_action

Limit of latency in micro-seconds for config actions. If a config action executed by cfg interpreter takes longer than its value, a message is printed in the logs, showing config path, line and action name when it is a module function, as well as internal action id.

Default value is 0 (disabled).

```
latency_limit_action=500
```

latency_limit_db

Limit of latency in us (micro-seconds) for db operations. If a db operation executed via DB API v1 takes longer that its value, a message is printed in the logs, showing the first 50 characters of the db query.

Default value is 0 (disabled).

```
latency_limit_db=500
```

latency_log

Log level to print the messages related to latency.

Default value is -1 (L_ERR).

```
latency_log=3
```

listen

Set the network addresses the SIP server should listen to. It can be an IP address, hostname or network iterface id or combination of protocol:address:port (e.g., udp:10.10.10.10:5060). This parameter can be set multiple times in same configuration file, the server listening on all addresses specified.

Example of usage:

```
    listen=10.10.10.10
    listen=eth1:5062
    listen=udp:10.10.10.10:5064
```

If you omit this directive then the SIP server will listen on all interfaces. On start the SIP server reports all the interfaces that it is listening on. Even if you specify only UDP interfaces here, the server will start the TCP engine too. If you don't want this, you need to disable the TCP support completely with the core parameter disable_tcp.

If you specify IPv6 addresses, you should put them into square brackets, e.g.:

```
    listen=udp:[2a02:1850:1:1::13]:5060
```

You can specify an advertise address (like ip:port) per listening socket - it will be used to build headers such as Via and Record-Route:

```
listen=udp:10.10.10.10:5060 advertise  11.11.11.11:5060
```

The advertise address must be the format 'address:port', the protocol is taken from the bind socket. The advertise address is a convenient alternative to advertised_address / advertised_port cfg parameters or set_advertised_address() / set_advertised_port() cfg functions.

A typical use case for advertise address is when running SIP server behind a NAT/Firewall, when the local IP address (to be used for bind) is different than the public IP address (to be used for advertising).

loadmodule

Loads a module for later usage in the configuration script. The modules is searched in the path specified by **loadpath**.

Prototype: **loadmodule “modulepath”**

If modulepath is only modulename or modulename.so, then Kamailio will try to search also for **modulename/modulename.so**, very useful when using directly the version compiled in the source tree.

Example of usage:

```
loadpath "/usr/local/lib/kamailio/:usr/local/lib/kamailio/modules/"

loadmodule "/usr/local/lib/kamailio/modules/db_mysql.so"
loadmodule "modules/usrloc.so"
loadmodule "tm"
loadmodule "dialplan.so"
```

loadpath

Alias name: mpath

Set the module search path. loadpath takes a list of directories separated by ':'. The list is searched in-order. For each directory d, $d/${module_name}.so and $d/${module_name}/${module_name}.so are tried.

This can be used to simplify the loadmodule parameter and can include many paths separated by colon. First module found is used.

Example of usage:

```
loadpath "/usr/local/lib/kamailio/modules:/usr/local/lib/kamailio/mymodules"

loadmodule "mysql"
loadmodule "uri"
loadmodule "uri_db"
loadmodule "sl"
loadmodule "tm"
```

The proxy tries to find the modules in a smart way, e.g: loadmodule “uri” tries to find uri.so in the loadpath, but also uri/uri.so.

log_engine_data

Set specific data required by the log engine. See also the **log_engine_type**.

```
log_engine_type="udp"
log_engine_data="127.0.0.1:9"
```

log_engine_type

Specify what logging engine to be used and its initialization data. A logging engine is implemented as a module. Supported values are a matter of the module.

For example, see the readme of **log_custom** module for more details.

```
log_engine_type="udp"
log_engine_data="127.0.0.1:9"
```

log_facility

If Kamailio logs to syslog, you can control the facility for logging. Very useful when you want to divert all Kamailio logs to a different log file. See the man page syslog(3) for more details.

For more see: http://www.kamailio.org/dokuwiki/doku.php/tutorials:debug-syslog-messages

Default value is LOG_DAEMON.

Example of usage:

```
log_facility=LOG_LOCAL0
```

log_name

Allows to configure a log_name prefix which will be used when printing to syslog – it is also known as syslog tag, and the default value is the application name or full path that printed the log message. This is useful to filter log messages when running many instances of Kamailio on same server.

```
log_name="kamailio-proxy-5080"
```

log_prefix

Specify the text to be prefixed to the log messages printed by Kamailio while processing a SIP message (that is, when executing route blocks). It can contain script variables that are evaluated at runtime. See log_prefix_mode about when/how evaluation is done.

If a log message is printed from a part of the code executed out of routing blocks actions (e.g., can be timer, evapi worker process, …), there is no log prefix set, because this one requires a valid SIP message structure to work with.

Example - prefix with message type (1 - request, 2 - response), CSeq and Call-ID:

```
log_prefix="{$mt $hdr(CSeq) $ci} "
```

log_prefix_mode

Control if log prefix is re-evaluated.

If set to 0 (default), then log prefix is evaluated when the sip message is received and then reused (recommended if the **log_prefix** has only variables that have same value for same message). This is the current behaviour of **log_prefix** evaluation.

If set to 1, then the log prefix is evaluated before/after each config action (needs to be set when the **log_prefix** has variables that are different based on the context of config execution, e.g., $cfg(line)).

Example:

```
log_prefix_mode=1
```

log_stderror

With this parameter you can make Kamailio to write log and debug messages to standard error. Possible values are:

- “yes” - write the messages to standard error

- “no” - write the messages to syslog

Default value is “no”.

For more see: http://www.kamailio.org/dokuwiki/doku.php/tutorials:debug-syslog-messages

Example of usage:

```
  log_stderror=yes
```

cfgengine

Set the config interpreter engine for execution of the routing logic inside the configuration file. Default is the native interpreter.

Example of usage:

```
  cfgengine="name"
  cfgengine "name"
```

If name is “native” or “default”, it expects to have in native config interpreter for routing logic.

The name can be the identifier of an embedded language interpreter, such as “lua” which is registered by the app_lua module:

```
  cfgengine "lua"
```

maxbuffer

The size in bytes not to be exceeded during the auto-probing procedure of descovering the maximum buffer size for receiving UDP messages. Default value is 262144.

Example of usage:

```
  maxbuffer=65536
```

max_branches

The maximum number of outgoing branches for each SIP request. It has impact on the size of destination set created in core (e.g., via append_branch()) as well as the serial and parallel forking done via tm module. It replaces the old defined constant MAX_BRANCHES.

The value has to be at least 1 and the upper limit is 31.

Default value: 12

Example of usage:

```
max_branches=16
```

max_recursive_level

The parameters set the value of maximum recursive calls to blocks of actions, such as sub-routes or chained IF-ELSE (for the ELSE branches). Default is 256.

Example of usage:

```
  max_recursive_level=500
```

max_while_loops

The parameters set the value of maximum loops that can be done within a “while”. Comes as a protection to avoid infinite loops in config file execution. Default is 100. Setting to 0 disables the protection (you will still get a warning when you start Kamailio if you do something like while(1) {…}).

Example of usage:

```
  max_while_loops=200
```

mcast

This parameter can be used to set the interface that should join the multicast group. This is useful if you want to **listen** on a multicast address and don't want to depend on the kernel routing table for choosing an interface.

The parameter is reset after each **listen** parameter, so you can join the right multicast group on each interface without having to modify kernel routing beforehand.

Example of usage:

```
  mcast="eth1"
  listen=udp:224.0.1.75:5060
```

mcast_loopback

It can be 'yes' or 'no'. If set to 'yes', multicast datagram are sent over loopback. Default value is 'no'.

Example of usage:

```
  mcast_loopback=yes
```

mcast_ttl

Set the value for multicast ttl. Default value is OS specific (usually 1).

Example of usage:

```
  mcast_ttl=32
```

memdbg

Alias name: mem_dbg

This parameter specifies on which log level the memory debugger messages will be logged. If memdbg is active, every request (alloc, free) to the memory manager will be logged. (Note: if compile option NO_DEBUG is specified, there will never be logging from the memory manager).

Default value: L_DBG (memdbg=3)

For example, memdbg=2 means that memory debugging is activated if the debug level is 2 or higher.

```
debug=3    # no memory debugging as debug level
memdbg=4   # is lower than memdbg
debug=3    # memory debugging is active as the debug level
memdbg=2   # is higher or equal memdbg
```

Please see also memlog and debug.

memlog

Alias name: mem_log

This parameter specifies on which log level the memory statistics will be logged. If memlog is active, Kamailio will log memory statistics on shutdown (or if requested via signal SIGUSR1). This can be useful for debugging of memory leaks.

Default value: L_DBG (memlog=3)

For example, memlog=2 means that memory statistics dumping is activated if the debug level is 2 or higher.

```
debug=3    # no memory statistics as debug level
memlog=4   # is lower than memlog
debug=3    # dumping of memory statistics is active as the
memlog=2   # debug level is higher or equal memlog
```

Please see also memdbg and debug.

mem_join

If set to 1, memory manger (e.g., q_malloc) does join of free fragments. It is effective if MEM_JOIN_FREE compile option is defined.

It can be set via config reload framework.

Default is 0 (disabled).

```
mem_join=1
```

To change its value at runtime, **kamcmd** needs to be used and the modules **ctl** and **cfg_rpc** loaded. Enabling it can be done with:

```
kamcmd cfg.set_now_int core mem_join 1
```

To disable, set its value to 0.

mem_safety

If set to 1, memory free operation does not call abort() for double freeing a pointer or freeing an invalid address. The server still prints the alerting log messages. If set to 0, the SIP server stops by calling abort() to generate a core file.

It can be set via config reload framework.

Default is 1 (enabled).

```
mem_safety=0
```

mem_status_mode

If set to 1, memory status dump for qm allocator will print details about used fragments. If set to 0, the dump contains only free fragments. It can be set at runtime via cfg param framework (e.g., via kamcmd).

Default is 0.

```
mem_status_mode=1
```

mem_summary

Parameter to control printing of mmemory debugging information displayed on exit or SIGUSR1. The value can be composed by following flags:

- 1 - dump all the pkg used blocks (status)
- 2 - dump all the shm used blocks (status)
- 4 - summary of pkg used blocks
- 8 - summary of shm used blocks

If set to 0, nothing is printed.

Default value: 3

Example:

```
mem_summary=15
```

mhomed

Set the server to try to locate outbound interface on multihomed host. This parameter affects the selection of the outgoing socket for forwarding requests. By default is off (0) - it is rather time consuming. When deactivated, the incoming socket will be used or the first one for a different protocol, disregarding the destination location. When activated, Kamailio will select a socket that can reach the destination (to be able to connect to the remote address). (Kamailio opens a UDP socket to the destination, then it retrieves the local IP which was assigned by the operating system to the new UDP socket. Then this socket will be closed and the retrieved IP address will be used as IP address in the Via/Record-Route headers)

Example of usage:

```
  mhomed=1
```

mlock_pages

Locks all Kamailio pages into memory making it unswappable (in general one doesn't want his SIP proxy swapped out)

```
mlock_pages = yes |no (default no)
```

modinit_delay

Number of microseconds to wait after initializing a module - useful to cope with systems where are rate limits on new connections to database or other systems.

Default value is 0 (no wait).

```
modinit_delay=100000
```

modparam

The modparam command will be used to set the options of the modules.

Example:

```
modparam("usrloc", "db_mode", 2)
modparam("usrloc", "nat_bflag", 6)
```

See the documenation of the respective module to find out the available options.

onsend_route_reply

If set to 1 (yes, on), onsend_route block is executed for received replies that are sent out. Default is 0.

```
  onsend_route_reply=yes
```

open_files_limit

If set and bigger than the current open file limit, Kamailio will try to increase its open file limit to this number. Note: Kamailio must be started as root to be able to increase a limit past the hard limit (which, for open files, is 1024 on most systems). “Files” include network sockets, so you need one for every concurrent session (especially if you use connection-oriented transports, like TCP/TLS).

Example of usage:

```
open_files_limit=2048
```

phone2tel

By enabling this feature, Kamailio internally treats SIP URIs with user=phone parameter as TEL URIs. If you do not want this behavior, you have to turn it off.

Default value: 1 (enabled)

```
phone2tel = 0
```

pmtu_discovery

If enabled, the Don't Fragment (DF) bit will be set in outbound IP packets.

```
pmtu_discovery = 0 | 1 (default 0)
```

port

The port the SIP server listens to. The default value for it is 5060.

Example of usage:

```
port=5080
```

pv_buffer_size

The size in bytes of internal buffer to print dynamic strings with pseudo-variables inside. The default value is 8192 (8kB). Please keep in mind that for xlog messages, there is a dedicated module parameter to set the internal buffer size.

Example of usage:

```
pv_buffer_size=2048
```

pv_buffer_slots

The number of internal buffer slots to print dynamic strings with pseudo-variables inside. The default value is 10.

Example of usage:

```
pv_buffer_slots=12
```

pv_cache_limit

The limit how many pv declarations in the cache after which an action is taken. Default value is 2048.

```
pv_cache_limit=1024
```

pv_cache_action

Specify what action to be done when the size of pv cache is exceeded. If 0, print an warning log message when the limit is exceeded. If 1, warning log messages is printed and the cache systems tries to drop a $sht(…) declaration. Default is 0.

```
pv_cache_action=1
```

rundir

Alias: run_dir

Set the folder for creating runtime files such as MI fifo or CTL unixsocket.

Default: /var/run/kamailio

Example of usage:

```
rundir="/tmp"
```

received_route_mode

Enable or disable the execution of event_route[core:msg-received] routing block or its corresponding Kemi callback.

Default value: 0 (disabled)

Example of usage:

```
received_route_mode=1
```

reply_to_via

If it is set to 1, any local reply is sent to the IP address advertised in top most Via of the request instead of the IP address from which the request was received. Default value is 0 (off).

Example of usage:

```
  reply_to_via=0
```

route_locks_size

Set the number of mutex locks to be used for synchronizing the execution of messages sharing the same Call-Id. In other words, enables Kamailio to execute sequentially the requests and replies received within the same dialog – a new message received within the same dialog waits until the previous one is routed out.

For smaller impact on parallel processing, its value it should be at least twice the number of kamailio processes (children

Example:

```
route_locks_size =256
```

server_id

A configurable unique server id that can be used to discriminate server instances within a cluster of servers when all other information, such as IP addresses are the same.

```
server_id = number
```

server_header

Set the value of Server header for replies generated by Kamailio. It must contain the header name, but not the ending CRLF.

Example of usage:

```
server_header="Server: My Super SIP Server"
```

server_signature

This parameter controls the “Server” header in any locally generated message.

Example of usage:

```
server_signature=no
```

If it is enabled (default=yes) a header is generated as in the following example:

```
Server: Kamailio (<version> (<arch>/<os>))
```

shm_force_alloc

Tries to pre-fault all the shared memory, before starting. When “on”, start time will increase, but combined with mlock_pages will guarantee Kamailio will get all its memory from the beginning (no more kswapd slow downs)

shm_force_alloc = yes | no (default no)

shm_mem_size

Set shared memory size (in Mb).

shm_mem_size = 64 (default 64)

sip_warning (noisy feedback)

Can be 0 or 1. If set to 1 (default value is 0) a 'Warning' header is added to each reply generated by Kamailio. The header contains several details that help troubleshooting using the network traffic dumps, but might reveal details of your network infrastructure and internal SIP routing.

Example of usage:

```
sip_warning=0
```

socket_workers

Number of workers to process SIP traffic per listen socket - typical use is before a **listen** global parameter.

- when used before **listen** on UDP or SCTP socket, it overwrites **children** or **sctp_children** value for that socket.
- when used before **listen** on TCP or TLS socket, it adds extra tcp workers, these handling traffic only on that socket.

The value of **socket_workers** is reset with next **listen** socket definition that is added, thus use it for each **listen** socket where you want custom number of workers.

If this parameter is not used at all, the values for **children**, **tcp_children** and **sctp_children** are used as usually.

Example for udp sockets:

```
children=4
socket_workers=2
listen=udp:127.0.0.1:5080
listen=udp:127.0.0.1:5070
listen=udp:127.0.0.1:5060
```

- it will start 2 workers to handle traffic on udp:127.0.0.1:5080 and 4 for each of udp:127.0.0.1:5070 and udp:127.0.0.1:5060. In total there are 10 worker processes

Example for tcp sockets:

```
children=4
socket_workers=2
listen=tcp:127.0.0.1:5080
listen=tcp:127.0.0.1:5070
listen=tcp:127.0.0.1:5060
```

- it will start 2 workers to handle traffic on tcp:127.0.0.1:5080 and 4 to handle traffic on both tcp:127.0.0.1:5070 and tcp:127.0.0.1:5060. In total there are 6 worker processes

sql_buffer_size

The size in bytes of the SQL buffer created for data base queries. For database drivers that use the core db_query library, this will be maximum size object that can be written or read from a database. Default value is 65535.

Example of usage:

```
sql_buffer_size=131070
```

statistics

Kamailio has built-in support for statistics counter. This means, these counters can be increased, decreased, read and cleared. The statistics counter are defined either by the core (e.g. tcp counters), by modules (e.g. 2xx_transactions by “tmx” module) or by the script writer using the “statistics” module.

The statistics counters are read/updated either automatically by Kamailio internally (e.g. tcp counters), by the script writer via the module functions of the “statistics” module, by the script writer using the $stat() pseudo variable (read-only), or via MI commands.

Following are some examples how to access statistics variables:

script

```
modparam("statistics", "variable", "NOTIFY")

(if method == "NOTIFY") {
  update_stat("NOTIFY", "+1");
}

xlog("Number of received NOTIFYs: $stat(NOTIFY)");
```

MI

```
# get counter value
kamctl fifo get_statistics NOTIFY
# set counter to zero
kamctl fifo reset_statistics NOTIFY
# get counter value and then set it to zero
kamctl fifo clear_statistics NOTIFY

# or use the kamcmd tool
kamcmd mi get_statistics 1xx_replies
```

tos

The TOS (Type Of Service) to be used for the sent IP packages (both TCP and UDP).

Example of usage:

```
  tos=IPTOS_LOWDELAY
  tos=0x10
  tos=IPTOS_RELIABILITY
```

udp_mtu

Fallback to another protocol (udp_mtu_try_proto must be set also either globally or per packet) if the constructed request size is greater then udp_mtu.

RFC 3261 specified size: 1300. Default: 0 (off).

```
udp_mtu = number
```

udp_mtu_try_proto

If udp_mtu !=0 and udp forwarded request size (after adding all the “local” headers) > udp_mtu, use this protocol instead of udp. Only the Via header will be updated (e.g. The Record-Route will be the one built for udp).

Warning: Although RFC3261 mandates automatic transport protocol changing, enabling this feature can lead to problems with clients which do not support other protocols or are behind a firewall or NAT. Use this only when you know what you do!

See also udp_mtu_try_proto(proto) function.

Default: UDP (off). Recommended: TCP.

```
udp_mtu_try_proto = TCP|TLS|SCTP|UDP
```

user

Alias name: uid

The user id to run Kamailio (Kamailio will suid to it).

Example of usage:

```
    user="kamailio"
```

user_agent_header

Set the value of User-Agent header for requests generated by Kamailio. It must contain header name as well, but not the ending CRLF.

```
user_agent_header="User-Agent: My Super SIP Server"
```

verbose_startup

Control if printing routing tree and udp probing buffer debug messages should be printed at startup.

Default is 0 (don't print); set to 1 to get those debug messages.

Example of usage:

```
    verbose_startup=1
```

version_table

Set the name of the table holding the table version. Useful if the proxy is sharing a database within a project and during upgrades. Default value is “version”.

Example of usage:

```
    version_table="version44"
```

workdir

Alias name: wdir

The working directory used by Kamailio at runtime. You might find it useful when it comes to generating core files :)

Example of usage:

```
    wdir="/usr/local/kamailio"
    or
    wdir=/usr/kam_wd
```

xavp_via_params

Set the name of the XAVP of which subfields will be added as local *Via* -header parameters.

If not set, XAVP to Via header parameter manipulation is not applied (default behaviour).

If set, local Via header gets additional parameters from defined XAVP. Core flag FL_ADD_XAVP_VIA_PARAMS needs to be set[1].

Example:

```
   xavp_via_params="via"
```

[1] See function *via_add_xavp_params()* from “corex” module.

xavp_via_fields

Set the name of xavp from where to take Via header field: address and port. Use them to build local Via header.

Example:

```
xavp_via_fields="customvia"

request_route {
  ...
  $xavp(customvia=>address)="1.2.3.4";
  $xavp(customvia[0]=>port)="5080";# must be string
  via_use_xavp_fields("1");
  t_relay();}
```

See function *via_use_xavp_fields()* from “corex” module.

DNS Parameters

Note: See also file doc/dns.txt for details about Kamailio's DNS client.

Kamailio has an internal DNS resolver with caching capabilities. If this caching resolver is activated (default setting) then the system's stub resolver won't be used. Thus, also local name resolution configuration like /etc/hosts entries will not be used. If the DNS cache is deactivated (use_dns_cache=no), then system's resolver will be used. The DNS failover functionality in the tm module references directly records in the DNS cache (which saves a lot of memory) and hence DNS based failover only works if the internal DNS cache is enabled.

DNS resolver comparison	internal resolver	system resolver
Caching of resolved records	yes	no*
NAPTR/SRV lookups with correct weighting	yes	yes
DNS based failover	yes	no

* Of course you can use the resolving name servers configured in /etc/resolv.conf as caching nameservers.

If the internal resolver/cache is enabled you can add/remove records by hand (using kamcmd or xmlrpc) using the DNS RPCs, e.g. dns.add_a, dns.add_srv, dns.delete_a a.s.o. For more info on DNS RPCs see http://www.kamailio.org/docs/docbooks/devel/rpc_list/rpc_list.html#dns.add_a

Note: During startup of Kamailio, before the internal resolver is loaded, the system resolver will be used (it will be used for queries done from module register functions or modparams fixups, but not for queries done from mod_init() or normal fixups).

Note: The dns cache uses the DNS servers configured on your server (/etc/resolv.conf), therefore even if you use the internal resolver you should have a working DNS resolving configuration on your server.

Kamailio also allows you to finetune the DNS resolver settings.

The maximum time a dns request can take (before failing) is (if dns_try_ipv6 is yes, multiply it again by 2; if SRV and NAPTR lookups are enabled, it can take even longer!):

```
(dns_retr_time*(dns_retr_no+1)*dns_servers_no)*(search_list_domains)
```

Note: During DNS lookups, the process which performs the DNS lookup blocks. To minimize the blocked time the following parameters can be used (max 2s):

```
dns_try_ipv6=no
dns_retr_time=1
dns_retr_no=1
dns_use_search_list=no
```

dns

This parameter controls if the SIP server will try doing a DNS lookup on the address in the Via header of a received sip request to decide if adding a received=<src_ip> parameter to the Via is necessary. Note that Vias containing DNS names (instead of IPs) should have received= added, so turning dns to yes is not recommended.

Default is no.

rev_dns

This parameter controls if the SIP server will try doing a reverse DNS lookup on the source IP of a sip request to decide if adding a received=<src_ip> parameter to the Via is necessary (if the Via contains a DNS name instead of an IP address, the result of the reverse dns on the source IP will be compared with the DNS name in the Via). See also dns (the effect is cumulative, both can be turned on and in that case if the DNS lookup test fails the reverse DNS test will be tried). Note that Vias containing DNS names (instead of IPs) should have received= added, so turning rev_dns to yes is not recommended.

Default is no.

dns_cache_del_nonexp

Alias name: dns_cache_delete_nonexpired

```
dns_cache_del_nonexp = yes | no (default: no)
  allow deletion of non-expired records from the cache when there is no more space
  left for new ones. The last-recently used entries are deleted first.
```

dns_cache_rec_pref

```
dns_cache_rec_pref = number (default 0)
  dns cache record preference, determines how new DNS records are stored internally in relation to existing entries.
  Possible values:
    0 - do not check duplicates
    1 - prefer old records
```

```
    2 - prefer new records
    3 - prefer records with longer lifetime
```

dns_cache_flags

```
dns_cache_flags = number (default 0) -
  dns cache specific resolver flags, used for overriding the default behaviour (low level).
  Possible values:
    1 - ipv4 only: only DNS A requests are performed, even if Kamailio also listens on ipv6 addresses.
    2 - ipv6 only: only DNS AAAA requests are performed. Ignored if dns_try_ipv6 is off or Kamailio
        doesn't listen on any ipv6 address.
    4 - prefer ipv6: try first to resolve a host name to an ipv6 address (DNS AAAA request) and only
        if this fails try an ipv4 address (DNS A request). By default the ipv4 addresses are preferred.
```

dns_cache_gc_interval

Interval in seconds after which the dns cache is garbage collected (default: 120 s)

```
dns_cache_gc_interval = number
```

dns_cache_init

If off, the dns cache is not initialized at startup and cannot be enabled runtime, that saves some memory.

```
dns_cache_init = on | off (default on)
```

dns_cache_max_ttl

```
dns_cache_max_ttl = time in seconds (default MAXINT)
```

dns_cache_mem

Maximum memory used for the dns cache in KB (default 500 K)

```
dns_cache_mem = number
```

dns_cache_min_ttl

```
dns_cache_min_ttl = time in seconds (default 0)
```

dns_cache_negative_ttl

Tells how long to keep negative DNS responses in cache. If set to 0, disables caching of negative responses. Default is 60 (seconds).

dns_naptr_ignore_rfc

If the DNS lookup should ignore the remote side's protocol preferences, as indicated by the Order field in the NAPTR records and mandated by RFC 2915.

```
  dns_naptr_ignore_rfc = yes | no (default yes)
```

dns_retr_no

Number of dns retransmissions before giving up. Default value is system specific, depends also on the '/etc/resolv.conf' content (usually 4).

Example of usage:

```
  dns_retr_no=3
```

dns_retr_time

Time in seconds before retrying a dns request. Default value is system specific, depends also on the '/etc/resolv.conf' content (usually 5s).

Example of usage:

```
  dns_retr_time=3
```

dns_search_full_match

When name was resolved using dns search list, check the domain added in the answer matches with one from the search list (small performance hit, but more safe)

```
dns_search_full_match = yes | no (default yes)
```

dns_servers_no

How many dns servers from the ones defined in '/etc/resolv.conf' will be used. Default value is to use all of them.

Example of usage:

```
  dns_servers_no=2
```

dns_srv_lb

Alias name: dns_srv_loadbalancing

Enable dns srv weight based load balancing (see doc/dns.txt)

```
dns_srv_lb = yes | no (default no)
```

dns_try_ipv6

Can be 'yes' or 'no'. If it is set to 'yes' and a DNS lookup fails, it will retry it for ipv6 (AAAA record). Default value is 'no'.

Note: If dns_try_ipv6 is off, no hostname resolving that would result in an ipv6 address would succeed - it doesn't matter if an actual DNS lookup is to be performed or the host is already an ip address. Thus, if the proxy should forward requests to IPv6 targets, this option must be turned on!

Example of usage:

```
  dns_try_ipv6=yes
```

dns_try_naptr

Enable NAPTR support according to RFC 3263 (see doc/dns.txt for more info)

```
dns_try_naptr = yes | no (default no)
```

dns_sctp_pref, dns_tcp_pref, dns_tls_pref, dns_udp_pref

Alias name: dns_sctp_preference, dns_tcp_preference, dns_tls_preference, dns_udp_preference

Set preference for each protocol when doing naptr lookups. By default dns_udp_pref=30, dns_tcp_pref=20, dns_tls_pref=10 and dns_sctp_pref=20. To use the remote site preferences set all dns_*_pref to the same positive value (e.g. dns_udp_pref=1, dns_tcp_pref=1, dns_tls_pref=1, dns_sctp_pref=1). To completely ignore NAPTR records for a specific protocol, set the corresponding protocol preference to -1 (or any other negative number). (see doc/dns.txt for more info)

```
dns_{udp,tcp,tls,sctp}_pref = number
```

dns_use_search_list

Can be 'yes' or 'no'. If set to 'no', the search list in '/etc/resolv.conf' will be ignored (⇒fewer lookups ⇒gives up faster). Default value is 'yes'.

HINT: even if you don't have a search list defined, setting this option to 'no' will still be “faster”, because an empty search list is in fact search “” (so even if the search list is empty/missing there will still be 2 dns queries, eg. foo+'.' and foo+“”+'.')

Example of usage:

```
dns_use_search_list=no
```

use_dns_cache

Tells if DNS responses are cached - this means that the internal DNS resolver (instead of the system's stub resolver) will be used. If set to “off”, disables caching of DNS responses and, as side effect, DNS failover. Default is “on”. Settings can be changed also during runtime (switch from internal to system resolver and back).

use_dns_failover

use_dns_failover = on | off (default off)

TCP Parameters

The following parameters allows to tweak the TCP behaviour.

disable_tcp

Global parameter to disable TCP support in the SIP server. Default value is 'no'.

Example of usage:

```
disable_tcp=yes
```

tcp_accept_aliases

If a message received over a tcp connection has “alias” in its via a new tcp alias port will be created for the connection the message came from (the alias port will be set to the via one).

Based on draft-ietf-sip-connect-reuse-00.txt, but using only the port (host aliases are dangerous, involve extra DNS lookups and the need for them is questionable)

See force_tcp_alias for more details.

Note: For NAT traversal of TCP clients it is better to not use tcp_accept_aliases but just use nathelper module and fix_nated_[contact|register] functions.

Default is “no” (off)

```
tcp_accept_aliases= yes|no
```

tcp_accept_haproxy

Enable the internal TCP stack to expect a PROXY-protocol-formatted header as the first message of the connection. Both the human-readable (v1) and binary-encoded (v2) variants of the protocol are supported. This option is typically useful if you are behind a TCP load-balancer, such as HAProxy or an AWS' ELB, and allows the load-balancer to provide connection information regarding the upstream client. This enables the use of IP-based ACLs, even behind a load-balancer.

Please note that enabling this option will reject any inbound TCP connection that does not conform to the PROXY-protocol spec.

For reference: A PROXY protocol - https://www.haproxy.org/download/1.8/doc/proxy-protocol.txt

Default value is **no**.

```
tcp_accept_haproxy=yes
```

tcp_accept_hep3

Enable internal TCP receiving stack to accept HEP3 packets. This option has to be set to **yes** on a Kamailio instance acting as Homer SIPCapture server that is supposed to receive HEP3 packets over TCP/TLS.

Default value is **no**.

```
tcp_accept_hep3=yes
```

tcp_accept_no_cl

Control whether to throw or not error when there is no Content-Length header for requests received over TCP. It is required to be set to **yes** for XCAP traffic sent over HTTP/1.1 which does not use Content-Length header, but splits large bodies in many chunks. The module **sanity** can be used then to restrict this permission to HTTP traffic only, testing in route block in order to stay RFC3261 compliant about this mandatory header for SIP requests over TCP.

Default value is **no**.

```
tcp_accept_no_cl=yes
```

tcp_accept_unique

If set to 1, reject duplicate connections coming from same source IP and port.

Default set to 0.

```
tcp_accept_unique =1
```

tcp_async

Alias name: tcp_buf_write

If enabled, all the tcp writes that would block / wait for connect to finish, will be queued and attempted latter (see also tcp_conn_wq_max and tcp_wq_max).

Note: It also applies for TLS.

```
tcp_async = yes | no (default yes)
```

tcp_children

Number of children processes to be created for reading from TCP connections. If no value is explicitly set, the same number of TCP children as UDP children (see “children” parameter) will be used.

Example of usage:

```
tcp_children=4
```

tcp_clone_rcvbuf

Control if the received buffer should be cloned from the TCP stream, needed by functions working inside the SIP message buffer (such as msg_apply_changes()).

Default is 0 (don't clone), set it to 1 for cloning.

Example of usage:

```
tcp_clone_rcvbuf=1
```

tcp_connection_lifetime

Lifetime in seconds for TCP sessions. TCP sessions which are inactive for longer than **tcp_connection_lifetime** will be closed by Kamailio. Default value is defined is 120. Setting this value to 0 will close the TCP connection pretty quick .

Note: As many SIP clients are behind NAT/Firewalls, the SIP proxy should not close the TCP connection as it is not capable of opening a new one.

Example of usage:

```
tcp_connection_lifetime=3605
```

tcp_connection_match

If set to 1, try to be more strict in matching outbound TCP connections, attempting to lookup first the connection using also local port, not only the local IP and remote IP+port.

Default is 0.

```
tcp_connection_match=1
```

tcp_connect_timeout

Time in seconds before an ongoing attempt to establish a new TCP connection will be aborted. Lower this value for faster detection of TCP connection problems. The default value is 10s.

Example of usage:

```
  tcp_connect_timeout=5
```

tcp_conn_wq_max

Maximum bytes queued for write allowed per connection. Attempting to queue more bytes would result in an error and in the connection being closed (too slow). If tcp_buf_write is not enabled, it has no effect.

```
tcp_conn_wq_max = bytes (default 32 K)
```

tcp_crlf_ping

Enable SIP outbound TCP keep-alive using PING-PONG (CRLFCRLF - CRLF).

```
tcp_crlf_ping = yes | no default: yes
```

tcp_defer_accept

Tcp accepts will be delayed until some data is received (improves performance on proxies with lots of opened tcp connections). See linux tcp(7) TCP_DEFER_ACCEPT or freebsd ACCF_DATA(0). For now linux and freebsd only.

WARNING: the linux TCP_DEFER_ACCEPT is buggy (⇐2. 6. 23)and doesn't work exactly as expected (if no data is received it will retransmit syn acks for ~ 190 s, irrespective of the set timeout and then it will silently drop the connection without sending a RST or FIN). Try to use it together with tcp_syncnt (this way the number of retrans. SYNACKs can be limited ⇒the timeout can be controlled in some way).

On FreeBSD:

```
tcp_defer_accept =  yes | no (default no)
```

On Linux:

```
tcp_defer_accept =  number of seconds before timeout (default disabled)
```

tcp_delayed_ack

Initial ACK for opened connections will be delayed and sent with the first data segment (see linux tcp(7) TCP_QUICKACK). For now linux only.

```
tcp_delayed_ack  = yes | no (default yes when supported)
```

tcp_fd_cache

If enabled FDs used for sending will be cached inside the process calling tcp_send (performance increase for sending over tcp at the cost of slightly slower connection closing and extra FDs kept open)

```
tcp_fd_cache = yes | no (default yes)
```

tcp_keepalive

Enables keepalive for tcp (sets SO_KEEPALIVE socket option)

```
tcp_keepalive = yes | no (default yes)
```

tcp_keepcnt

Number of keepalives sent before dropping the connection (TCP_KEEPCNT socket option). Linux only.

```
tcp_keepcnt = number (not set by default)
```

tcp_keepidle

Time before starting to send keepalives, if the connection is idle (TCP_KEEPIDLE socket option). Linux only.

```
tcp_keepidle  = seconds (not set by default)
```

tcp_keepintvl

Time interval between keepalive probes, when the previous probe failed (TCP_KEEPINTVL socket option). Linux only.

```
tcp_keepintvl = seconds (not set by default)
```

tcp_linger2

Lifetime of orphaned sockets in FIN_WAIT2 state (overrides tcp_fin_timeout on, see linux tcp(7) TCP_LINGER2). Linux only.

```
tcp_linger2 = seconds (not set by default)
```

tcp_max_connections

Maximum number of tcp connections (if the number is exceeded no new tcp connections will be accepted). Default is defined in tcp_init.h: #define DEFAULT_TCP_MAX_CONNECTIONS 2048

Example of usage:

```
  tcp_max_connections=4096
```

tcp_no_connect

Stop outgoing TCP connects (also stops TLS) by setting tcp_no_connect to yes.

You can do this any time, even even if Kamailio is already started (in this case using the command “kamcmd cfg.set_now_int tcp no_connect 1”).

tcp_poll_method

Poll method used (by default the best one for the current OS is selected). For available types see io_wait.c and poll_types.h: none, poll, epoll_lt, epoll_et, sigio_rt, select, kqueue, /dev/poll

Example of usage:

```
  tcp_poll_method=select
```

tcp_rd_buf_size

Buffer size used for tcp reads. A high buffer size increases performance on server with few connections and lot of traffic on them, but also increases memory consumption (so for lots of connection is better to use a low value). Note also that this value limits the maximum message size (SIP, HTTP) that can be received over tcp.

The value is internally limited to 16MByte, for higher values recompile Kamailio with higher limit in tcp_options.c (search for “rd_buf_size” and 16777216). Further, you may need to increase the private memory, and if you process the message stateful you may also have to increase the shared memory.

Default: 4096, can be changed at runtime.

```
tcp_rd_buf_size=65536
```

tcp_send_timeout

Time in seconds after a TCP connection will be closed if it is not available for writing in this interval (and Kamailio wants to send something on it). Lower this value for faster detection of broken TCP connections. The default value is 10s.

Example of usage:

```
  tcp_send_timeout=3
```

tcp_source_ipv4, tcp_source_ipv6

Set the source IP for all outbound TCP connections. If setting of the IP fails, the TCP connection will use the default IP address.

```
tcp_source_ipv4 = IPv4 address
tcp_source_ipv6 = IPv6 address
```

tcp_syncnt

Number of SYN retransmissions before aborting a connect attempt (see linux tcp(7) TCP_SYNCNT). Linux only.

```
tcp_syncnt = number of syn retr. (default not set)
```

tcp_wq_blk_size

Block size used for tcp async writes. It should be big enough to hold a few datagrams. If it's smaller then a datagram (in fact a tcp write()) size, it will be rounded up. It has no influenced on the number of datagrams queued (for that see tcp_conn_wq_max or tcp_wq_max). It has mostly debugging and testing value (can be ignored).

Default: 2100 (~ 2 INVITEs), can be changed at runtime.

tcp_wq_max

Maximum bytes queued for write allowed globally. It has no effect if tcp_buf_write is not enabled.

```
tcp_wq_max = bytes (default 10 Mb)
```

tcp_reuse_port

Allows reuse of TCP ports. This means,for example, that the same TCP ports on which Kamailio is listening on, can be used as source ports of new TCP connections when acting as an UAC. Kamailio must have been compiled in a system implementing SO_REUSEPORT (Linux > 3.9.0, FreeBSD, OpenBSD, NetBSD, MacOSX). This parameter takes effect only if also the system on which Kamailio is running on supports SO_REUSEPORT.

```
tcp_reuse_port = yes (default no)
```

TLS Parameters

Most of TLS layer attributes can be configured via TLS module parameters.

tls_port_no

The port the SIP server listens to for TLS connections.

Default value is 5061.

Example of usage:

```
  tls_port_no=6061
```

tls_max_connections

Maximum number of ls connections (if the number is exceeded no new ls connections will be accepted). It cannot exceed tcp_max_connections.

Default value is 2048.

Example of usage:

```
  tls_max_connections=4096
```

SCTP Parameters

disable_sctp

Global parameter to disable SCTP support in the SIP server. see enable_sctp

Default value is 'auto'.

Example of usage:

```
  disable_sctp=yes
```

enable_sctp

```
enable_sctp = 0/1/2  - SCTP disabled (0)/ SCTP enabled (1)/auto (2),
                       default auto (2)
```

sctp_children

sctp children no (similar to udp children)

```
sctp_children = number
```

sctp_socket_rcvbuf

Size for the sctp socket receive buffer

Alias name: sctp_socket_receive_buffer

```
sctp_socket_rcvbuf = number
```

sctp_socket_sndbuf

Size for the sctp socket send buffer

Alias name: sctp_socket_send_buffer

```
sctp_socket_sndbuf = number
```

sctp_autoclose

Number of seconds before autoclosing an idle association (default: 180 s). Can be changed at runtime, but it will affect only new associations. E.g.:

```
$ kamcmd cfg.set_now_int sctp autoclose 120
sctp_autoclose = seconds
```

sctp_send_ttl

Number of milliseconds before an unsent message/chunk is dropped (default: 32000 ms or 32 s). Can be changed at runtime, e.g.:

```
$ kamcmd cfg.set_now_int sctp send_ttl 180000
sctp_send_ttl = milliseconds - n
```

sctp_send_retries

How many times to attempt re-sending a message on a re-opened association, if the sctp stack did give up sending it (it's not related to sctp protocol level retransmission). Useful to improve reliability with peers that reboot/restart or fail over to another machine.

WARNING: use with care and low values (e.g. 1-3) to avoid “multiplying” traffic to unresponding hosts (default: 0).Can be changed at runtime.

```
sctp_send_retries = 1
```

sctp_assoc_tracking

Controls whether or not sctp associations are tracked inside Kamailio. Turning it off would result in less memory being used and slightly better performance, but it will also disable some other features that depend on it (e.g. sctp_assoc_reuse). Default: yes.

Can be changed at runtime (“kamcmd sctp assoc_tracking 0”), but changes will be allowed only if all the other features that depend on it are turned off (for example it can be turned off only if first sctp_assoc_reuse was turned off).

Note: turning sctp_assoc_tracking on/off will delete all the tracking information for all the currently tracked associations and might introduce a small temporary delay in the sctp processing if lots of associations were tracked.

Config options depending on sctp_assoc_tracking being on: sctp_assoc_reuse.

```
sctp_assoc_tracking = yes/no
```

sctp_assoc_reuse

Controls sctp association reuse. For now only association reuse for replies is affected by it. Default: yes. Depends on sctp_assoc_tracking being on.

Note that even if turned off, if the port in via corresponds to the source port of the association the request was sent on or if rport is turned on (force_rport() or via containing a rport option), the association will be automatically reused by the sctp stack. Can be changed at runtime (sctp assoc_reuse), but it can be turned on only if sctp_assoc_tracking is on.

```
sctp_assoc_reuse = yes/no
```

sctp_max_assocs

Maximum number of allowed open sctp associations. -1 means maximum allowed by the OS. Default: -1. Can be changed at runtime (e.g.: “kamcmd cfg.set_now_int sctp max_assocs 10”). When the maximum associations number is exceeded and a new associations is opened by a remote host, the association will be immediately closed. However it is possible that some SIP packets get through (especially if they are sent early, as part of the 4-way handshake).

When Kamailio tries to open a new association and the max_assocs is exceeded the exact behaviour depends on whether or not sctp_assoc_tracking is on. If on, the send triggering the active open will gracefully fail, before actually opening the new association and no packet will be sent. However if sctp_assoc_tracking is off, the association will first be opened and then immediately closed. In general this means that the initial sip packet will be sent (as part of the 4-way handshake).

```
sctp_max_assocs = number
```

sctp_srto_initial

Initial value of the retr. timeout, used in RTO calculations (default: OS specific).

Can be changed at runtime (sctp srto_initial) but it will affect only new associations.

```
sctp_srto_initial = milliseconds
```

sctp_srto_max

Maximum value of the retransmission timeout (RTO) (default: OS specific).

WARNING: values lower then the sctp sack_delay will cause lots of retransmissions and connection instability (see sctp_srto_min for more details).

Can be changed at runtime (sctp srto_max) but it will affect only new associations.

```
sctp_srto_max = milliseconds
```

sctp_srto_min

Minimum value of the retransmission timeout (RTO) (default: OS specific).

WARNING: values lower then the sctp sack_delay of any peer might cause retransmissions and possible interoperability problems. According to the standard the sack_delay should be between 200 and 500 ms, so avoid trying values lower then 500 ms unless you control all the possible sctp peers and you do make sure their sack_delay is higher or their sack_freq is 1.

Can be changed at runtime (sctp srto_min) but it will affect only new associations.

```
sctp_srto_min = milliseconds
```

sctp_asocmaxrxt

Maximum retransmissions attempts per association (default: OS specific). It should be set to sctp_pathmaxrxt * no. of expected paths.

Can be changed at runtime (sctp asocmaxrxt) but it will affect only new associations.

```
sctp_asocmaxrxt   = number
```

sctp_init_max_attempts

Maximum INIT retransmission attempts (default: OS specific).

Can be changed at runtime (sctp init_max_attempts).

```
sctp_init_max_attempts = number
```

sctp_init_max_timeo

Maximum INIT retransmission timeout (RTO max for INIT). Default: OS specific.

Can be changed at runtime (sctp init_max_timeo).

```
sctp_init_max_timeo = milliseconds
```

sctp_hbinterval

sctp heartbeat interval. Setting it to -1 will disable the heartbeats. Default: OS specific.

Can be changed at runtime (sctp hbinterval) but it will affect only new associations.

```
sctp_hbinterval = milliseconds
```

sctp_pathmaxrxt

Maximum retransmission attempts per path (see also sctp_asocmaxrxt). Default: OS specific.

Can be changed at runtime (sctp pathmaxrxt) but it will affect only new associations.

```
sctp_pathmaxrxt = number
```

sctp_sack_delay

Delay until an ACK is generated after receiving a packet. Default: OS specific.

WARNING: a value higher then srto_min can cause a lot of retransmissions (and strange problems). A value higher then srto_max will result in very high connections instability. According to the standard the sack_delay value should be between 200 and 500 ms.

Can be changed at runtime (sctp sack_delay) but it will affect only new associations.

```
sctp_sack_delay = milliseconds
```

sctp_sack_freq

Number of packets received before an ACK is sent (without waiting for the sack_delay to expire). Default: OS specific.

Note: on linux with lksctp up to and including 1.0.9 is not possible to set this value (having it in the config will produce a warning on startup).

Can be changed at runtime (sctp sack_freq) but it will affect only new associations.

```
sctp_sack_freq = number
```

sctp_max_burst

Maximum burst of packets that can be emitted by an association. Default: OS specific.

Can be changed at runtime (sctp max_burst) but it will affect only new associations.

```
sctp_max_burst = number
```

UDP Parameters

udp4_raw

Enables raw socket support for sending UDP IPv4 datagrams (40-50% performance increase on linux multi-cpu).

Possible values: 0 - disabled (default), 1 - enabled, -1 auto.

In “auto” mode it will be enabled if possible (sr started as root or with CAP_NET_RAW). udp4_raw can be used on Linux and FreeBSD. For other BSDs and Darwin one must compile with -DUSE_RAW_SOCKS. On Linux one should also set udp4_raw_mtu if the MTU on any network interface that could be used for sending is smaller then 1500.

The parameter can be set at runtime as long as sr was started with enough privileges (core.udp4_raw).

```
udp4_raw = on
```

udp4_raw_mtu

MTU value used for UDP IPv4 packets when udp4_raw is enabled. It should be set to the minimum MTU of all the network interfaces that could be used for sending. The default value is 1500. Note that on BSDs it does not need to be set (if set it will be ignored, the proper MTU will be used automatically by the kernel). On Linux it should be set.

The parameter can be set at runtime (core.udp4_raw_mtu).

udp4_raw_ttl

TTL value used for UDP IPv4 packets when udp4_raw is enabled. By default it is set to auto mode (-1), meaning that the same TTL will be used as for normal UDP sockets.

The parameter can be set at runtime (core.udp4_raw_ttl).

Blacklist Parameters

dst_blacklist_expire

Alias name: dst_blacklist_ttl

How much time a blacklisted destination will be kept in the blacklist (w/o any update).

```
dst_blacklist_expire = time in s (default 60 s)
```

dst_blacklist_gc_interval

How often the garbage collection will run (eliminating old, expired entries).

```
dst_blacklist_gc_interval = time in s (default 60 s)
```

dst_blacklist_init

If off, the blacklist is not initialized at startup and cannot be enabled runtime, that saves some memory.

```
dst_blacklist_init = on | off (default on)
```

dst_blacklist_mem

Maximum shared memory amount used for keeping the blacklisted destinations.

```
dst_blacklist_mem = size in Kb (default 250 Kb)
```

use_dst_blacklist

Enable the destination blacklist: Each failed send attempt will cause the destination to be added to the blacklist. Before any send, this blacklist will be checked and if a match is found, the send is no longer attempted (an error is returned immediately).

Note: using the blacklist incurs a small performance penalty.

See also doc/dst_blacklist.txt.

```
use_dst_blacklist = on | off (default off)
```

Real-Time Parameters

real_time

Sets real time priority for all the Kamailio processes, or the timers (bitmask).

```
   Possible values:   0  - off
                      1  - the "fast" timer
                      2  - the "slow" timer
                      4  - all processes, except the timers
   Example: real_time= 7 => everything switched to real time priority.
real_time = <int> (flags) (default off)
```

rt_policy

Real time scheduling policy, 0 = SCHED_OTHER, 1= SCHED_RR and 2=SCHED_FIFO

```
rt_policy= <0..3> (default 0)
```

rt_prio

Real time priority used for everything except the timers, if real_time is enabled.

```
rt_prio = <int> (default 0)
```

rt_timer1_policy

Alias name: rt_ftimer_policy

Like rt_policy but for the “fast” timer.

```
rt_timer1_policy=<0..3> (default 0)
```

rt_timer1_prio

Alias name: rt_fast_timer_prio, rt_ftimer_prio

Like rt_prio but for the “fast” timer process (if real_time & 1).

```
rt_timer1_prio=<int> (default 0)
```

rt_timer2_policy

Alias name: rt_stimer_policy

Like rt_policy but for the “slow” timer.

```
rt_timer2_policy=<0..3> (default 0)
```

rt_timer2_prio

Alias name: rt_stimer_prio

Like rt_prio but for the “slow” timer.

```
rt_timer2_prio=<int> (default 0)
```

Core Functions

Functions exported by core that can be used in route blocks.

add_local_rport

Add **rport** parameter to local generated Via header – see RFC3581. In effect for forwarded SIP requests.

Example of usage:

```
add_local_rport();
```

avpflags

break

'break' statement can be used to end a 'case' block in a 'switch' statement or exit from a 'while' statement.

drop

Stop the execution of the configuration script and alter the implicit action which is done afterwards.

If the function is called in a 'branch_route' then the branch is discarded (implicit action for 'branch_route' is to forward the request).

If the function is called in the default 'onreply_route' then you can drop any response. If the function is called in a named 'onreply_route' (transaction stateful) then any provisional reply is discarded. (Implicit action for 'onreply_route' is to send the reply upstream according to Via header.)

Example of usage:

```
  onreply_route {
      if(status=="200") {
          drop(); # this works
      }
  }
  onreply_route[FOOBAR] {
      if(status=="200") {
          drop(); # this is ignored
      }
  }
```

exit

Stop the execution of the configuration script – it has the same behaviour as return(0). It does not affect the implicit action to be taken after script execution.

```
route {
  if (route(2)) {
    xlog("L_NOTICE","method $rm is INVITE\n");
  } else {
    xlog("L_NOTICE","method is $rm\n");
  };
}
route[2] {
  if (is_method("INVITE")) {
    return(1);
  } else if (is_method("REGISTER")) {
    return(-1);
  } else if (is_method("MESSAGE")) {
    sl_send_reply("403","IM not allowed");
    exit;
  };
}
```

error

exec

force_rport

Force_rport() adds the rport parameter to the first Via header of the received message. Thus, Kamailio will add the received port to the top most Via header in the SIP message, even if the client does not indicate support for rport. This enables subsequent SIP messages to return to the proper port later on in a SIP transaction.

This is useful for NAT traversal, to enforce symmetric response signaling.

The rport parameter is defined in RFC 3581.

Note: there is also a force_rport parameter which changes the gobal behavior of the SIP proxy.

Example of usage:

```
force_rport();
```

add_rport

Alias for force_rport();

force_send_socket

Force to send the message from the specified socket (it _must_ be one of the sockets specified with the “listen” directive). If the protocol doesn't match (e.g. UDP message “forced” to a TCP socket) the closest socket of the same protocol is used.

This function does not support pseudo-variables, use the set_send_socket function from the corex module instead.

Example of usage:

```
force_send_socket(10.10.10.10:5060);
force_send_socket(udp:10.10.10.10:5060);
```

force_tcp_alias

Alias name: add_tcp_alias

force_tcp_alias(port)

adds a tcp port alias for the current connection (if tcp). Useful if you want to send all the trafic to port_alias through the same connection this request came from [it could help for firewall or nat traversal]. With no parameters adds the port from the message via as the alias. When the “aliased” connection is closed (e.g. it's idle for too much time), all the port aliases are removed.

forward

Forward the SIP request to destination stored in $du in stateless mode.

Example of usage:

```
$du = "sip:10.0.0.10:5060;transport=tcp";
forward();
```

isavpflagset

isflagset

Test if a flag is set for current processed message (if the flag value is 1). The value of the parameter can be in range of 0..31.

For more see: https://www.kamailio.org/wiki/tutorials/kamailio-flag-operations.

Example of usage:

```
if(isflagset(3)) {
    log("flag 3 is set\n");
};
```

Kamailio also supports named flags. They have to be declared at the beginning of the config file with:

```
flags  flag1_name[:position],  flag2_name ...
```

Example:

```
flags test, a:1, b:2 ;
route{
       setflag(test);
       if (isflagset(a)){ # equiv. to isflagset(1)
         ....
       }
       resetflag(b);  # equiv. to resetflag(2)
```

is_int

Checks if a pseudo variable argument contains integer value.

```
if(is_int("$avp(foobar)")) {
  log("foobar contains an integer\n");
}
```

log

Write text message to standard error terminal or syslog. You can specify the log level as first parameter.

For more see: http://www.kamailio.org/dokuwiki/doku.php/tutorials:debug-syslog-messages

Example of usage:

```
log("just some text message\n");
```

prefix

Add the string parameter in front of username in R-URI.

Example of usage:

```
  prefix("00");
```

resetavpflag

resetflag

return

The return() function allows you to return any integer value from a called route() block. You can test the value returned by a route using $retcode or $? variable.

return(0) is same as exit();

In bool expressions:

- Negative is FALSE
- Positive is TRUE

If no value is specified, or a route reaches its end without executing a return statement, it returns 1. If return is used in the top level route is equivalent with exit [val].

Example usage:

```
route {
  if (route(2)) {
    xlog("L_NOTICE","method $rm is INVITE\n");
  } else {
    xlog("L_NOTICE","method $rm is REGISTER\n");
  };
}
route[2] {
  if (is_method("INVITE")) {
    return(1);
  } else if (is_method("REGISTER")) {
    return(-1);
  } else {
    return(0);
  };
}
```

revert_uri

Set the R-URI to the value of the R-URI as it was when the request was received by server (undo all changes of R-URI).

Example of usage:

```
  revert_uri();
```

rewritehostport

Alias name: sethostport, sethp

Rewrite the domain part and port of the R-URI with the value of function's parameter. Other parts of the R-URI like username and URI parameters remain unchanged.

Example of usage:

```
rewritehostport("1.2.3.4:5080");
```

rewritehostporttrans

Alias name: sethostporttrans, sethpt

Rewrite the domain part and port of the R-URI with the value of function's parameter. Also allows to specify the transport parameter. Other parts of the R-URI like username and URI parameters remain unchanged.

Example of usage:

```
rewritehostporttrans("1.2.3.4:5080");
```

rewritehost

Alias name: sethost, seth

Rewrite the domain part of the R-URI with the value of function's parameter. Other parts of the R-URI like username, port and URI parameters remain unchanged.

Example of usage:

```
rewritehost("1.2.3.4");
```

rewriteport

Alias name: setport, setp

Rewrites/sets the port part of the R-URI with the value of function's parameter.

Example of usage:

```
rewriteport("5070");
```

rewriteuri

Alias name: seturi

Rewrite the request URI.

Example of usage:

```
rewriteuri("sip:test@kamailio.org");
```

rewriteuserpass

Alias name: setuserpass, setup

Rewrite the password part of the R-URI with the value of function's parameter.

Example of usage:

```
rewriteuserpass("my_secret_passwd");
```

rewriteuser

Alias name: setuser, setu

Rewrite the user part of the R-URI with the value of function's parameter.

Example of usage:

```
rewriteuser("newuser");
```

route

Execute route block given in parameter. Parameter may be name of the block or a string valued expression.

Examples of usage:

```
route(REGISTER_REQUEST);
route(@received.proto + "_proto_" + $var(route_set));
```

set_advertised_address

Same as 'advertised_address' but it affects only the current message. It has priority if 'advertised_address' is also set.

Example of usage:

```
set_advertised_address("kamailio.org");
```

set_advertised_port

Same as 'advertised_port' but it affects only the current message. It has priority over 'advertised_port'.

Example of usage:

```
set_advertised_port(5080);
```

set_forward_no_connect

The message will be forwarded only if there is already an existing connection to the destination. It applies only to connection oriented protocols like TCP and TLS (TODO: SCTP), for UDP it will be ignored. The behavior depends in which route block the function is called:

- normal request route: affects stateless forwards and tm. For tm it affects all the branches and the possible retransmissions (in fact there are no retransmission for TCP/TLS).
- onreply_route[0] (stateless): equivalent to set_reply_*() (it's better to use set_reply_* though)
- onreply_route[!=0] (tm): ignored
- branch_route: affects the current branch only (all messages sent on this branch, like possible retransmissions and CANCELs).
- onsend_route: like branch route

Example of usage:

```
route {
  ...
  if (lookup()) {
    //requests to local users. They are usually behind NAT so it does not make sense to try
    //to establish a new TCP connection
    set_forward_no_connect();
    t_relay();
  }
  ...
}
```

set_forward_close

Try to close the connection (the one on which the message is sent out) after forwarding the current message. Can be used in same route blocks as set_forward_no_connect().

Note: Use with care as you might not receive the replies anymore as the connection is closed.

set_reply_no_connect

Like set_forward_no_connect(), but for replies to the current message (local generated replies and replies forwarded by tm). The behavior depends in which route block the function is called:

- normal request route: affects all replies sent back on the transaction (either local or forwarded) and all local stateless replies (sl_reply()).
- onreply_route: affects the current reply (so the send_flags set in the onreply_route will be used if the reply for which they were set is the winning final reply or it's a provisional reply that is forwarded)
- branch_route: ignored.
- onsend_route: ignored

Example of usage:

```
route[4] {
  //requests from local users. There are usually behind NAT so it does not make sense to try
  //to establish a new TCP connection for the replies
  set_reply_no_connect();
  // do authentication and call routing
  ...
```

```
}
```

set_reply_close

Like set_reply_no_connect, but closes the TCP connection after sending. Can be used in same route blocks as set_reply_no_connect.

Example of usage:

```
route {
  ...
  if (...caller-is-not-registered...) {
    // reject unregistered client
    // if request was received via TCP/TLS close the connection, as
    // this may trigger re-registration of the client.
    set_reply_close();
    sl_send_reply("403","REGISTER first");
    exit;
  }
  ...
}
```

setavpflag

setflag

Set a flag for current processed message. The value of the parameter can be in range of 0..31. The flags are used to mark the message for special processing (e.g., accounting) or to keep some state (e.g., message authenticated).

For more see: https://www.kamailio.org/wiki/tutorials/kamailio-flag-operations .

Example of usage:

```
setflag(3);
```

strip

Strip the first N-th characters from username of R-URI (N is the value of the parameter).

Example of usage:

```
strip(3);
```

strip_tail

Strip the last N-th characters from username of R-URI (N is the value of the parameter).

Example of usage:

```
strip_tail(3);
```

udp_mtu_try_proto(proto)

- proto - TCP|TLS|SCTP|UDP - like udp_mtu_try_proto global parameter but works on a per packet basis and not globally.

Example:

```
if($rd=="10.10.10.10")
    udp_mtu_try_proto(SCTP);
```

userphone

Add “user=phone” parameter to R-URI.

Custom Global Parameters

These are parameters that can be defined by the writer of kamailio.cfg in order to be used inside routing blocks. One of the important properties for custom global parameters is that their value can be changed at runtime via RPC commands, without restarting Kamailio.

The definition of a custom global parameter must follow the pattern:

```
group.variable = value desc "description"
```

The value can be a quoted string or integer number.

Example:

```
pstn.gw_ip="1.2.3.4" desc "PSTN GW Address"
```

The custom global parameter can be accessed inside a routing block via:

```
$sel(cfg_get.group.variable)
```

Example:

```
$ru ="sip:"+ $rU +"@"+ $sel(cfg_get.pstn.gw_ip);
```

Note: Some words cannot be used as (part of) names for custom variables or groups, and if they are used a syntax error is logged by kamailio. These keywords are: “yes”, “true”, “on”, “enable”, “no”, “false”, “off”, “disable”, “udp”, “UDP”, “tcp”, “TCP”, “tls”, “TLS”, “sctp”, “SCTP”, “ws”, “WS”, “wss”, “WSS”, “inet”, “INET”, “inet6”, “INET6”, “sslv23”, “SSLv23”, “SSLV23”, “sslv2”, “SSLv2”, “SSLV2”, “sslv3”, “SSLv3”, “SSLV3”, “tlsv1”, “TLSv1”, “TLSV1”

Routing Blocks

The routing blocks are the parts of the configuration file executed by kamailio at runtime. They can be seen as blocks of actions similar to functions (or procedures) from common programming languages.

A routing block is identified by a specific token, followed by a name in between square brackets and actions in between curly braces.

```
route_block_id[NAME]{
```

```
  ACTIONS
}
```

The name can be any alphanumeric string, with specific routing blocks enforcing a particular format.

Note: route(number) is equivalent to route(“number”).

Route blocks can be executed on network events (e.g., receiving a SIP message), timer events (e.g., retransmission timeout) or particular events specific to modules.

There can be so called sub-route blocks, which can be invoked from another route blocks, like a function. Invocation is done with 'route' followed by the name of sub-route to execute, enclosed in between parentheses.

Example:

```
request_route{
  ...
  route("test");
  ...
}

route["test"]{
  ...
}
```

request_route

Request routing block - is executed for each SIP request.

It contains a set of actions to be executed for SIP requests received from the network. It is the equivalent of *main()* function for handling the SIP requests.

For backward compatibility reasons, the main request 'route' block can be identified by 'route{…}' or 'route[0]{…}'.

The implicit action after execution of the main route block is to drop the SIP request. To send a reply or forward the request, explicit actions (e.g., sl_send_reply(), forward(), t_relay()) must be called inside the route block.

Example of usage:

```
request_route {if(is_method("OPTIONS")){# send reply for each options request
        sl_send_reply("200","ok");exit();}
     route(FWD);}
route[FWD]{# forward according to uri
     forward();}
```

route

This block is used to define 'sub-routes' - group of actions that can be executed from another routing block. Originally targeted as being executed from 'request_route', it can be executed now from all the other blocks. Be sure you put there the actions valid for the root routing block executing the sub-route.

The definition of the sub-route block follows the general rules, with a name in between square brackets and actions between curly braces. A sub-route can return an integer value back to the routing block that executed it. The return code can be retrieved via $rc variables.

Evaluation of the return of a subroute is done with following rules:

- negative value is evaluated as false
- 0 - is interpreted as **exit**
- positive value is evaluated as true

```
request_route {if(route(POSITIVE)){
    xlog("return number is positive\n");}if(! route(NEGATIVE)){
    xlog("return number is negative\n");}if( route(ZERO)){
    xlog("this log message does not appear\n");}}

route[POSITIVE]{return10;}

route[NEGATIVE]{return-8;}

route[ZERO]{return0;}
```

A sub-route can execute another sub-route. There is a limit to the number of recursive levels, avoiding ending up in infinite loops – see **max_recursive_level** global parameter.

The sub-route blocks allow to make the configuration file modular, simplifying the logic and helping to avoid duplication of actions.

branch_route

Request's branch routing block. It contains a set of actions to be taken for each branch of a SIP request. It is executed only by TM module after it was armed via t_on_branch(“branch_route_index”).

Example of usage:

```
request_route {
    lookup("location");
    t_on_branch("OUT");if(!t_relay()){
        sl_send_reply("500","relaying failed");}}
branch_route[OUT]{if(uri=~"10\.10\.10\.10"){# discard branches that go to 10.10.10.10
        drop();}}
```

failure_route

Failed transaction routing block. It contains a set of actions to be taken each transaction that received only negative replies (>=300) for all branches. The 'failure_route' is executed only by TM module after it was armed via t_on_failure(“failure_route_index”).

Note that in 'failure_route' is processed the request that initiated the transaction, not the reply .

Example of usage:

```
request_route {
    lookup("location");
    t_on_failure("TOVOICEMAIL");if(!t_relay()){
        sl_send_reply("500","relaying failed");}}
failure_route[TOVOICEMAIL]{if(is_method("INVITE")){# call failed - relay to voice mail
         t_relay_to_udp("voicemail.server.com","5060");}}
```

reply_route

Main SIP response (reply) handling block - it contains a set of actions to be executed for SIP replies. It is executed for all replies received from the network.

It does not have a name and it is executed by the core, before any other module handling the SIP reply. It is triggered only by SIP replies received on the network.

There is no network route that can be enforced for a SIP reply - it is sent based on Via header, according to SIP RFC3261 - therefore no dedicated actions for forwarding the reply must be used in this block.

This routing block is optional, if missing, the SIP reply is sent to the address in 2nd Via header.

One can decide to drop a SIP reply by using **drop** action.

Example:

```
reply_route {if(status=="128"){
    drop;}}
```

Note: for backward compatibility reasons, the main 'reply' routing block can be also identified by 'onreply_route {…}' or 'onreply_route[0] {…}'.

onreply_route

SIP reply routing block executed by **tm** module. It contains a set of actions to be taken for SIP replies in the contect of an active transaction.

The 'onreply_route' must be armed for the SIP requests whose replies should be processed within it, via t_on_reply("onreply_route_index").

Core 'reply_route' block is executed before a possible **tm** 'onreply_route' block.

```
request_route {
    lookup("location");
    t_on_reply("LOGRPL");if(!t_relay()){
        sl_send_reply("500","relaying failed");}}

reply_route {if(!t_check_trans()){
        drop;}}

onreply_route[LOGRPL]{if(status=~"1[0-9][0-9]"){log("provisional response\n");}}
```

onsend_route

The route is executed in when a SIP request is sent out. Only a limited number of commands are allowed (drop, if + all the checks, msg flag manipulations, send(), log(), textops::search()).

In this route the final destination of the message is available and can be checked (with snd_ip, snd_port, to_ip, to_port, snd_proto, snd_af).

This route is executed only when forwarding requests - it is not executed for replies, retransmissions, or locally generated messages (e.g. via fifo uac).

Example:

```
onsend_route {if(to_ip==1.2.3.4 &&!isflagset(12)){log(1,"message blocked\n");
    drop;}}
```

- snd_ip, snd_port - behave like src_ip/src_port, but contain the ip/port Kamailio will use to send the message
- to_ip, to_port - like above, but contain the ip/port the message will be sent to (not to be confused with dst_ip/dst_port, which are the destination of the original received request: Kamailio's ip and port on which the message was received)
- snd_proto, snd_af - behave like proto/af but contain the protocol/address family that Kamailio will use to send the message
- msg:len - when used in an onsend_route, msg:len will contain the length of the message on the wire (after all the changes in the script are applied, Vias are added a.s.o) and not the lentgh of the original message.

event_route

Generic type of route executed when specific events happen.

Prototype: event_route[groupid:eventid]

- groupid - should be the name of the module that triggers the event
- eventid - some meaningful short text describing the event

Core Event Routes

Implementations:

- **event_route[core:worker-one-init]** - executed by core after the first udp sip worker process executed the child_init() for all modules, before starting to process sip traffic
  - note that due to forking, other sip workers can get faster to listening for sip traffic

```
event_route[core:worker-one-init]{
    xlog("L_INFO","Hello world\n");}
```

- **event_route[core:msg-received]** - executed when a message is received from the network. It runs with a faked request and makes available the $rcv(key) variables to access what was received and related attribtues.
  - it has to be enabled with received_route_mode global parameter. For usage via Kemi, set kemi.received_route_callback global parameter.
  - if drop is executed, the received message is no longer processed

```
event_route[core:msg-received]{
  xlog("rcv on $rcv(af)/$rcv(proto): ($rcv(len)) [$rcv(buf)] from [$rcv(srcip):$rcv(srcport)] to [$rcv(rcvip):$rcv(rcvport)]\n");if($rcv(srcip)=="1.2.3.4"){
    drop;}}
```

Module Event Routes

Here are only a few examples, to see if a module exports event_route blocks and when they are executed, check the readme of the module.

- **event_route[htable:mod-init]** - executed by **htable** module after all modules have been initialised. Good for initialising values in hash tables.

```
modparam("htable","htable","a=>size=4;")

event_route[htable:mod-init]{
  $sht(a=>calls-to::10.10.10.10)=0;
  $sht(a=>max-calls-to::10.10.10.10)=100;}

request_route {if(is_method("INVITE")&&!has_totag()){switch($rd){case"10.10.10.10":
        lock("calls-to::10.10.10.10");
        $sht(a=>calls-to::10.10.10.10)=
```

```
            $sht(a=>calls-to::10.10.10.10)+1;
        unlock("calls-to::10.10.10.10");if($sht(a=>calls-to::10.10.10.10)>$sht(a=>max-calls-to::10.10.10.10)){
           sl_send_reply("500","To many calls to .10");exit;}break;
      ...
    }}}
```

- **event_route [tm:local-request]** - executed on locally generated requests.

```
event_route [tm:local-request]{# Handle locally generated requests
  xlog("L_INFO","Routing locally generated $rm to <$ru>\n");
  t_set_fr(10000,10000);}
```

- **event_route [tm:branch-failure]** - executed on all failure responses.

```
request_route {
    ...
    t_on_branch_failure("myroute");
    t_relay();}

event_route[tm:branch-failure:myroute]{
  xlog("L_INFO","Handling $T_reply_code response to $rm to <$ru>\n");if(t_check_status("430")){# Outbound flow failed
    unregister("location","$tu","$T_reply_ruid");if(t_next_contact_flow()){
      t_relay();}}}
```

Script Statements

if

IF-ELSE statement

Prototype:

```
    if(expr) {
       actions;
    } else {
       actions;
    }
```

The 'expr' should be a valid logical expression.

The logical operators that can be used in 'expr':

```
  ==      equal
  !=      not equal
  =~      regular expression matching: Note: Posix regular expressions will be used, e.g. use [[:digit:]]{3} instead of \d\d\d
  !~      regular expression not-matching (NOT PORTED from Kamailio 1.x, use '!(x =~ y)')
  >       greater
  >=      greater or equal
  <       less
  <=      less or equal
  &&      logical AND
```

```
||      logical OR
!       logical NOT
[ ... ] test operator - inside can be any arithmetic expression
```

Example of usage:

```
if(is_method("INVITE"))
{
    log("this sip message is an invite\n");
} else {
    log("this sip message is not an invite\n");
}
```

switch

SWITCH statement - it can be used to test the value of a pseudo-variable.

IMPORTANT NOTE: 'break' can be used only to mark the end of a 'case' branch (as it is in shell scripts). If you are trying to use 'break' outside a 'case' block the script will return error – you must use 'return' there.

Example of usage:

```
    route {
        route(1);
        switch($retcode)
        {
        case -1:
            log("process INVITE requests here\n");
        break;
        case 1:
            log("process REGISTER requests here\n");
        break;
        case 2:
        case 3:
            log("process SUBSCRIBE and NOTIFY requests here\n");
        break;
        default:
            log("process other requests here\n");
}

 # switch of R-URI username
 switch($rU)
 {
     case "101":
         log("destination number is 101\n");
     break;
     case "102":
         log("destination number is 102\n");
     break;
     case "103":
     case "104":
         log("destination number is 103 or 104\n");
     break;
     default:
```

```
            log("unknown destination number\n");
    }
}

route[1]{
    if(is_method("INVITE"))
    {
        return(-1);
    };
    if(is_method("REGISTER"))
        return(1);
    }
    if(is_method("SUBSCRIBE"))
        return(2);
    }
    if(is_method("NOTIFY"))
        return(3);
    }
    return(-2);
}
```

NOTE: take care while using 'return' - 'return(0)' stops the execution of the script.

while

while statement

Example of usage:

```
$var(i) = 0;
while($var(i) < 10)
{
    xlog("counter: $var(i)\n");
    $var(i) = $var(i) + 1;
}
```

Script Operations

Assignments together with string and arithmetic operations can be done directly in configuration file.

Assignment

Assignments can be done like in C, via '=' (equal). The following pseudo-variables can be used in left side of an assignment:

- Unordered List Item AVPs - to set the value of an AVP
- script variables ($var(…)) - to set the value of a script variable
- shared variables ($shv(…))
- $ru - to set R-URI
- $rd - to set domain part of R-URI
- $rU - to set user part of R-URI

- $rp - to set the port of R-URI
- $du - to set dst URI
- $fs - to set send socket
- $br - to set branch
- $mf - to set message flags value
- $sf - to set script flags value
- $bf - to set branch flags value

```
$var(a) = 123;
```

For avp's there a way to remove all values and assign a single value in one statement (in other words, delete existing AVPs with same name, add a new one with the right side value). This replaces the := assignment operator from kamailio < 3.0.

```
$(avp(i:3)[*]) = 123;
$(avp(i:3)[*]) = $null;
```

String Operations

For strings, '+' is available to concatenate.

```
$var(a) = "test";
$var(b) = "sip:" + $var(a) + "@" + $fd;
```

Arithmetic Operations

For numbers, one can use:

- + : plus
- - : minus
- / : divide
- * : multiply
- % : modulo (Kamailio uses 'mod' instead of '%')
- | : bitwise OR
- & : bitwise AND
- ^ : bitwise XOR
- ~ : bitwise NOT
- << : bitwise left shift
- >> : bitwise right shift

Example:

```
$var(a) = 4 + ( 7 & ( ~2 ) );
```

NOTE: to ensure the priority of operands in expression evaluations do use *parenthesis*.

Arithmetic expressions can be used in condition expressions.

```
if( $var(a) & 4 )
    log("var a has third bit set\n");
```

Operators

1. type casts operators: (int), (str).
2. string comparison: eq, ne
3. integer comparison: ieq, ine

Note: The names are not yet final (use them at your own risk). Future version might use ==/!= only for ints (ieq/ine) and eq/ne for strings (under debate). They are almost equivalent to == or !=, but they force the conversion of their operands (eq to string and ieq to int), allowing among other things better type checking on startup and more optimizations.

Non equiv. examples:

0 == “” (true) is not equivalent to 0 eq “” (false: it evaluates to “0” eq “”).

“a” ieq “b” (true: (int)“a” is 0 and (int)“b” is 0) is not equivalent to “a” == “b” (false).

Note: internally == and != are converted on startup to eq/ne/ieq/ine whenever possible (both operand types can be safely determined at start time and they are the same).

1. Kamailio tries to guess what the user wanted when operators that support multiple types are used on different typed operands. In general convert the right operand to the type of the left operand and then perform the operation. Exception: the left operand is undef. This applies to the following operators: +, == and !=.

Special case: undef as left operand:

```
For +: undef + expr -> undef is converted to string => "" + expr.
For == and !=:   undef == expr -> undef is converted to type_of expr.
If expr is undef, then undef == undef is true (internally is converted
to string).
```

1. expression evaluation changes: Kamailio will auto-convert to integer or string in function of the operators:

int(undef)==0, int(“”)==0, int(“123”)==123, int(“abc”)==0

```
 str(undef)=="", str(123)=="123".
```

1. script operators for dealing with empty/undefined variables

defined expr - returns true if expr is defined, and false if not.

```
                Note: only a standalone avp or pvar can be
                undefined, everything else is defined.
  strlen(expr) - returns the lenght of expr evaluated as string.
  strempty(expr) - returns true if expr evaluates to the empty
                   string (equivalent to expr=="").
  Example: if (defined $v && !strempty($v)) $len=strlen($v);
```

- Old revisions
- Backlinks
- Back to top