tm Module

Jiri Kuthan
FhG FOKUS

Edited by
Jiri Kuthan

tm Module

Edited by and Jiri Kuthanand Jiri Kuthan

Revision History

Revision $Revision: 1.1.2.1 $ $Date: 2003/08/05 21:48:08 $

Table of Contents

List of Examples

Chapter 1. User's Guide

1.1. Overview

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

From user's perspective, there are two major functions : t_relay and t_relay_to. Both setup transaction state, absorb retransmissions from upstream, generate downstream retransmissions and correlate replies to requests. t_relay forwards to current URI (be it original request's URI or a URI changed by some of URI-modifying functions, such as sethost). t_relay_to forwards to a specific address.

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

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

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

1.2. Dependencies

1.2.1. SER Modules

The following modules must be loaded before this module:

- *No dependencies on other SER modules.*

1.2.2. External Libraries or Applications

The following libraries or applications must be installed before running SER with this module loaded:

- *None*.

1.3. Exported Parameters

1.3.1. `fr_timer` (integer)

Timer which hits if no final reply for a request or ACK for a negative INVITE reply arrives (in seconds).

Default value is 30 seconds.

Example 1-1. Set `fr_timer` parameter

```
...
modparam("tm", "fr_timer", 10)
...
```

1.3.2. `fr_inv_timer` (integer)

Timer which hits if no final reply for an INVITE arrives after a provisional message was received (in seconds).

Default value is 120 seconds.

Example 1-2. Set `fr_inv_timer` parameter

```
...
modparam("tm", "fr_inv_timer", 200)
...
```

1.3.3. `wt_timer` (integer)

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

Default value is 5 seconds.

Example 1-3. Set `wt_timer` parameter

```
...
modparam("tm", "wt_timer", 10)
...
```

1.3.4. `delete_timer` (integer)

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

Default value is 2 seconds.

Example 1-4. Set `delete_timer` parameter

```
...
modparam("tm", "delete_timer", 5)
...
```

1.3.5. `retr_timer1p1` (integer)

Retransmission period.

Default value is 1 second.

Example 1-5. Set `retr_timer1p1` parameter

```
...
modparam("tm", "retr_timer1p1", 2)
...
```

1.3.6. `retr_timer1p2` (integer)

Retransmission period.

*Default value is 2 * `retr_timer1p1` second.*

Example 1-6. Set `retr_timer1p2` parameter

```
...
modparam("tm", "retr_timer1p2", 4)
...
```

1.3.7. `retr_timer1p3` (integer)

Retransmission period.

*Default value is 4 * `retr_timer1p1` second.*

Example 1-7. Set `retr_timer1p4` parameter

```
...
modparam("tm", "retr_timer1p3", 8)
...
```

1.3.8. `retr_timer2` (integer)

Maximum retransmission period.

Default value is 4 seconds.

Example 1-8. Set `retr_timer2` parameter

```
...
modparam("tm", "retr_timer2", 8)
...
```

1.3.9. `noisy_ctimer` (integer)

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

Default value is 0 (false).

Example 1-9. Set `noisy_ctimer` parameter

```
...
modparam("tm", "noisy_ctimer", 1)
...
```

1.4. Exported Functions

1.4.1. `t_relay_to(ip, port)`

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

Meaning of the parameters is as follows:

- *ip* - IP address where the message should be sent.
- *port* - Port number.

Example 1-10. `t_relay_to` usage

```
...
t_relay_to("1.2.3.4", "5060");
...
```

1.4.2. `t_relay()`

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

Example 1-11. `t_relay` usage

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

1.4.3. `t_on_negative(reply_route)`

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

Meaning of the parameters is as follows:

- *reply_route* - Reply route block to be called.

Example 1-12. `t_on_negative` usage

```
...
route {
    t_on_negative("1");
    t_relay();
}

reply_route[1] {
    revert_uri();
    setuser("voicemail");
    append_branch();
}
...
```

See test/onr.cfg for a more complex example of combination of serial with parallel forking.

1.4.4. `append_branch()`

Similarly to `t_fork_to`, it extends destination set by a new entry. The difference is that current URI is taken as new entry.

Example 1-13. `append_branch` usage

```
...
set_user("john");
t_fork();
set_user("alice");
t_fork();
t_relay();
...
```

1.4.5. `t_newtran()`

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

Example 1-14. `t_newtran` usage

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

See test/uas.cfg for more examples.

1.4.6. `t_reply(code, reason_phrase)`

Sends a stateful reply after a transaction has been established. See `t_newtran` for usage.

Meaning of the parameters is as follows:

- *code* - Reply code number.
- *reason_phrase* - Reason string.

Example 1-15. `t_reply` usage

```
...
t_reply("404", "Not found");
...
```

1.4.7. `t_lookup_request()`

Checks if a transaction exists. Returns a positive value if so, negative otherwise. Most likely you will not want to use it, as a typicall application of a looku-up is to introduce a new transaction if none was found. However this is safely (atomically) done using `t_newtran`.

Example 1-16. `t_lookup_request` usage

```
...
if (t_lookup_request()) {
    ...
};
...
```

1.4.8. `t_retransmit_reply()`

Retransmits a reply sent previously by UAS transaction.

Example 1-17. `t_retransmit_reply` usage

```
...
t_retransmit_reply();
...
```

1.4.9. `t_release()`

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

Example 1-18. `t_release` usage

```
...
t_release();
...
```

1.4.10. `t_forward_nonack(ip, port)`

mainly for internal usage--forward a non-ACK request statefuly.

Meaning of the parameters is as follows:

- *ip* - IP address where the message should be sent.
- *port* - Port number.

Example 1-19. `t_forward_nonack` usage

```
...
t_forward_nonack("1.2.3.4", "5060");
...
```

1.4.11. External Usage of TM

There are applications which would like to generate SIP transactions without too big involvement in SIP stack, transaction management, etc. An example of such an application is sending instant messages from a website. To address needs of such apps, SER accepts requests for new transactions via fifo pipes too. If you want to enable this feature, start FIFO server with configuration option.

fifo="/tmp/ser_fifo"

Then, an application can easily launch a new transaction by writing a transaction request to this named pipe. The request must follow very simple format, which is

```
:t_uac_from:[<file_name>]\n
<method>\n
<sender's uri>\n
<dst uri>\n
<CR_separated_headers>\n
<body>\n
.\n
\n
```

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

Note the the request write must be atomic, otherwise it might get intermixed with writes from other writers. You can easily use it via Unix command-line tools, see the following example:

```
[jiri@bat jiri]$ cat > /tmp/ser_fifo
:t_uac_from:xxx
MESSAGE
sip:sender@iptel.org
sip:mrx@iptel.org
header:value
foo:bar
bznk:hjhjk
p_header: p_value

body body body
yet body
end of body
.
```

or cat test/transaction.fifo > /tmp/ser_fifo

1.4.12. Known Issues

- We don't have authentication merging on forking.
- Local ACK/CANCELs copy'n'pastes Route and ignores deleted Routes.
- 6xx should be delayed.
- Possibly, performance could be improved by not parsing non-INVITEs, as they do not be replied with 100, and do not result in ACK/CANCELs, and other things which take parsing. However, we need to rethink whether we don't need parsed headers later for something else. Remember, when we now conserver a request in sh_mem, we can't apply any pkg_mem operations to it any more. (that might be redesigned too).
- Another performance improvement may be achieved by not parsing CSeq in replies until reply branch matches branch of an INVITE/CANCEL in transaction table.
- `t_replicate` should be done more cleanly--Vias, Routes, etc. should be removed from a message prior to replicating it (well, does not matter any longer so much as there is a new replication module).
- SNMP support (as nobody cares about SNMP, in particular for TM, I will drop this item soon).

Chapter 2. Developer's Guide

The module does not provide any sort of API to use in other SER modules.

2.1. Defines

- ACK_TAG enables stricter matching of acknowledgemnts including to-tags. Without it, to-tags are ignored. It is disabled by default for two reasons:
  - It eliminates an unlikely race condition in which transaction's to-tag is being rewritten by a 200 OK whereas an ACK is being looked up by to-tag.
  - It makes UACs happy who set wrong to-tags.

  It should not make a difference, as there may be only one negative reply sent upstream and 200/ACKs are not matched as they consititute another transaction. It will make no difference at all when the new magic cookie matching is enabled anyway.

- CANCEL_TAG similarly enables strict matching of CANCELs including to-tags--act of mercy to UACs, who screw up the to-tags (however, it still depends on how forgiving the downstream UAS is). Like with ACK_TAG, all this complex transactions matching goes with RFC3261 (http://www.ietf.org/rfc/rfc3261.txt)'s magic cookie away anyway.

2.2. Functions

2.2.1. `register_tmcb(cb_type, cb_func)`

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

Meaning of the parameters is as follows:

- *cb_type* - Callback type.
- *cb_func* - Callback function.

2.2.2. `load_tm(*import_structure)`

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

Meaning of the parameters is as follows:

- *import_structure* - Pointer to the import structure.

Chapter 3. Frequently Asked Questions

1. Where can I find more about SER?

Take a look at http://iptel.org/ser.

2. Where can I post a question about this module?

First at all check if your question was already answered on one of our mailing lists:

- http://mail.iptel.org/mailman/listinfo/serusers
- http://mail.iptel.org/mailman/listinfo/serdev

E-mails regarding any stable version should be sent to `<serusers@iptel.org>` and e-mail regarding development versions or CVS snapshots should be send to `<serdev@iptel.org>`.

If you want to keep the mail private, send it to `<serhelp@iptel.org>`.

3. How can I report a bug?

Please follow the guidelines provided at: http://iptel.org/ser/bugs