NAME

mod_cc, **DECLARE_CC_MODULE**, **CCV** - Modular Congestion Control

SYNOPSIS

#include <netinet/tcp.h>
#include <netinet/cc/cc.h>
#include <netinet/cc/cc_module.h>

DECLARE_CC_MODULE(*ccname*, *ccalgo*);

CCV(*ccv*, *what*);

DESCRIPTION

The **mod_cc** framework allows congestion control algorithms to be implemented as dynamically loadable kernel modules via the kld(4) facility. Transport protocols can select from the list of available algorithms on a connection-by-connection basis, or use the system default (see mod_cc(4) for more details).

mod_cc modules are identified by an ascii(7) name and set of hook functions encapsulated in a *struct cc_algo*, which has the following members:

```
struct cc_algo {
        char    name[TCP_CA_NAME_MAX];
        int     (*mod_init) (void);
        int     (*mod_destroy) (void);
        size_t  (*cc_data_sz)(void);
        int     (*cb_init) (struct cc_var *ccv, void *ptr);
        void    (*cb_destroy) (struct cc_var *ccv);
        void    (*conn_init) (struct cc_var *ccv);
        void    (*ack_received) (struct cc_var *ccv, uint16_t type);
        void    (*cong_signal) (struct cc_var *ccv, uint32_t type);
        void    (*post_recovery) (struct cc_var *ccv);
        void    (*after_idle) (struct cc_var *ccv);
        int     (*ctl_output)(struct cc_var *, struct sockopt *, void *);
        void    (*rttsample)(struct cc_var *, uint32_t, uint32_t, uint32_t);
        void    (*newround)(struct cc_var *, uint32_t);
};
```

The *name* field identifies the unique name of the algorithm, and should be no longer than TCP_CA_NAME_MAX-1 characters in length (the TCP_CA_NAME_MAX define lives in

<*netinet/tcp.h*> for compatibility reasons).

The *mod_init* function is called when a new module is loaded into the system but before the registration process is complete. It should be implemented if a module needs to set up some global state prior to being available for use by new connections. Returning a non-zero value from *mod_init* will cause the loading of the module to fail.

The *mod_destroy* function is called prior to unloading an existing module from the kernel. It should be implemented if a module needs to clean up any global state before being removed from the kernel. The return value is currently ignored.

The *cc_data_sz* function is called by the socket option code to get the size of data that the *cb_init* function needs. The socket option code then preallocates the modules memory so that the *cb_init* function will not fail (the socket option code uses M_WAITOK with no locks held to do this).

The *cb_init* function is called when a TCP control block *struct tcpcb* is created. It should be implemented if a module needs to allocate memory for storing private per-connection state. Returning a non-zero value from *cb_init* will cause the connection set up to be aborted, terminating the connection as a result. Note that the ptr argument passed to the function should be checked to see if it is non-NULL, if so it is preallocated memory that the cb_init function must use instead of calling malloc itself.

The *cb_destroy* function is called when a TCP control block *struct tcpcb* is destroyed. It should be implemented if a module needs to free memory allocated in *cb_init*.

The *conn_init* function is called when a new connection has been established and variables are being initialised. It should be implemented to initialise congestion control algorithm variables for the newly established connection.

The *ack_received* function is called when a TCP acknowledgement (ACK) packet is received. Modules use the *type* argument as an input to their congestion management algorithms. The ACK types currently reported by the stack are CC_ACK and CC_DUPACK. CC_ACK indicates the received ACK acknowledges previously unacknowledged data. CC_DUPACK indicates the received ACK acknowledges data we have already received an ACK for.

The *cong_signal* function is called when a congestion event is detected by the TCP stack. Modules use the *type* argument as an input to their congestion management algorithms. The congestion event types currently reported by the stack are CC_ECN, CC_RTO, CC_RTO_ERR and CC_NDUPACK. CC_ECN is reported when the TCP stack receives an explicit congestion notification (RFC3168). CC_RTO is reported when the retransmission time out timer fires. CC_RTO_ERR is reported if the retransmission time out timer fired in error. CC_NDUPACK is reported if N duplicate ACKs have been received back-to-back, where N is the fast retransmit duplicate ack threshold (N=3 currently as per RFC5681).

The *post_recovery* function is called after the TCP connection has recovered from a congestion event. It should be implemented to adjust state as required.

The *after_idle* function is called when data transfer resumes after an idle period. It should be implemented to adjust state as required.

The *ctl_output* function is called when getsockopt(2) or setsockopt(2) is called on a tcp(4) socket with the *struct sockopt* pointer forwarded unmodified from the TCP control, and a *void ** pointer to algorithm specific argument.

The *rttsample* function is called to pass round trip time information to the congestion controller. The additional arguments to the function include the microsecond RTT that is being noted, the number of times that the data being acknowledged was retransmitted as well as the flightsize at send. For transports that do not track flightsize at send, this variable will be the current cwnd at the time of the call.

The *newround* function is called each time a new round trip time begins. The montonically increasing round number is also passed to the congestion controller as well. This can be used for various purposes by the congestion controller (e.g Hystart++).

Note that currently not all TCP stacks call the *rttsample* and *newround* function so dependancy on these functions is also dependant upon which TCP stack is in use.

The **DECLARE_CC_MODULE**() macro provides a convenient wrapper around the DECLARE_MODULE(9) macro, and is used to register a **mod_cc** module with the **mod_cc** framework. The *ccname* argument specifies the module’s name. The *ccalgo* argument points to the module’s *struct cc_algo*.

mod_cc modules must instantiate a *struct cc_algo*, but are only required to set the name field, and optionally any of the function pointers. Note that if a module defines the *cb_init* function it also must define a *cc_data_sz* function. This is because when switching from one congestion control module to another the socket option code will preallocate memory for the *cb_init* function. If no memory is allocated by the modules *cb_init* then the *cc_data_sz* function should return 0.

The stack will skip calling any function pointer which is NULL, so there is no requirement to implement any of the function pointers (with the exception of the cb_init <-> cc_data_sz dependancy noted above). Using the C99 designated initialiser feature to set fields is encouraged.

Each function pointer which deals with congestion control state is passed a pointer to a *struct cc_var*, which has the following members:

```
struct cc_var {
        void            *cc_data;
        int             bytes_this_ack;
        tcp_seq         curack;
        uint32_t  flags;
        int             type;
        union ccv_container {
                struct tcpcb            *tcp;
                struct sctp_nets    *sctp;
        } ccvc;
        uint16_t  nsegs;
        uint8_t         labc;
};
```

struct cc_var groups congestion control related variables into a single, embeddable structure and adds a layer of indirection to accessing transport protocol control blocks. The eventual goal is to allow a single set of **mod_cc** modules to be shared between all congestion aware transport protocols, though currently only tcp(4) is supported.

To aid the eventual transition towards this goal, direct use of variables from the transport protocol's data structures is strongly discouraged. However, it is inevitable at the current time to require access to some of these variables, and so the **CCV**() macro exists as a convenience accessor. The *ccv* argument points to the *struct cc_var* passed into the function by the **mod_cc** framework. The *what* argument specifies the name of the variable to access.

Apart from the *type* and *ccv_container* fields, the remaining fields in *struct cc_var* are for use by **mod_cc** modules.

The *cc_data* field is available for algorithms requiring additional per-connection state to attach a dynamic memory pointer to. The memory should be allocated and attached in the module's *cb_init* hook function.

The *bytes_this_ack* field specifies the number of new bytes acknowledged by the most recently received ACK packet. It is only valid in the *ack_received* hook function.

The *curack* field specifies the sequence number of the most recently received ACK packet. It is only valid in the *ack_received*, *cong_signal* and *post_recovery* hook functions.

The *flags* field is used to pass useful information from the stack to a **mod_cc** module. The CCF_ABC_SENTAWND flag is relevant in *ack_received* and is set when appropriate byte counting (RFC3465) has counted a window’s worth of bytes has been sent. It is the module’s responsibility to clear the flag after it has processed the signal. The CCF_CWND_LIMITED flag is relevant in *ack_received* and is set when the connection’s ability to send data is currently constrained by the value of the congestion window. Algorithms should use the absence of this flag being set to avoid accumulating a large difference between the congestion window and send window.

The *nsegs* variable is used to pass in how much compression was done by the local LRO system. So for example if LRO pushed three in-order acknowledgements into one acknowledgement the variable would be set to three.

The *labc* variable is used in conjunction with the CCF_USE_LOCAL_ABC flag to override what labc variable the congestion controller will use for this particular acknowledgement.

SEE ALSO

cc_cdg(4), cc_chd(4), cc_cubic(4), cc_dctcp(4), cc_hd(4), cc_htcp(4), cc_newreno(4), cc_vegas(4), mod_cc(4), tcp(4)

ACKNOWLEDGEMENTS

Development and testing of this software were made possible in part by grants from the FreeBSD Foundation and Cisco University Research Program Fund at Community Foundation Silicon Valley.

FUTURE WORK

Integrate with sctp(4).

HISTORY

The modular Congestion Control (CC) framework first appeared in FreeBSD 9.0.

The framework was first released in 2007 by James Healy and Lawrence Stewart whilst working on the NewTCP research project at Swinburne University of Technology’s Centre for Advanced Internet Architectures, Melbourne, Australia, which was made possible in part by a grant from the Cisco University Research Program Fund at Community Foundation Silicon Valley. More details are available at:

http://caia.swin.edu.au/urp/newtcp/

AUTHORS

The **mod_cc** framework was written by Lawrence Stewart <*lstewart@FreeBSD.org*>, James Healy <*jimmy@deefa.com*> and David Hayes <*david.hayes@ieee.org*>.

This manual page was written by David Hayes <*david.hayes@ieee.org*> and Lawrence Stewart <*lstewart@FreeBSD.org*>.