Pluggable TCP & SCTP Congestion Control Design Specification

Artem Kachitchkine

January 12, 2010

Abstract

We propose to replace hard-coded congestion control in Solaris TCP and SCTP implementations with a pluggable module framework. Based on this new framework, we intend to implement several alternative congestion control algorithms that can achieve better bandwidth utilization, among other useful properties, compared to the existing NewReno algorithm.

Contents

1 TCP-style Congestion Control Overview

Network congestion occurs when nodes attempt to send more data than the network can accommodate, most commonly leading to router buffer overflow. Congestion control is different from flow control, the latter being used to avoid receiver's buffer overflows. Methods of controlling congestion can be divided into:

- *Network-assisted*, where routers proactively send packets asking the sender to reduce its sending rate.
- *End-to-end*, where the sender has to infer congestion from events such as packet loss. These methods can be misled by some types of links, e.g. wireless, that routinely drop packets without congestion.

Initially TCP only provided flow control (receiver window) but no congestion control. The seminal 1988 paper by Van Jacobson [1] introduced basic principles of TCP congestion control, including sender's congestion window, slow start and congestion avoidance. Van Jacobson's algorithm evolved into what is known today as NewReno and is what implemented in Solaris.

In the following years, much research was done to improve on the original, to pursue various goals [2], such as to optimize for networks with a high bandwidth-delay product (aka long fat pipes) or to avoid congestion before it happens by sensing round-trip tendencies. Many algorithms have been developed, all within the constraints of the original slow start/congestion avoidance paradigm (as required by the standards), but one thing remains unchanged: there is no one algorithm that works well in every possible network.

Hence the need to have a pluggable framework for different algorithms. Linux has had it for some time and that's where most of research has been happening lately. FreeBSD has experimental support for it. We propose to add such a framework to Solaris.

SCTP congestion control have been derived from TCP, with a multihoming twist. For each destination address, a separate set of flow and congestion control parameters is maintained, so from the network perspective, an SCTP association with N paths behaves similarly to N TCP connections. This makes SCTP congestion control TCP-friendly, which means it can fairly share bandwidth with TCP on the same network.

This proposal covers both TCP and SCTP.

2 Administrative Model

TCP-style congestion control algorithms will be provided by a new type of kernel module, `tcpcong`. Such modules will be installed under `/kernel/tcpcong` and loaded on demand whenever a connection is created that requires a particular algorithm. See Section 5 for the kernel interfaces between modules and the rest of the networking stack.

`"tcpcong"` was chosen to reflect that this is specifically TCP-style congestion control - even though it will be used by SCTP as well, it is essentially the same mechanism involving inherently TCP notions such as congestion window, slow start, ACKs, etc. It would be incorrect to use `"tpcong"` for "transport protocol congestion" as other transport protocols might have entirely type of congestion control; `"ipcong"` and `"netcong"` do not quite fit either; and simply `"cong"` would wrongly suggest that congestion control is limited to networking. We feel that while the proposed term is slightly confusing, it is less confusing than the alternatives.

Before an algorithm can be used, it should be registered with the system. We propose to extend `ipadm(1M)` [4] for this purpose. To register/unregister an algorithm:

```
ipadm add-cong algorithm
ipadm remove-cong algorithm
```

To list available algorithms:

```
ipadm show-cong
```

There is no subcommand to list 'registrable' algorithms - it is our intention that all congestion algorithms will be automatically registered when they are installed on the system. The only time a sysadmin should need to use the `add-cong` subcommand is to undo her previous `remove-cong` invocation.

The new `cong` property can be used to override the default `newreno` algorithm:

```
ipadm set-prop -p cong=algorithm protocol
```

where `protocol` is either `tcp` or `sctp`. When invoked from within a zone, the property will be applied only to that zone. Future work may provide even finer granularity based on connection 4-tuples.

3 Application Interfaces

Since the majority of congestion research in recent years has been done in Linux, we propose to provide Linux-compatible APIs to encourage developers. Same interfaces have been adopted by FreeBSD as well. For instance, the widely used network performance tool `iperf` can be simply recompiled on Solaris to enable its congestion algorithm command line option.

Socket options are an established mechanism for retrieving and setting, using `getsockopt(3SOCKET)` and `setsockopt(3SOCKET)`, protocol parameters on a per connection basis.

3.1 TCP_CONGESTION

New socket option `TCP_CONGESTION` can be used to get or set the current congestion control algorithm. It takes a variable-length string value for algorithm name.

3.2 SCTP_CONGESTION

New socket option `SCTP_CONGESTION` is the SCTP counterpart of `TCP_CONGESTION`.

3.3 TCP_INFO

New socket option `TCP_INFO` can be used to get current connection's congestion window and slow start threshold among others. It returns `struct tcp_info` shown below. Fields that have no meaning in Solaris will be set to zero.

```
struct tcp_info {
        uint8_t    tcpi_state;
        uint8_t    tcpi_ca_state;
        uint8_t    tcpi_retransmits;
        uint8_t    tcpi_probes;
        uint8_t    tcpi_backoff;
        uint8_t    tcpi_options;
        uint8_t    tcpi_snd_wscale : 4, tcpi_rcv_wscale : 4;

        uint32_t   tcpi_rto;       /* in usec */
        uint32_t   tcpi_ato;       /* in usec */
        uint32_t   tcpi_snd_mss;
        uint32_t   tcpi_rcv_mss;

        /* these counters are in segments */
        uint32_t   tcpi_unacked;
        uint32_t   tcpi_sacked;
        uint32_t   tcpi_lost;
        uint32_t   tcpi_retrans;
        uint32_t   tcpi_fackets;

        /* Times, in msec */
        uint32_t   tcpi_last_data_sent;
        uint32_t   tcpi_last_ack_sent;
        uint32_t   tcpi_last_data_recv;
        uint32_t   tcpi_last_ack_recv;

        /* Metrics. */
```

```
        uint32_t   tcpi_pmtu;
        uint32_t   tcpi_rcv_ssthresh;  /* in segments */
        uint32_t   tcpi_rtt;           /* in usec */
        uint32_t   tcpi_rttvar;        /* in usec */
        uint32_t   tcpi_snd_ssthresh;  /* in segments */
        uint32_t   tcpi_snd_cwnd;      /* in segments */
        uint32_t   tcpi_advmss;
        uint32_t   tcpi_reordering;

        uint32_t   tcpi_rcv_rtt;       /* in usec */
        uint32_t   tcpi_rcv_space;

        uint32_t   tcpi_total_retrans;
};
```

3.4 SCTP_GET_PEER_ADDR_INFO

Socket option `SCTP_GET_PEER_ADDR_INFO` is defined in IETF Draft: *Sockets API Extensions for Stream Control Transmission Protocol (SCTP)* [5] and is already supported in Solaris. This option can be used to get information about an SCTP peer address of an association. The following structure is used:

```
struct sctp_paddrinfo {
        sctp_assoc_t    spinfo_assoc_id;
        struct sockaddr_storage spinfo_address;
        int32_t         spinfo_state;
        uint32_t        spinfo_cwnd;
        uint32_t        spinfo_srtt;
        uint32_t        spinfo_rto;
        uint32_t        spinfo_mtu;
};
```

4 Security Considerations

In its current form, this proposal does not require special privileges for processes to choose their congestion control algorithm via socket option (setting the default algorithm via `ipadm` does require `sys_ip_config` privilege). Some algorithms may be more aggressive in the face of congestion than others, or simply behave differently from what network administrators are used to. It should be noted, however, that all algorithms we plan to implement are TCP-friendly, have been present on the Internet for some time, and do not even come close to the havoc that can be wreaked by UDP on a congested network. Undesirable algorithms can be easily disabled by using `ipadm remove-cong`, though admittedly, that's quite a hammer. Comments from reviewers of this document are welcome.

5 Kernel Interfaces

This being our initial foray into the pluggable congestion control, the kernel interfaces proposed herein are intended to be Consolidation Private to reflect their experimental nature, even though the best effort was made to future-proof them.

Note that we limit modules to sending rate adjustment; we do not presently allow other types of transport protocol tweaking that are sometimes bundled in the congestion control algorithms, such as Vegas's fast retransmit modifications.

5.1 Loadable Module Interfaces

Transport protocols can load congestion control algorithms by name:

```
        tcpcong_handle_t tcpcong_load(const char *name, tcpcong_ops_t **);
        void             tcpcong_unload(tcpcong_handle_t);
```

`tcpcong_load()` returns a pointer to the ops structure, as well as an opaque handle, which should be used to unload the module. Reference counting will make sure modules are kept in memory as long as there are consumers. Congestion modules are allowed to load other modules, for instance, some algorithms may use NewReno in certain situations.

Each `tcpcong` module should define `tcpcong_ops_t` and `struct modltcpcong` structures and pass them via modlinkage along with the global `mod_tcpcongops` symbol:

```
tcpcong_ops_t xxops = {
        ...     /* see next section for details */
};

extern struct mod_ops mod_tcpcongops;

static struct modltcpcong xxlinkage = {
        &mod_tcpcongops,
        "xx congestion control module",
        &xxops
};

static struct modlinkage modlinkage = {
        MODREV_1,
        (void *)&xxlinkage,
        NULL
};

int
_init()
{
... mod_install(&modlinkage) ...
}
```

5.2 tcpcong_ops_t

Each module should export this ops structure:

```
typedef struct tcpcong_ops_s {
        int     co_version;                     /* interface version */
        char    *co_name;                       /* module/algorithm name */
        int     co_flags;                       /* module flags */

        /* per peer, private state */
        size_t  (*co_state_size)(int);
        void    (*co_state_init)(tcpcong_args_t *);
        void    (*co_state_fini)(tcpcong_args_t *);

        /* events */
        void    (*co_ack)(tcpcong_args_t *);      /* ack recvd, increase cwnd */
        void    (*co_loss)(tcpcong_args_t *);     /* packet loss */
        void    (*co_enter_fr)(tcpcong_args_t *); /* fast recovery entered */
        void    (*co_exit_fr)(tcpcong_args_t *);  /* fast recovery ended */
        void    (*co_congestion)(tcpcong_args_t *); /* explicit congestion */
        void    (*co_after_idle)(tcpcong_args_t *); /* after no activity */
} tcpcong_ops_t;
```

- `co_version` should be set to `TCPCONG_VERSION`.
- `co_flags` is a bitwise-or of the following flags:
  - `TCPCONG_PROTO_TCP` if module supports TCP;
  - `TCPCONG_PROTO_SCTP` if module supports SCTP;
  - `TCPCONG_SND_TIMESTAMP` if module requires timestamping upon segment transmission. Algorithms such as Vegas use this for their fine-grained RTT estimation.
- `co_state_init()` is called when a connection is associated with the congestion control algorithm. `co_state_fini()` is the opposite and should release any resources allocated by `co_state_init()`.

  In case of SCTP, `co_state_init()` will be called for each peer address of the association. We provide no means for the module to relate between peers - which is consistent with our reading of the SCTP standard. A hypothetical algorithm that tries to do congestion control on the entire SCTP association would violate the TCP-friendliness principle.
- `co_state_size()` tells the networking stack how much memory to allocate for the module's per connection state. This state is intended to be embeddable in the protocol-specific structure, such as `tcp_t` or `sctp_faddr_t`, thus providing better cache locality than separately allocated memory. If the module requires more than the protocol-specific structure can hold, additional memory will be allocated. The integer argument specifies transport protocol, see `TCPCONG_PROTO_` flags above.
- `co_ack()` is called every time a segment of data is acked by the receiver. Depending on the algorithm, the congestion window will be increased. Some algorithms may also collect samples for estimating delay or end-to-end bandwidth.
- `co_loss()` is called when sent data is considered lost. The passed flag provides more detail on the type of loss:
  - `TCPCONG_DUPACK` multiple (typically three) duplicate ACKs were received.
  - `TCPCONG_RTO` retransmit timeout.
- `co_enter_fr()` is called when the connection enters fast recovery phase, right after retransmitting the lost segment as part of fast retransmit. `co_exit_fr()` is called when fast recovery ends and moves back to congestion avoidance.
- `co_congestion()` is called when an explicit congestion signal is received. The passed flag provides more detail:
  - `TCPCONG_ECN` ECN Congestion Experienced.
  - `TCPCONG_ICMP` ICMP Source Quench
- `co_after_idle()` is called after the connection has been idle for a long enough time as to potentially invalidate the latest state of the congestion control algorithm.

5.3 `tcpcong_args_t`

Common to all events is the argument data structure.

```
typedef struct tcpcong_args_s {
        void            *ca_state;              /* algorithm state */
        int             ca_flags;               /* flags */
        uint32_t        *ca_ssthresh;           /* slow start threshold */
        uint32_t        *ca_cwnd;               /* congestion window, bytes */
        uint32_t        *ca_cwnd_cnt;           /* cwnd in cong avoidance */
        const uint32_t  *ca_cwnd_max;           /* max cwnd */
```

```
        const uint32_t  *ca_mss;                /* max segment size */
        const uint32_t  *ca_bytes_acked;        /* bytes acknowledged */
        const uint32_t  *ca_flight_size;        /* bytes unacknowledged */
        const uint32_t  *ca_dupack_cnt;         /* consecutive duplicate acks */
        const clock_t   *ca_srtt;               /* smoothed RTT */
        const clock_t   *ca_rttdev;             /* RTT deviation */
        const hrtime_t  *ca_snd_timestamp;      /* last segment timestamp */
} tcpcong_args_t;
```

- `ca_state` points to the private algorithm state.
- `ca_flags` contains either `TCPCONG_PROTO_TCP` or `TCPCONG_PROTO_SCTP` to specify the caller stack, plus additional flags depending on the entry point.
- `ca_cwnd` and `ca_ssthresh` can be rewritten by the module: recalculating these variables is the congestion control modules' raison d'etre.
- Other parameters are read-only and can be used for internal logic. Not all parameters apply to all functions. The use of pointers, rather than values, obviate the need for the caller to update the structure every time it calls a `tcpcong` function. Instead, the pointers can be initialized once to point to the actual members of the protocol-specific structure such as `tcp_t`.

5.4 Sidenote: Linux Comparison

It is hard to avoid questions about how our design compares to Linux's, even though such comparison is, in our biased opinion, of very limited use. Our design strives to provide substantial interface stability and safety. The additional requirement to support SCTP has also left a considerable imprint on our decisions. Generally, we keep congestion modules on a shorter leash. You will notice that transport protocols call into congestion modules, but not vice versa; in Linux, the calls are two-way, sometimes intertwined and hard to follow. Modules in Linux receive the entire TCP state structure (think `tcp_t` in Solaris) as an argument, effectively making congestion interfaces dependent on completely unrelated changes in TCP.

6 Observability Features

In addition to the socket options described in Section 3, we propose the following observability features.

6.1 pfiles(1)

`pfiles(1)` already shows useful socket information. We propose to extend it to include connection's congestion control algorithm:

```
9: S_IFSOCK mode:0666 dev:337,0 ino:10285 uid:0 gid:0 size:0
   O_RDWR|O_NONBLOCK
     SOCK_STREAM
     SO_SNDBUF(49152),SO_RCVBUF(49640)
     sockname: AF_INET 129.146.104.83  port: 65506
     peername: AF_INET 192.18.34.10  port: 5001
     congestion: newreno
```

6.2 kstats

Solaris already implements several meaningful kstats related to congestion. We propose to complement them with these new kstats:

- `tcp_cong_signals (name: tcpstat, class: net)`
  The number of multiplicative downward congestion window adjustments due to all forms of congestion signals, including Fast Retransmit, ECN, and timeouts.

- `sctp_cong_signals (name: sctpstat, class: net)`
  The SCTP counterpart of `tcp_cong_signals`.

The project team also reviewed *RFC 4898 TCP Extended Statistics MIB*, which provides several congestion related objects, but not yet implemented in Solaris. Due to the size of effort, we feel that it warrants a separate project. We are considering adding some congestion-related counters to the kernel SNMP walker in a way that does not affect SNMP consumers, but is reported via `netstat -v`. This document will be updated for ARC commitment review with the final decision.

6.3 Congestion Window

Unlike receiver's advertized window (`rwnd`), which is a field in the transport protocol header and can be captured and analyzed with the rest of the traffic, congestion window (`cwnd`) is a sender's internal variable. Tools such as Wireshark allow to infer `cwnd` by estimating the amount of in-flight/unacknowledged data - and that's a good enough approximation for most practical purposes.

The exact `cwnd` transitions are of interest to a small group of users: congestion control researchers, developers and testers. We intend to provide DTrace probes and DTrace-based tools as part of the test suite, but feel that they are too specialized to be included in the architecture. The DTrace probes may be added to this document by ARC commitment review with a low stability level.

7 New Congestion Control Algorithms

In addition to the built-in `newreno` algorithm, we propose to implement the following new algorithms:

Algorithm	Solaris Name	Description
HighSpeed [6]	`highspeed`	This is one of the best known and simplest modifications of NewReno for high speed networks.
CUBIC [7]	`cubic`	It is quite popular and is currently the default algorithm in Linux 2.6. CUBIC changes congestion avoidance phase from linear window increase to a cubic function.
Westwood+ [8]	`westwood`	Westwood uses RTT to estimate available bandwidth and performs well in high-loss environments such as wireless.
Vegas [9]	`vegas`	This is a classic delay-based algorithm that attempts to predict congestion without triggering actual loss.

8 Interface table

Interface	Stability	Description
`ipadm add-cong`	Committed	ipadm subcommand
`ipadm remove-cong`	Committed	ipadm subcommand
`ipadm show-cong`	Committed	ipadm subcommand
`cong`	Committed	ipadm property name
`newreno`	Committed	algorithm name
`highspeed`	Committed	algorithm name
`cubic`	Committed	algorithm name
`westwood`	Committed	algorithm name
`vegas`	Committed	algorithm name
`TCP_CONGESTION`	Uncommitted	socket option
`SCTP_CONGESTION`	Uncommitted	socket option
`TCP_INFO`	Uncommitted	socket option
`struct tcp_info`	Uncommitted	`TCP_INFO` argument
`tcpcong_load()`	Consolidation Private	load algorithm
`tcpcong_unload()`	Consolidation Private	unload algorithm
`mod_tcpcongops`	Consolidation Private	tcpcong module ops
`struct modltcpcong`	Consolidation Private	tcpcong module linkage
`tcpcong_ops_t`	Consolidation Private	algorithm ops
`tcpcong_args_t`	Consolidation Private	algorithm arguments
`TCPCONG_*`	Consolidation Private	various flags
`tcp_cong_signals`	Uncommitted	kstat
`sctp_cong_signals`	Uncommitted	kstat

Requested binding: Minor. Should the need arise to backport this to Solaris 10, a new case will be filed, since the `ipadm(1M)` dependency is unlikely to be backported, so at the very least, we would have to specify how to use `ndd(1M)` instead.

References

[1] Van Jacobson and Michael J. Karels. Congestion Avoidance and Control. November 1988.

[2] `http://en.wikipedia.org/wiki/Taxonomy_of_congestion_control`

[3] RFC 5681 TCP Congestion Control

[4] PSARC/2009/306 Brussels II - ipadm and libipadm

[5] Sockets API Extensions for Stream Control Transmission Protocol (SCTP)

[6] RFC 3649 HighSpeed TCP for Large Congestion Windows

[7] BIC and CUBIC web page

[8] TCP Westwood+ web page

[9] Lawrence S. Brakmo. TCP Vegas: End to End Congestion Avoidance on a Global Internet. October 1995.