Pluggable TCP & SCTP Congestion Control Design Specification

Artem Kachitchkine

May 20, 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 is currently 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 Administration

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 by the networking stack. See Section 5 for kernel interfaces.

> A note on naming: "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 of congestion window, slow start, ACKs, etc. It would be incorrect to use "tpcong" for "transport protocol congestion" as other transport protocols might use entirely different type of congestion control; "ipcong" and "netcong" do not quite fit either; and simply "cong" would incorrectly 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.

We propose to manage congestion control via `ipadm(1M)` [4] properties:

- `cong_enabled` is a read-write property containing a comma-separated list of enabled algorithms. Administrator can disable an algorithm by removing it from the list using -= modifier, and reenable using += modifier. The possible value for this property lists algorithms that are installed in the system and is updated automatically when algorithm modules are installed and removed. This property is special in that it is not stored in ipadm's persistent store, but rather generated based on the state of SMF services, see Section 2.2 below for details.
- `cong_default` is a read-write property that specifies which algorithm to be used for connections by default, i.e. when applications do not set it explicitly via socket option (see Section 3). Currently `cong_default` has effect on the entire zone, global or non-global, but in the future we may support individual interfaces and even src,port,dst,port 4-tuples for finer granularity.

Algorithms may optionally provide private properties. As supported by ipadm(1M) design, private properties are not registered in libipadm, not listed by `ipadm show-prop`, but can be shown and set individually with the -p option. These private properties should follow the naming convention `protocol_cong_algorithm_property`, e.g. `tcp_cong_cubic_beta`.

Usage examples:

```
# ipadm show-prop tcp
PROTO PROPERTY              PERM CURRENT      PERSISTENT   DEFAULT      POSSIBLE
tcp   ecn                     rw   passive      --           passive      never,passive,
                                                                         active
...
tcp   cong_default            rw   newreno      --           newreno      -
tcp   cong_enabled            rw   newreno,cubic, --         newreno      newreno,cubic,
                                   highspeed,                             highspeed,vegas
                                   vegas
# ipadm set-prop -p cong_enabled-=vegas tcp
# ipadm set-prop -p cong_default=cubic tcp
# ipadm show-prop -p tcp_cong_cubic_beta tcp
PROTO PROPERTY              PERM CURRENT      PERSISTENT   DEFAULT      POSSIBLE
tcp   tcp_cong_cubic_beta     rw   820          --           820          1-65536
# ipadm set-prop -p tcp_cong_cubic_beta=600 tcp
# ipadm show-prop -p cong_default,tcp_cong_cubic_beta tcp
PROTO PROPERTY              PERM CURRENT      PERSISTENT   DEFAULT      POSSIBLE
tcp   cong_default            rw   cubic        cubic        newreno      -
tcp   tcp_cong_cubic_beta     rw   600          600          820          1-65536
```

2.1 Persistence

ipadm(1M) properties are different from dladm(1M) properties in the way persistence is managed. GLDv3 drivers can be unloaded from the kernel and their persistent properties reapplied whenever drivers are loaded again. This works because the kernel notifies the dlmgmtd daemon via door upcall whenever properties need to be reapplied for a datalink.

By contrast, ipadm(1M)/ipmgmtd do not provide a similar mechanism (it wasn't a viable solution as ip_ddi_init() is called before any daemons are spawned), instead relying on the fact that 'ip' module is never unloaded. Properties are pushed into the kernel early at boot by `ipadm init-prop` that runs from ipmgmtd service start method. Afterwards, individual properties may be changed by users through `ipadm set-prop`, but there is never a need to reapply whole sets of persistent properties because TCP/IP stack remains in the kernel memory at all times.

The implication for congestion module properties is that the only way to insure persistence is to load all enabled algorithms at the same time as 'ip', let `ipadm init-prop` apply the properties and never unload congestion modules unless they get disabled (by removing from the `cong_enabled` list).

2.2 Packaging

Transition from SVR4 to IPS has brought significant changes. Class action scripts are no longer supported, with no way to run any sort of registration command during package install/uninstall. Instead, IPS relies on SMF and recommends that new modules are discovered dynamically. Also, shared files a la /etc/sock2path, to which multiple packages contribute entries, are strongly discouraged. Following these guiding principles, we arrived at the proposed design.

For each `tcpcong` kernel module and each supported transport protocol, a new transient SMF service instance should be delivered by its package:

```
svc:/network/<protocol>/congestion-control:<algorithm>
```

For instance, CUBIC module supporting TCP and SCTP will add two new service instances:

```
svc:/network/tcp/congestion-control:cubic
svc:/network/sctp/congestion-control:cubic
```

`svc:/network/ip-interface-management:default` service will have an `"optional_all"` dependency on these services and will use them to dynamically discover congestion control algorithms and store algorithm properties in the SMF repository. These services are not intended to be managed by users directly, rather ipadm(1M) will make changes behind the scenes.

More specifically, when `ipadm init-prop` is invoked, `cong_enabled` property will be generated based on congestion control services. An algorithm is included in the possible value if its service instance exists, and in the current value if the service instance is enabled. This covers scenarios such as `pkg image-update` into a new BE followed by reboot.

For live upgrade scenarios, we will use the `refresh_fmri` actuator (see pkg(5) man page, section Actuators), so that the service gets refreshed after package install/uninstall. Unlike the start method, however, the refresh method cannot initialize all properties, because that would lose temporary values. Instead, the refresh method will invoke:

```
ipadm init-prop -p cong_enabled
```

which will only regenerate the specified property. The `-p` option for the `init-prop` subcommand is a new addition introduced by this project.

In practice, these IPS scenarios will not actually happen upon initial delivery of this project, because all congestion algorithms will be part of core networking packages that are always present on the system. This also allows us to circumvent an existing SMF bug that prevents multiple manifests from contributing new instances of the same service (6517953). The SMF group is very aware of this issue, as it affects several other projects. This bug will have been fixed by the time we or 3rd parties may want to deliver congestion modules in separate packages.

While the user experience with congestion algorithm private properties will not be any different from any other ipadm(1M) private properties, internal implementation in libipadm will be different in that they will be stored in the SMF repository, rather than `/etc/ipadm/ipadm.conf`. This trivially solves the problem of property cleanup upon algorithm uninstall, since SMF service instances are removed along with the package.

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`. It takes the following variable-length structure:

```
struct sctp_congestion {
        sctp_assoc_t    sc_assoc_id;
        char            sc_name[1];
};
```

If socket is a one-to-many style socket, `sc_assoc_id` refers to the association of interest; otherwise it is ignored.

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 disabled by using `ipadm set-prop -p cong_enabled-=`, though admittedly that's not very flexible.

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. A consumer in this case is any caller of tcpcong_load(), which include netstack instances and individual connections. Congestion modules are also allowed to load other congestion modules, for instance, some algorithms may use NewReno in certain cases. Congestion modules whose reference counter reaches zero will be unloaded - this can happen when an algorithm is disabled and not used by any connections.

`tcpcong` modules should define `tcpcong_ops_t` and `struct modltcpcong` structures and pass them via modlinkage along with the global `mod_tcpcongops` symbol. They should also register their ops structure during _init(9E) and unregister during _fini(9E) using the following functions:

```
        int             tcpcong_mod_register(tcpcong_ops_t *);
        int             tcpcong_mod_unregister(tcpcong_ops_t *);
```

Here's an example:

```
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()
{
        int ret;

        if ((ret = tcpcong_mod_register(&xxops)) != 0) {
                return (ret);
        }
        if ((ret = mod_install(&xxlinkage)) != 0) {
                (void) tcpcong_mod_unregister(&xxops);
        }
        return (ret);
}
```

5.2 tcpcong_ops_t

Modules must provide this ops structure:

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

        /* private property management (optional) */
        mod_prop_info_t *(*co_prop_info_alloc)(uint_t);
        void (*co_prop_info_free)(mod_prop_info_t *, uint_t);

        /* 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, adjust 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 */
        void    (*co_mss_update)(tcpcong_args_t *); /* MSS update */
} tcpcong_ops_t;
```

- `co_version` should be set to `TCPCONG_VERSION`. If the version of the module does not match the version of the kernel, the module will fail to load and a syslog message will be printed.
- `co_flags` is a bitwise-or of the following flags:
  - `TCPCONG_PROTO_TCP` if module supports TCP;
  - `TCPCONG_PROTO_SCTP` if module supports SCTP;
- `co_prop_info_alloc()` must allocate and initialize a NULL-terminated array of private property handling structures. These are the same structures that are being used for public properties, per Brussels II design [4]. In the current incarnation, each netstack instance will allocate itself a copy, but this may be expanded in the future. The uint_t argument specifies protocol using one of `MOD_PROTO_*` constants. `co_prop_info_free()` is used to free the array. These ops can be NULL if no private properties are supported.
- `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. `ca_bytes_acked` argument specifies how many bytes were acked, or zero in case of a duplicate ACK. `co_flags` may contain additional flags to describe the state of protocol:
  - `TCPCONG_REXMIT` retransmit;
  - `TCPCONG_FAST_REXMIT` fast retransmit;
  - `TCPCONG_FAST_RECOVERY` fast recovery;
- `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.
- `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. `ca_idle_time` argument contains the idle time value.
- `co_mss_update()` is called when maximum segment size changes. `ca_mss` argument contains the new value.

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, bytes */
        uint32_t        *ca_cwnd;               /* congestion window, bytes */
        int32_t         *ca_cwnd_cnt;           /* cwnd in cong avoidance */
        const uint32_t  *ca_cwnd_max;           /* max cwnd, bytes */
        const uint32_t  *ca_maxburst;           /* Max.Burst per RFC 4960, bytes */
        const uint32_t  *ca_mss;                /* max segment size, bytes */
        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, msec */
        const clock_t   *ca_rttdev;             /* RTT deviation, msec */
        const clock_t   *ca_idle_time;          /* conn's idle time, msec */
} 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. 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`. Not all parameters apply to all functions (see below), but we think this is a better alternative to passing individual values, as some functions would have to receive 7, 8 or even more arguments.

The following table shows which arguments are valid for which operation. `ca_state` and `ca_flags` are always valid.

	co_state _init/fini	co_ack	co_loss	co_cong estion	co_enter /exit_fr	co_after _idle	co_mss _update
ca_ssthresh		+	+	+	+	+	
ca_cwnd		+	+	+	+	+	
ca_cwnd_cnt		+	+	+	+		
ca_cwnd_max	+	+	+	+	+	+	+
ca_maxburst	+	+	+	+	+	+	+
ca_mss	+	+	+	+	+	+	+
ca_bytes_acked		+					
ca_flight_size		+	+	+	+		
ca_dupack_cnt		+	+				
ca_srtt		+	+	+	+		
ca_rttdev		+	+	+	+		
ca_idle_time						+	

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 control: 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.

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 several static kernel DTrace probes (Project Private stability) for such low-level purposes:

- {`tcp`,`sctp`}`_cwnd_update` cwnd value changed
- {`tcp`,`sctp`}`_ssthresh_update` ssthresh value changed
- {`tcp`,`sctp`}`_cwnd_ssthresh_update` both cwnd and ssthresh values changed
- `load__tcpcong__module` tcpcong kernel module loaded

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.
Vegas [8]	`vegas`	This is a classic delay-based algorithm that attempts to predict congestion without triggering actual loss.

8 Interface table

Interface	Stability	Description
`cong_default`	Committed	ipadm property name
`cong_enabled`	Committed	ipadm property name
`newreno`	Committed	algorithm name
`highspeed`	Committed	algorithm name
`cubic`	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 , PSARC/2010/080 Brussels II addendum

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