TAS Documentation

Antoine Kaufmann

Oct 12, 2020

CONTENTS:

CHAPTER

ONE

USER GUIDE

1.1 Quick Start

1.1.1 Building

Requirements:

- TAS is built on top of Intel DPDK for direct access to the NIC. We have tested this version with dpdk versions 17.11.9, 18.11.5, 19.11.

Assuming that dpdk is installed in `~/dpdk-inst` TAS can be built as follows (for a system installation of dpdk the `RTE_SDK` variable does not need to be passed explicitly):

```
make RTE_SDK=~/dpdk-inst
```

This will build the TAS service (binary `tas/tas`), client libraries (in `lib/`), and a few debugging tools (in `tools/`).

1.1.2 Running

Before running TAS the following steps are necessary:

- Make sure `hugetlbfs` is mounted on `/dev/hugepages` and enough huge pages are allocated for TAS and dpdk.
- Binding the NIC to the dpdk driver, as with any other dpdk application (for Intel NICs use `vfio` because `uio` does not support multiple interrupts).

```
sudo modprobe vfio-pci
sudo mount -t hugetlbfs nodev /dev/hugepages
echo 1024 | sudo tee /sys/devices/system/node/node*/hugepages/hugepages-2048kB/nr_
↪hugepages
sudo ~/dpdk-inst/sbin/dpdk-devbind  -b vfio-pci 0000:08:00.0
```

To run (`--ip-addr` and `--fp-cores-max` are the minimum arguments typically needed to run tas):

```
sudo code/tas/tas --ip-addr=10.0.0.1/24 --fp-cores-max=2
```

Once tas is running, applications that directly link to `libtas` or `libtas_sockets` can be run directly. To run an unmodified application with sockets interposition run as follows (for example):

```
sudo LD_PRELOAD=lib/libtas_interpose.so ../benchmarks/micro_rpc/echoserver_linux 1234␣
↪1 foo 8192 1
```

1.1.3 In Qemu/KVM

For functional testing and development TAS can run in Qemu (with or without acceleration through KVM). We have tested this with the `virtio` dpdk driver. By default, the qemu virtio device only provides a single queue, and thus only allows TAS to run on a single core. To run a virtual machine with support for multiple queue, qemu requires a tap device with multi-queue support enabled.

Here is an example sequence of commands to create a tap device with multi queue support and then start a qemu instance that binds this tap device to a multi-queue virtio device:

```
sudo ip link add tastap0 type tuntap
sudo ip tuntap add mode tap multi_queue name tastap0
sudo ip link set dev tastap0 up
qemu-system-x86_64 \
    -machine q35 -cpu host \
    -drive file=vm1.qcow2,if=virtio \
    -netdev tap,ifname=tastap0,script=no,downscript=no,vhost=on,queues=8,id=nInt\
    -device virtio-net-pci,mac=52:54:00:12:34:56,vectors=18,mq=on,netdev=nInt \
    -serial mon:stdio -m 8192 -smp 16 -display none -enable-kvm
```

Inside the virtual machine, the following sequence of commands first takes the linux network interface down, binds it to the `uio_pci_generic` driver that the dpdk virtio PMD supports, and then reserves huge pages:

```
sudo ifconfig enp0s2 down
sudo modprobe uio
sudo modprobe uio_pci_generic
sudo dpdk-devbind.py -b uio_pci_generic 0000:00:02.0
echo 1024 | sudo tee /sys/devices/system/node/node*/hugepages/hugepages-2048kB/nr_
↪hugepages
```

Virtio does not support all the NIC features that we depend on in physical NICs. In particular virtio does not support transmit checksum offload or the RSS redirection table TAS uses for scaling up and down. The dpdk virtio PMD also does not support multiple MSI-X interrupts. To run TAS given these constraints, the following command line parameters disable the use of these features (note that this implies busy polling and no autoscaling):

```
sudo code/tas/tas --ip-addr=10.0.0.1/24 --fp-cores-max=8 \
    --fp-no-xsumoffload --fp-no-ints --fp-no-autoscale
```

1.1.4 Kernel NIC Interface

TAS supports the DPDK kernel NIC interface (KNI) to pass packets to the Linux kernel network stack. With KNI enabled, TAS becomes an opt-in fastpath where TAS-enabled applications operate through TAS, and other applications can use the Linux network stack as before, sharing the same physical NIC.

To run TAS with KNI the first step is to load the `rte_kni` kernel module. Next, when run with the `--kni-name=` option, TAS will create a KNI dummy network interface with the specified name. After assigning an IP address to this network interface, the Linux network stack can send and receive packets through this interface as long as TAS is running. Here is the complete sequence of commands:

```
sudo modprobe rte_kni
sudo code/tas/tas --ip-addr=10.0.0.1/24 --kni-name=tas0
# in separate terminal
sudo ifconfig tas0 10.0.0.1/24 up
```

1.2 TAS Command-Line Parameters

1.2.1 IP Configuration

- `--ip-addr=ADDR[/PREFIXLEN]`

  Set local IP address. Currently only exactly one IP address is supported.

- `--ip-route=DEST[/PREFIX],NEXTHOP`

  Add an IP route for the destination subnet `DEST/PREFIX` via `NEXTHOP`. Can be specified more than once. For example, a default route could be `--ip-route=0.0.0.0/0,192.168.1.1`.

1.2.2 Fast Path Configuration

- `--fp-cores-max=CORES`

  Maximum number of cores to use for fast-path. (default: 1)

- `--fp-no-ints`

  Disable receive interrupts in the NIC driver, switches over to just polling.

- `--fp-no-xsumoffload`

  Disable transmit checksum offloads, primarily useful to run TAS with NICs that do not support checksum offload, but comes at a slight performance cost.

- `--fp-no-autoscale`

  Disable auto scaling, instead fix the number of cores used by the fast path to the maximum.

- `--fp-no-hugepages`

  Do not use huge pages for the shared memory region between TAS and applications. (DPDK still uses huge pages for it's buffers unless explicitly disabled through `--dpdk-extra`)

- `--dpdk-extra=ARG`

  Pass `ARG` through as a parameter to the dpdk EAL. (see https://doc.dpdk.org/guides/linux_gsg/linux_eal_parameters.html)

1.2.3 TCP Protocol Parameters

- `--tcp-rtt-init=RTT`

  Initial RTT used for congestion control. Is updated with actual measurements when they arrive.

- `--tcp-link-bw=BANDWIDH`

  Link bandwidth in GBPS. TODO: what is this used for? (default: 10).

- `--tcp-rxbuf-len=LEN`

  Connection receive buffer len in bytes (default: 8,192).

- `--tcp-txbuf-len=LEN`

  Connection transmit buffer len in bytes (default: 8,192).

- `--tcp-handshake-timeout=TIMEOUT`

  TCP handshake timeout in microseconds (default 10,000us).

- `--tcp-handshake-retries=RETRIES`

  Maximum retries for timeouts during handshake. (default: 10).

1.2.4 Congestion Control Parameters

- `--cc=ALGORITHM`

  Choose which congestion control algorithm to use. The supported options are:

  - `dctcp-rate`: dctcp algorithm adapted to directly operate on the connection rate.
  - `dctcp-win`: original dctcp algorithm with the window converted to a rate for enforcement.
  - `timely`: latency-based TIMELY control law.
  - `const-rate`: set all connections to a constant rate (effectively disables congestion control, useful for debugging).

- `--cc-control-interval=INT`

  Control interval length as multiples of the connection's RTT. (default: 2)

- `--cc-control-granularity=G`

  Minimal control loop granularity. Control loop is only executed at most once every `G` microseconds. (default: 50)

- `--cc-rexmit-ints=INTERVALS`

  Number of connection cnotrol intervals before TAS triggers a re-transmit. (default: 4).

DCTCP

For the `dctcp-rate` and `dctcp-win` algorithm:

- `--cc-dctcp-weight=WEIGHT`

  EWMA weight for dctcp's ECN rate (alpha, default: 0.0625).

- `--cc-dctcp-mimd=INC_FACT`

  Enable mutliplicative increase by `INC_FACT` (disabled by default, only used for tests).

- `--cc-dctcp-min=RATE`

  Minimum rate to set for flows (kbps, default: 10000).

Timely

Parameters for the `timely` algorithm:

- `--cc-timely-tlow=TIME`

  Tlow threshold in microseconds. (default: 30)

- `--cc-timely-thigh=TIME`

  Thigh threshold in microseconds. (default: 150)

- `--cc-timely-step=STEP`

  Additive increase step size in kbps (default: 10000)

- `--cc-timely-init=RATE`

  Initial connection rate in kbps (default: 10000)

- `--cc-timely-alpha=FRAC`

  EWMA weight for rtt diff. (default: 0.02)

- `--cc-timely-beta=FRAC`

  Multiplicative decrease factor. (default: 0.8)

- `--cc-timely-minrtt=RTT`

  Minimal RTT without queueing in microseconds. (default: 11)

- `--cc-timely-minrate=RTT`

  Minimal connection rate to use in kbps (default: 10000)

Constant Rate

For the `const-rate` "algorithm" the following configuration options apply:

- `--cc-const-rate=RATE`

  Sets the rate to use in kbps.

1.2.5 ARP Protocol Parameters

- `--arp-timeout=TIMEOUT`

  Initial ARP request timetout in microseconds. This doubles with every retry (default: 500).

- `--arp-timeout-max=TIMEOUT`

  Maximal ARP timeout in microseconds. If the retry-timeout grows larger than this, the request fails. (default: 10,000,000 us)

1.2.6 Slowpath Queues

- `--nic-rx-len=LEN`

  Number of entries in TAS slowpath receive queue. (default: 16,384).

- `--nic-tx-len=LEN`

  Number of entries in TAS slowpath transmit queue. (default: 16,384).

- `--app-kin-len=LEN`

  Application slow path receive queue length in bytes. (default: 1,048,576).

- `--app-kout-len=LEN`

  Application slow path transmit queue length in bytes. (default: 1,048,576).

1.2.7 Host Kernel Interface

- `--kni-name=NAME`

  Enables the DPDK kernel network interface, by creating a dummy network interface with the name `NAME`. (default: disabled)

1.2.8 Miscellaneous

- `--quiet`

  Disable non-essential logging.

- `--ready-fd=FD`

  Causes TAS to write to file descriptor `FD` when ready. Can be used by supervisor processes to detect when TAS is ready, e.g. used in full system tests.

1.3 TAS Troubleshooting

CHAPTER

TWO

DEVELOPER GUIDE

2.1 Code Structure

- `tas/`: service implementation
  - `tas/fast`: TAS fast path
  - `tas/slow`: TAS slow path
- `lib/`: client libraries
  - `lib/tas`: lowlevel TAS client library (interface: `lib/tas/include/tas_ll.h`)
  - `lib/sockets`: socket emulation layer
- `tools/`: debugging tools

CHAPTER

THREE

API

The full API documentation extracted by doxygen can be found here.

3.1 TAS Low-Level Application API

int **flextcp_init**(void)
Initializes global flextcp state, must only be called once.

Return 0 on success, < 0 on failure

3.1.1 Contexts

struct flextcp_context
A flextcp context is per-thread state for the stack. (opaque) This includes:

- admin queue pair to kernel
- notification queue pair to flexnic

int **flextcp_context_create**(**struct** *flextcp_context* *ctx)
Create a flextcp context.

int **flextcp_context_poll**(**struct** *flextcp_context* *ctx, int *num*, **struct** *flextcp_event* *events)
Poll events from a flextcp socket.

Warning: doxygenfunction: Cannot find function "flextcp_block" in doxygen xml output for project "TAS" from directory: xml/

3.1.2 Connections

struct flextcp_connection
TCP connection. (opaque)

int **flextcp_connection_open**(**struct** *flextcp_context* *ctx, **struct** *flextcp_connection* *conn, uint32_t *dst_ip*, uint16_t *dst_port*)
Open a connection (asynchronous).

int **flextcp_connection_close**(**struct** *flextcp_context* *ctx, **struct** *flextcp_connection* *conn)
Close a connection (asynchronous).

int **flextcp_connection_rx_done**(**struct** *flextcp_context* *ctx, **struct** *flextcp_connection* *conn, size_t *len*)
Receive processing for `len' bytes done.

ssize_t **flextcp_connection_tx_alloc**(**struct** *flextcp_connection* *conn, size_t *len*, void **buf*)
Allocate transmit buffer for `len' bytes, returns number of bytes allocated.

NOTE: short allocs can occur if buffer wraps around

ssize_t **flextcp_connection_tx_alloc2**(**struct** *flextcp_connection* *conn, size_t *len*, void **buf_1*, size_t **len_1*, void **buf_2*)
Allocate transmit buffer for `len' bytes, returns number of bytes allocated. May be split across two buffers, in case of wrap around.

int **flextcp_connection_tx_send**(**struct** *flextcp_context* *ctx, **struct** *flextcp_connection* *conn, size_t *len*)
Send previously allocated bytes in transmit buffer

int **flextcp_connection_tx_close**(**struct** *flextcp_context* *ctx, **struct** *flextcp_connection* *conn)
Send previously allocated bytes in transmit buffer

int **flextcp_connection_tx_possible**(**struct** *flextcp_context* *ctx, **struct** *flextcp_connection* *conn)
Make sure there is room in the context send queue (not send buffer)

Returns 0 if transmit is possible, -1 otherwise.

int **flextcp_connection_move**(**struct** *flextcp_context* *ctx, **struct** *flextcp_connection* *conn)
Move connection to specfied context

3.1.3 Listeners

struct flextcp_listener
TCP listening "socket". (opaque)

FLEXTCP_LISTEN_REUSEPORT

int **flextcp_listen_open**(**struct** *flextcp_context* *ctx, **struct** *flextcp_listener* *lst, uint16_t *port*, uint32_t *backlog*, uint32_t *flags*)
Open a listening socket (asynchronous).

int **flextcp_listen_accept**(**struct** *flextcp_context* *ctx, **struct** *flextcp_listener* *lst, **struct** *flextcp_connection* *conn)
Accept connections on a listening socket (asynchronous). This can be called more than once to register multiple connection handles.

3.1.4 Events

enum flextcp_event_type
Types of events that can occur in flextcp contexts

Values:

enumerator FLEXTCP_EV_LISTEN_OPEN
flextcp_listen_open() result.

enumerator FLEXTCP_EV_LISTEN_NEWCONN
New connection on listening socket arrived.

enumerator FLEXTCP_EV_LISTEN_ACCEPT
Accept operation completed

enumerator FLEXTCP_EV_CONN_OPEN
flextcp_connection_open() result

enumerator FLEXTCP_EV_CONN_CLOSED
Connection was closed

enumerator FLEXTCP_EV_CONN_RECEIVED
Data arrived on connection

enumerator FLEXTCP_EV_CONN_SENDBUF
More send buffer available

enumerator FLEXTCP_EV_CONN_RXCLOSED
Receive stream closed

enumerator FLEXTCP_EV_CONN_TXCLOSED
transmit stream closed

enumerator FLEXTCP_EV_CONN_MOVED
Connection moved to new context

struct flextcp_event
Events that can occur on flextcp contexts.

Public Members

struct *flextcp_event*::**[anonymous]**::**[anonymous]** **listen_open**
For *FLEXTCP_EV_LISTEN_OPEN*

struct *flextcp_event*::**[anonymous]**::**[anonymous]** **listen_newconn**
For *FLEXTCP_EV_LISTEN_NEWCONN*

struct *flextcp_event*::**[anonymous]**::**[anonymous]** **listen_accept**
For *FLEXTCP_EV_LISTEN_ACCEPT*

struct *flextcp_event*::**[anonymous]**::**[anonymous]** **conn_open**
For *FLEXTCP_EV_CONN_OPEN*

struct *flextcp_event*::**[anonymous]**::**[anonymous]** **conn_received**
For *FLEXTCP_EV_CONN_RECEIVED*

struct *flextcp_event*::**[anonymous]**::**[anonymous]** **conn_sendbuf**
For *FLEXTCP_EV_CONN_SENDBUF*

struct *flextcp_event*::**[anonymous]**::**[anonymous]** **conn_rxclosed**
For *FLEXTCP_EV_CONN_RXCLOSED*

struct *flextcp_event*::**[anonymous]**::**[anonymous]** **conn_txclosed**
For *FLEXTCP_EV_CONN_TXCLOSED*

struct *flextcp_event*::**[anonymous]**::**[anonymous]** **conn_moved**
For *FLEXTCP_EV_CONN_MOVED*

struct *flextcp_event*::**[anonymous]**::**[anonymous]** **conn_closed**
For *FLEXTCP_EV_CONN_CLOSED*

3.2 TAS Sockets API

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

F