**A Guide to enable mTCP and F-Stack with EFD and Comparison of the two with Vanilla Kernel**

# **Brief Description**:

In this document we describe how to enable the mTCP [1] and f-stack [2] applications on a multicore processor and how we can benchmark them. mTCP is a highly scalable user-level TCP stack for multicore systems. F-stack is also a user-level application that along with other features implements a user-level TCP. Both mTCP and F-Stack are implemented on top of Intel’s Data Plane Development Kit (DPDK) [3].

The main goals of both the mTCP and the F-stack projects are to bypass the kernel driver and process the packets from the NIC card directly in the userspace. Both mTCP and F-stack implement user-level TCP. By processing the packets directly in the userspace, the overhead of copying packets between kernel and the userspace as it occurs in traditional Linux user/kernel space paradigm can be avoided. The result is a light weight implementation that helps achieve better performance[[1]](#footnote-1). The rest of documents talks about how to run them in a multicore environment.

# **mTCP:**

For mTCP, it is possible to run the experiments using two possible configurations of the Network Interface Card (NIC). These configurations are: 1) **Receive Side Scaling** (RSS) and; 2) **Ethernet Flow Director** (EFD). The default configuration is RSS. In RSS mode, the packets are randomly sent to different cores for processing. Whereas, in the EDF packets are sent to the dedicated which needs to consume the packets. The default configuration of the NIC card is RSS. One can use ethtool to configure the NIC card [4,5].

The two NIC cards that implement RSS and EFD are: 1). ixgbe and; 2). i40e. The NIC configuration of our interest is EDF. This is because in data center a dedicated application is running on a dedicated core. So depending upon the IP and destination port of application, we want the EFD in the NIC to direct the packets to the destined core.

## **Steps to Run Redis on Top of mTCP with NIC in EFD mode and Understanding mTCP:**

### **Cloning and Modifying the Source:**

1. git clone ssh://mchowdh1@git-amr-1.devtools.intel.com:29418/mtcp-source && cd "mtcp-source" && git config user.name "Muktadir Chowdhury" && git config user.email "[muktadir.chowdhury@intel.com](mailto:muktadir.chowdhury@intel.com)" && scp -P 29418 [mchowdh1@git-amr-1.devtools.intel.com:hooks/commit-msg .git/hooks/](mailto:mchowdh1@git-amr-1.devtools.intel.com:hooks/commit-msg%20.git/hooks/)

Use you username, name, and windows password.

1. After getting the source need to checkout the libc-wrapper branch: git checkout –b <name of branch of your choice> libc-wrapper
2. The code in this branch of mTCP does not support setting of ethernet flow rules for children process by the parent process in ixgbe driver. I ran my experiment on ncc7 and interface that is connected to the switch has ixgbe driver. So had to change the code such that each children process sets its own flow rule. The corresponding code is in: /pnpdata/dpdk-mtcp/mtcp-source/mtcp/src/dpdk\_module.c. In i40e nic, the original code will work without any modification. Changes for Ethernet flow director for ixgbe driver are done in mtcp-source/mtcp/src/dpdk\_module.c. These changes are shown below. Please note that the original branch from asim will work perfectly fine and these changes are not required if you are using i40e driver.

diff --git a/mtcp/src/dpdk\_module.c b/mtcp/src/dpdk\_module.c

index 26e8ee72..027bcf9f 100644

--- a/mtcp/src/dpdk\_module.c

+++ b/mtcp/src/dpdk\_module.c

@@ -813,8 +813,8 @@ attach\_eth\_filter(int nif, int qid, void \*arg\_ip\_port, struct rte\_flow\_error \*er

struct rte\_flow\_action\_queue queue = { .index = qid };

struct rte\_flow\_item\_eth eth\_spec;

struct rte\_flow\_item\_eth eth\_mask;

- struct rte\_flow\_item\_vlan vlan\_spec;

- struct rte\_flow\_item\_vlan vlan\_mask;

+ //struct rte\_flow\_item\_vlan vlan\_spec;

+ //struct rte\_flow\_item\_vlan vlan\_mask;

struct rte\_flow\_item\_ipv4 ip\_spec;

struct rte\_flow\_item\_ipv4 ip\_mask;

struct rte\_flow\_item\_tcp tcp\_spec;

@@ -861,12 +861,12 @@ attach\_eth\_filter(int nif, int qid, void \*arg\_ip\_port, struct rte\_flow\_error \*er

\* since in this case we want to get to all the

\* way to L4, we also set this level to allow all.

\*/

- memset(&vlan\_spec, 0, sizeof(struct rte\_flow\_item\_vlan));

+ /\*memset(&vlan\_spec, 0, sizeof(struct rte\_flow\_item\_vlan));

memset(&vlan\_mask, 0, sizeof(struct rte\_flow\_item\_vlan));

pattern[1].type = RTE\_FLOW\_ITEM\_TYPE\_VLAN;

pattern[1].spec = &vlan\_spec;

pattern[1].mask = &vlan\_mask;

-

+ \*/

/\*

\* setting the third level of the pattern (ip).

\* in this case this is the level we care about

@@ -878,9 +878,9 @@ attach\_eth\_filter(int nif, int qid, void \*arg\_ip\_port, struct rte\_flow\_error \*er

ip\_mask.hdr.dst\_addr = 0xffffffff;

ip\_spec.hdr.src\_addr = 0;

ip\_mask.hdr.src\_addr = 0x0;

- pattern[2].type = RTE\_FLOW\_ITEM\_TYPE\_IPV4;

- pattern[2].spec = &ip\_spec;

- pattern[2].mask = &ip\_mask;

+ pattern[1].type = RTE\_FLOW\_ITEM\_TYPE\_IPV4;

+ pattern[1].spec = &ip\_spec;

+ pattern[1].mask = &ip\_mask;

/\*

\* setting the fourth level of the pattern (tcp).

@@ -892,17 +892,37 @@ attach\_eth\_filter(int nif, int qid, void \*arg\_ip\_port, struct rte\_flow\_error \*er

tcp\_spec.hdr.dst\_port = addr->sin\_port;

tcp\_spec.hdr.src\_port = 0;

tcp\_mask.hdr.dst\_port = 0xffff;

- pattern[3].type = RTE\_FLOW\_ITEM\_TYPE\_TCP;

- pattern[3].spec = &tcp\_spec;

- pattern[3].mask = &tcp\_mask;

+ pattern[2].type = RTE\_FLOW\_ITEM\_TYPE\_TCP;

+ pattern[2].spec = &tcp\_spec;

+ pattern[2].mask = &tcp\_mask;

/\* the final level must be always type end \*/

- pattern[4].type = RTE\_FLOW\_ITEM\_TYPE\_END;

-

- /\* finally set the Ethernet flow director filter \*/

- rc = rte\_flow\_validate(nif, &attr, pattern, action, err);

- if (!rc)

- flow = rte\_flow\_create(nif, &attr, pattern, action, err);

+ pattern[3].type = RTE\_FLOW\_ITEM\_TYPE\_END;

+

+ if (qid == 0) {

+ /\* finally set the Ethernet flow director filter \*/

+ rc = rte\_flow\_validate(nif, &attr, pattern, action, err);

+ if (!rc)

+ flow = rte\_flow\_create(nif, &attr, pattern, action, err);

+

+ qid = 1;

+ int portNo = 9002;

+ for (; qid < CONFIG.num\_cores; qid++, portNo++) {

+ queue.index = qid;

+ action[0].conf = &queue;

+ tcp\_spec.hdr.dst\_port = htons(portNo);

+ pattern[2].spec = &tcp\_spec;

+

+ /\* finally set the Ethernet flow director filter \*/

+ rc = rte\_flow\_validate(nif, &attr, pattern, action, err);

+

+ if (!rc)

+ flow = rte\_flow\_create(nif, &attr, pattern, action, err);

+ }

+ } else {

+ /\*dummy value \*/

+ flow = (struct rte\_flow \*) &dpdk\_module\_func;

+ }

#if 0

rte\_flow\_flush(nif, err);

rte\_eth\_dev\_stop(nif);

1. redis-intel/src/ae-epoll.c at line 48 change 1024 to 10,0000

### **Building the mTCP Source:**

1. Go to mtcp directory: $ cd mtcp
2. Run the following script. If you already have dpdk installed in your system then you can pass the dpdk installation path ($RTE\_SDK) to the script. If you don’t pass the argument the script then will clone dpdk. They have used dpdk-18.02/ as their DPDK driver. FYI, you can pass a different dpdk source directory as command line argument.

$ bash setup\_mtcp\_dpdk\_env.sh [<path to $RTE\_SDK]

The script will go to dpdk directory and run the dpdk set-up script from …./dpdk/usertools/dpdk-setup.sh

After that you have to do the following:

Press [14] to compile x86\_64-native-linuxapp-gcc version (if libnuma is not there: yum install numactl-devel)

Press [17] to install the driver (depmod, )

Press [20] to setup 2048 2MB hugepages

Press [23] to register the Ethernet ports

0000:00:1f.6 'Ethernet Connection (3) I219-LM 15b9' if=enp0s31f6 drv=e1000e unused=igb\_uio \*Active\*

0000:01:00.0 '82599ES 10-Gigabit SFI/SFP+ Network Connection 10fb' if=enp1s0f0 drv=ixgbe unused=igb\_uio

0000:01:00.1 '82599ES 10-Gigabit SFI/SFP+ Network Connection 10fb' if=enp1s0f1 drv=ixgbe unused=igb\_uio

0000:17:00.0 '82571EB/82571GB Gigabit Ethernet Controller D0/D1 (copper applications) 105e' if=enp23s0f0 drv=e1000e unused=igb\_uio \*Active\*

0000:17:00.1 '82571EB/82571GB Gigabit Ethernet Controller D0/D1 (copper applications) 105e' if=enp23s0f1 drv=e1000e unused=igb\_uio

0000:67:00.0 'Ethernet Connection X722 for 10GbE SFP+ 37d0' if=enp103s0f0 drv=i40e unused=igb\_uio

0000:67:00.1 'Ethernet Connection X722 for 10GbE SFP+ 37d0' if=enp103s0f1 drv=i40e unused=igb\_uio

0000:67:00.2 'Ethernet Connection X722 for 10GbE SFP+ 37d0' if=enp103s0f2 drv=i40e unused=igb\_uio

0000:67:00.3 'Ethernet Connection X722 for 10GbE SFP+ 37d0' if=enp103s0f3 drv=i40e unused=igb\_uio

Select 01:00.0 is connected to the switch.

(**Help**: It is not possible to register \*\*active\*\* interfaces. Need to bring down the interface before registering, e.g. ifdown <interface-name>)

Press [22] to make sure your ports are registerd properly under dpdk driver.

Press [34] to quit the tool

After that the interfaces which are now bound to dpdk-compatible driver will be shown and they are renamed to **dpdkX** mTCP.

1. Next bring the dpdk-registered interfaces up, and then set RTE\_SDK and RTE\_TARGET environment variables.

ifconfig dpdk0 192.168.232.50 netmask 255.255.255.0 up

$ sudo ifconfig dpdk0 x.x.x.x netmask 255.255.255.0 up

Note: Here you can assign any address to the interface. Better to use private address of the form 192.168.1.x

$ export RTE\_SDK=`echo $PWD`/dpdk

export RTE\_SDK=/pnpdata/dpdk-mtcp/mtcp/dpdk

$ export RTE\_TARGET=x86\_64-native-linuxapp-gcc

1. Setup mtcp library:

$./configure --with-dpdk-lib=$RTE\_SDK/$RTE\_TARGET

$ make

- By default, mTCP assumes that there are 16 CPUs in your system.

You can set the CPU limit, e.g. on a 32-core system, by using the following command:

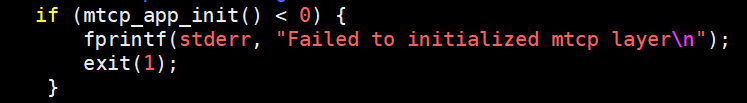
# ./configure --with-dpdk-lib=$RTE\_SDK/$RTE\_TARGET CFLAGS="-DMAX\_CPUS=32"

In the system, I installed mtcp has 36 cpus, so I used the following command:

./configure --with-dpdk-lib=$RTE\_SDK/$RTE\_TARGET CFLAGS="-DMAX\_CPUS=36"

### **Building Redis-Intel and Running the Server**:

1. Change the redis code to init mtcp stack.



1. Get redis-intel from <https://github.intel.com/DSLO/redis-intel>. Make sure to get the redis-4.0.8-dpdk branch.

$ git clone --single-branch -b redis-4.0.8-dpdk <https://github.intel.com/DSLO/redis-intel>

1. Go to redis-intel directory. $ cd redis-intel. We need to change src/Makefile so that compiler flags contain the correct include and library path of dpdk and mtcp.

3.1. $ vi src/Makefile

(**Help**: Search for dpdk or mtcp, wherever they appear you will have to change those lines.)

3.2. Change the include path of dpdk and mtcp. In my case it was the following: -I /pnpdata/dpdk-mtcp/mtcp/dpdk/x86\_64-native-linuxapp-gcc/include/ -I /pnpdata/dpdk-mtcp/mtcp/mtcp/include

3.3. Change library path path of dpdk and mtcp. In my case it was the following: -L /pnpdata/dpdk-mtcp/mtcp/dpdk/x86\_64-native-linuxapp-gcc/lib -L /pnpdata/dpdk-mtcp/mtcp/mtcp/li

1. $ make
2. **Before running** need to do the following:

5.1. Make sure there is a mtcp.conf file in the system, and MTCP\_CONFIG environment variable is set. There is a sample copy of mtcp.conf in …../mtcp/config/ directory.

$ export MTCP\_CONFIG=<path to mtcp.conf>

e.g. export MTCP\_CONFIG=/pnpdata/dpdk-mtcp/redis-intel/mtcp.conf

You need to make the following changes in your **mtcp.conf**:

5.1.1. port = dpdk0

5.1.2. num\_cores = 1

5.1.3. stat\_print = dpdk0

A sample mtcp.conf used for the experiments is here:

############### mtcp configuration file ###############

# The underlying I/O module you want to use.

io = dpdk

#io = netmap #if mtcp is interfaced with netmap driver.

# Number of cores settings

num\_cores = 2

# Number of memory channels per processor socket

num\_mem\_ch = 4

# User port

port = dpdk0

#port = <netmap configure port name>

# Ethernet filter config

eth\_filter\_conf = fdir

# Maximum concurrency per core

max\_concurrency = 16384

# Maximum number of socket buffers per core

max\_num\_buffers = 16384 # 16384

# Receive buffer size of sockets

rcvbuf = 16384 #32768 # 8192 # 32K tcprmem

# Send buffer size of sockets

sndbuf = 16384 #32768 # 8192 # 32K tcpwmmem

# TCP timeout seconds (-1 can disable the timeout check)

tcp\_timeout = 240

# TCP timewait seconds

tcp\_timewait = 0

# Used in multi-process configuration

# 1 - Master mode, 0 - slave mode, undefine = single instance mode

multiprocess = 1

#Prints the runtime stats from DPDK driver, you can comment it out

#stat\_print = dpdk0

5.2. Create a “config” directory inside “redis-intel”, then create arp.conf file, which contains the mac address and IP address of the redis-client. There is a sample copy of arp.conf in …../mtcp/config/ directory.

ARP\_ENTRY 1

192.168.232.18/24 00:25:90:4E:E3:0F

A sample arp.conf used for the experiments looked like this

#################################################################

# This file is to configure static arp tables.

# Rename this file to arp.conf and set the appropriate values.

# Please save this file as config/arp.conf. Put the config/

# directory in the same directory where the binary lies.

#

# (Destination IP address/IP\_prefix) (Destination MAC address)

ARP\_ENTRY 10

#192.168.232.17/32 00:25:90:4E:9C:84

192.168.232.2/32 3C:FD:FE:A5:51:68

192.168.232.3/32 3C:FD:FE:A5:4E:C8

192.168.232.4/32 3C:FD:FE:A5:4E:98

192.168.232.5/32 3C:FD:FE:A5:45:D0

192.168.232.6/32 68:05:CA:3A:AC:E8

192.168.232.7/32 3C:FD:FE:A1:30:A8

192.168.232.8/32 3C:FD:FE:A5:4E:30

192.168.232.9/32 90:E2:BA:1B:E0:C8

192.168.232.10/32 3C:FD:FE:A5:50:A8

192.168.232.11/32 3C:FD:FE:A5:4E:38

192.168.232.12/32 3C:FD:FE:A1:3B:B8

192.168.232.13/32 90:E2:BA:1C:E7:C4

192.168.232.14/32 3C:FD:FE:A1:35:90

192.168.232.15/32 68:05:CA:3A:AB:99

192.168.232.16/32 3C:FD:FE:A5:4F:50

192.168.232.18/32 68:05:CA:3A:A9:A0

192.168.232.19/32 3C:FD:FE:A1:3A:F0

192.168.232.20/32 68:05:CA:3A:AC:F0

192.168.232.21/32 3C:FD:FE:A1:30:C8

192.168.232.22/32 3C:FD:FE:A1:2F:48

192.168.232.23/32 3C:FD:FE:A1:34:E8

192.168.232.24/32 3C:FD:FE:A1:3B:28

192.168.232.25/32 3C:FD:FE:A1:2B:D8

192.168.232.26/32 3C:FD:FE:A1:31:28

192.168.232.27/32 3C:FD:FE:A1:3A:E8

#################################################################

5.3. The above two config files are for mtcp’s use. You also need to edit redis.conf. Look for the keyword “bind” and “port”, and assign ip address (of dpdk0) and port number accordingly.

When making a lot of connections make sure to set the tcp-backlog properly.

Also make sure to uncomment the maxclients 10000 number.

1. $ make

5. **Before running** need to do the following:

5.1. Make sure there is a mtcp.conf file in the system, and MTCP\_CONFIG environment variable is set.

There is a sample copy of mtcp.conf in …../mtcp/config/ directory.

$ export MTCP\_CONFIG=<path to mtcp.conf>

e.g. export MTCP\_CONFIG=/pnpdata/dpdk-mtcp/redis-intel/mtcp.conf

You need to make the following changes in your **mtcp.conf**:

5.1.1. port = dpdk0

5.1.2. num\_cores = 1

5.1.3. stat\_print = dpdk0

A sample mtcp.conf used for the experiments looks like what is pasted below:

############### mtcp configuration file ###############

# The underlying I/O module you want to use.

io = dpdk

#io = netmap #if mtcp is interfaced with netmap driver.

# Number of cores settings

num\_cores = 2

# Number of memory channels per processor socket

num\_mem\_ch = 4

# User port

port = dpdk0

#port = <netmap configure port name>

# Ethernet filter config

eth\_filter\_conf = fdir

# Maximum concurrency per core

max\_concurrency = 16384

# Maximum number of socket buffers per core

max\_num\_buffers = 16384 # 16384

# Receive buffer size of sockets

rcvbuf = 16384 #32768 # 8192 # 32K tcprmem

# Send buffer size of sockets

sndbuf = 16384 #32768 # 8192 # 32K tcpwmmem

# TCP timeout seconds (-1 can disable the timeout check)

tcp\_timeout = 240

# TCP timewait seconds

tcp\_timewait = 0

# Used in multi-process configuration

# 1 - Master mode, 0 - slave mode, undefine = single instance mode

multiprocess = 1

#Prints the runtime stats from DPDK driver, you can comment it out

#stat\_print = dpdk0

5.2. Create a “config” directory inside “redis-intel”, then create arp.conf file, which contains the mac address and IP address of the redis-client. There is a sample copy of arp.conf in …../mtcp/config/ directory.

ARP\_ENTRY 1

192.168.232.18/24 00:25:90:4E:E3:0F

A sample arp.conf used for the experiments looked like this

#################################################################

# This file is to configure static arp tables.

# Rename this file to arp.conf and set the appropriate values.

# Please save this file as config/arp.conf. Put the config/

# directory in the same directory where the binary lies.

#

# (Destination IP address/IP\_prefix) (Destination MAC address)

ARP\_ENTRY 10

#192.168.232.17/32 00:25:90:4E:9C:84

192.168.232.2/32 3C:FD:FE:A5:51:68

192.168.232.3/32 3C:FD:FE:A5:4E:C8

192.168.232.4/32 3C:FD:FE:A5:4E:98

192.168.232.5/32 3C:FD:FE:A5:45:D0

192.168.232.6/32 68:05:CA:3A:AC:E8

192.168.232.7/32 3C:FD:FE:A1:30:A8

192.168.232.8/32 3C:FD:FE:A5:4E:30

192.168.232.9/32 90:E2:BA:1B:E0:C8

192.168.232.10/32 3C:FD:FE:A5:50:A8

192.168.232.11/32 3C:FD:FE:A5:4E:38

192.168.232.12/32 3C:FD:FE:A1:3B:B8

192.168.232.13/32 90:E2:BA:1C:E7:C4

192.168.232.14/32 3C:FD:FE:A1:35:90

192.168.232.15/32 68:05:CA:3A:AB:99

192.168.232.16/32 3C:FD:FE:A5:4F:50

192.168.232.18/32 68:05:CA:3A:A9:A0

192.168.232.19/32 3C:FD:FE:A1:3A:F0

192.168.232.20/32 68:05:CA:3A:AC:F0

192.168.232.21/32 3C:FD:FE:A1:30:C8

192.168.232.22/32 3C:FD:FE:A1:2F:48

192.168.232.23/32 3C:FD:FE:A1:34:E8

192.168.232.24/32 3C:FD:FE:A1:3B:28

192.168.232.25/32 3C:FD:FE:A1:2B:D8

192.168.232.26/32 3C:FD:FE:A1:31:28

192.168.232.27/32 3C:FD:FE:A1:3A:E8

#################################################################

5.3. The above two config files are for mtcp’s use. You also need to edit redis.conf.

Look for the keyword “bind” and “port”, and assign ip address (of dpdk0) and port number respectively.

When making a lot of connections make sure to set the tcp-backlog properly.

Also make sure to uncomment the maxclients 10000 number.

6. Run redis-server. Redis server binary is in (redis-intel/src/redis-server):

$ redis-intel/src/redis-server <path to redis.conf>

Or you can specify the IP and port from the command line:

$ redis-intel/src/redis-server –bind <ip-address of dpdk0> <port-no>

(**Note**: Make sure the port is open , otherwise redis client won’t be able to access it.

Check the iptable rule. In fedora a port (say port no. 9001) can be opened for tcp connection like this: iptables -I INPUT 1 -p tcp --dport 9001 -j ACCEPT)

You should be able to see on the terminal that mtcp is initilaizing and redis-server has started. The statistics (tx, rx etc) for the interface (dpdk0) will be printed on terminal.

(**Note**: If you can’t see the statistics then check the IP address of your dpdk0 interface.

If it is not there you need to assign it again. You can stop the network manager to prevent it from happening again.)

7. From the client side you can test the connectivity by pinging the IP address. If it works fine, then you can run a redis client by specifying the redis server’s IP address and port number. If your client cannot connect to specific IP and port make sure the port is open on the server side as described in step 6.

### **Running Experiments:**

ncc7 (running redis server) <----> switch <-----> (client6 – client 27) 20 client machine running redis client

The IP of **ncc7** where the server runs is 10.242.51.139 and

The IP of **client18** is :10.242.51.173 . Password for client: spec123.

The server and clients are in same subnet 192.168.232.\* configured from the switch. ncc7 is connected to the switch using enp1s0f0 interface.

#### **Launching the Servers:**

For testing your server you can use one of the following tools:

1. memtier\_benchmark -p <port number> -s <IP of server> -d 32 -c 1 -n 1 –t 1
2. iptables -I INPUT 1 -p tcp --dport 9001 -j ACCEPT
3. redis-cli --latency -h <IP of server> -p <port number>

For the experiment, we used memtier benchmark.

To launch multiple servers on the server side, the following script can be used:

#/bin/bash

noOfServers=2

dataSize=64

fileName="vanila\_case\_${noOfServers}\_20\_client\_c\_500\_t\_1"

interface=dpdk0

#cd /pnpdata/dpdk-mtcp/redis-intel/src

cd /pnpdata/dpdk-mtcp/redis-3.2.8-mtcp/src

# run master process, which will create interfaces and map memory for other processes

export MTCP\_CORE\_ID=0

#./redis-server /pnpdata/dpdk-mtcp/redis-dpdk-1.conf &

./redis-server /pnpdata/dpdk-mtcp/redis-3.2.8-mtcp/redis-1.conf &

#./redis-server /pnpdata/dpdk-mtcp/redis-3.2.8-vanila/redis-1.conf &

sleep 10s # wait for sometime to make sure initialization is done

#perf record &

# Now run other processes

coreNo=1

for i in `seq 1 $((noOfServers-1))`;

do

export MTCP\_CORE\_ID=$coreNo

#./redis-server /pnpdata/dpdk-mtcp/redis-dpdk-$((i+1)).conf &

./redis-server /pnpdata/dpdk-mtcp/redis-3.2.8-mtcp/redis-$((i+1)).conf &

#./redis-server /pnpdata/dpdk-mtcp/redis-3.2.8-vanila/redis-$((i+1)).conf &

coreNo=$((coreNo+1))

done

#./netspeed.sh $interface > netspeed\_${fileName}

#touch perf\_${fileName}

#sar 1 > sar\_${fileName} &

#### **Launching the Clients:**

The following script was used to launch the clients.

#!/bin/bash

expType=$1 #"mtcp"

#expType="vanila"

#expType="fstack"

expType="${1}\_vm"

noOfMBs=2 # should be equal to noOfServers

connectionNo=500

threadNo=1

dataSize=32 # in bytes

reconnect\_interval=100

test\_time=60

serverIp=192.168.232.50

interface='eth2'

#fileName="${expType}\_case\_${noOfMBs}\_server\_c\_${connectionNo}\_reconnect\_${reconnect\_interval}"

fileName="${expType}\_case\_${noOfMBs}\_server\_c\_${connectionNo}"

startClient=18 #26 #23 #18 #6 #2

endClient=27 #27

noOfClient=$((endClient-startClient+1))

sleep\_time=$((noOfMBs\*noOfClient))

#./netspeed.sh $interface > netspeed\_${fileName}

for ((client=$startClient; client<=$endClient; client++))

do

coreNo=1

portNo=9001

for i in `seq 1 $noOfMBs`;

do

if [ $client -eq 17 ] || [ $client -eq 16 ] # skip client17, client16

then

continue

fi

if [ $client -eq 26 ]

then # client 26 is not reachable, so need to use client 31

ssh root@192.168.232.31 "sleep $sleep\_time && taskset -c ${coreNo} memtier\_benchmark -p ${portNo} -s ${serverIp} -d $dataSize -c ${connectionNo} --ratio=1:4 --key-pattern G:G --key-maximum 1000001 --test-time=${test\_time} --thread=${threadNo} --pipeline=1 > /pnpdata/redis\_dpdk/${fileName}\_clientId\_${client}\_${i}" &

else

ssh root@192.168.232.$client "sleep $sleep\_time && taskset -c ${coreNo} memtier\_benchmark -p ${portNo} -s ${serverIp} -d $dataSize -c ${connectionNo} --ratio=1:4 --key-pattern G:G --key-maximum 1000001 --test-time=${test\_time} --thread=${threadNo} --pipeline=1 > /pnpdata/redis\_dpdk/${fileName}\_clientId\_${client}\_${i}" &

fi

sleep\_time=$((sleep\_time-1))

coreNo=$((coreNo+1))

portNo=$((portNo+1))

sleep 1

done

done

wait

rm -rf ${fileName}

mkdir ${fileName}

for ((client=$startClient; client<=$endClient; client++))

do

if [ $client -eq 17 ] || [ $client -eq 16 ]

then

continue

fi

if [ $client -eq 26 ]

then # client 26 is not reachable, so need to use client 31

scp root@192.168.232.31:/pnpdata/redis\_dpdk/${fileName}\_clientId\_${client}\_\* /pnpdata/redis\_dpdk/${fileName}/

ssh root@192.168.232.31 "rm -f /pnpdata/redis\_dpdk/${fileName}\_clientId\_${client}\_\*"

else

scp root@192.168.232.$client:/pnpdata/redis\_dpdk/${fileName}\_clientId\_${client}\_\* /pnpdata/redis\_dpdk/${fileName}/

ssh root@192.168.232.$client "rm -f /pnpdata/redis\_dpdk/${fileName}\_clientId\_${client}\_\*"

fi

#./netspeed.sh $interface > netspeed\_${fileName}

done

#./parse\_log ${fileName}

rm -f ${fileName}\_clientId\_18\_\*

### **Experimental Setup and Results:**

The experimental setup consist of 1 server machine and 10 client machines. The machine used as a server was ncc7 that has Intel ® Xeon ® Gold 6130M CPU running at 2.10 Ghz, a RAM of 376 G and a NIC of 10 Gbps. Each client had a specification of Intel ® Xeon ® CPU E5-2690 @ 2.90 Ghz, a RAM of 60 GB. The block diagram of experimental setup is shown in Figure 1 below.



Figure 1: Experimental Setup with 1 Server and 10 Clients.

#### **Comparison of Vanilla Kernel and mTCP with Ethernet Flow Director – Core Scaling**

For the setup shown in figure 1, a comparison of the mTCP with vanilla kernel for core scaling was performed. The results are shown in the figure 2 below. Clearly, mTCP with Ethernet flow director enabled performs better than the vanilla kernel. The results were obtained using the following configuration:

* # of Redis Server = 1, 2, 4, 8, 16
* Benchmark Program/ Client Machine = 1, 2, 4, 8, 16
* # of Connections/ Server = 5K
* Data size = 32 B
* Redis version 3.2.8

Figure 2: Core Scaling Comparison of mTCP with Vanilla Kernel

#### **Comparison of Vanilla Kernel and mTCP using Ethernet Flow Director – Connection Scaling**

For the setup shown in Figure 1, we performed experiments where we increased the number of connections for both the vanilla and mTCP with Ethernet Flow Director enabled. The results are shown in Figure 3 below. The results show that the mTCP scales better as the number of connections increase. The details of the experimental configuration for these experiments are:

* # of Redis Server = 16
* Benchmark Program/ Client Machine = 16
* Run Benchmark from 2, 10, and 20 machines for 1K, 5K, and 10K connection.
* Data size = 32 B
* Redis version 3.2.8

Figure 3: Connection scaling of mTCP and Vanilla Kernel Using EFD

### **Redis and mTCP Interaction:**

The interaction between Redis and mTCP is shown in the figure below. The mtcp\_app\_init() function initializes mTCP stack. During the mTCP stack initialization, an mTCP thread is created. This thread reads new packets from the nic queue in bursts. Whenever there is a new packet, mTCP thread will add that event to listening socket and mTCP event queue. The mTCP epoll has access to the mTCP event queue and to the socket. Simultaneously, Redis thread is monitoring the socket using epoll\_wait() for new event. During the mTCP initialization function, the pointers of socket functions will point to mTCP socket functions. Consequently, when Redis thread calls epoll\_wait(), mtcp\_epoll\_wait() is invoked. And the mtcp\_epoll has access to the event queue where mTCP thread is adding events.



### **mTCP Description and Code Flow Diagram:**

Here is a brief description of the mTCP flow using DPDK.

1. mtcp\_init () is used to create, load and apply network as well as device configurations from a file or user passed options. During compile time an I/O module can be selected. The I/O module can either be dpdk or any other module supported by the mtcp stack.
2. There are two threads. The first thread is application thread (called RunServerThread) that runs in application context. The second thread (called the RunMainLoop) defined in core.c under mtcp src directory is responsible for reading and writing packets to the NIC.
3. In a normal server application such as epserver.c. The application thread (RunServerThread) is waiting on epoll events and is asleep.
4. The mtcp RunMainLoop thread runs continuously and checks if there are any packets to send or receive.
5. In order to receive packets the mtcp RunMainLoop thread calls mtcp->iom->recv packet module.
6. Depending upon the module selected the mtcp->iom->recv module translates to a particular underlying module. In our case it is dpdk\_recv\_pkts which internally calls rte\_eth\_rx\_burst().
7. Depending upon the number of received packets, the RunMainLoop reads data packet by packet by calling the wrapper get->iom->rptr(). This internally calls dpdk\_get\_rptr and returns the pointer of the received packet.
8. The pointer of received packet is then passed to the ProcessPacket API which depending upon the type of packet indicated in the ETH\_P\_IP field of the packet calls the respective API. In our case it is ProcessIPv4Packet.
9. The ProcessIPv4Packet looks at the iphdr and depending on the iphdr it calls respective function. In our case it is ProcessTCPPacket.
10. The ProcessTCPPacket calls Handle\_TCP\_ST\_ESTABLISHED which then calls ProcessTCPPayload.
11. Finally the RBput copies the data from the mbuf to TCPRecieveRingBuf that is used by application directly.

### 

# **F-stack:**

Just like mTCP, it is possible to configure the NIC card in RSS and EFD mode. Our interest is to configure the NIC card in EFD mode and run redis application on top of F-stack.

## **Steps to Run Redis on Top of F-stack with NIC in EFD mode and Understanding F-stack:**

### **Cloning and Modifying the Source:**

1. systemctl disable firewalld
2. git clone <https://github.com/F-Stack/f-stack.git>
3. commit 031be553ab2ea4926fee726130cedf3c17eb9604
4. cd f-stack
5. To run fstack in multi-process single-nic mode (where multiple processes are using the same ethernet interface), we need to use ethernet flow director. The implementation is in ncc7 (10.242.51.139. path: /pnpdata/f-stack). It is implemented in a separate git branch named ethernet flow director. To see the changes I have made: git diff master..ethernet\_flow\_director lib/ff\_dpdk\_if.c. Checkout the branch and do the following:
6. f-stack/lib/ff\_dpdk\_if.c:699, change the IP address according to your server’s IP address. [Using this tool](https://www.silisoftware.com/tools/ipconverter.php), convert IP to hexdecimal format, then put it in the line.

Note that, when we set flow director, the port number is strating from 9001. And its hardocded. If you have other port number need to change it in the f-stack/lib/ff\_dpdk\_if.c:953

diff --git a/lib/ff\_dpdk\_if.c b/lib/ff\_dpdk\_if.c

index 21e13074..c95d504d 100644

--- a/lib/ff\_dpdk\_if.c

+++ b/lib/ff\_dpdk\_if.c

@@ -50,6 +50,7 @@

#include <rte\_ip.h>

#include <rte\_tcp.h>

#include <rte\_udp.h>

+#include <rte\_flow.h>

#include "ff\_dpdk\_if.h"

#include "ff\_dpdk\_pcap.h"

@@ -136,6 +137,46 @@ static struct rte\_eth\_conf default\_port\_conf = {

.txmode = {

.mq\_mode = ETH\_MQ\_TX\_NONE,

},

+ .fdir\_conf = {

+ .mode = RTE\_FDIR\_MODE\_PERFECT,

+ .pballoc = RTE\_FDIR\_PBALLOC\_64K,

+ .status = RTE\_FDIR\_REPORT\_STATUS,

+ .drop\_queue = 127,

+ },

+};

+

+

+static struct rte\_eth\_conf flow\_port\_conf = {

+ .rxmode = {

+ .max\_rx\_pkt\_len = ETHER\_MAX\_LEN,

+ /\*.offloads = (DEV\_RX\_OFFLOAD\_CRC\_STRIP |

+ DEV\_RX\_OFFLOAD\_CHECKSUM

+ DEV\_RX\_OFFLOAD\_TCP\_LRO

+ ),\*/

+ .split\_hdr\_size = 0,

+ .header\_split = 0, /\*\*< Header Split disabled \*/

+ .hw\_ip\_checksum = 1, /\*\*< IP checksum offload enabled \*/

+ .hw\_vlan\_filter = 0, /\*\*< VLAN filtering disabled \*/

+ .jumbo\_frame = 0, /\*\*< Jumbo Frame Support disabled \*/

+ .hw\_strip\_crc = 1, /\*\*< CRC stripped by hardware \*/

+ .enable\_lro = 1, /\*\*< Enable LRO \*/

+ },

+ .rx\_adv\_conf = {

+ .rss\_conf = {

+ .rss\_key = NULL,

+ .rss\_hf = ETH\_RSS\_TCP | ETH\_RSS\_UDP |

+ ETH\_RSS\_IP | ETH\_RSS\_L2\_PAYLOAD

+ },

+ },

+ .txmode = {

+ .mq\_mode = ETH\_MQ\_TX\_NONE,

+ },

+ .fdir\_conf = {

+ .mode = RTE\_FDIR\_MODE\_PERFECT,

+ .pballoc = RTE\_FDIR\_PBALLOC\_64K,

+ .status = RTE\_FDIR\_REPORT\_STATUS,

+ .drop\_queue = 127,

+ },

};

struct mbuf\_table {

@@ -584,6 +625,132 @@ set\_rss\_table(uint16\_t port\_id, uint16\_t reta\_size, uint16\_t nb\_queues)

}

}

+struct rte\_flow \*

+create\_flow(uint16\_t nif, uint16\_t qid, uint16\_t portNo,

+ struct rte\_flow\_error \*err)

+{

+ struct rte\_flow\_attr attr;

+ struct rte\_flow\_item pattern[5];

+ struct rte\_flow\_action action[5];

+ struct rte\_flow\_action\_queue queue = { .index = qid };

+ struct rte\_flow\_item\_eth eth\_spec;

+ struct rte\_flow\_item\_eth eth\_mask;

+ //struct rte\_flow\_item\_vlan vlan\_spec;

+ //struct rte\_flow\_item\_vlan vlan\_mask;

+ struct rte\_flow\_item\_ipv4 ip\_spec;

+ struct rte\_flow\_item\_ipv4 ip\_mask;

+ struct rte\_flow\_item\_tcp tcp\_spec;

+ struct rte\_flow\_item\_tcp tcp\_mask;

+ //struct sockaddr\_in \*addr;

+ struct rte\_flow \*flow;

+ int rc;

+

+ //addr = (struct sockaddr\_in \*)arg\_ip\_port;

+ flow = NULL;

+ memset(pattern, 0, sizeof(pattern));

+ memset(action, 0, sizeof(action));

+

+ /\*

+ \* set the rule attribute.

+ \* in this case only ingress packets will be checked.

+ \*/

+ memset(&attr, 0, sizeof(struct rte\_flow\_attr));

+ attr.ingress = 1;

+

+ /\*

+ \* create the action sequence.

+ \* one action only, move packet to queue

+ \*/

+ action[0].type = RTE\_FLOW\_ACTION\_TYPE\_QUEUE;

+ action[0].conf = &queue;

+ action[1].type = RTE\_FLOW\_ACTION\_TYPE\_END;

+

+ /\*

+ \* set the first level of the pattern (eth).

+ \* since in this case, we get to L4, we set this level

+ \* to allow all.

+ \*/

+ memset(&eth\_spec, 0, sizeof(struct rte\_flow\_item\_eth));

+ memset(&eth\_mask, 0, sizeof(struct rte\_flow\_item\_eth));

+ eth\_spec.type = 0;

+ eth\_mask.type = 0;

+ pattern[0].type = RTE\_FLOW\_ITEM\_TYPE\_ETH;

+ pattern[0].spec = &eth\_spec;

+ pattern[0].mask = &eth\_mask;

+

+ /\*

+ \* setting the second level of the pattern (vlan).

+ \* since in this case we want to get to all the

+ \* way to L4, we also set this level to allow all.

+ \*/

+ /\*memset(&vlan\_spec, 0, sizeof(struct rte\_flow\_item\_vlan));

+ memset(&vlan\_mask, 0, sizeof(struct rte\_flow\_item\_vlan));

+ pattern[1].type = RTE\_FLOW\_ITEM\_TYPE\_VLAN;

+ pattern[1].spec = &vlan\_spec;

+ pattern[1].mask = &vlan\_mask;

+ \*/

+ /\*

+ \* setting the third level of the pattern (ip).

+ \* in this case this is the level we care about

+ \* so we set it according to the parameters.

+ \*/

+ memset(&ip\_spec, 0, sizeof(struct rte\_flow\_item\_ipv4));

+ memset(&ip\_mask, 0, sizeof(struct rte\_flow\_item\_ipv4));

+ ip\_spec.hdr.dst\_addr = htonl(0xC0A8E832);

+ ip\_mask.hdr.dst\_addr = 0xffffffff;

+ ip\_spec.hdr.src\_addr = 0;

+ ip\_mask.hdr.src\_addr = 0x0;

+ pattern[1].type = RTE\_FLOW\_ITEM\_TYPE\_IPV4;

+ pattern[1].spec = &ip\_spec;

+ pattern[1].mask = &ip\_mask;

+

+ /\*

+ \* setting the fourth level of the pattern (tcp).

+ \* in this example this is the level we \*really\* care about

+ \* so we set it according to the parameters.

+ \*/

+ memset(&tcp\_spec, 0, sizeof(struct rte\_flow\_item\_tcp));

+ memset(&tcp\_mask, 0, sizeof(struct rte\_flow\_item\_tcp));

+ tcp\_spec.hdr.dst\_port = htons(portNo);

+ tcp\_spec.hdr.src\_port = 0;

+ tcp\_mask.hdr.dst\_port = 0xffff;

+ pattern[2].type = RTE\_FLOW\_ITEM\_TYPE\_TCP;

+ pattern[2].spec = &tcp\_spec;

+ pattern[2].mask = &tcp\_mask;

+

+ /\* the final level must be always type end \*/

+ pattern[3].type = RTE\_FLOW\_ITEM\_TYPE\_END;

+

+ rc = rte\_flow\_validate(nif, &attr, pattern, action, err);

+ if (!rc)

+ flow = rte\_flow\_create(nif, &attr, pattern, action, err);

+

+ /\*

+ if (qid == 0) {

+ rc = rte\_flow\_validate(nif, &attr, pattern, action, err);

+ if (!rc)

+ flow = rte\_flow\_create(nif, &attr, pattern, action, err);

+

+ qid = 1;

+ int portNo = 9002;

+ for (; qid < CONFIG.num\_cores; qid++, portNo++) {

+ queue.index = qid;

+ action[0].conf = &queue;

+ tcp\_spec.hdr.dst\_port = htons(portNo);

+ pattern[2].spec = &tcp\_spec;

+

+ rc = rte\_flow\_validate(nif, &attr, pattern, action, err);

+

+ if (!rc)

+ flow = rte\_flow\_create(nif, &attr, pattern, action, err);

+ }

+ } else {

+ flow = (struct rte\_flow \*) &dpdk\_module\_func;

+ }

+ \*/

+ return flow;

+}

+

static int

init\_port\_start(void)

{

@@ -686,6 +853,12 @@ init\_port\_start(void)

pconf->hw\_features.rx\_csum = 1;

}

+ /\* config for flow director\*/

+ port\_conf.fdir\_conf.mode = RTE\_FDIR\_MODE\_PERFECT;

+ port\_conf.fdir\_conf.pballoc = RTE\_FDIR\_PBALLOC\_64K;

+ port\_conf.fdir\_conf.status = RTE\_FDIR\_REPORT\_STATUS;

+ port\_conf.fdir\_conf.drop\_queue = 127;

+

if ((dev\_info.tx\_offload\_capa & DEV\_TX\_OFFLOAD\_IPV4\_CKSUM)) {

printf("TX ip checksum offload supported\n");

pconf->hw\_features.tx\_csum\_ip = 1;

@@ -718,7 +891,7 @@ init\_port\_start(void)

if (rte\_eal\_process\_type() != RTE\_PROC\_PRIMARY) {

continue;

}

-

+

int ret = rte\_eth\_dev\_configure(port\_id, nb\_queues, nb\_queues, &port\_conf);

if (ret != 0) {

return ret;

@@ -776,6 +949,18 @@ init\_port\_start(void)

printf("set port %u to promiscuous mode error\n", port\_id);

}

}

+

+ // attach eth filter

+ uint16\_t portNo=9001;

+ for (q = 0; q < 0 /\*nb\_queues\*/; q++, portNo++) {

+ struct rte\_flow\_error error;

+ struct rte\_flow\* flow = create\_flow(port\_id, q, portNo, &error);

+

+ if (!flow) {

+ printf("Failed to attach Ethernet flow firector: (%d) %s\n", error.type, error.message);

+ rte\_exit(EXIT\_FAILURE, "port[%d] set flow director failed\n", port\_id);

+ }

+ }

1. cd dpdk/usertools
2. ./dpdk-setup.sh # build and compile dpdk

Press [14] to compile x86\_64-native-linuxapp-gcc version (if libnuma is not there: yum install numactl-devel)

Press [17] to install the driver (depmod, )

Press [20] to setup 2048 2MB hugepages

Press [23] to register the Ethernet ports

0000:00:1f.6 'Ethernet Connection (3) I219-LM 15b9' if=enp0s31f6 drv=e1000e unused=igb\_uio \*Active\*

0000:01:00.0 '82599ES 10-Gigabit SFI/SFP+ Network Connection 10fb' if=enp1s0f0 drv=ixgbe unused=igb\_uio

0000:01:00.1 '82599ES 10-Gigabit SFI/SFP+ Network Connection 10fb' if=enp1s0f1 drv=ixgbe unused=igb\_uio

0000:17:00.0 '82571EB/82571GB Gigabit Ethernet Controller D0/D1 (copper applications) 105e' if=enp23s0f0 drv=e1000e unused=igb\_uio \*Active\*

0000:17:00.1 '82571EB/82571GB Gigabit Ethernet Controller D0/D1 (copper applications) 105e' if=enp23s0f1 drv=e1000e unused=igb\_uio

0000:67:00.0 'Ethernet Connection X722 for 10GbE SFP+ 37d0' if=enp103s0f0 drv=i40e unused=igb\_uio

0000:67:00.1 'Ethernet Connection X722 for 10GbE SFP+ 37d0' if=enp103s0f1 drv=i40e unused=igb\_uio

0000:67:00.2 'Ethernet Connection X722 for 10GbE SFP+ 37d0' if=enp103s0f2 drv=i40e unused=igb\_uio

0000:67:00.3 'Ethernet Connection X722 for 10GbE SFP+ 37d0' if=enp103s0f3 drv=i40e unused=igb\_uio

1. Select 01:00.0 is connected to the switch.

(**Note**: It is not possible to register \*\*active\*\* interfaces. Need to bring down the interface before registering, e.g. ifdown <interface-name>)

Press [22] to make sure your ports are registerd properly under dpdk driver.

Press [34] to quit the tool

1. After that the interfaces which are now bound to dpdk-compatible driver will be shown and they are renamed to **dpdkX** mTCP.

### **Build F-Stack Application:**

1. export FF\_PATH=/pnpdata/f-stack
2. export FF\_DPDK=/pnpdata/f-stack/dpdk/x86\_64-native-linuxapp-gcc
3. cd ../../lib
4. make # build f-stack library

### **Build Redis With F-stack:**

cd ../app/redis-3.2.8/src

make # build redis app

### **Start an Application with F-Stack:**

Since F-Stack is multi-process architecture, every F-Stack application process should call ff\_init(argc, argv) to initialize the environments. For example, if lcore\_mask=f in config.ini, you can start your app like this:

${bin} --conf config.ini --proc-type=primary --proc-id=0

${bin} --conf config.ini --proc-type=secondary --proc-id=1

${bin} --conf config.ini --proc-type=secondary --proc-id=2

${bin} --conf config.ini --proc-type=secondary --proc-id=3

For example it can be run as below:

./app/redis-3.2.8/src/redis-server --conf config.ini --proc-type=primary --proc-id=0 redis-1.conf

./app/redis-3.2.8/src/redis-server --conf config.ini --proc-type=secondary --proc-id=1 redis-2.conf

### **Running Experiments:**

#### **Launching the Servers:**

Before launching the server make sure to have a config.ini file in f-stack/config.ini. The one we used looks like this.

f-stack/config.ini

[dpdk]

# Hexadecimal bitmask of cores to run on.

lcore\_mask=0x3

# Number of memory channels.

channel=4

# Specify base virtual address to map.

#base\_virtaddr=0x7f0000000000

# Promiscuous mode of nic, defualt: enabled.

promiscuous=1

numa\_on=1

# TCP segment offload, default: disabled.

tso=0

# HW vlan strip, default: enabled.

vlan\_strip=1

# sleep when no pkts incomming

# unit: microseconds

idle\_sleep=0

# enabled port list

#

# EBNF grammar:

#

# exp ::= num\_list {"," num\_list}

# num\_list ::= <num> | <range>

# range ::= <num>"-"<num>

# num ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

#

# examples

# 0-3 ports 0, 1,2,3 are enabled

# 1-3,4,7 ports 1,2,3,4,7 are enabled

port\_list=0

# Number of vdev.

nb\_vdev=0

# Port config section

# Correspond to dpdk.port\_list's index: port0, port1...

[port0]

addr=192.168.232.150

netmask=255.255.225.0

broadcast=192.168.1.255

gateway=192.168.1.1

# lcore list used to handle this port

# the format is same as port\_list

# lcore\_list= 0

# Packet capture path, this will hurt performance

#pcap=./a.pcap

# Vdev config section

# orrespond to dpdk.nb\_vdev's index: vdev0, vdev1...

# iface : Shouldn't set always.

# path : The vuser device path in container. Required.

# queues : The max queues of vuser. Optional, default 1, greater or equal to the number of processes.

# queue\_size : Queue size.Optional, default 256.

# mac : The mac address of vuser. Optional, default random, if vhost use phy NIC, it should be set to the phy NIC's mac.

# cq : Optional, if queues = 1, default 0; if queues > 1 default 1.

#[vdev0]

##iface=/usr/local/var/run/openvswitch/vhost-user0

#path=/var/run/openvswitch/vhost-user0

#queues=1

#queue\_size=256

#mac=00:00:00:00:00:01

#cq=0

# Kni config: if enabled and method=reject,

# all packets that do not belong to the following tcp\_port and udp\_port

# will transmit to kernel; if method=accept, all packets that belong to

# the following tcp\_port and udp\_port will transmit to kernel.

#[kni]

#enable=1

#method=reject

# The format is same as port\_list

#tcp\_port=80,443

#udp\_port=53

# FreeBSD network performance tuning configurations.

# Most native FreeBSD configurations are supported.

[freebsd.boot]

hz=100

# Block out a range of descriptors to avoid overlap

# with the kernel's descriptor space.

# You can increase this value according to your app.

fd\_reserve=1024

kern.ipc.maxsockets=262144

net.inet.tcp.syncache.hashsize=4096

net.inet.tcp.syncache.bucketlimit=100

net.inet.tcp.tcbhashsize=65536

kern.ncallout=262144

[freebsd.sysctl]

kern.ipc.somaxconn=32768 # comment out this line when you have more than this many connections

kern.ipc.maxsockbuf=16777216

net.link.ether.inet.maxhold=5

net.inet.tcp.fast\_finwait2\_recycle=1

net.inet.tcp.sendspace=16384

net.inet.tcp.recvspace=8192

net.inet.tcp.nolocaltimewait=1

net.inet.tcp.cc.algorithm=cubic

net.inet.tcp.sendbuf\_max=16777216

net.inet.tcp.recvbuf\_max=16777216

net.inet.tcp.sendbuf\_auto=1

net.inet.tcp.recvbuf\_auto=1

net.inet.tcp.sendbuf\_inc=16384

net.inet.tcp.recvbuf\_inc=524288

net.inet.tcp.sack.enable=1

net.inet.tcp.blackhole=1

net.inet.tcp.msl=2000

net.inet.tcp.delayed\_ack=0

net.inet.udp.blackhole=1

net.inet.ip.redirect=0

After you have created this file, the following script was used to run the F-stack. You can paste this in a [filename].sh and run it.

#!/bin/bash

noOfServers=1

clientThread=1

noOfConnection=300

portNo=9001

#dataSize=8192

#./run\_emon.sh emon\_$fileName

/root/workspace\_rohan/f-stack/app/redis-3.2.8/src/redis-server --conf /root/workspace\_rohan/f-stack/config.ini --proc-type=primary --proc-id=0 /root/workspace\_rohan/f-stack/app/redis-3.2.8/redis-1.conf &

sleep 5

/root/workspace\_rohan/f-stack/app/redis-3.2.8/src/redis-server --conf /root/workspace\_rohan/f-stack/config.ini --proc-type=secondary --proc-id=1 /root/workspace\_rohan/f-stack/app/redis-3.2.8/redis-2.conf &

#perf record &

proc\_id=1

for i in `seq 2 $noOfServers`;

do

echo "Starting server $i"

#./app/redis-3.2.8/src/redis-server --conf config.ini --proc-type=secondary --proc-id=$proc\_id redis-$i.conf &

portNo=$((portNo+1))

proc\_id=$((proc\_id+1))

done

#./netspeed.sh $interface > netspeed\_${fileName}

#sar 1 > sar\_${fileName}

#emon -stop

#mv perf.data perf\_${fileName}

#### **Launching the Clients:**

The following script was used to launch the clients. It is same as what we used for mTCP.

#!/bin/bash

expType=$1 #"mtcp"

#expType="vanila"

#expType="fstack"

expType="${1}\_vm"

noOfMBs=2 # should be equal to noOfServers

connectionNo=500

threadNo=1

dataSize=32 # in bytes

reconnect\_interval=100

test\_time=60

serverIp=192.168.232.50

interface='eth2'

#fileName="${expType}\_case\_${noOfMBs}\_server\_c\_${connectionNo}\_reconnect\_${reconnect\_interval}"

fileName="${expType}\_case\_${noOfMBs}\_server\_c\_${connectionNo}"

startClient=18 #26 #23 #18 #6 #2

endClient=27 #27

noOfClient=$((endClient-startClient+1))

sleep\_time=$((noOfMBs\*noOfClient))

#./netspeed.sh $interface > netspeed\_${fileName}

for ((client=$startClient; client<=$endClient; client++))

do

coreNo=1

portNo=9001

for i in `seq 1 $noOfMBs`;

do

if [ $client -eq 17 ] || [ $client -eq 16 ] # skip client17, client16

then

continue

fi

if [ $client -eq 26 ]

then # client 26 is not reachable, so need to use client 31

ssh root@192.168.232.31 "sleep $sleep\_time && taskset -c ${coreNo} memtier\_benchmark -p ${portNo} -s ${serverIp} -d $dataSize -c ${connectionNo} --ratio=1:4 --key-pattern G:G --key-maximum 1000001 --test-time=${test\_time} --thread=${threadNo} --pipeline=1 > /pnpdata/redis\_dpdk/${fileName}\_clientId\_${client}\_${i}" &

else

ssh root@192.168.232.$client "sleep $sleep\_time && taskset -c ${coreNo} memtier\_benchmark -p ${portNo} -s ${serverIp} -d $dataSize -c ${connectionNo} --ratio=1:4 --key-pattern G:G --key-maximum 1000001 --test-time=${test\_time} --thread=${threadNo} --pipeline=1 > /pnpdata/redis\_dpdk/${fileName}\_clientId\_${client}\_${i}" &

fi

sleep\_time=$((sleep\_time-1))

coreNo=$((coreNo+1))

portNo=$((portNo+1))

sleep 1

done

done

wait

rm -rf ${fileName}

mkdir ${fileName}

for ((client=$startClient; client<=$endClient; client++))

do

if [ $client -eq 17 ] || [ $client -eq 16 ]

then

continue

fi

if [ $client -eq 26 ]

then # client 26 is not reachable, so need to use client 31

scp root@192.168.232.31:/pnpdata/redis\_dpdk/${fileName}\_clientId\_${client}\_\* /pnpdata/redis\_dpdk/${fileName}/

ssh root@192.168.232.31 "rm -f /pnpdata/redis\_dpdk/${fileName}\_clientId\_${client}\_\*"

else

scp root@192.168.232.$client:/pnpdata/redis\_dpdk/${fileName}\_clientId\_${client}\_\* /pnpdata/redis\_dpdk/${fileName}/

ssh root@192.168.232.$client "rm -f /pnpdata/redis\_dpdk/${fileName}\_clientId\_${client}\_\*"

fi

#./netspeed.sh $interface > netspeed\_${fileName}

done

#./parse\_log ${fileName}

rm -f ${fileName}\_clientId\_18\_\*

### **Experimental Setup and Results:**

We had the exact same setup for F-stack as described in the mTCP experimental setup and results section.

#### **Comparison of Vanilla Kernel and f-stack with Ethernet Flow Director – Core Scaling**

For the setup shown in figure 1, a comparison of the f-stack with vanilla kernel for core scaling was performed. The results are shown in the figure 4 below. Clearly, F-stack with Ethernet flow director enabled performs better than the vanilla kernel. The results were obtained using the following configuration:

* # of Redis Server = 1, 2, 4, 8, 16
* Benchmark Program/ Client Machine = 1, 2, 4, 8, 16
* # of Connections/ Server = 5K
* Data size = 32 B
* Redis version 3.2.8

Figure 4: Core Scaling Comparison of F-stack With Vanilla Kernel When EDF is enabled

#### **Comparison of Vanilla Kernel and f-Stack using Ethernet Flow Director – Connection Scaling**

For the setup shown in Figure 1, we performed experiments where we increased the number of connections for both the vanilla and f-stack with Ethernet Flow Director enabled. The results are shown in Figure 5 below. The results show that the f-stack scales better as the number of connections increase. The details of the experimental configuration for these experiments are:

* # of Redis Server = 16
* Benchmark Program/ Client Machine = 16
* Run Benchmark from 2, 10, and 20 machines for 1K, 5K, and 10K connection.
* Data size = 32 B
* Redis version 3.2.8

Figure 5: Connection scaling of mTCP and Vanilla Kernel Using EFD

### **Port an application to F-stack:**

1. Change Makefile to include fstack library and header.
2. Initialize f-stack with this line: ff\_init(argc, argv)
3. ff\_mod\_init(): its in anet\_ff.c: it contains the wrapper of standard socket functions (bind, connect, listen etc.)
4. it is different for unix and redis. Why?
5. If the application require command line argument, need to adjust command line arguments.
6. ff\_run(loop, server.el)
7. Define a loop() function in the **int loop(void\* arg)** format.

### **F-Stack Code Flow Diagram:**



# Comparison of Vanilla Kernel, mTCP and F-stack using EFD

For setup in figure 1, the results here show the comparison of vanilla kernel with mTCP and F-stack.

## Core Scaling:

For the setup shown in figure 1, a comparison of the f-stack with vanilla kernel for core scaling was performed. The results are shown in the figure 6 below. Clearly, mTCP with Ethernet flow director enabled performs better than the other two approaches. The results were obtained using the following configuration:

* # of Redis Server = 1, 2, 4, 8, 16
* Benchmark Program/ Client Machine = 1, 2, 4, 8, 16
* # of Connections/ Server = 5K
* Data size = 32 B
* Redis version 3.2.8

Figure 6: Comparison of Vanilla Kernel, mTCP and F-Stack for Core Scaling

## Connection Scaling:

For the setup shown in Figure 1, we performed experiments where we increased the number of connections for both the vanilla and f-stack with Ethernet Flow Director enabled. The results are shown in Figure 7 below. The results show that the mTCP scales better then the other two approaches as the number of connections increase. The details of the experimental configuration for these experiments are:

* # of Redis Server = 16
* Benchmark Program/ Client Machine = 16
* Run Benchmark from 2, 10, and 20 machines for 1K, 5K, and 10K connection.
* Data size = 32 B
* Redis version 3.2.8

Figure 7: Comparison of Vanilla Kernel, mTCP and F-Stack With Connection Scaling

# References:

1. mTCP (<https://github.com/mtcp-stack/mtcp>)
2. F-Stack (<http://www.f-stack.org/>)
3. DPDK (<https://www.dpdk.org/>)
4. RSS and EFD (<https://software.intel.com/en-us/articles/setting-up-intel-ethernet-flow-director>)
5. Ethernet Flow Director White Paper (<https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf>)

1. More details about mTCP, F-stack and DPDK can be read by clicking the reference number and following the link in the reference. [↑](#footnote-ref-1)