Deep-dive into Data Plane Development Kit (DPDK)

CS 744

Slides by: Diptyaroop Maji, Nilesh Unhale

Course Instructor: Prof. Umesh Bellur

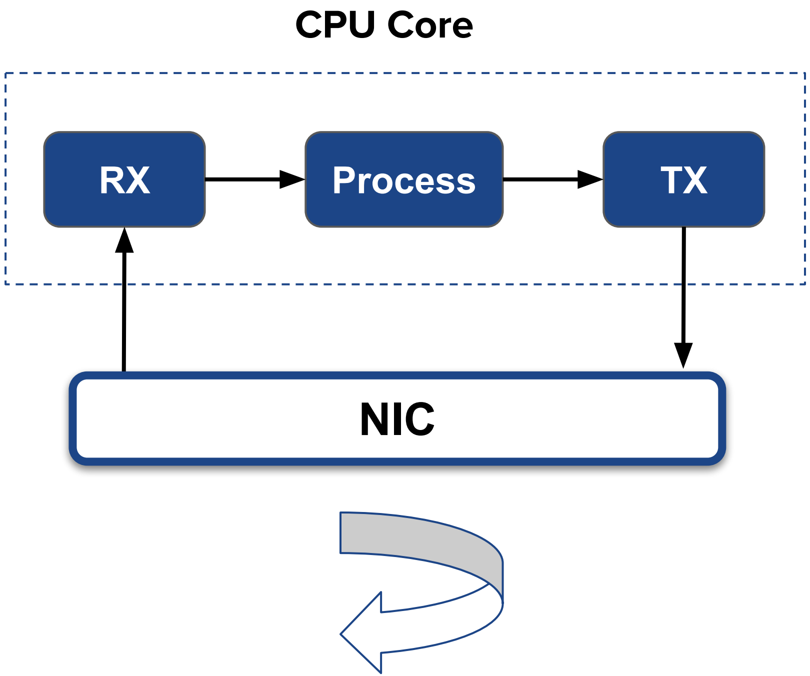
Department of Computer Science & Engineering
Indian Institute of Technology Bombay

Prologue

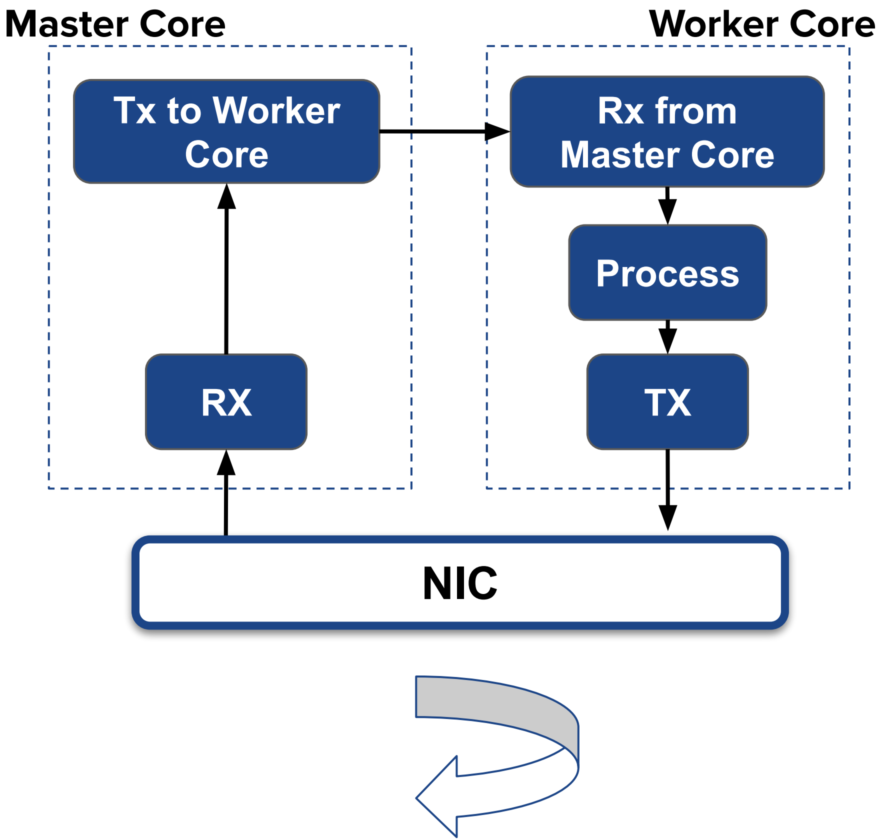
- **Prereq:** Lecture on kernel bypass
- Kernel bypass lecture handles theoretical aspects
- This lecture: more on implementation/hands-on approach
- Implementation designs: Run-to-completion vs Pipeline
- Installing & Compiling DPDK
- Running a simple DPDK application

Implementation designs

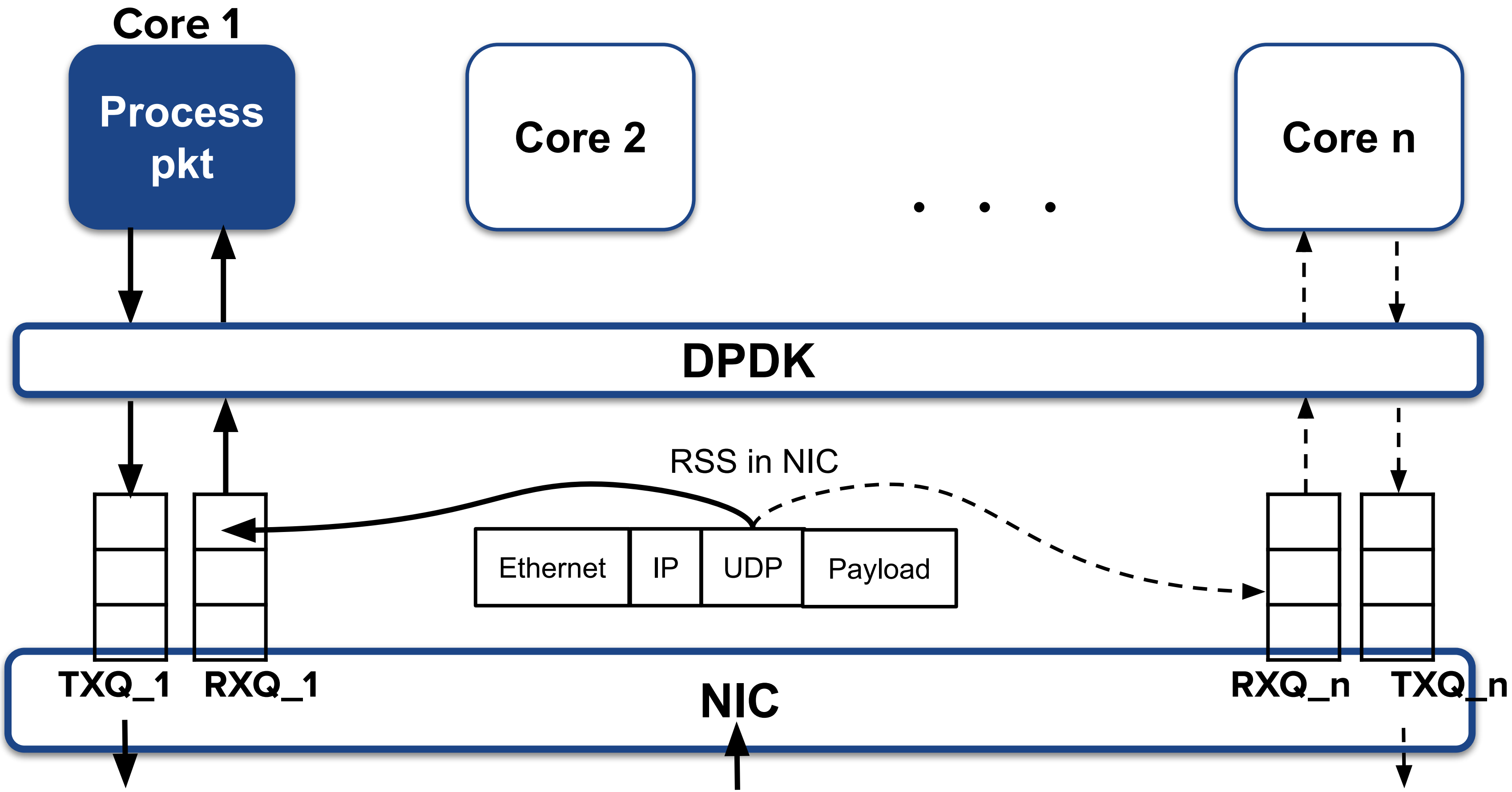
Run-to-Completion



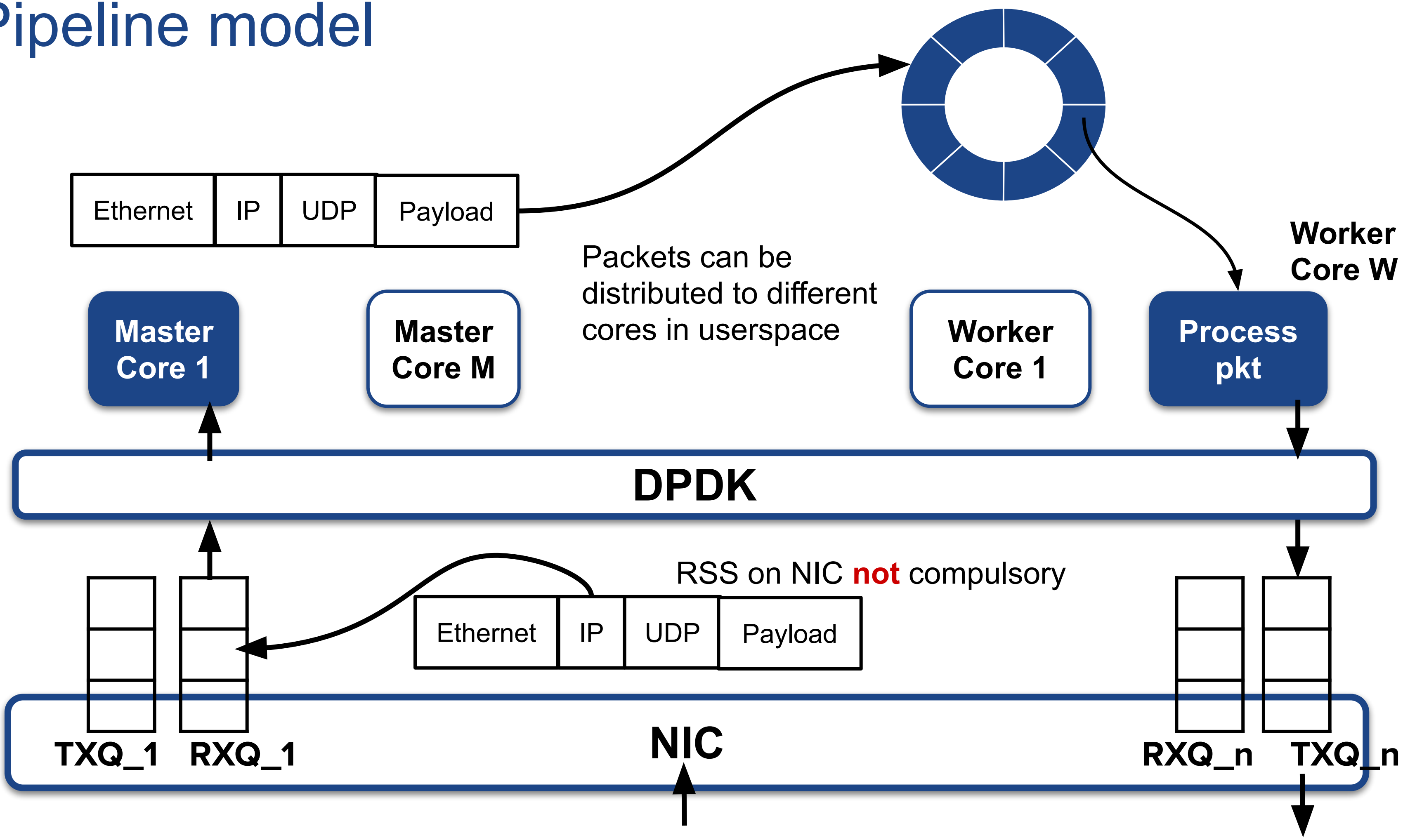
Pipeline



Run-to-completion model



Pipeline model

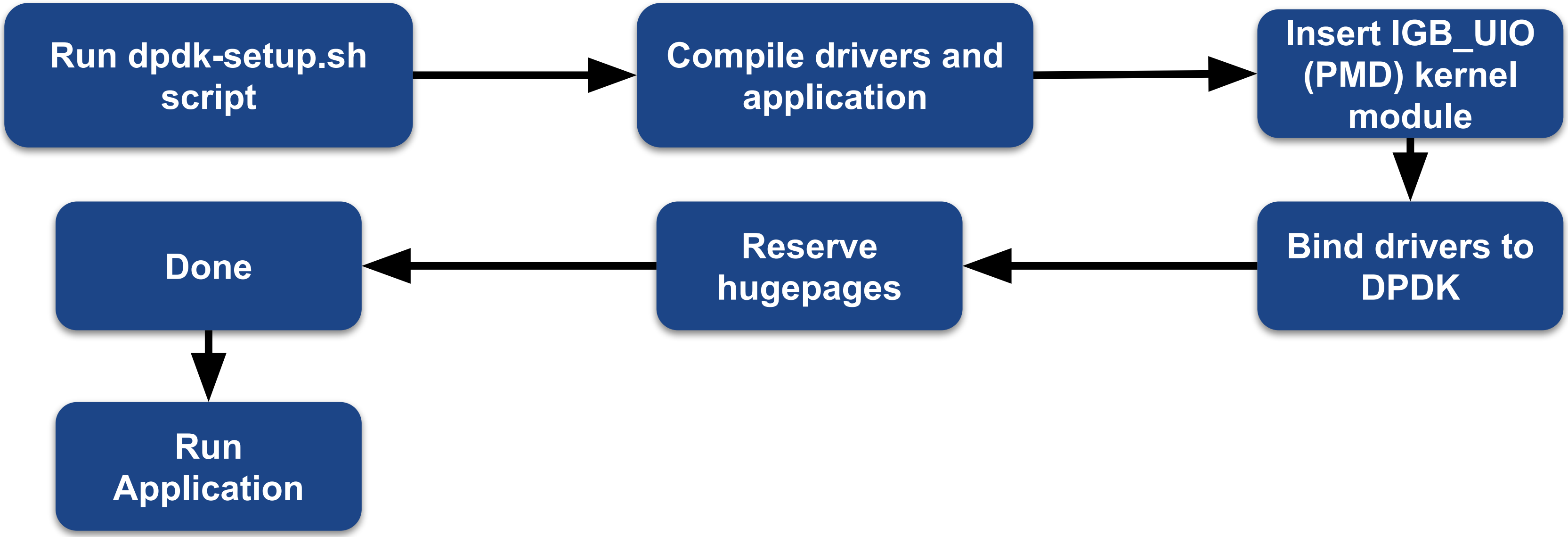


Run-to-Completion(RTC) vs Pipeline

	Run-to-completion (RTC)	Pipeline
Pros	❖ Easily scalable. ❖ Less userspace processing overhead (no inter-core communication)	❖ Easily scalable. ❖ No h/w support needed to distribute packets to other cores.
Cons	❖ H/W support needed (Eg. To distribute packets to different Rx queues, good RSS function support required in NIC)	❖ More userspace processing overhead (inter-core communication via rings)

Installing DPDK

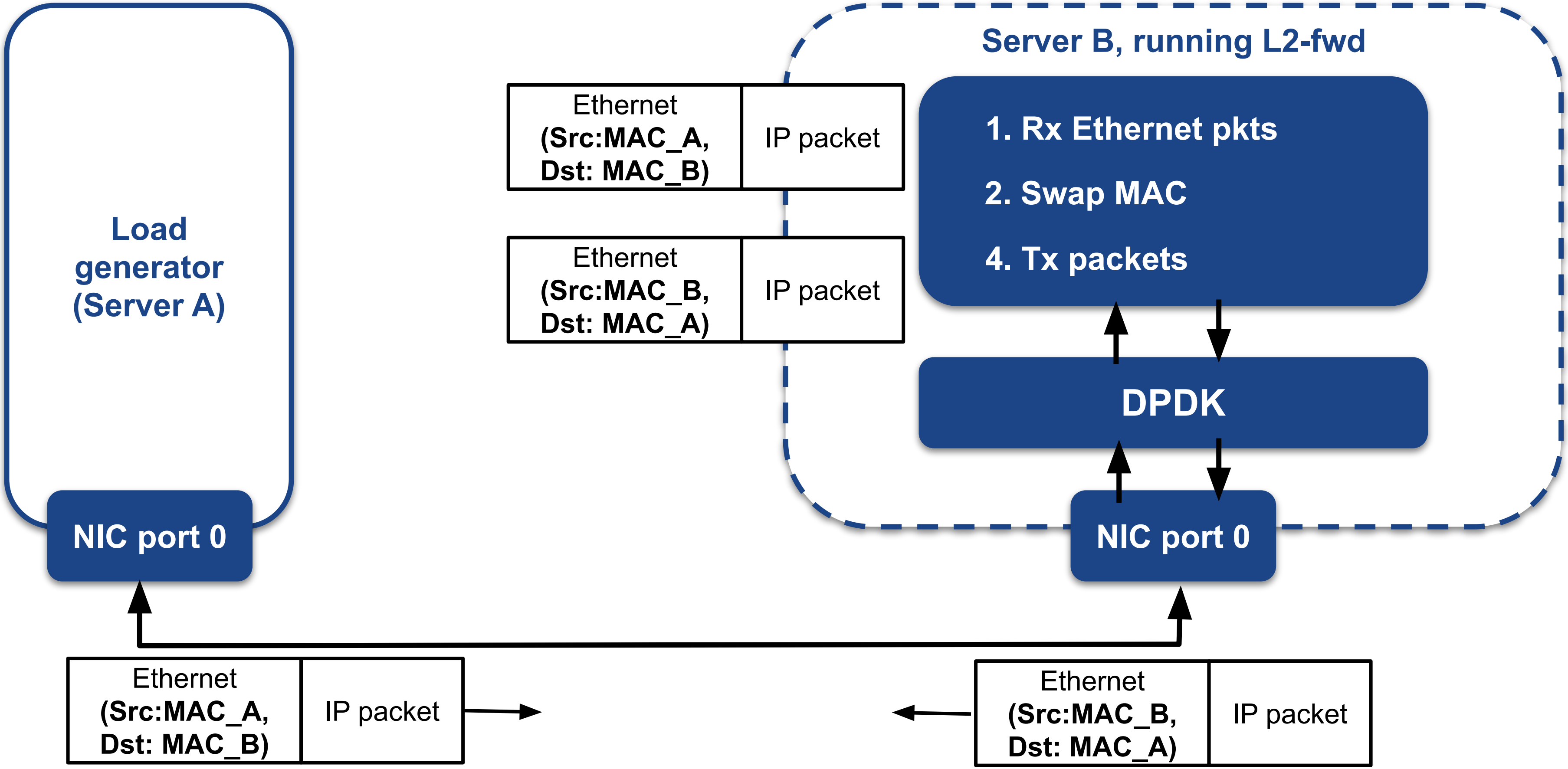
- Check if h/w supports DPDK (DPDK compatibility)
- Clone DPDK from GIT (DPDK GIT repository)
- Run the following steps (DPDK version <= 19.11):
  - (running script can be found in <path_to_dpdk>/usertools)



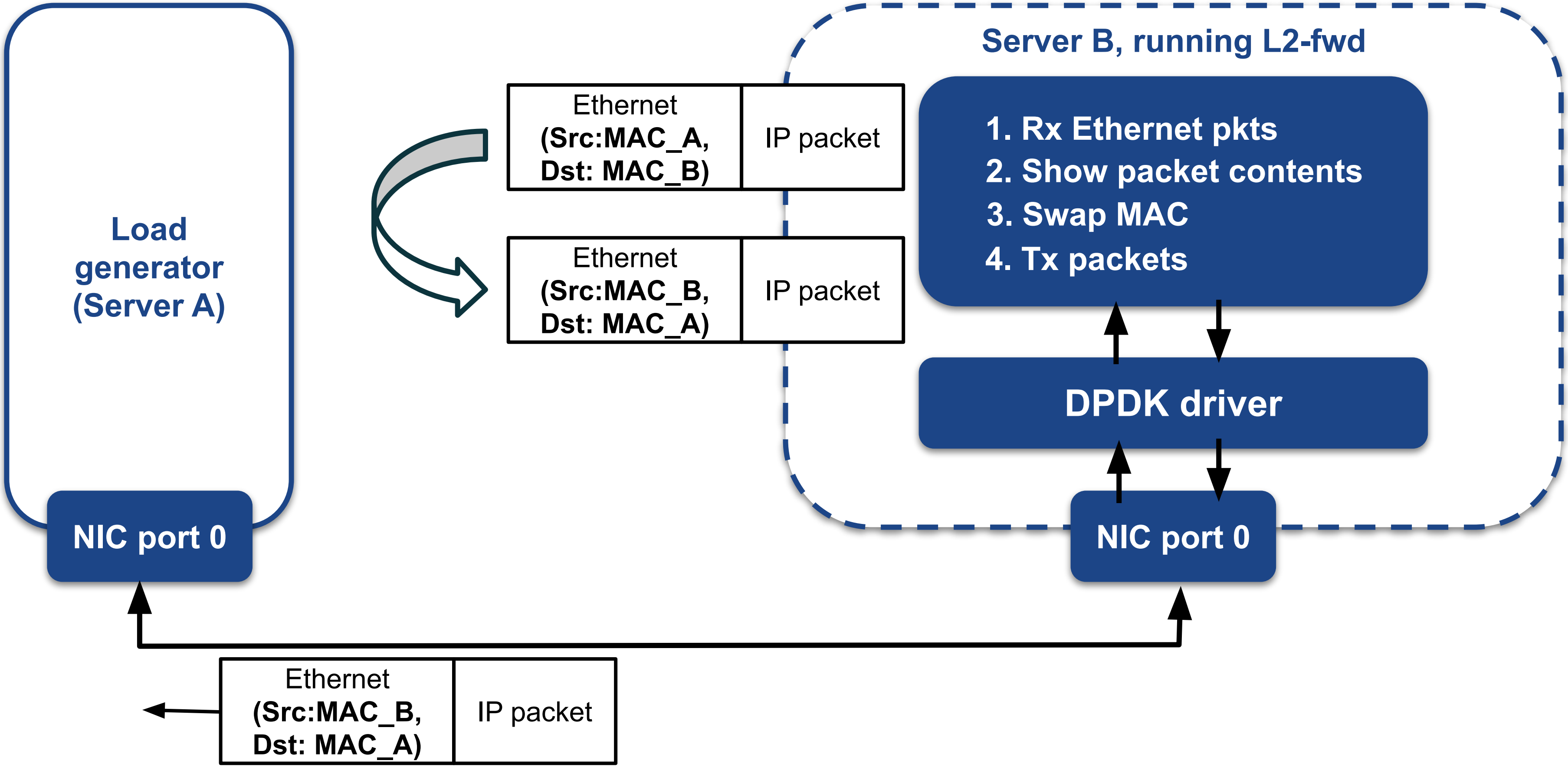
Let’s look at an example

<u>L2 forwarding sample application</u>
(click <u>here</u> to know more)

L2 forwarding sample application



L2 forwarding sample application



L2-fwd: Explanation

- **Command line arguments (EAL parameters):**
  - **-w <port bus address>: Whitelisting the port to be used**
    - **-w 81:00.1**
  - **-c <mask>: Core mask in hex, specifying no. of cores to be used**
    - **-c 1**
  - **Check out this link to know more.**

- **Application specific arguments:**
  - **-p <portMask>: Port mask in hex, specifying no. of ports to be used**
    - **-p 0x1**

Ex : sudo ./build/l2fwd -w 81:00.1 -c 1 -- -p 0x1

L2-fwd: Explanation

- **Initializing the Environment Abstraction Layer (EAL):**
  - **This should be the first API to be called. It initializes the EAL layer & makes way for the application to use the DPDK framework.**

```
ret = rte_eal_init(argc, argv);
```

  - ***argc:* No. of command line arguments (both EAL & application specific parameters)**
  - ***argv:* Array storing the command line arguments**
  - ***ret:* On success, *ret* stores the no. of parsed arguments, which is equal to the no. of EAL parameters passed. The application can now use *argc* & *argv* to parse application specific parameters like any other normal C/C++ program using *int main(int argc, char *argv[]).***

L2-fwd: Explanation

- **Setting up ports/queues:**
  - **Firstly, the NIC port must be configured.**
    `rte_eth_dev_configure();`
    - **No. of Rx/Tx queues, whether NIC should perform RSS etc.**
  - **Setting up Tx queue(s).**
    `rte_eth_tx_queue_setup();`
    - **No. of Tx ring descriptors to allot, what Tx offloading feature to be enabled etc.**
  - **Setting up Rx queue(s).**
    `rte_eth_rx_queue_setup();`
    - **No. of Rx ring descriptors to allot, what Rx offloading feature to be enabled etc.**
  - **Finally, starting the respective port.**
    `rte_eth_dev_start();`
    - **The respective port can now start receiving & transmitting packets.**
- **You can check out in detail about these APIs** [here]

L2-fwd: Explanation

- **Receiving packets:**

```
nb_rx = rte_eth_rx_burst(portid, qNo, pkts_burst, MAX_PKT_BURST);
```

- ***portid:*** **ID of NIC port which will receive the incoming packets.**
- ***qNo:*** **This is actually the Rx queue no. of that particular port where packets will be received and queued till the DPDK driver sends them to the userspace.**
- ***pkts_burst:*** `struct rte_mbuf *pkts_burst[MAX_PKT_BURST];`

  Array of structure to store the incoming packets.
- ***MAX_PKT_BURST:*** **Max. no. of packets permitted to be received at a time.**
- ***nb_rx:*** **Actual no. of packets received (< = *MAX_PKT_BURST*)**

L2-fwd: Explanation

- **Transmitting packets:**

```
nb_tx = rte_eth_tx_burst(portid, qNo, pkts_burst, MAX_PKT_BURST);
```

- ***portid:*** **ID of NIC port which will transmit the outgoing packets.**
- ***qNo:*** **Tx queue no. of that particular port where packets will be queued and till NIC sends them out.**
- ***pkts_burst:*** `struct rte_mbuf *pkts_burst[MAX_PKT_BURST];`

  Array of structure to store the outgoing packets.
- ***MAX_PKT_BURST:*** **Max. no. of packets permitted to be transmitted at a time.**
- ***nb_tx:*** **Actual no. of packets transmitted (< = *MAX_PKT_BURST*)**

DPDK and modern NICs

- DPDK provides many APIs to take advantage of feature on NICs
- Some packet processing can be offloaded to h/w (NIC)
- Some features include
  - Checksum verification/computation offloading (L3 and L4)
  - Distributing packets to separate rx queues based on particular header (RSS)
  - Parsing L3/L4 headers and taking some simple actions (drop/forward etc.)
  - To check (or set) what offloading feature NIC has
    - In DPDK Application -- API
    - Without DPDK: ethtool -k <iface> (link1, link2)

Easy startup guide

- Quick Start: https://core.dpdk.org/doc/quick-start/

Compiling DPDK Applications

- Exporting Environment variables
  - **RTE_SDK** - Points to the DPDK installation directory.
    - Eg: *export RTE_SDK=$HOME/DPDK*
  - **RTE_TARGET** - Points to the DPDK target environment directory.
    - Eg: *export RTE_TARGET=x86_64-native-linux-gcc*
- Go to desired DPDK Application provided
  - Eg. *cd path/to/dpdk/examples/helloworld*
- Compile (generally using *make)*
- Application executable will be built in *<path_to_app>/build/app/*

Common packet generators

- Testpmd
- nping
- iperf
- others

Further Reading

- DPDK in layman's terms: link1 link2 link3
- DPDK overview : https://doc.dpdk.org/guides/prog_guide/overview.html
- <path_to_dpdk>/apps and <path_to_dpdk>/examples
  - L3fwd (user guide)
  - helloworld
  - **Testpmd (user guide)**
- Short Notes on DPDK installation and app: Click Here
- DPDK APIs -- (Comprehensive list of APIs)
  - Ethernet devices APIs (Eg. Rx/Tx, configuring queues)
  - DPDK ring (Lockless FIFO queue)
  - DPDK packet data structure -- similar to sk_buff(kernel socket buffer) which holds network packets
  - Launching a function on particular CPU core
- **Below are optional references**
  - User level TCP stack : mTCP [paper]
  - OpenVSwitch with DPDK: getting started
  - DPDK on SRIOV [link] VFs: link1 link2

Errata

- Video time 26:25 -- L3/L4 headers are parsed in the **NIC** itself (in the video, it is mistakenly said that L3/L4 headers are parsed in the RSS itself)
- In the demo, while calculating TX speed, no. of packets sent is wrong. It's value is equal to total packets received instead of total packets sent. However, we can calculate the TX speed using the formula:

  Tx speed (Gbps) = No. of packets (Mpps) * Packet Size * 8 / (Time * (10^9))

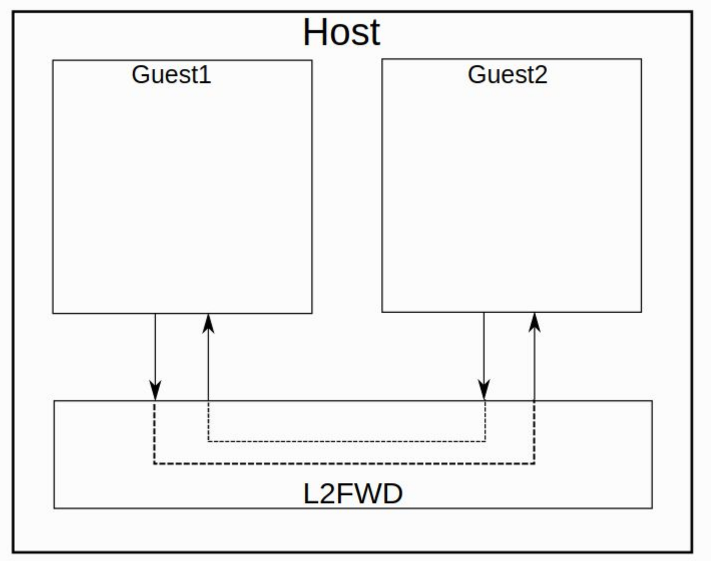
  In this case, no. of packets sent = 193913799 (& **NOT** 193920956)

  Packet size = 642 B

  Time = 30 secs

  Therefore, Tx speed = **33.198 Gbps** (~ Rx speed)

Backup Slides


Host
Guest1
Guest2
L2FWD