Alleviating Bundle Throughput Constriction for Delay Tolerant Networking (DTN) Bundles with Software-Defined Networking (SDN)

Stephanie Booth, Alan Hylton, Rachel Dudukovich,
Nadia Kortas, Blake LaFuente, and Brian Tomko
NASA Glenn Research Center
Cleveland, Ohio

***Abstract*—A load-balancing technique is proposed, executed, and tested against a Delay Tolerant Network (DTN) implementation with well-known characteristics. This would prove that transparently inserting software defined networking (SDN) to achieve load balancing without re-configuring the DTN portion is possible.**

Two routes were taken to alleviate a DTN bottleneck threat. The first used a P4 networking switch. This manual load-balancing test will balance the incoming packets without the users at the end-points knowing that their original packet destinations and/or sources may have been changed. The second route utilized a High-rate Delay Tolerant Networking (HDTN) receiving node instead of the typical delay tolerance networking (DTN) implementation used. Bench-marking results of the DTN implementation receiving node and the HDTN receiving node will be compared.

***Index Terms*—Delay Tolerant Networking (DTN), Software Defined Network (SDN), Interplanetary Overlay Network (ION), High Rate Delay Tolerance Networking (HDTN), Space Communications, Load-Balancing Network Switch**

I. Background

As space networks expand and increase complexity, bundle (and packet) throughput will also increase. This leads to the idea that individual nodes must be able to handle increased traffic, particularly from multiple sources and to multiple destinations. Consider an example with two nodes. Node 1 is sending bundles, at its processing limit, and likewise node 2 is receiving bundles at its limit. If a third node was introduced to this system sending any number of packets to node 2, then node 2 will be over capacity. In this case, the best-case scenario is that bundles must be re-transmitted. The question becomes how cases like these could be handled at a system level. An approach for future networks to load-balance themselves through a Software Defined Networking (SDN) switch or a High-rate Delay Tolerant Networking (HDTN) capability is outlined in this paper [1].

Space networks are well-known for links featuring high propagation times and intermittent connectivity. These difficulties give rise to Delay Tolerant Networking (DTN), which is an experimental network protocol designed for this environment. The DTN implementation used in this paper is the Interplanetary Overlay Network (ION), which is NASA's current implementation of delay tolerant networking for space communication applications [2]. Previous work shows ION's performance limitation capabilities where it can process, at most, low-thousands of bundles per second [1]. This provides concerns with scaling networks and limiting throughput. Since ION imposes clear constraints on network performance, it is easy to see noticeable improvement within the system if the load-balancing alleviates bundle throughput.

SDN switches range in capabilities due to hardware configuration limitations and the networking software. For the experiments within this paper, an Aurora 710 networking switch by Netberg was used. This particular switch provides 32 x 100GbE four-lane Quad Small Form-factor Pluggable (QSFP) interfaces with programmable pipelines using the Programming Protocol-Independent Packet Processors (P4) language. The Intel Tofino switching integrated circuit (IC) is capable of 3.2T switching and, therefore, will not be a bottleneck in this experiment [3]. ION was configured to transmit bundles over the User Datagram Protocol (UDP), and hence the Aurora 710 P4 networking switch was programmed to seamlessly load balance UDP packets. The authors hasten to add that any protocol, including custom protocols, could have been used.

P4 is one of many different network switch languages that can be used to change how a typical network switch works, but was chosen as it is open-source and has been around since 2013. Prior to the language's first appearance, vendors of data plane devices had total control over any device's functionality. However, now it is possible to implement specific behavior in the network within minutes by avoiding firmware and hardware development. Since P4's initial debut, the language has gone through two major reworks, known as $P4_{14}$ and $P4_{16}$ [4]. The Aurora network switch is compatible with both, but $P4_{16}$ was used for this paper.

Next generation space networks require systems that can handle longer latency times than naturally occur on Earth. HDTN is a performance-optimized implementation of DTN used in this paper, and is optimized for a nearby network of space systems [1]. HDTN is designed with a message bus architecture to attempt to eliminate software bottlenecks caused by shared memory and related locking mechanisms, such as semaphores and mutexes. HDTN decodes bundles

and metadata into internal messages to control how data flows through the system.

Figure 1 shows a high level diagram of one of the network components discussed in this paper. Two ION nodes send bundles over UDP to HDTN through the P4 switch. Bundles are received by HDTN's ingress module, which decodes the bundle header to determine the message's desired destination and time-to-live (TTL). If there is no route available to reach the destination, bundles are saved to disk by HDTN's storage module. The scheduler determines if a link is available to transfer bundles to another node. HDTN can either send bundles directly through to the egress module, or storage will retain the bundle until it receives a release message. For the experiments conducted in this paper, bundles were stored to disk when they were received.

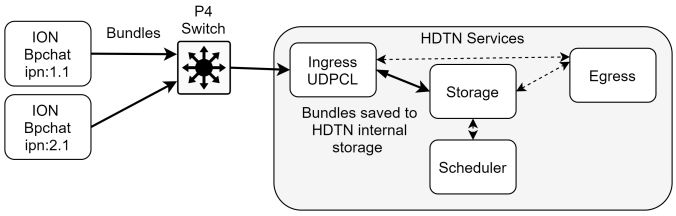

Figure 1: High Level Network Components

II. SDN Development Procedure

The Aurora 710 networking switch was provided blank, that is, without any inherent protocol support. The switch was first coded in P4 to perform packet switching on Layer 3/Internet Protocol (IP) logical addressing. Once completed, Layer 4 was taken into account. Because bundles were sent over UDP to port `4556`, which is normal for DTN, the switch can easily determine if incoming packets were bundles or not. The network design was straight-forward, and is illustrated in Figure 2. Recall that in ION, node names are numbers; in our case these are the final octets of the node's IP address. Hence `Rho`, which has IP `10.10.10.1`, is node `ipn:1`.

After basic Layer 3 switching and packet header checks, network address translations (NAT) happen next. To start with, if the final octet of the IP address ends in `10` to `19`, its original destination will remain unchanged. However, to load balance the traffic, if the source IP address's final octet is between `20` and `29`, then the destination IP will change to the first non-zero digit. For example, if the source IP ends in `.20`, then its destination IP end is changed from `.1` to `.2`. When reversing the traffic flow, source IP address will be analyzed. If the source IP address does not end in a `.1`, the source IP address will change back to the original `10.10.10.1` address. See Table I for the NAT rules for traffic on port `4556`. All the IP address renaming happens on the networking switch in real time. The checksums were then checked before the packet left the switch and were updated if necessary.

Table I: NAT Rules for Traffic on Port 4556

Original		Modification
IP Source	IP Destination	
10.10.10.[10-19, 100]	10.10.10.1	No change
10.10.10.[20-29, 100]	10.10.10.1	Destination → 10.10.10.2
10.10.10.[30-39, 100]	10.10.10.1	Destination → 10.10.10.3
10.10.10.1	10.10.10.[10-19, 100]	No change
10.10.10.2	10.10.10.[20-29, 100]	Source → 10.10.10.1
10.10.10.3	10.10.10.[30-39, 100]	Source → 10.10.10.1

Once the P4 code was compiled and uploaded to the SDN switch, we were ready to start. We enabled the hardware ports, wrote the switching tables, and manually configured the Address Resolution Protocol (ARP) tables (statically) between the machines. An example of the hardware ports and switching tables are shown in Figures 3 and 4. Each port shown in Figure 3 connects to another computer with a corresponding IP address to Table I. For example, port `1` could be connected to the machine with IP address `10.10.10.100` and port

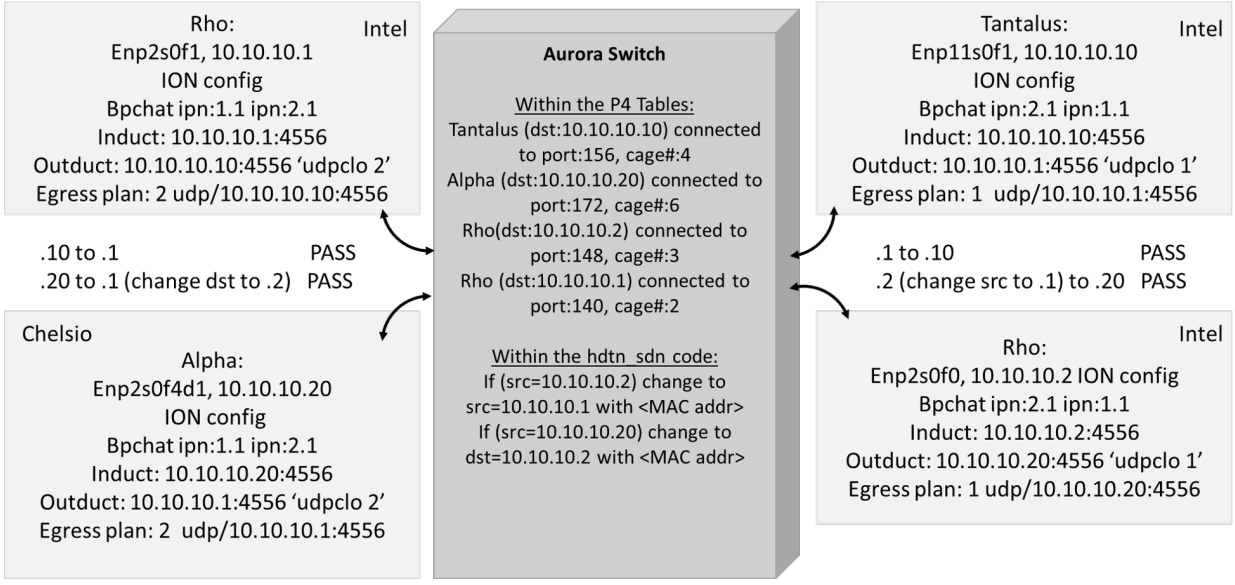

Figure 2: DTN to DTN Roadmap

2 is connected to machine with IP address 10.10.10.1. These addresses might change per test group performed due to computer malfunctions and/or limitations. Figure 4 tells the switch where to send the packet depending on its destination address via port number. Terminology overlaps here where the port number in Figure 3 is the hardware cage and lane number used. The port number in Figure 4 connects with the D_P column for Device Port (logical pipe) in Figure 3.

```
bf-sde> pm show
-----+---+---+---+-------+----+---+---+---+--------+---------------+---------------+-
PORT |MAC |D_P|P/PT|SPEED  |FEC |RDY|ADM|OPR|LPBK    |FRAMES RX      |FRAMES TX      |E
-----+---+---+---+-------+----+---+---+---+--------+---------------+---------------+-
1/0  |23/0|132|2/ 4|40G    |NONE|YES|ENB|UP |  NONE  |            213|              0|
2/0  |22/0|140|2/12|40G    |NONE|YES|ENB|UP |  NONE  |            219|              0|
3/0  |21/0|148|2/20|40G    |NONE|YES|ENB|UP |  NONE  |            220|              0|
4/0  |20/0|156|2/28|40G    |NONE|YES|ENB|UP |  NONE  |            216|              0|
5/0  |19/0|164|2/36|40G    |NONE|YES|ENB|UP |  NONE  |              3|              0|
6/0  |18/0|172|2/44|40G    |NONE|YES|ENB|UP |  NONE  |            218|              0|
```

Figure 3: Aurora 710 Hardware Port Diagnostics

```
Table ipv4_host:
----- ipv4_host Dump Start -----
Default Entry:
Entry data (action : NoAction):

pipe.Ingress.ipv4_host entries for action: Ingress.send
hdr.ipv4.dst_addr    port
------------------   ------
0x0A0A0A0A           0x9C
0x0A0A0A14           0xAC
0x0A0A0A02           0xA4
0x0A0A0A64           0x84
0x0A0A0A01           0x8C
```

Figure 4: P4 Switching Table for Table Labeled ipv4_host

Once operational, preliminary tests were conducted of the P4 software using a packet creation software called Scapy[1] before adding in the DTN implementation. Following Table I, the results showed that the code alters packets only if it was a UDP packet on port 4556 and all other UDP and TCP packets were switched as normal.

III. SDN Manually Load Balancing ION to ION

The users from each endpoint should be hidden from the knowledge that their packet destination was altered in the middle of its way to the destination. Hence, between IP addresses ending between .1 and .10, packets are switched without modification. Between .2 and .20 the switch changes the source of .2 to .1 or the destination (if packet is coming from .20) to .1. For a visual of this, see Figure 5. Here the switch shows that it can manually load balance based on what IP address it came from by changing the destination address. However, due to how Linux's TCP/IP stack is designed and a computer malfunctioning, the DTN implementation tests were broken up between simple switching and modified switching; see Figure 2 for details of each DTN implementation link tested.

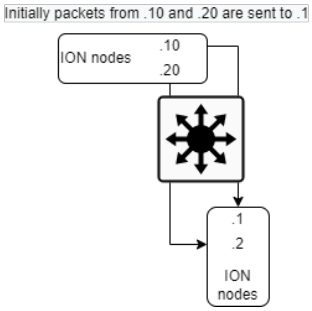

Figure 5: DTN to DTN Load Balancing Overview

[1]See https://scapy.net/.

The ION implementation's version of bpchat was used to test link connection but was also verified with Wireshark[2]; this software is often used diagnostically to send text messages over bundles. Because the Address Resolution Protocol (ARP) was not implemented in P4, the ARP tables were manually edited to establish the links. The first test completed used simple UDP switching with an UDP destination port equaling 4556. This test had no changes to the IP and hardware addresses as expected. See results in Figure 6.

Figure 6: Simple Switching with the SDN Switch

The second test required the SDN switch to modify the packets and kept the UDP destination port as 4556. For bpchat to work, it required IPv4 and UDP checksums to be correct at the receiving node. The address changes are shown in Figure 7 and follow Table I.

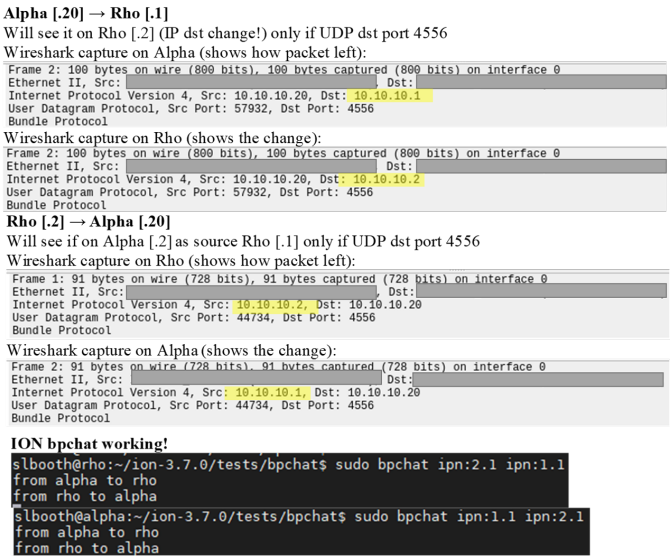

Figure 7: Modified Switching with the SDN Switch

The last test of the manual load balancing efforts with the Aurora 710 network switch found throughput limitations for each DTN

[2]See https://www.wireshark.org/.

link. This was to give a baseline prior to sending traffic from two DTN nodes to a single recipient DTN node with HDTN. Typical performance-measuring tools for DTN, and particularly ION, include `bpdriver` and `bpsink`, which can send fixed numbers of bundles or stream bundle continuously. Both are designed to work as quickly as possible.

These results are shown in Table II. At 1000 bundles the time-to-live (TTL) value was hit. We recall that in the context of DTN, TTL is not hop-based, but rather time-based. The tool `bpdriver` uses a default value of 30 seconds. The SDN switch was able to send each packet through but the destination DTN node could not process as many before the TTL expired. This shows that the DTN implementation used is asymmetric with itself and that in this scenario for 1000 bundles, a node to node link fails.

Table II: Preliminary DTN to DTN Baseline Results

Tx Computer	Rx Computer	Status
Rho	**Eta**	
Stopping bpdriver. Total bundles: 100 Time (seconds): 0.535 Total bytes: 100000 Throughput (Mbps): 1.495	Stopping bpcounter; bundles received: 100 Time (seconds): 9.486 Total bytes: 100000 Throughput (Mbps): 0.084	PASS
Stopping bpdriver. Total bundles: 1000 Time (seconds): 5.338 Total bytes: 1000000 Throughput (Mbps): 1.499	Stopping bpcounter; bundles received: 438 Time (seconds): 20.130 Total bytes: 438000 Throughput (Mbps): 0.174	FAIL
Omicron	**Eta**	
Stopping bpdriver. Total bundles: 100 Time (seconds): 0.558 Total bytes: 100000 Throughput (Mbps): 1.433	Stopping bpcounter; bundles received: 100 Time (seconds): 12.765 Total bytes: 100000 Throughput (Mbps): 0.063	PASS
Stopping bpdriver. Total bundles: 1000 Time (seconds): 2.656 Total bytes: 1000000 Throughput (Mbps): 3.012	Stopping bpcounter; bundles received: 810 Time (seconds): 23.921 Total bytes: 810000 Throughput (Mbps): 0.271	FAIL

IV. Balancing Multiple DTN nodes with HDTN

The DTN implementation transmitting machine names for these tests are named Rho (with IP address ending in `.1`) and Omicron (with IP address ending in `.2`). The HDTN node resides on Eta with an IP address of `10.10.10.100`. The Aurora 710 network switch will be between the ION nodes and the HDTN node. Here the HDTN node should only see packets from `10.10.10.1` and therefore will be listening for only that IP, as shown in Figure 8.

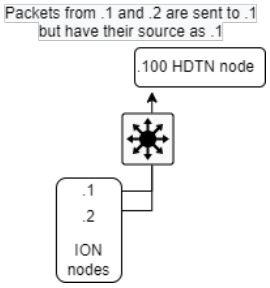

Figure 8: DTN to HDTN Load Balancing Overview

The HDTN node was configured first to non-volatile storage. Once results were received, `bpsink` capabilities were tested. No differences of the results proved one configuration was more ideal than the other for these tests with ION.

For a preliminary test of DTN to HDTN, the same test of Table II was conducted except Eta was configured with HDTN. The results brought forth all successes for both ION node links. In addition, when both ION nodes were transmitting 1000 bundles, the HDTN node received all as shown in Table III. Since the previous results of the DTN implementation used had failures (as described in Section III), the PASS results for HDTN under the same conditions are prodigious.

Table III: DTN to HDTN Test Results

Tx Computer	Rx Computer	Status
Rho -hdtn_host1.rc **Omicron -hdtn_benchmark1.rc**	**Eta ./runscript.sh** **Only to Storage**	
@rho: bpdriver 100 ipn:1.1 ipn:2.1 -1000 t30 Stopping bpdriver. Total bundles: 100 Time (seconds): 0.580 Total bytes: 100000 Throughput (Mbps): 1.378 **@omicron:** bpdriver 200 ipn:1.1 ipn:2.1 -1000 t30 Stopping bpdriver. Total bundles: 200 Time (seconds): 0.782 Total bytes: 200000 Throughput (Mbps): 2.046	**@eta:** hdtn m_bundleCountStorage: 302 m_bundleCountEgress: 0 m_bundleCount: 302 m_bundleData: 312083	PASS
@rho: bpdriver 1000 ipn:1.1 ipn:2.1 -1000 t30 Stopping bpdriver. Total bundles: 1000 Time (seconds): 2.565 Total bytes: 1000000 Throughput (Mbps): 3.119 **@omicron:** bpdriver 1000 ipn:1.1 ipn:2.1 -1000 t30 Stopping bpdriver. Total bundles: 1000 Time (seconds): 3.109 Total bytes: 1000000 Throughput (Mbps): 2.573	**@eta:** hdtn m_bundleCountStorage: 2002 m_bundleCountEgress: 0 m_bundleCount: 2002 m_bundleData: 2083906	PASS

ION's `bpdriver` sends its first bundle twice which does not pose counting issues with its counterpart, `bpcounter`. Since HDTN counts every packet going to storage, even if it is a duplicate, it counts that first packet from both transmitting DTN nodes. This results in the extra two packets counted at the receiver.

V. Summary

With increasing network throughput and capability needs, bottlenecks are important to avoid. Therefore, it is crucial to circumvent performance restrictions of the network. Thankfully, some options have been found for DTN implementation nodes. Manually load balancing packets with a SDN switch and/or using an HDTN receiver node instead are both viable options for the modern network system.

SDN development continues to provide solutions to 100G data rates for High-rate DTN (HDTN). Current research in teaching the Aurora network switch platform how to automatically load balance traffic between ports without the need of the IP changes hard-coded into the P4 code continues. In addition, future work will incorporate the bundle egress to neighboring node and finding limitations to HDTN's capabilities.

References

[1] A. Hylton, D. Raible, and G. Clark, "A delay tolerant networking-based approach to a high data rate architecture for spacecraft," *IEEE Aerospace Conference*, 2019.
[2] NASA, "Interplanetary overlay network additional information."
[3] Netberg, "Aurora 710."
[4] O. N. F. ONF, "P4 open source programming language," 2021.