Lab5: Switches Do Dream Of Machine Learning! (Report)

Group number:	Group 11
Group members:	Adithya Vasisth, Rohaan Almeida, Sudarsan Sivakumar
Slip days used:	3 + 2
Bonus claims:	Handling other traffic, Aggregating floating points & Combination

1 Basic In-Network Aggregation

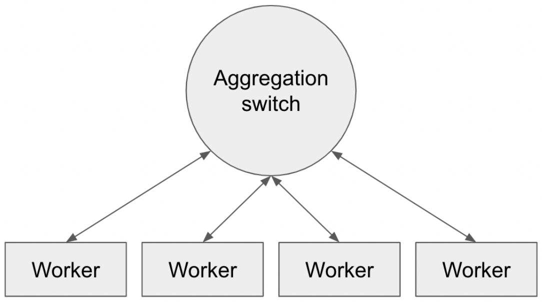


Figure 1: SwitchML architecture

In distributed machine learning, each worker computes local gradients for each iteration and synchronizes with other workers to aggregate these gradients. The SwitchML[1] simplifies this process by leveraging Top-of-Rack switches as aggregation points for workers within the same rack, as shown in Figure 1. Rather than requiring complex communication between workers, each worker breaks its gradients into small chunks and sends them to the switch. The switch then aggregates the Chunk (C_i) from all workers(W_j) and broadcasts the result to the workers. This strategy allows for in-network computation, reducing communication complexity between workers.

$$RC_i = \sum_{j=0}^{WC} C_{ji}$$

This report will describe how we have implemented the concept of in-network computation using P4 programming, focusing on the Ethernet, UDP, and UDP with reliability guarantee communication protocols. Specifically, we will detail how workers, upon receiving the Resultant Chunk (RC_i), will send the next Chunk (C_{i+1}) until the entire gradient is aggregated. This process will be repeated until all gradients have been fully aggregated.

1.1 Ethernet Communication

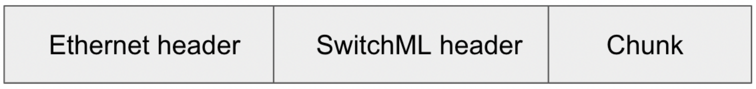


Figure 2: Ethernet frame composition

Worker

The worker generates an ethernet frame(as shown in Figure 2) to be sent to the switch for aggregation with the following specifications:

- EtherType: 0x8777
- Destination MAC Address: ff:ff:ff:ff:ff:ff
- Payload: It consists of the SwitchML header and the Chunk

The SwitchML header within the payload includes the following fields:

- num_workers: a 32-bit integer indicating the total number of workers in the system
- chunk_size: a 32-bit integer indicating the size of the chunk

To facilitate the transmission of this information, we have modified the network.py program to pass the NUM_WORKERS value to worker.py through command line arguments. This allows the workers to access the total number of workers in the system and simplifies the implementation.

Switch

Upon receiving an Ethernet frame, the switch verifies the EtherType to ensure it is 0x8777. If the EtherType is correct, the switch parses the SwitchML header and Chunk held within the payload. The switch's control plane then creates a multicast group comprising all ports connected to the workers. After the switch has completed its computation (as described in greater detail in Section 2), the ingress stage marks the frame for transmission to the multicast group. The frame is then sent back to all the workers.

1.2 UDP Communication

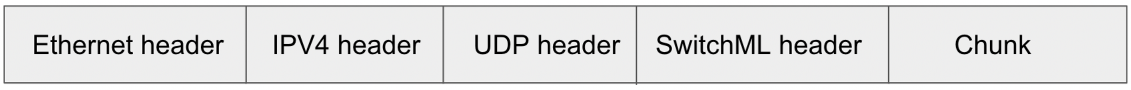


Figure 3: UDP packet composition

Worker

We create a UDP (DGRAM) socket in the worker and bind it to port 8000. The payload of the UDP packet(Figure 3) contains the SwitchML header and the Chunk (similar to the Ethernet frame). The worker then sends the packet to the address "10.0.0.0:8000". After sending the packet, the worker receives the aggregated value from the socket and repeats this process with the next chunk. By fixing all communication to port 8000, we simplify the implementation.

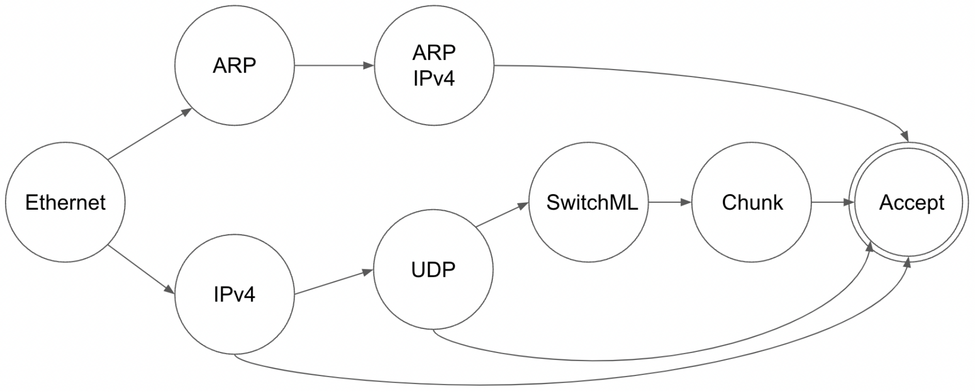


Figure 4: UDP parse graph

Switch

Upon receiving a UDP packet, the switch examines the destination port to determine if it is 8000. If the destination port is 8000, the switch parses the packet for the SwitchML header and Chunk. The overview of the parse graph for UDP-based SwitchML is shown in Figure 4. After completing the computation for all worker chunks, the packet then multicasts the aggregated output to all workers. To facilitate this process, the switch's control plane creates a multicast group comprising all ports connected to the workers. In the egress stage, we install a table from the control plane that matches the *egress_port* and provides the IP and MAC address of the node connected to that port as action parameters. The action modifies the packet's source and destination MAC and IP addresses. This allows the switch to route the multicast packet to the workers correctly.

ARP handling

In order to establish communication via sockets, the worker sends an ARP request at the start to identify the destination MAC address. Since we are only expecting SwitchML traffic in the system, we configured the switch to reply to all the ARP requests with its MAC address (Later, for all traffic in bonuses, we had to change it). To implement this, we hard-coded "00.00.00.00.00.01" as the MAC address of the switch in P4. While it would be more robust to obtain the MAC address of the switch from the control plane, we felt that hard-coding the MAC address was sufficient for the purposes of this assignment.

2 Basic Implementation

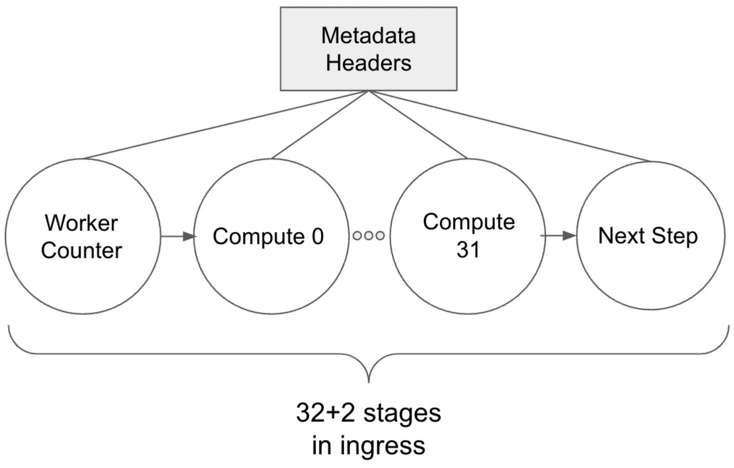


Figure 5: Basic Ingress Stages

To effectively aggregate the chunks, the switch must store and access chunks across packets, requiring the use of stateful memory in the program. However, there is a constraint on the Intel Tofino switch that limits access to stateful memory to **one read** and **one write** per stage per packet per circulation. As a result, we designed the flow of the program to adhere to these constraints. Figure 5 illustrates the flow of the program across stages.

1. **Worker Counter**: This stage serves the purpose of keeping track of the number of workers arriving at the switch to complete one aggregation cycle. To do this, we store a counter in a register. When the packets from the first worker arrive, we initialize the counter value to be one less than the number of workers reported in the packet. For subsequent workers, we continue to decrease the counter value until it reaches zero. Additionally, we maintain a flag in the metadata called *first_ last_ flag*. This flag is set to 0 and 1 for the first and last workers, respectively. By tracking the progress of the aggregation process in this way, we can ensure that all workers have contributed to the aggregates before proceeding to the next iteration.

2. **Compute**: These stages are responsible for aggregating the chunks received from all the workers. Each compute stage is assigned to handle the aggregation of a specific index of the chunk. Since we have a limited number of stages (32) and only support one circulation of the packet, we set the maximum possible chunk size to 32. When processing a chunk, the compute stage checks if the index of the chunk is less than the *chunk_ size* specified in the packet. It then checks the value of the *first_ last_ flag*. If the flag is 0, the value in the packet at the given index is copied to the register at the same index. If the flag is not 0, the program reads the value in the register at the given index, updates the register by adding the value from the packet at the same index, and writes the computed value back into the packet. This process is repeated until all chunks have been fully aggregated. If the packet being processed is from the last worker, it is broadcasted to all workers. This step is handled by the next stage.

3. **Next step**: After all workers have contacted the switch, the aggregated results must be broadcasted to all workers. When a packet from the last worker reaches this stage, the *first_ last_ flag* will have a value of 1. If this is the case, the packet is multicast to all workers using the methods described in the communication section for each protocol.

2.1 How to Run It

We have three variables that we have provided the flexibility to handle in our code.

- **NUM_WORKERS**: This variable is inside the *network.py* file. We support up to 8 workers in our setup.
- **CHUNK_SIZE**: This variable is inside the *worker.py* file. This denotes how many values we can pack inside a packet to continue the aggregation. We support the maximum chunk size of 32.
- **ITER_NUM**: This variable is inside the *worker.py* file. This denotes how many iterations of the tests we can run in our setup. We have tested up to 32 iterations.

To run any of our programs, we can do so by varying any of the variables above. We then go ahead and set up our mininet. These are the steps:

1. We run *start.sh* to spin up the mininet.
2. We can then run the SwitchML program by running *py net.run_ workers()*
3. We can verify the results in the log folder.

3 Fault-Tolerant Aggregation Protocol

In this scenario, we have to provide the aggregation over a lossy network. We overcome message losses through timeout-based retry till we receive a reply for the expected chunk. But the existing aggregation operation is not idempotent. An additional mechanism is needed to ensure correctness. In this section, we will describe how we extended the above UDP solution for this scenario. Let's start with the fields we added to the SwitchML header.

- worker_rank: a 32-bit field that denotes the worker's rank
- chunk_id: a 32-bit field that denotes the index of the chunk, the value ranges from [1, CHUNK_COUNT]
- total_chunks: a 32-bit field that denotes CHUNK_COUNT
- slot_mod: an 8-bit field that determines the slot for that chunk computation. The value of this field will be chunk_id(mod)2. We fix the number of slots to 2(We will discuss slots in the next section).

4 Fault-Tolerant Implementation

First, we begin by identifying the properties of the system

1. A message loss can happen either at the worker sending chunk to switch or the switch sending back aggregated reply back to the worker.
2. We identified that the workers are in synchronization in every iteration. All the workers in the system are either waiting for ith chunk reply or i+1th chunk reply.

Hence we decided to have two slots to store the data of the previous chunk(ith chunk) in case some workers still didn't receive a reply while some other workers have started to send i+1th chunk. To alternate slots between successive chunks, the worker does chunk_index(mod)2 and provides that value in slot_mod of the SML header (We do this mod in worker instead of switch because it's not supported in the Tofino switch).In addition, we added another stage in ingress to check if the packet is duplicate and additional checks to existing models.

- To identify duplicates, we added a stage at the start which maintains an array of stateful counters where each index is for a worker. It marks a packet as not duplicate(metadata.process is set to 1) and updates the local value(value stored in that worker's index) to the chunk_id(Retrieved from sml header) if the chunk_id of that packet is greater than local value. If not, the chunk_id is equal to the local value or increment of it is equal((chunk_id + 1 == local_value)||(chunk_id == local_value)), then we mark that packet as a duplicate(metadata.process is set to 0). If none of the above cases match, then that could be the 1st packet from the next iteration. Hence, we mark the packet as not a duplicate.
- Depending on whether a packet is a duplicate(metadata.process==0), other stages have functionality differences. The worker counter stage updates the number of workers sent the chunk for that slot only if it's not a duplicate. Compute stages updates local value only if the packet is not a duplicate.

4.1 How to Run It

There are no changes to this section, and the process is the same as how it is described in 2.1.

5 Bonus 1: Handling Other Traffic

5.1 Implementation

The implementation of this bonus is quite simple. To make the switch handle traffic of all types of IPv4 packets, we follow these few simple steps:

1. Insert table entries into the switch. As this is a simple implementation with 8 hosts and 1 switch. We need to add table entries in switch 0. Each table entry has the key matching the Destination IPv4 address, and the parameters include the next hop's mac address and the output port. In this case, the destination mac is the same as the next hop's mac address.
2. **Parser**: In the *main.p4*, we first have the parser function as it should. We accept all types of traffic as default, and if it is SML traffic, we parse it further as needed for the actual Switch ML implementation.
3. **Ingress**: We implement the routing only if the packet doesn't have an SML header. This allows us to take a different path for the SML packets. For the IPv4, we have a switch table, which uses the destination IPv4 address as the key. On match, the action specified in step 1 will be executed. We perform the basic routing operations here, similar to the implementation in Lab 4. We set the packet's destination mac address and egress port as provided by the table entry's parameters. We also reduce the TTL.

5.2 Interesting phenomena observed

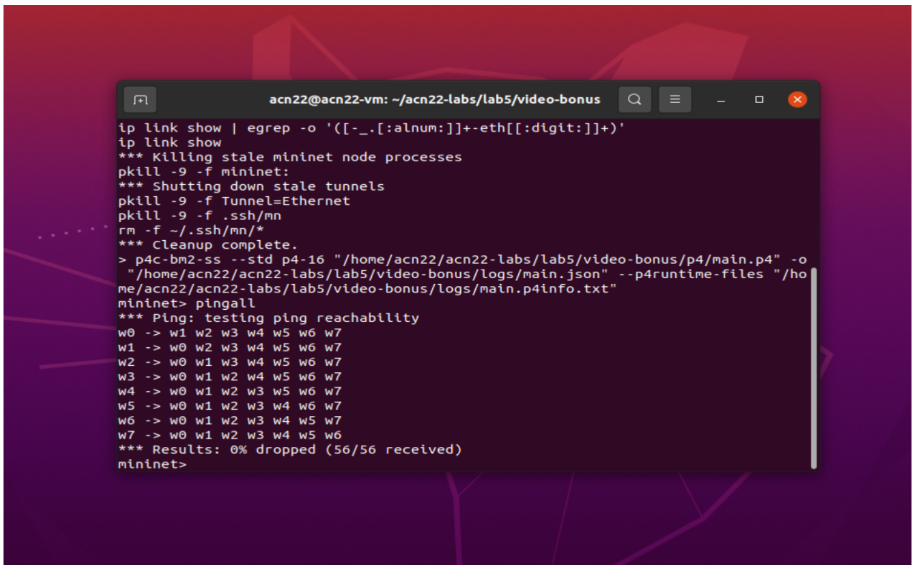
We described a naive way of handling ARP requests in Section 1.2. At that time, we thought the system only had SML requests. When we extended the solution for all traffic, we observed a phenomenon. When one worker node sends an ARP request to another worker node, the switch replies with its own MAC address. But, when the UDP packets are being sent from that node(Destination MAC address as "00:00:00:00:00:01"), the switch modifies the destination and source MAC address to the actual destination node's MAC and switch MAC, respectively as we have specified in the IPv4 routing stage. The switch acts like a Layer 2 proxy.

5.3 How to Run It

1. We run *start.sh* to spin up the mininet.

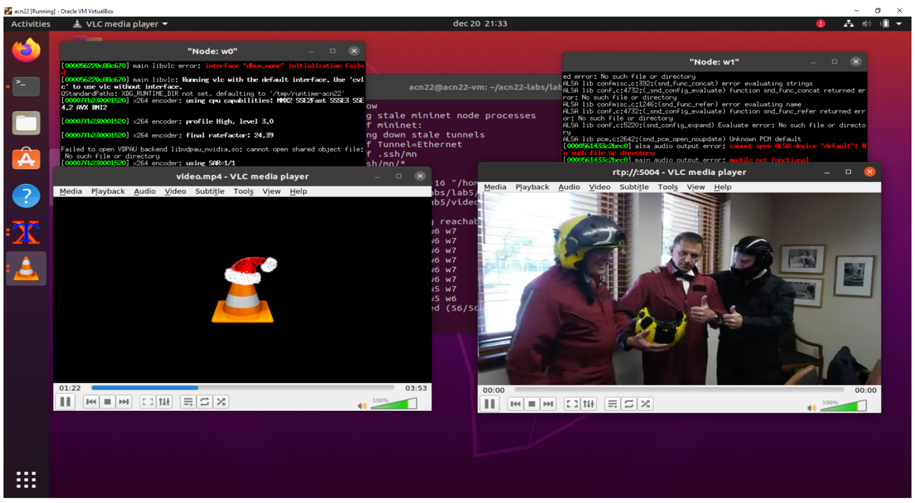
2. We can then run *pingall* command. This will work correctly in our implementation.
3. **Streaming Video**:
   (a) We can run *xterm w0 w1*
   (b) In w0, we can spin up the server by running *vlc-server.sh*
   (c) In w1, we can run set up the client by running *vlc-client.sh*

5.4 Results



```
ip link show | egrep -o '([-_.[:alnum:]]+-eth[[:digit:]]+)'
ip link show
*** Killing stale mininet node processes
pkill -9 -f mininet:
*** Shutting down stale tunnels
pkill -9 -f Tunnel=Ethernet
pkill -9 -f .ssh/mn
rm -f ~/.ssh/mn/*
*** Cleanup complete.
> p4c-bm2-ss --std p4-16 "/home/acn22/acn22-labs/lab5/video-bonus/p4/main.p4" -o
 "/home/acn22/acn22-labs/lab5/video-bonus/logs/main.json" --p4runtime-files "/ho
me/acn22/acn22-labs/lab5/video-bonus/logs/main.p4info.txt"
mininet> pingall
*** Ping: testing ping reachability
w0 -> w1 w2 w3 w4 w5 w6 w7
w1 -> w0 w2 w3 w4 w5 w6 w7
w2 -> w0 w1 w3 w4 w5 w6 w7
w3 -> w0 w1 w2 w4 w5 w6 w7
w4 -> w0 w1 w2 w3 w5 w6 w7
w5 -> w0 w1 w2 w3 w4 w6 w7
w6 -> w0 w1 w2 w3 w4 w5 w7
w7 -> w0 w1 w2 w3 w4 w5 w6
*** Results: 0% dropped (56/56 received)
mininet>
```

Figure 6: Pingall command working across all the eight workers


acn22 [Running] - Oracle VM VirtualBox
Activities
VLC media player
dec 20 21:33
"Node: w0"
video.mp4 - VLC media player
Media Playback Audio Video Subtitle Tools View Help
01:22
03:53
"Node: w1"
rtp://:5004 - VLC media player
Media Playback Audio Video Subtitle Tools View Help
00:00
00:00
acn22@acn22-vm: ~/acn22-labs/lab

```
[000056220c88c670] main libvlc error: interface "dbus,none" initialization faile
d
[000056220c88c670] main libvlc: Running vlc with the default interface. Use 'cvl
c' to use vlc without interface.
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-acn22'
[00007fb230001520] x264 encoder: using cpu capabilities: MMX2 SSE2Fast SSSE3 SSE
4.2 AVX BMI2

[00007fb230001520] x264 encoder: profile High, level 3.0

[00007fb230001520] x264 encoder: final ratefactor: 24.39

Failed to open VDPAU backend libvdpau_nvidia.so: cannot open shared object file:
 No such file or directory
[00007fb230001520] x264 encoder: using SAR=1/1
```

```
ed error: No such file or directory
ALSA lib confmisc.c:392:(snd_func_concat) error evaluating strings
ALSA lib conf.c:4732:(_snd_config_evaluate) function snd_func_concat returned er
ror: No such file or directory
ALSA lib confmisc.c:1246:(snd_func_refer) error evaluating name
ALSA lib conf.c:4732:(_snd_config_evaluate) function snd_func_refer returned err
or: No such file or directory
ALSA lib conf.c:5220:(snd_config_expand) Evaluate error: No such file or directo
ry
ALSA lib pcm.c:2642:(snd_pcm_open_noupdate) Unknown PCM default
[0000561433c2bec0] alsa audio output error: cannot open ALSA device "default": N
o such file or directory
[0000561433c2bec0] main audio output error: module not functional
```

Figure 7: Worker 0 is acting as a server and is streaming the video to Worker 1. Worker 1 is acting as the client and can view the video. Here we see that Worker 1 can view the video.

6 Bonus 4: Aggregating Quantized Weights

The Intel Tofino switch doesn't support floating point computation (In fact, no switches till now support floating point computation). But mostly, the gradients contain floating points. To support it, SwitchML paper uses block floating point representation. The general approach is the worker scales the floating point to a fixed point, and the Switch aggregates the integers and sends it back. The Worker scales it back down to a floating point. The scaling should also guarantee to have minimum error bound and prevent overflow of values during aggregation (As derived in Appendix E of the SwitchML paper).

6.1 Implementation

1. **Worker**: The scaling factor of a chunk is calculated as specified in the SwitchML paper. We round the absolute maximum value in the chunk to the nearest 2-power integer(K). The $\log_2$(K) is a $\bar{M}$ value. The Switch helps in deciding the maximum M value across workers(M = Max($\bar{M}_j$; for all j in NUM_WORKERS)) for a chunk. The workers compute the scaling factor through the following equation $(2^{31}-1)/(N*2^M)$, where N is the number of workers.

   To identify the maximum M value across workers for a chunk, we have included another field in the SML header, that is **bit<32> m_val**. We piggyback the local $\bar{M}$ value for the i+1th chunk when sending scaled ith chunk to the Switch. The Switch replaces that field with a max M value when replying. To find the M value for the initial chunk, we send a dummy chunk along with the locally computed $\bar{M}$ value for chunk 1. Once the Switch replies with the Max M value, we discard that dummy chunk.

2. **Switch**: In Switch, we have added another stage after the compute stages, which stores the max M value seen for that chunk in a register. We update the register only if the packet is not a duplicate packet and the received M value is greater than the current M value in that register slot (Slot chosen based on slot mod as discussed in reliable communication) or the packet received is the first for that chunk. It also replaces the M value in the packet.

6.2 How To Run It

There are no changes to this section, and the process is the same as described in 2.1.

7 Bonus 5: Combining Bonus 1 and Bonus 4

7.1 Implementation

The implementation is quite simple. It is the combination of the implementation of the routing of all types of traffic and the implementation of the floating point computation.

7.2 How To Run It

1. We run *start.sh* to spin up the mininet.
2. We can then run *pingall* command. This will work correctly in our implementation.
3. **Streaming Video**:
   (a) We can run *xterm w0 w1*
   (b) In w0, we can spin up the server by running *vlc-server.sh*
   (c) In w1, we can run set up the client by running *vlc-client.sh*
4. **Computation**: We can then run the SwitchML program by running *py net.run_ workers()*
5. We can verify the results in the log folder.

References

[1] Amedeo Sapio et al. "Scaling Distributed Machine Learning with In-Network Aggregation". In: *CoRR* abs/1903.06701 (2019). arXiv: 1903.06701. URL: http://arxiv.org/abs/1903.06701.