PADERBORN UNIVERSITY

Advanced Networked Systems (SS25)

Lab3: Switches Do Dream of Machine Learning

Maximum points:	16
Submission:	zipped source code on PANDA
Deadline:	15.07.2025 23:59
Contact:	lin.wang@upb.de with subject prefix "[ANS-SS25] Lab3"

1 Introduction

In this lab, we will use in-network computing to accelerate AllReduce [5], an important collective operation, and central to data-parallel distributed deep neural network (DNN) training [4]. The main goal of this lab is to implement a simplified version of SwitchML [8] at three levels of complexity:

1. In-network AllReduce over Ethernet
2. In-network AllReduce over UDP
3. In-network AllReduce over UDP with reliability

This lab will put your overall networking (and P4) knowledge to the test, as you will need to implement everything (almost) from scratch. **Waste no time and start immediately!**

Please run `git pull` in the labs codebase to retrieve the code template for this lab. You will see a new folder `lab3` and you are supposed to work in that directory for this lab. Before you start, you should first run command `bash ./install_p4_env.sh` in the course VM to prepare the P4 development environment. Note that this script only works for Ubuntu 20.04. If you are using the course VM with Vagrant, there should not be any problem when directly running the prepared script; otherwise, you might face multiple issues. You may consult this page for troubleshooting.

2 In-Network Aggregation

Before reading any further please make sure you have read, and understood, what AllReduce is [5], and, to a reasonable degree, the main ideas of the SwitchML paper [8].

There are N workers (i.e., servers here) connected to a single Top-of-Rack (ToR) switch that is P4-programmable. Workers are identified by a unique rank $r \in \{0, 1, ..., N-1\}$. Each worker is connected to the switch port that corresponds to its rank. Each worker is running a loop to train the deep neural network (DNN) on its local subset of the data. At iteration i, worker with rank r produces a vector V_i^r that holds its contribution to the DNN update. To be able to proceed to the next iteration, each worker must receive the global DNN update $\overline{V_i}$ by aggregating the local updates from all the workers:

$$\overline{V_i} = V_i^0 + V_i^1 + ... + V_i^{N-1} \tag{1}$$

To compute $\overline{V_i}$ all workers collectively invoke AllReduce on their local vectors. That is, for each rank r, $\overline{V_i} =$ `AllReduce(`V_i^r`)`. Your task for this assignment is to implement this function using a host-network co-design, following a simplified version of the SwitchML protocol.

2.1 In-network Aggregation à la SwitchML

In-network AllReduce works by having workers stream their local vectors, in chunks of size C, to the switch. The switch aggregates those chunks, and after N aggregations, it broadcasts the result to all the workers. In particular, for every `AllReduce(`V_i^r`)` call, the behavior of workers is as follows: Until V_i^r is entirely streamed out, worker r does the following:

1. Send $V_i^r[s : t]$, where $t - s = C$ and wait for a response,
2. Write the response to $\overline{V_i}[s : t]$,
3. Increase s and t by chunk size C and repeat the above.

The switch behavior is as follows. Upon receiving a chunk:

1. If it is not the last chunk for this aggregation round, aggregate the values in the chunk, store the aggregation result on the switch, and drop the packet.
2. If it is the last chunk,
    a. aggregate the values,
    b. write the aggregation result to the packet,
    c. prepare the memory for reuse, and
    d. multicast the packet to the involved workers.

In the above scheme, each worker sends a single packet containing one chunk at a time, and waits for a single response (the aggregation result for this round) in order to proceed. Thus, the switch only needs enough memory to aggregate only a single chunk at a time. This is equivalent to a single-slot version of the SwitchML protocol. More elaborate schemes (closer to the real SwitchML implementation) are offered as bonuses (see §7).

2.2 Requirements and Assumptions

You solutions to all three levels of this lab are required to respect the following:

1. Vector elements are 32-bit unsigned integers.
2. Vectors at the workers may have arbitrary lengths, although all vectors involved in the same AllReduce invocation will always have equal lengths.
3. Your solution should work with any value of $N \in [1, 8]$.
4. Chunk size C should always be greater than one.
5. A single switch pipeline can perform at most 32 aggregations per packet traversal. You can double that amount by doing 64-bit register accesses, but you are not required to.
6. A general requirement is that your P4 code respects (as much as possible) a real switch architecture.

Some of the constraints to respect for a real switch architecture are listed below.

Operations. On a real Intel Tofino switch, the multiplication (×), division (/) and modulo (%) operations are not available. Although P4 and the BMv2 software switch (we are currently using for this lab) allow these, using them in your code will cost you points. Fortunately, the remaining operators (see [1, §8]) are more than enough for a correct implementation.

Accessing memory. A byproduct of the the stage-local resources model for the RMT architecture is that a real Tofino switch allows accessing a register (stateful memory as an extern) exactly once, at the exact time when the packet is processed by the stage the register is allocated to. The Tofino Native Architecture (TNA) exposes a `RegisterAction` extern [2, §7.13] for performing atomic read-modify-write operations on registers. The code inside a `RegisterAction` can read the register exactly once, perform limited logic (e.g., to decide if a new value should be written), and potentially write the register, exactly once. In contrast, BMv2 switches allow unrestricted register accesses through a register's `read` and `write` methods [7]. This not only allows to write code that will never compile to real hardware but creates an additional complication: data races. Fortunately, the P4 language defines an `@atomic` annotation [1, §18.4.1] that allows to group statements for atomic execution w.r.t. other packets that might be processed and thus access the same memory concurrently. A good solution is expected to access registers atomically, exactly once[1], and with minimal logic.

ℹ You can use a `@atomic` control blocks to create your own read-modify-write primitives for registers, effectively emulating TNA's `RegisterAction`:

[1] Your code may branch arbitrarily, but no path should attempt more than one read and write.

```
control MyAtomicAction (in register<T> r, out T old, ... ) {
  apply {
    @atomic {
        r.read(old);
        /* other code */
    }
  }}
```

In addition, the BMv2 `simple_switch_grpc` [3] software switch we use comes with its own set of restrictions, the most important of which are listed below (but do check [3] for more):

1. Header sizes must be multiples of 8 bits.
2. You can only `shift` by an 8-bit value.
3. No branching is allowed in the deparser (see the hint below for some inspiration).

ℹ `pkt.emit(hdr.x)` will emit header `x` only if `x.isValid()` is true.

You may make the following assumptions to make your job easier:

1. Number of workers N and chunk size C are compile time constants that do not change at run time.
2. Vector lengths are always even multiples of C, and greater than 0.
3. Vector values are always in range [`0x0`, `0xffff`], so aggregation overflow is impossible.
4. You do not have to deal with other network traffic (e.g., normal IP traffic) and may just drop those packets not intended for SwitchML.
5. Workers are always correct (assuming you programmed them correctly). That is, you do not have to consider Byzantine faults.
6. You are free to use any P4 construct available to `simple_switch_grpc`.
7. Any P4 trickery is allowed. Surprise us!

2.3 Project Skeleton

Each level of this lab has its own (sub)directory. In general you are expected to follow the comments in the provided code template and complete the following files:

1. `network.py`: Mininet and control plane initialization code
2. `worker.py`: your worker code
3. `p4/main.p4`: your switch code

To run your solution you should first run in your terminal:

```
$ sudo ./start.sh
```

This will start Mininet, run your control plane configuration, and open a Mininet CLI. Then from the Mininet CLI, you type in

```
mininet> py net.run_workers()
```

which will run your workers and wait for their completion. A basic version of this function is already implemented for you, but feel free to modify it. To debug your code you may inspect the logs and pcaps under the `logs/` directory. A very useful file is the switch's log `p4s.<switch_name>.log`. You may also manually start workers, each on its own `xterm`. To do so, first open `xterm` windows for workers:

```
mininet> xterm w0 w1
```

and then run one worker at each window:

```
$ python worker.py <rank>
```

Our framework uses p4app [6] to create a Mininet network of P4Runtime-enabled switches, automatically compile your P4 code and run it, etc. For the biggest part you will not interact with p4app at all. However, p4app does expose a simplified P4Runtime API that allow you to programmatically control the switch, which you *might* find useful. You can find the relevant functions in `lib/p4app/src/p4_mininet.py` and you can easily invoke them like this:

```
s = net.get("my_switch_name").do_something()
```

We have also included a testing infrastructure. Worker code, before invoking `AllReduce`, will write its data to a file, and after the `AllReduce` invocation, it will check if the received aggregation result is correct. The test outcome is written in files found under `logs/test/<test_name>`. An example output of a passing test for an `AllReduce` with two workers looks like this:

```
[+] Running test: udp-iter-0, rank: 0, ts: 17:17:09.908774
[+] From data files:
  /home/acn22/acn-sml-solution/sml-udp-reliable/logs/test/test-udp-iter-0/data-rank-0.csv
  /home/acn22/acn-sml-solution/sml-udp-reliable/logs/test/test-udp-iter-0/data-rank-1.csv
[+] Result: PASS
```

Currently, it is set up such that every worker runs a test in the data it receives. You can find the testing infrastructure and other utilities you will be using throughout the lab under `lib/*.py`. They are all well-documented so feel free to read them!

3 Level 1: In-network AllReduce Over Ethernet

For the first level of this assignment your should implement `AllReduce` directly over Ethernet. You should design an aggregation protocol, including a packet format that carries aggregation data and control, as well as how such packets are processed by both the switch and the workers. The protocol you design here will form the basis for the other two levels of the lab later.

Your worker code should craft Ethernet frames, whose payload is your own `AllReduce` protocol header. You should use Scapy APIs to craft, send and receive packets directly to/from `eth0`. Your switch code should be able to understand your protocol, perform the required aggregations, and communicate the results.

ℹ The `addMulticastGroup` function from `lib/p4app/src/p4_mininet.py` allows you to create a multicast group `standard_metadata.mcast_grp` for multicasting packets (see here).

You can assume the network is reliable and congestion-free. That is, every packet sent, by both the workers and the switch, is guaranteed to be received intact.

Some questions you will need to answer for this level include:

- Where does a worker actually send to?
- How do hosts and switch distinguish your protocol over other Ethernet frames?
- How much state should the switch and workers maintain and/or share?
- What should be the value of chunk size C and how many values should a packet contain?
- How to allocate stateful memory to store intermediate aggregations on the switch?
- How to (correctly) update and reuse stateful memory?

4 Level 2: In-network AllReduce Over UDP

You will now transfer your Level 1 solution to a more realistic setting. That is, all communication should now happen over UDP sockets. Thus, you can no longer use Scapy for L2 communication. However, if you wish, you may still use Scapy to craft packets of your custom protocol.

ℹ `bytes(p)` will return a byte array from a given Scapy packet `p`.

Socket communication raises some important complications that you now need to deal with. First of all, a socket sends to, and receives from, an `ip:port` pair. But to actually put bits on the wire, your hosts' network stack needs to know the MAC address for a given target IP. But, unlike Level 1, you no longer have access to the Ethernet frames. How can you solve this? Additionally, if you sniff `eth0` and see that data is received, but not delivered to your socket, what could be the reason?

In Level 1, Scapy took care of endianess. Since you are no longer using scapy to send/receive packets, you now have to explicitly deal with this issue. You may find Python's `struct` module useful for this task.

You may still assume the network is reliable and congestion-free. That is, every packet sent, by both the workers and the switch, is guaranteed to be received intact.

Some of the questions you will need to answer here include:

- How do hosts and switch distinguish packets in your protocol over other IP packets?
- Should the switch know the workers' IP addresses? And if so how?
- How to make sure a broadcast is correctly received by all workers' sockets?

ℹ The `insertTableEntry` function from `lib/p4app/src/p4_mininet.py` allows you to insert entries from the control plane to any table defined in your P4 program.

5 Level 3: In-network AllReduce Over UDP with Reliability

In the final level, you will extend your Level 2 solution to handle packet loss. In our considered network, packet loss may occur in two places and for two reasons:

1. The switch will drop a packet if the packet is corrupted, or there is congestion (buffers are full) at the switch ingress or egress ports.
2. A host's NIC will drop a packet if the packet is corrupted or if the NIC receive buffer is full.

Your solution is expected to be able to handle both.

SwitchML handles packet loss at the hosts with a simple timeout mechanism. A worker starts a timer when it sends out a packet. If the (expected) response is received before the timeout then the worker proceeds to the next chunk. If a timeout occurs, the worker re-sends the same packet, until the correct response is received. Since a broadcast packet may not reach all the workers (i.e., a retransmission will be triggered), the switch can no longer discard a result right after the broadcast. Instead, the switch needs to store the result until the result is no longer needed. When the switch receives a re-transmitted packet, it simply copies the result and unicasts the result to the sender. This copy is maintained until no more retransmissions are possible for the given chunk.

ℹ Workers proceed to chunk j, only after having received the result for chunk $j-1$. Hence, a worker can never be (and retransmit for) two or more chunks behind.

However, a retransmitted packet may be for a chunk whose aggregation is not yet complete. The switch should be able to recognize this and not aggregate the sender's data more than once. To this end, a simple counter is no longer enough to ensure correctness, and a more involved mechanism is required in order to keep track of which worker's chunk has been aggregated. Incorporating this while respecting the hardware switch restrictions as outlined in §2.2 will (most likely) significantly complicate your switch logic.

To help you test your program we have included the `unreliable_send` and `unreliable_receive` functions in `lib/comm.py`. You may use these instead of the normal `send/receive` to exchange packets with delay, or simulate packet drops. Please consult their docs for how to use them!

Some additional questions you will need to answer here include:

- What does it mean for a packet to be corrupted, and how do you recognize this?

- How should the protocol be adapted for reliability and how much extra state (on both the switch and the workers) is required?
- How to update the switch state following the single-access memory semantics discussed in §2.2?

6 Grading Criteria

The grading will be done based on an in-person interview-style oral examination. The detailed schedule for the interview will be announced when the submission deadline approaches. For fairness consideration, you must upload your code in a zip file. Please use the naming convention `Lab3_GroupX.zip` and rename the folder before you zip it. Here `X` is your group number. You must upload your code by the specified deadline and use the uploaded version for the interview. No interview will be scheduled for you if there is no code upload; this is a strict rule.

There are in total 16 points for this lab (excluding the bonuses) and these points are distributed as follows:

- Your implementation of "L1: AllReduce over Ethernet" works correctly and passes the tests with different numbers of workers (1 to 8), and different vector sizes. (6 points)
- Your implementation of "L2: AllReduce over UDP" works correctly and passes the tests with different numbers of workers (1 to 8), and different vector sizes. (4 points)
- Your implementation of "L3: AllReduce over UDP with reliability" works correctly and passes the tests with different numbers of workers (1 to 8), different vector sizes, and different parameters in the `unreliable_send` and `unreliable_receive` functions. (6 points)

Please note that quality of your solution matters. This includes things like, good usage of switch resources, whether or not you respect hardware switch restrictions in your P4 code, doing unnecessary things that would hamper performance in a real setting, and so on.

7 Bonuses

In order for a bonus to be considered, you must have successfully implemented Level 3 of the basic assignment. That is, any bonus should be implemented by extending your Level 3 solution. For each bonus you attempt, you should create a new folder under the root folder of this lab.

7.1 B1: Handling Other Traffic

For this bonus your switch has to be able to handle other, non-SwitchML, network traffic. In particular, while the switch is performing aggregations, `pingall` from the Mininet CLI should succeed. Your control plane logic should go in `network.py`. You may extend the topology to add more hosts.

You will receive **2 extra points** if you complete this bonus task successfully.

7.2 B2: Multi-slot SwitchML

So far, you have implemented a single-slot version of SwitchML. That is, workers send a single chunk, and wait for a single response, at a time. For this bonus you will have to implement the multi-slot version of the SwitchML protocol. That is, workers now have multiple chunks in flight, expecting multiple responses, but also dealing with (potentially) multiple packet losses.

Another major problem you have to solve is how chunks are mapped to aggregation slots on the switch and who decides which slot a chunk will be aggregated to. You are free to implement the solution from the SwitchML paper, modify it, or devise your own.

You will receive **3 extra points** if you complete this bonus task successfully.

7.3 B3: Recirculation

SwitchML uses all four pipes of a Tofino switch to quadruple the number of values that can be aggregated at a time. This effectively increases aggregation throughput because:

1. Chunk size $C \times 4 \times 4$ bytes do fit in a packet, and
2. Link latency is a couple of microseconds, whereas the combined processing time of all pipes is a couple hundred nanoseconds.

For this bonus you will have to extend your solution such that it uses recirculation to aggregate 4× the elements per packet. BMv2 switches are single-pipe, but we can easily emulate 3 more with the following idea.

Packets are recirculated in the same pipe, and each time your program should be able to distinguish which (virtual) pipe it is processing a packet for. For example, if you are doing pipe 0 processing (first time you see a packet for a new chunk), then you should only aggregate part of the chunk, at indices 0 to $C/4$, then recirculate the packet to (virtual) pipe 1 which aggregates indices $C/4$ to $C/2$, and so on. You will also need to increase your register sizes such that aggregations for different pipes happen at different register indices.

In this scheme, a packet takes the following (virtual) path inside the switch:

`ingress 0` → `ingress 1` → `ingress 2` → `ingress 3` → `egress 0`

In reality, every recirculation is to pipe 0, but each time different data is accessed. You may read more about the recirculation behavior of the BMv2 switch(es) here.

To make your job easier, you may assume that, on the switch side, congestion may only occur at the ingress and egress of pipe 0 only. That is, your packet will either traverse all 4 (virtual) pipes, or none. Thus, there is never the case that only part of a chunk is aggregated.

If you are combining B1 with this bonus, you may assume that there are more ports than the number of workers N available, per pipe, and the extra servers you connect to the switch are all connected to ports in pipe 0.

You will receive **4 extra points** if you complete this bonus task successfully.

7.4 B4: Aggregating Quantized Weights

In SwitchML, vectors contain floating point weights of a DNN layer. However, the switch, having no FPUs, only aggregates 32 bit integers. This works by having workers quantize the weights before invoking `AllReduce`. For this bonus you will have to implement SwitchML's quantization approach. You may now use the `GenFloats` functions from `lib/gen.py` instead of `GenInts`, to generate floating point values. Then, you have to extend your protocol such that all workers quantize those values correctly (see the paper for more details). Then, you perform `AllReduce` on integers and de-quantize the result back to an array of floating points. You can use `RunFloatTest` instead of `RunIntTest`, to check if you received the correct values (with some precision loss).

You will receive **4 extra points** if you complete this bonus task successfully.

References

[1] The P4 Language Consortium. *P4_16 Language Specification*. 2022. URL: https://p4.org/p4-spec/docs/P4-16-v-1.2.3.html.

[2] Intel. *P4_16 Intel Tofino Native Architecture - Public Version*. 2021. URL: https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf.

[3] P4Lang - Behavioral Model. *The BMv2 Simple Switch target*. 2019. URL: https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md.

[4] Preferred Networks. *Technologies behind Distributed Deel Learning: AllReduce*. 2018. URL: https://tech.preferred.jp/en/blog/technologies-behind-distributed-deep-learning-allreduce.

[5] NVIDIA. *NCCL Collective Operations, AllReduce*. 2024. URL: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html.

[6] P4Lang - p4app. *p4app*. URL: https://github.com/p4lang/p4app/tree/rc-2.0.0.

[7] P4Lang - P4C. *v1model.p4*. 2021. URL: https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4#L305.

[8] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. "Scaling Distributed Machine Learning with In-Network Aggregation". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2021, pp. 785–808.