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

Group number:	5
Group members:	Maciej Kozub, Max Anderson, Paulo Liedtke
Slip days used:	3
Bonus claims:	0

1 Basic In-Network Aggregation

The network is initialized by creating a switch as well as NUM_WORKERS number of hosts to act as workers. All of these workers are linked to the switch and nothing else, creating a star topology. Additionally, a multicast group is defined on the switch, in order for the switch to send aggregation results to every worker.
Each worker sends its data to the switch in equal-sized chunks, which are aggregated by the switch by adding them together. To accomplish this, the switch uses stateful memory to store how many chunks it has received, as well as the current aggregation value. Once all workers have sent a chunk for the current aggregation, the result is broadcast to all workers, which save the aggregated value and proceed to the next chunk.

1.1 Ethernet Communication

Over Ethernet, the protocol involves simply setting the EtherType field in the Ethernet header to 0x05FF, and arbitrarily chosen value that isn't in popular use. Packets are crafted, sent, and received using the Scapy library, which broadcasts the packets from the workers and receives broadcasts back from the switch.

1.2 UDP Communication

Over UDP, rather than relying on the EtherType (which is no longer accessible), the SML protocol involves using the port 11037 on top of adhering to the packet format. This time, Scapy is only used to craft packets. Sending and receiving is done via Python UDP sockets, which send to the s0 switch directly (10.0.2.15) and bound to receive from the local broadcast address (10.0.2.255).
As a result of locating the all workers on the same subnet as the switch, this local broadcast address can be used by the switch to send packets to all hosts. Therefore, the switch has no need to store IP or MAC addresses for the workers.
In order for worker sockets to accept broadcasts on this address, the switch takes care to set the destination MAC address to the broadcast address (ff:ff:ff:ff:ff:ff), the destination IP address to the broadcast address, and the source addresses to match the switch's own. Additionally, the udp checksum is recalculated as the packet exits the switch to accomodate the packet's updated values, something that wasn't necessary when only Ethernet headers were used.
In order for the workers to send packets to the switch, they require a MAC address, but are only given an IP address. To alleviate this, the s0 switch is given a MAC address during the control

plane configuration, and each host has an entry added to its ARP table mapping the IP address to the MAC.

2 Basic Implementation

Once the program has begun, each worker generates an array of values to be used in that iteration and then begins broadcasting chunks on the network interface "eth0". Since each worker only has 1 link, the broadcast is only received by the switch. After each chunk is broadcast, the worker waits to receive a packet in the same format, which it saves to memory before continuing to the next chunk.
As far as maintained state goes, workers need to maintain which chunk is currently being sent, while s0 needs to maintain the current aggregation value, as well as how many chunks have contributed to it.
The switch s0 awaits incoming packets of the correct protocol and aggregates the values of these packets. To do this, s0 makes use of a register to save information across incoming packets. The register is (CHUNK_SIZE * 32) + 8 bits large, in order to store all of the incoming integer elements in the chunk as well as an 8-bit value NUM_CHUNKS_RECEIVED indicating the number of chunks received during this aggregation so far. 8 extra bits is more than sufficient as the switch only interacts with 1 chunk per worker during each aggregation, and the maximum value of NUM_WORKERS is 8. Indeed, 8 was chosen over a smaller number due to the BMv2 requirement that header sizes be multiples of 8 bits.
The size of each packet is defined by CHUNK_SIZE, which was chosen to be 63 because the register being used to store data has a maximum width of 2,048 bits, allowing for a maximum of 64 32-bit integers to be stored. Since the switch is also storing other values (e.g. NUM_CHUNKS_RECEIVED in this version), CHUNK_SIZE is reduced to 1 below this maximum to make room without resorting to the creation of additional registers.
When a packet arrives, s0 reads the current value of the register in its entirety into a temporary buffer. Preprocessor directives are used to #define array indices based on the CHUNK_SIZE and other input values, so that these values are only hard-coded once. These array indices are used to access the temporary buffer's bit array as though it were holding multiple discrete values, for instance *reg_val[NUM_CHUNKS_RECEIVED_INDICES]*. In the case that all chunks for the current aggregation have been received, the switch broadcasts the aggregated value as the "vector" field in an SML packet and then writes 0 to the register instead of the chunk count and aggregated values.
Despite being stored as an array, the vector elements received by s0 are aggregated together as though the entire array were a single value consisting of all elements concatenated together. There is no need to manage the vector elements separately, for as long as no individual value overflows (an assumption guaranteed by the assignment), adding the numbers in their concatenated form produces the same result as adding them separately. For demonstration, consider how the following two computations are equivalent to their concatenated version:

$$0x02 + 0x01 = 0x03$$
$$0x04 + 0x52 = 0x56$$

$$0x0204 + 0x0152 = 0x0356$$

This saves the switch the effort of separating out the 63 individual elements and then performing 63 rounds of addition and storage per aggregation. Moreover, this facilitates only reading from the register a single time, as the entire aggregation operation can be performed in a single read, a single addition operation, and a single write.
To adhere to guidelines and requirements for using stateful memory in the switch, @atomic tag is used to enclose operations which may conflict with other workers if the same memory is accessed concurrently. The register is only read from once at the beginning of the @atomic tag, and is

only written to once at the end of the tag. Additionally, logic within the tag is minimized where ever possible, for instance aggregating all values at once rather than separating them out and iterating through them.

3 Fault-Tolerant Aggregation Protocol

Introducing fault-tolerance to the protocol brought the design notably closer to the one discussed in the article by Sapio et al. To begin, the switch now more explicitly uses the "pool" concept by using a register with two different entries for tracking aggregation progress. This allows the switch to re-broadcast the previous aggregation results, in case a worker was unable to receive it, while also proceeding with aggregation for the current round.
To recover from faults, the workers have also been introduced with the capacity to cease waiting for a response to a given transmission or to declare a response invalid. In these cases, the original packet is retransmitted, and the worker continues to wait for its response.

4 Fault-Tolerant Implementation

Packet corruption is defined in this case as any failure for the switch or worker to receive or interpret a packet. This can include incorrect values, formatting, or routing data. This is identified in this implementation by simply setting a timeout value for the socket, such that after 1 second a socket will "give up" waiting to receive a certain response, resend the original packet for that aggregation round, and try again to receive the answer for the current round.
This means that there are two actives pools being worked with at once. To avoid confusing which pool a packet is involved in, the SML headers have been updated to include both the rank of the worker sending a chunk as well as which chunk is being sent. In this way, the switch won't accidentally aggregate a specific chunk multiple times, and workers can identify which chunk a given packet is a response to.
The switch identifies which pool to use for a given chunk aggregation by the parity of the chunk_num. Due to the fact that the number of chunks in any iteration is always even (as guaranteed by the assignment), chunk_num values will always alternate between even and odd, even across different iterations and different runs of the program. Thus, when a packet comes in, its chunk_num is compared to the CHUNK_NUM of the pool with the same parity. If the numbers are equal, the chunk is aggregated in that pool. If the numbers do not match, it must mean that this is the first packet in a new aggregation, as if the current contents of that pool were valid then it would mean the existence of three different active pools (the one referenced in the current packet and the two currently stored in pools). As the previous aggregation must necessarily be stored in the pool of opposite parity, the pool matching the incoming packet's parity is declared old and erased to begin the new aggregation round.
Additionally, to track whether a specific worker has contributed to a specific aggregation pool, a WORK_RECEIVED field linked to the aggregation pool is stored. Each of the 8 bits in the field tracks whether work has been received from a different worker using bitmasks (recall that the maximum value of NUM_WORKERS is 8). This has replaced NUM_CHUNKS_RECEIVED, as that metric became unreliable when it became possible for a given worker to transmit the same chunk multiple times.
Workers do not need to store additional state as compared to the "basic" solution. The main difference in their activity is merely that they may need to remain within a certain chunk's send/receive step for longer, as well as to transmit their rank and the current chunk_num with the SML packet.