COMP 436 Project Report

0 Administrative

Our team consists of the following members:

Pinglan "Paul" Gao (pg22)
Alan Ji (abj3)
Kevin Xu (kx7)

We completed all three milestones for this project. In addition, we completed the extra credit for Milestone 1 (versioned key-value store). Please note that the versioned KVS is not a part of the MS2 or MS3 implementation.

1 Project Structure

This directory (the directory that this project report resides in) serves as the home directory for our project. Relative to this home directory, there are three directories, "MS1/", "MS2/", and "MS3/" that correspond to each of the milestones for this project. Note that the extra credit for MS1 is included directly as part of the MS1 implementation, and is does not reside as a standalone part.

This project report contains all the details regarding design choices, implementation, and the testing mechanisms for our project. It does not include any information on how to run each of the milestones. Such information is contained in the README.md file for each milestone. Although the operation of each of the milestones is rather similar, it is important to read each individual README.md file as there are slight differences in both the usage as well as the way we choose to display output to the user.

At a high level, each of the milestone folders will have the following files:

- A .p4 file for each switch implemented in that milestone
- A receive.py file to receive packets from the switch, and a send.py file for each client implemented in that milestone
- Makefile
- Shell scripts for each automated test for that milestone
- A README.md file with instructions to run each milestone
- topology.json file
- Runtime configuration files
- A TestSuite.sh file that, when invoked, runs all automated tests for that milestone
- A TestUtil.py file that provides support for the TestSuite.sh script

2 Milestone 1

2.1 Topology

The topology for milestone 1 is rather simple. There is a single host, connected to a single switch, which handles all queries from the client.

2.2 Python Send Client

We created two special packet types that is used in all milestones for this project. The first one of interest is KVSQuery. The KVSQuery packet has the following fields:

- protocol – specifies the protocol for the next layer
- key – the key used in the query
- key2 – the second key used in the query (for SELECT and RANGE queries)
- value – the value used in the query (for PUT queries)
- versionNum – the version number to get
- switchID – indicates which switch handled the query (more detail in Section 3)
- pingPong – indicates if it is a PING/PONG packet (more detail in Section 3)
    - 0 for regular packets, 1 for pings, 2 for pongs, 3 for failure bound reached)
- queryType – the type of query
    - 0 for GET, 1 for PUT, 2 for RANGE/SELECT
- padding – padding for the packet, used as a boolean to determine if the packet came from the switch (so as to ignore packets sent by itself)

The second special packet type of interest is Response, which is the header that we will use to construct our header stack. The Response packet has the following fields:

- value – the value returned by the database
- isNull – indicates if the value is null, as null values cannot be sent over the network
- nextType – has value 1 if it is the last header in the header stack, 0 otherwise
- padding – padding for the packet

The packet layers are bound in the following way: Ether/Response/IP/KVSQuery/TCP.

When the user types in a query into the terminal, the send client will perform a series of preliminary checks to make sure that the query is legitimate. When given a bogus query, the client will reject it and exit immediately without attempting to send this request to the switch. Errors that fall in this category include incorrect usage and out-of-range values – these errors, in our view, do not need to involve the switch at all.

Upon being given a large RANGE query, our client will split this RANGE query into smaller queries that are of size 10. Since SELECT queries are a special case of RANGE queries, they are also treated this way (and sent as RANGE queries). The logic of this is handled in the splitRange function within send.py.

2.3 Python Receive Client

The receive client is rather self-explanatory. It simply takes every packet received from the switch, and prints the appropriate information depending on the type of query (for example, printing the header stack for RANGE queries, or printing NULL if the isNull field has been set).

2.4 Switch Implementation

The switch code for Milestone 1 relies on a few key data structures to hold the requisite information. The two key ones to note are, the database itself, implemented as a register of bit<32>'s, and another register of bit<1>'s, called isFilled. The purpose of the latter register is to distinguish between zero values and null values, as P4 does not support null values.

When a packet is received by the switch, the first step is to parse the headers. For the most part, this is not much different than any P4 program, the packet is parsed in the layer order described in section 2.2. The one feature of interest is that while parsing the response layer, the parser will parse through each header in the header stack, by checking the nextType field in the response header, before moving onto the next layer.

In the ingress processing, we use an action table that matches against the query type in order to perform the appropriate action in the database. These operations are rather straightforward – PUT requests simply update the database/null tracker, and read operations get the value from the database. RANGE queries (and SELECT queries) are handled in a special way. Since there is no "loop" feature in P4, we first read from the starting index, add a response header to the header stack, and increment the current key. Then, in the apply section, we recirculate the packet if it is a RANGE query and we have not read all the necessary keys yet. This repeats until all of the requested range has been read and stored in the header stack.

2.5 Multi-version Requests

In order to support multi-version requests, we made a couple of tweaks to the existing design. First, since up to 6 versions (0-5) can be stored, we expanded the size of the database register to 1025 * 6 of int<32>'s. Instead of creating a database for each variable, we simply use index manipulation to store all 6 versions, making the code much less voluminous. Secondly, we make a new register of size 1025, storing the latest current version number for each key in the register. Finally, we allow users to specify which version to read from as a command-line parameter for SELECT, GET, and RANGE requests.

Since we can assume that only 6 versions are stored, we made the choice to reject a PUT request to a particular key if, for that key, there have already been 6 versions stored. As such, we avoid any technicalities involving which version to evict, if a new version needs to be stored. This rejection is done by the switch, as the client has no knowledge of how many versions have been stored.

Other issues stemming from the addition of versioning, such as invalid version numbers, are rejected directly by the client.

3 Milestone 2

3.1 Topology

Figure 1 shows the topology for Milestone 2. All traffic from the sender is first routed to the load balancer, which is Switch 0. Then, the load balancer balances requests based on key between Switch 1 and Switch 2, which then sends their responses to the receiver. The load balancer also sends all requests to Switch 3, but Switch 3 does not respond unless there is a failure in Switch 1 or Switch 2. Note that we are still using a single-host implementation, but the diagram separates the sender and receiver for clarity.

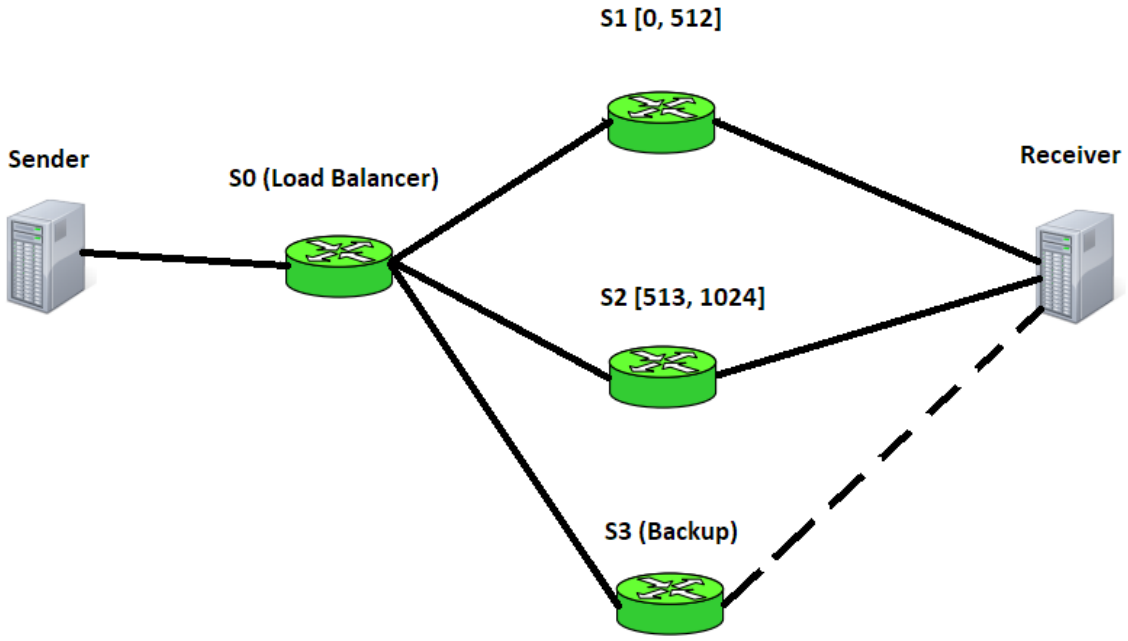

Figure 1: MS2 Network Topology

3.2 Client Implementation

The send and receive clients are, for the most part, identical to those described in Milestone 1. The only notable difference is that the receiver will check for two additional packet types: whether the packet is a ping/pong packet, and whether the packet is an error packet (notifies the receiver that a switch has failed).

3.3 Load Balancing and PING/PONG pairs

In this milestone, each switch is implemented in its own file. The process starts with packets being sent to S0, the load balancer. Based on the key in the request, the load balancer will set the egress port to the appropriate switch that should be handling that key.

The load balancer makes use of three registers to keep track of important information. This first of these registers keeps track of two counters that count the number of requests handled so far. Once the first counter reaches 10 requests, the switch will issue a PING packet to check the health of S1 and S2. Then, this counter will be reset. Likewise, once the second counter reaches 15 requests, the switch will check the difference between the number of PINGs sent and the number of PONGs received. The second and third registers keep counts of the number of PINGs sent and the number of PONGs received for each switch, in order to support the aforementioned failure checking.

The load balancer switch makes use of the clone function in P4 to issue these PING packets to the other switches. Our runtime configuration defines that when cloning packets, we will send the PING packets to all other switches in the network. But since the clone function sends packets exactly how they are received, we need a way to distinguish PING packets from normal packets. We handled this problem by setting the pingPong field in KVSQuery to 1 (to indicate it is a PING) if it is a cloned packet in the egress processing. Then, when switch 1 or 2 receives the cloned packet, they will proceed to set this field to 2, and send it back to the load balancer to update its registers.

There is another special case for using clone – to send PUT requests to the standby switch. To handle this, we simply defined a new clone session ID in the runtime for this specific scenario, and if the request is a PUT, then we simply use clone with this session ID (to distinguish from PINGs) to send the request to S3, the backup switch.

The switch must also have a way to distinguish forward traffic (from the host) and returning traffic (PONGS from switches). We simply check the ingress port of the packet to handle this. If the packet is coming from the host, then we simply perform load balancing based on key, update the PING counts, and clone the packet every 10^{th} request to send out PINGs as described earlier. If the packet is coming from switches, then we update the PONG counts, and check the PING/PONG difference if we are currently on the 15^{th} request since our last check. If the difference is too high (set to be >10), we know that a failure has occurred, and we send a “failure” packet over the network.

Other switches in the network are rather straightforward. Switch 1 and Switch 2 are the same as the switch in Milestone 1, except the key-value store is now half the size. The switches just use index calculations to handle this appropriately. Switch 3 is implemented exactly the same as the switch in MS1 as well, with the only exception being that it does not forward anything to the load balancer, and consequentially the client. We did not implement the extra credit for this milestone, in which Switch 3 would take over if any of the other backend switches have failed.

3.4 Other Design Decisions

In this milestone, we decide to forward the PONG packets to the client, as well as print out information denoting which switch handled a particular request. This would not make sense in a real system – the user does not care about PING/PONG packets; such health

checking is only the business of switches in the network. Likewise, the user does not care which switch handled their request; they only want to know the result of their query. However, we decided to do this in order to effectively debug and test the correctness of our implementation. Since logging/interpreting log files in P4 is rather cumbersome, we wanted to handle this in Python instead, which required that this information be sent to the client. We limited the printing of this "extraneous" information to MS2 only.

4 Milestone 3

4.1 Topology

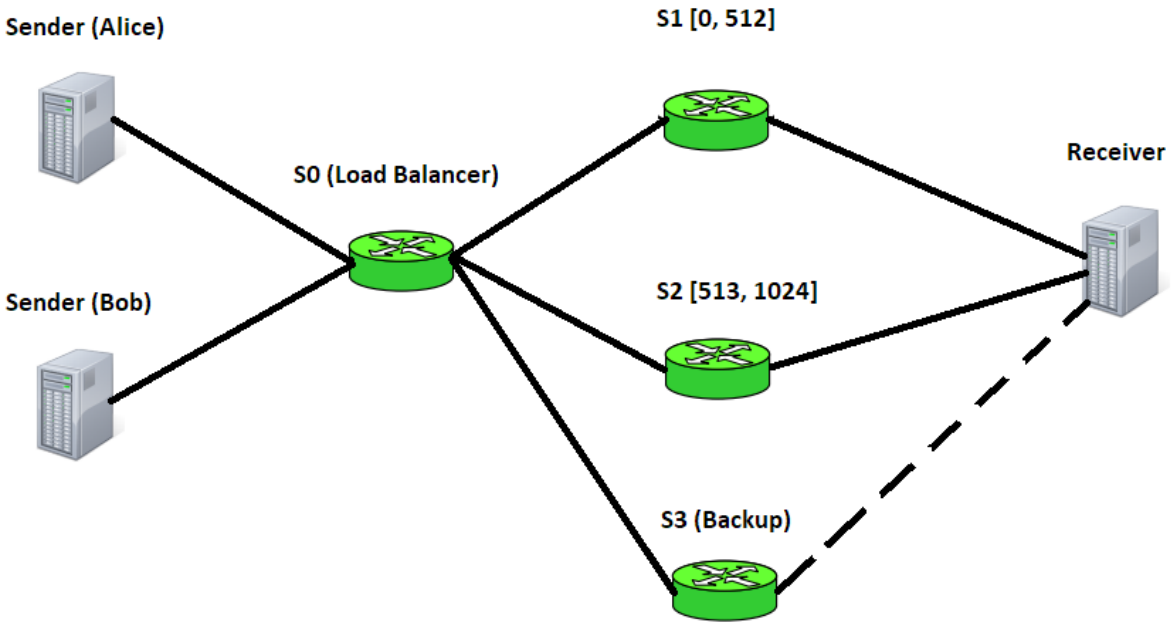

Figure 2: Topology for Milestone 3

Figure 2 shows the network topology for Milestone 3. It is the same as Milestone 2, except Alice and Bob are now modeled as different clients. In our implementation, we decided to have both Alice and Bob utilize the same host.

4.2 Client Implementation

The send client in this milestone has a couple of added fields within KVSQuery:

- clientID- integer storing the client ID of the sender (0 for Alice, 1 for Bob)
- upperBound- integer indicating the highest-most key in any RANGE request
- readWriteAccess – integer storing the type of access error (0 if all access allowed, 1 if denied read access, or 2 if denied write access)
- rateLimitReached – used as a boolean to denote whether the user's rate limit has been reached

The file sendA.py models Alice's client, and the file sendB.py models Bob's client. They are exactly the same, except for the fact that they will send packets with different client ID's.

The receiver will also use this client ID field, as well as the aforementioned fields denoting read/write access and rate limits, to print out an informative message specifying that a particular client does not have read/write access to a certain key, or has reached their rate limit.

In our model, we have one unified receiver which will print out the results of both Alice and Bob's queries. While this may not make much sense in a real-world system (Alice and Bob don't want to see the results of each other's queries), since we are using a single-host implementation, having a single receiver is functionally equivalent to having separate receivers for Alice and Bob. They are both receiving each other's packets anyways, and all the python script would be doing is filtering them out based on client ID. So in order to achieve consistency with the other milestones, simulate the perspective of the network owner (not necessarily the clients themselves), and ensure unification and ease-of-use in testing, we just have a single receiver show all the requests that have been made in the system.

4.3 Access Control Lists and Rate Limits

The switches are implemented in the same way as Milestone 2, but of course, now the load balancer is also responsible for handling ACLs and Rate Limits. A count of the number of requests Alice and Bob have sent is now also stored in a register. We chose not to store the access bounds for Alice and Bob, instead we just directly compare the queried key(s) against their access limits.

In order to delineate between Alice and Bob's requests, the load balancer has an action table that matches against the client ID of the incoming packet from the host. The switch then calls the appropriate function, depending on whence the packet came, that checks that the specified key is within the allowed range of read/write accesses for the client. If it is not, the readWriteAccess field is modified to indicate this, as mentioned above.

Using the stored information about how many requests have been sent by each user so far, the switch checks to see if the rate limit has been reached. If so, the rateLimitReached field is marked accordingly. Otherwise, the number of requests sent so far is incremented by 1, irrespective of whether the request is fulfilled.

In both the case of denied access as well as the case of exceeding the rate limit, the packet is returned to the client by setting the egress port appropriately. If neither are the case, the packet is sent to one of the other switches (also by setting the egress port) in an identical fashion to Milestone 2.

4.4 Other Design Considerations

We made two important design decisions in this milestone. First, if part of the query is outside the user's permissible access range (for RANGE and SELECT queries), we choose to reject the request in its entirety. We believe that a request is either valid or invalid – and if part of the request is invalid, then the entire request is invalid, and there is no reason to give a "partial response" to the request. Furthermore, in some systems, the owner of the database may not want to disclose to the users which range of keys they have access to, as it could be a security risk. In such cases, we do not want the user to be able to efficiently find out this information through large RANGE or SELECT queries. Furthermore, we've used the upperBound field to indicate, for RANGE an SELECT queries, a read access denial if the client does not have access to read the upperBound as a key. Thus, in these cases, we simply return a message to the user indicating that they do not have access to one or more keys in the specified range.

Additionally, since large RANGE and SELECT queries are split into a number of smaller requests, we decide to count each of these smaller requests against the user's rate limit. We believe that this makes sense because even though the user can send the same request in many different ways, the load on the system/network traffic is effectively equivalent whether the user sends one large RANGE request or several smaller ones. Therefore, we think that it makes sense to keep track of the rate limit from the switch's perspective (the number of packets received by the load balancer).

5 Testing

5.1 Overview

We conducted our testing both through manually trying various requests as well as a series of automated scripts. The automated scripts, and the tests that comprise them, are included in the submission in each milestone folder.

5.2 Testing Apparatus

In each of the milestone folders, you will see one or more shell scripts that have a series of requests used as test inputs. For example, MS1GetAndPutTest.sh tests some basic functionalities of GET and PUT requests. These tests also have comments that explain what we are trying to exercise. To make use of the tests, simply use one terminal window to run the shell script of the test you want to run, and run './receive.py' on the other terminal. You can then verify that the results returned by the key-value store are indeed what you expected.

Each milestone folder also contains a TestSuite.sh file as well as a MS[milestone number]ExpectedOutput.out file. This gives you the ability to run every test in that folder and automatically check the output against the expected output file. It will generate a MS[milestone number]TestOutput.out file that is safe to delete after completion of the test. The python script TestUtil.py is automatically invoked by the test suite as it helps provide

support such as stripping extraneous lines and comparing outputs. To run the TestSuite, simply run ‘./TestSuite.sh’ in one of the terminals created from Mininet. As seen in Figure 3, a successful run of the test suite will yield a message stating, “All tests passed.”

Figure 3: Successful run of the test suite

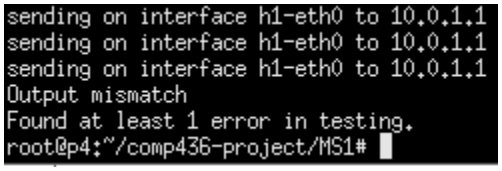

Figure 4: Test suite run showing errors

Unsuccessful runs of the test suite will show output similar to Figure 4. Generally, an “output mismatch” means that the number of lines is not the same between the actual and expected output. Getting an error without an “output mismatch” usually means that the result of a query does not match what is expected.

Note that tests may be failing if you do not restart Mininet after every test. This behavior is explained further in Section 5.3.

5.3 Testing Design

Because values stored in the database persist until Mininet is terminated, the tests will not exhibit independent and deterministic behavior if Mininet is left open across tests. For example, a test that stores values into the database could affect a later test which attempts to get values. Depending on the order in which tests are run, the results may vary.

As we do not implement a way to easily purge the database, the correct way to run tests is to launch a fresh instance of Mininet for each test that you wish to run. Otherwise, the result of the tests may not be what you might expect.

Due to this type of behavior, we chose not to include expected output for each individual test. The file containing expected output shows what is expected for running the entire test suite. It relies on the tests running in a predetermined order, so therefore any modifications to the database done by a previous test is factored into the results. This way, users can easily verify the correctness of the entire build in one command after starting Mininet.

5.4 Tested Functionalities

Milestone 1 contains four automated tests:

- MS1GetAndPutTest.sh tests ordinary GET and PUT requests.
- MS1RangeAndSelectTest.sh tests ordinary RANGE and SELECT requests.
- MS1VersionTest.sh tests that the integrity of the database is preserved across multiple versions, and that there are no logical errors.
- MS1ErrorTest.sh tests for edge cases in MS1. This file tests for issues such as: invalid usage, out-of-range requests, trying to put too many times, version numbers that are not stored yet, correct NULL vs 0 determination, etc.

Milestone 2 contains a single test:

- MS2LoadBalanceTest.sh tests the load balancing feature to make sure that the appropriate switch is handling the requests. This test also implicitly exercises the PING/PONG feature as well. Since MS2 automatically prints out PONG packets that have been redirected to the client, we also use this test to ascertain that the correct number of PONG packets are received (that it is indeed 1 every 10 requests). It also makes sure that the 10^{th} request itself is not affected by the PING/PONG mechanic.

Milestone 3 contains two tests:

- MS3ReadWriteAccessTest.sh tests that Alice and Bob are able to read/write within their allowed ranges, and are unable to do so outside of their allowed ranges. It also makes sure that when Alice and Bob attempt to operate on the same part of the database (such as Bob overwriting Alice's previously stored value), there is no unintended behavior.
- MS3RateLimitTest.sh tests that Alice and Bob's requests are not fulfilled after their rate limit has been reached. It also makes sure that illegal requests made by Alice and Bob are counting against their rate limit.