COMP 536 Course Project Report

1. Project Team

- Yu-Hsi Teng yt55@rice.edu
- Jiaqi He jh166@rice.edu

Notes: Our team has only two members due to the unexpected withdrawal of our third member from the course. However, we have successfully obtained approval of a reduced group size from Professor Chen.

2. Project Overview

There are three primary directories for this project: `DB_MS1/`, `DB_MS2/`, and `DB_MS3/`, each corresponding to a specific milestone as detailed in the project description. Every directory

contains the code, configurations, and scripts required for grading each milestone independently, assuming the directory structure specified in the project description.

Each `DB_MS*/` directory shares a similar structure as follows:

- `README.md`
- `Makefile`
- `DB_MS*_s*.p4`: P4 program for switch s*
- `send.py`: A scapy-based python script that supports sending PUT, GET, RANGE, and SELECT requests to key/value store
- `receive.py`: A scapy-based python script that supports receiving PUT, GET, RANGE, and SELECT requests from key/value store
- `query_h.py`: A scapy-based python script that defines custom header formats, specifies how these packets are encapsulated within other protocols (e.g., Ethernet frames, IP packets, and TCP segments), and handles packet field parsing and processing
- `s*-runtime.json`: Configuration file for switch s* to specify how incoming packets should be processed and forwarded
- `topology.json`: Topology configuration file to specify network layout and connections
- `run_tests.sh`: A test runner that executes all tests `test*.sh` in current directory
- `test*.sh`: Individual test script to evaluate a collection of operations
- `expected/`: A directory that contains the expected output files, `test*.out`, for each respective test
- `output/`: A directory that stores the test outputs generated upon the completion of each test run

3. Milestone 1: In-network key/value store in a single P4 programmable switch

3.1. Topology

- One switch **s1** connected with one host **h1**

3.2. Custom Header Types

We have created two custom headers, `query_t` and `multiVal_t`, to encode the packet types for key/value store queries and support different query operations. These headers correspond to the `Query` and `MultiVal` packets generated in the python script `query_h.py`

- `query_t:` Storing information from client's request
  - Format

```
header query_t {
   bit<2>   queryType; // 4 types: PUT/GET/RANGE/SELECT
   bit<6>   padding;   // BMv2's bit constraints
   bit<32>  key1;      // key
   bit<32>  key2;      // second key for RANGE/SELECT
   bit<32>  value;     // value for PUT query
   bit<32>  version;   // version number [0, 5]
   bit<32>  count;     // query counter for RANGE/SELECT
   bit<16>  protocol;  // next layer header/protocol
}
```

  - We use EtherType `0x0801` to identify `Query` packets

- `multiVal_t:` Storing information retrieved from the key/value store based on client's query. Also useful when emulating loop behavior if the query is RANGE or SELECT
  - Format

```
header multiVal_t {
   bit<32>  value;     // value retrieved from database
   bit<1>   has_val;   // 0: no value
   bit<1>   has_next;  // 0: this is the last multiVal header,
                       // used for emulating loop behavior
   bit<6>   padding;
}
```

  - We use EtherType `0x0802` to identify `MultiVal` packets
- Packet layer bindings: Ethernet / Query / MultiVal / IPv4 / TCP

3.3. Key/Value Store Implementation

In our P4 program, we use three registers, `db_value`, `lastest_version`, and `key_exist` that are persistent through all packets to implement a key/value store database that supports versioning. Details of these registers are as follows:

```
// stores database values with indices as keys
// supports key range: [0, 1024]
// supports version number range: [0, 5]
// For each entry, we allocate contiguous space for its 6 versions in the register
register<bit<32>>(1025 * 6) db_value;

// stores the latest version number of a database entry with index as key
register<bit<32>>(1025) lastest_version;

// stores the flag indicating whether an entry exist with index as key
// 0: not exist, 1: exist
register<bit<1>>(1025) key_exist;
```

- For example, to retrieve a value from our database with a specific key (`k`) and version number (`n`) and store it in a local variable (`res`), we can use the following syntax: `db_value.read(res, k * 6 + n);`

3.4. Multi-Version Request Implementation

To support PUT, GET, RANGE, and SELECT requests for our key/value store with versioning, we define the following actions in our ingress processing logic:

- `put`: Handles PUT queries
  - If `query.key1` does not exist in database, store `query.value` at the corresponding location within `db_value`
  - If `query.key1` does exist, store `query.value` as a new version in the database, using the version number stored in `lastest_version` as an offset to locate the correct position in `db_value`
  - We will reject PUT request for an entry (`query.key1, query.value`) if there are already 6 versions of values for `query.key1` stored in the database
- `get`: Handles GET queries
  - If `query.key1` exist and `query.version` is less than or equal to the version number stored in `lastest_version`, store `query.value` in `multiVal.value`
- `range_get`: Handles RANGE and SELECT queries
  - Emulate loop behavior by shifting the header stack by one position, invalidating the first element
    - Then, in the egress logic, recirculate the packet and increment the key `query.key1`, if the query type is RANGE or SELECT and not all keys satisfying the RANGE or SELECT condition have been read
  - Continue retrieving values from the database using similar logic in the GET operation until `query.key1` is equal to `query.key2`

3.5. Additional Design Choices

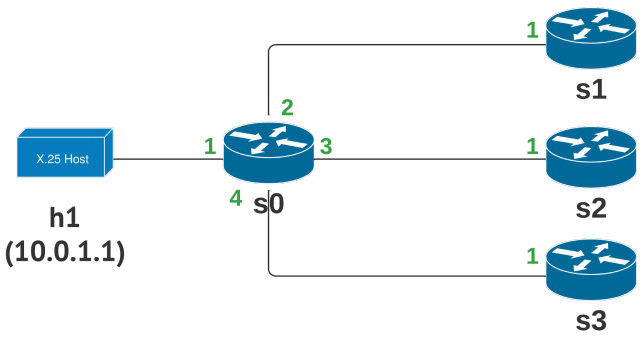
- Handling large-size RANGE/SELECT queries
  - To prevent the response of RANGE/SELECT queries from becoming too large to fit into a single packet, we split large RANGE/SELECT queries into several smaller queries with a size limit of 10 in our python script `send.py`. This ensures that the response for any RANGE/SELECT query will fit into one packet
- Predicate format of SELECT query
  - For the operands in the predicates of SELECT queries, client should use the following strings on the command line to match the operands specified in the project description:
    - `>` `gt`
    - `>=` `gteq`
    - `<` `ls`
    - `<=` `lseq`

        - `==` `eq`
- Retrieving a value or version that does not exist in the database
    - Prints out `NULL` on the receiver side
- Retrieving a value without providing version number
    - The default-to-highest-version feature is not implemented. If user does not supply version number, prints out an error message on sender side informing user of the correct usage of the operation

4. Milestone 2: Key/value store partitions, load balancing, fault tolerance

4.1. Topology

- One load balancing switch **s0**, connected with one host **h1**
- Two key/value store partition switches **s1** and **s2**
- One standby switch **s3**



4.2. Load Balancing Overview

We have implemented a frontend switch, s0, that performs the load balancing across two switches, s1 and s2, which maintains two key/value store partitions, based on the following key ranges:

- **s1**: keys in [0, 512]
- **s2**: keys in (512, 1024]

We have also implemented a standby switch, s3, which is connected to s0 and holds the values for the entire key range [0, 1024]. However, s3 does not respond to any client requests directly.

4.3. Custom Header Types

To implement the PING/PONG protocol for s0 to monitor s1 and s2 and check their status online, we have created a custom header `pingPong_t`, which corresponds to the `PingPong` packet generated by `query_h.py`. We have also modified header `query_t` as follows, with additional fields highlighted and commented:

- `pingPong_t`

```
header pingPong_t {
   bit<2>   type; // 0: normal packet, 1: PING, 2: PONG
   bit<6>   padding;
}
```

  - We use EtherType `0x0803` to identify `PingPong` packets

- `query_t`

```
header query_t {
   bit<2>   queryType;
   bit<1>   isFeedback; // 1: is feedback from normal request
   bit<1>   s1_dead;    // 0: s1 is alive, 1: s1 is dead
   bit<1>   s2_dead;    // 0: s2 is alive, 1: s2 is dead
   bit<3>   padding;
   bit<32>  responder;  // who returns the packet: s1,s2,s3
   bit<32>  key1;
   bit<32>  key2;
   bit<32>  value;
   bit<32>  version;
   bit<32>  count;
   bit<16>  protocol;
}
```

- Packet layer bindings: Ethernet / PingPong / Query / MultiVal / IPv4 / TCP

4.4. Failure Detection Implementation (PING/PONG)

We use a register `request_tracker` to store information required to be persistent across all packets to handle PING/PONG correctly:

- `request_tracker`

```
// 0: number of total requests
// 1: number of ping to s1
// 2: number of ping to s2
// 3: number of pong from s1
// 4: number of pong from s2
register<bit<32>>(5) request_tracker;
```

The ingress processing logic in switch s0's P4 program is outlined as follows:

- If this is a response packet from a normal request, send back to h1 (port 1)
- if this is a PONG packet from s1 or s2, increment PONG counters and forward to port 1
- If this is a standard request packet,
  - If this is a multiple-of-ten-th request (10th, 20th, …),
    - Clone the packet
    - Convert it into a PING packet
    - Send to both s1 and s2
    - Then, proceed with processing the request as usual

    - If this is a multiple-of-fifteen-th request (15th, 30th, …),
        - [**Failure Detection**] Verify if s1 and s2 are still operational by examining whether the difference between the counts of PING and PONG is less than 5. If not, declare switch failure by setting the flag `query.s1_dead` and/or `query.s2_dead` to 1
        - Then, proceed with processing the request as usual
    - [**Load Balancing**] Process the request as follows:
        - If `query.key1` is in [0, 512], send request to s1 (port 2)
        - If `query.key1` is in (512, 1024], send to s2 (port 3)
        - Additionally, if this is a PUT request, clone and send the request packet to standby switch s3

s1, s2, and s3 share similar ingress processing logic as in **Milestone 1**, but with the following differences:

- Set `query.responder` to 1, 2, or 3, indicating packet is a response from s1, s2, or s3
- For s1 and s2
    - If this is a normal request packet, convert it into a response packet by setting `query.isFeedback` flag to 1
    - If this is a PING packet, convert it into a PONG packet by setting `pingPong.type` to 2
- For s3
    - Convert this request packet to a response packet by setting `query.isFeedback` flag to 1

4.5. Additional Design Choices

- Failure bound for PING/PONG protocol
    - Considering the inherent latency between the PING and PONG requests, we define the FAILURE_BOUND as 5. Exceeding this threshold indicates the failure of switch
- Handling modulo in P4
    - Since P4 does not support the modulo operation we need for determining the current sequence of packet, we use the following bit masking and shifting operations:
        - Convert to PING packet for every 10-th request packet:
            - Need modulo: `num_request % 10 == 9`
            - P4 alternative: `num_request & 0b1111) == 0b1001`
        - Failure bound check for every 15-th request packet
            - Need modulo: `num_request % 15 == 14`
            - P4 alternative:
```
(num_request & 0b1111) == 0b1110 &&
((num_request >> 4) & 0b111) == 0b111
```
- Handling database failure on client side
    - When s1 or s2 fails, prints out an error message on the receiver side indicating which switch fails, and exits gracefully

5. Milestone 3: Storage with access control lists

5.1. Topology

- Same as in **Milestone 2**

5.2. Custom Header Types

We have created a new header `access_t` to implement rules for Access Control List (ACL), which corresponds to `Access` packets generated by `query_h.py`:

- `access_t`

```
header access_t {
   bit<32>  clientID;        // 0: Alice, 1: Bob
   bit<32>  key1_wholeQuery; // key1 before split
   bit<32>  key2_wholeQuery; // key2 before split
   bit<1>   no_read_access;  // 0: can read,  1: can't read
   bit<1>   no_write_access; // 0: can write, 1: can't write
   bit<6>   padding;
}
```

- Packet layer bindings: Ethernet / PingPong / Access / Query / MultiVal / IPv4 / TCP

5.3. Access Control Lists Implementation

The ingress logic for s0 is similar to that in **Milestone 2**, with the following new actions added to support ACL rules:

- `set_access_Alice`: Manages access control for Alice
  - For PUT requests, if `query.key1` is greater than 512, revoking write access by setting `access.no_write_access` flag to 1
- `set_access_Bob`: Manages access control for Bob
  - For PUT requests, if `query.key1` is greater than 256, revoke write access by setting `access.no_write_access` flag to 1
  - For GET requests, if `query.key1` is greater than 256, revoke read access by setting `access.no_read_access` flag to 1
  - For RANGE/SELECT requests, if `query.key2_wholeQuery` is greater than 256, revoke read access by setting `access.no_read_access` flag to 1

After processing the access control logic, if either the `no_read_access` or `no_write_access` flag is set, return the packet back to host h1 (port 1). Otherwise, proceed with the request processing in a manner similar to what described in **Milestone 2**.

5.4. Additional Design Choices

- Handling large-size RANGE/SELECT queries

    - As in **Milestone 1**, to avoid oversized RANGE/SELECT query responses, we divide large RANGE/SELECT queries into smaller ones with a maximum size of 10 using our `send.py` python script
- Handling access denial on client side
    - If a client (identified by `clientID`) is determined to lack read or write access to a specific key range due to ACL rules, prints out an error message on the receiver side and waits for the next query

6. Testing

To evaluate our implementation for each milestone, we have developed a series of individual test scripts, `test*.sh`, designed to test various database operations and behaviors. Additionally, we have implemented a test runner, `run_tests.sh`, which collectively executes all test scripts within each milestone directory, and provides a summary upon completion.

Tests for Milestone 1

- `test1.sh`: Tests basic PUT and GET requests without versioning
    - Issue multiple PUT requests to store key-value pairs with keys in the range [0, 1024] in the database
        - **Expected outcome**: Correct storage of values
    - Issue multiple GET requests to retrieve the stored values by providing keys, using version number 0 for requests
        - **Expected outcome**: Correct retrieval of values
- `test2.sh`: Examines additional versioned PUT and GET requests, including edge cases
    - Issue more PUT requests to the key/value store database
        - **Expected outcome**: Correct storage of values
    - Issue a PUT request to store a key-value pair that already has 6 versions in the database
        - **Expected outcome**: Retrieve only the first 6 versions. No overwrites
    - Issue GET requests for out-of-bound keys
        - **Expected outcome**: Display error message on the sender side
    - Issue GET requests for non-existent keys or versions in the database
        - **Expected outcome**: Retrieve `NULL` as output
- `test3.sh`: Tests versioned RANGE requests, and versioned SELECT requests with different predicates
    - Issue RANGE requests with different version numbers
        - **Expected outcome**: Large requests are split into smaller queries with a size limit of 10, and a message is displayed on the client side. Values are retrieved correctly
    - Issue SELECT requests with different version numbers and predicates

        - **Expected outcome**: Query splitting occurs, and values are retrieved correctly

Tests for Milestone 2

- `test1.sh`: Tests basic PUT and GET requests without versioning, along with load balancing and PING/PONG-based failure detection mechanisms
    - Issue multiple PUT and GET requests
        - **Expected outcome**: Proper storage of values in both the standby switch (s3) and the database switch (s1 or s2). Accurate retrieval of values from the correct switch. PONG messages are displayed (note that due to latency, PONG may not occur precisely at the 10th request)
- `test2.sh`: Tests versioned PUT and GET requests with edge cases, as well as load balancing and PING/PONG-based failure detection mechanisms
    - Issue additional PUT and GET requests, as in Tests for Milestone 1
        - **Expected outcome**: Edge cases such as out-of-bound keys and non-existent keys or versions are managed correctly, as previously described. Values are stored in and retrieved from the appropriate switches. PONG messages are displayed
- `test3.sh`: Tests versioned RANGE requests, versioned SELECT requests with diverse predicates, load balancing, and PING/PONG-based failure detection mechanisms
    - Issue RANGE request, and SELECT request with various predicates
        - **Expected outcome**: Query splitting occurs, and values are accurately retrieved from the correct switch. PONG messages are displayed

Tests for Milestone 3

- `test1.sh`: Tests versioned PUT and GET requests with ACL, focusing on clients with different levels of key range read/write access
    - Sending requests as Alice
        - Issue PUT requests within [0, 512]
            - **Expected outcome**: Proper storage of values in both the standby switch (s3) and the database switch (s1 or s2)
        - Issue PUT requests outside [0, 512]
            - **Expected outcome**: An error message is displayed, indicating that Alice does not have write access to the provided keys
        - Issue GET requests for the entire key range [0, 1024]
            - **Expected outcome**: Values are accurately retrieved from the correct switch
    - Sending requests as Bob
        - Issue PUT requests within [0, 256]
            - **Expected outcome**: Proper storage of values to s3 and s1/s2
        - Issue PUT requests outside [0, 256]
            - **Expected outcome**: An error message is displayed, indicating that Bob does not have write access to the provided keys

        - Issue GET requests within [0, 256]
            - **Expected outcome**: Proper retrieval of values from correct switch
        - Issue GET requests outside [0, 256]
            - **Expected outcome**: An error message is displayed, indicating that Bob does not have read access to the provided keys
    - Note that PONG messages are also expected to be displayed appropriately
- `test2.sh`: Tests versioned RANGE request with ACL, and versioned SELECT requests with ACL and various predicates, focusing on clients with different levels of key range read/write access
    - Issue multiple RANGE and SELECT requests as Alice or Bob
        - **Expected outcome**: If the current client has access, accurately retrieve values from the appropriate switch. If not, display an error message indicating that the current client does not have read access to the provided keys. PONG messages are displayed