Secure and Cloud Computing Course project

- **Estimated duration:** September 17 -- November 17 (2 months).

- **Goal:** You will be building a secure, in-network, load balanced key/value store using a set of P4-based programmable switches.
    - Three milestones, required for all students.
    - MS1: with 5% "extra credit"
    - MS2: with 5% "extra credit"
    - Hardware component: 5% "extra credit"
    - Extra credits are upper bounded by 5%.

- **Relevant background:** Completing this project will require in-depth mastery of all three sections of this course (cloud networking, cloud systems, and cloud security). In particular, the following sessions are particularly important:
    - Sessions 5--6: Programmable switches and P4
    - Session 7: Load balancing strategies
    - Session 13: Cloud storage, key/value stores
    - Sessions 18-19: Security, access control
    - Programming environment: P4 VM running Mininet bmv2 model.

- **Notes on deploying different programs to different switches:**
    - This project requires to run different P4 programs on several switches. Mininet by default only supports one single program across all switches. Therefore, the following setup is needed:
    - Assign different P4 programs to different switches.
        - Modify the Makefile under tutorials/utils/Makefile. Change the line `compiled_json := $(BUILD_DIR)/$(outfile)` to `compiled_json := $(addprefix $(BUILD_DIR)/,$(outfile))`. This assumes the following directory structure:
```
+----utils
          +----Makefile (change this file)
+----basic
+----courseproject
          +----Makefile (no need to change)
          +----yourprogram.p4
          +----s*-runtime.json
```
        - Modify the first few lines in s*-runtime.json. For example, if you want to run 'frontend.p4' for switch 2, then modify 's2-runtime.json':
```
"p4info": "build/frontend.p4.p4info.txt",
"bmv2_json": "build/frontend.json"
```
        - You can use the above to do different setups for different switches.

- **Project description and milestones:**
    - Cloud switches are programmable in P4. This allows a range of “in-network” cloud functions to be deployed inside the switches. Such functions enjoy higher performance, due to the hardware speeds of switch programs; they also have lower latency than regular, end host-based systems, because the response time to the clients is shorter. It has been demonstrated that, for instance, consensus algorithms, key/value stores, DDoS defense, and many other useful cloud functions can be offloaded to in-network P4 implementations.
    - This course project will particularly focus on key/value stores and algorithms that operate on key/value stores. A key/value store (KVS) is a database-like system, but its operations are modeled after matching on a key and returning a value or a set of values.
        - A GET(k) request will look up the KVS, and return a value v if a key/value pair (k,v) can be found in the database. Otherwise, it returns a NULL value.
        - A PUT(k,v) request will insert the key/value pair (k,v) to the database if the key k does not exist; it will update the value v if the key k already exists.
        - A RANGE(k1,k2) is a special GET request. It specifies that keys in the range (k1,k2) should be looked up and returned to the client. This can be implemented in a set of GET(k) requests, or a “range-based” lookup for efficiency. We’ll do the latter.
        - A multi-version key/value store will maintain multiple versions of the same key/value pair for data persistence. For instance, a PUT(k,v), if it hits an existing key, will create a new version instead of deleting the old value. Concretely, a key/value pair will be represented by (k,v,t), where t is the timestamp -- either the wallclock time, or more commonly a logical counter that increments for every new insertion.
        - A SELECT(predicate(k)) is another special GET request, which runs a function / predicate over k, and only selects key/value pairs that match the predicate. For instance, a predicate could be “k>800”; in this case only key/value pairs with keys greater than 800 will be returned. In other words, RANGE can be viewed as a specific case of SELECT. Since there are many types of SELECT predicates, we will limit them to predicates of the form “k OP val” where OP is an operand in “>, >=, <, <=, ==” and ‘val’ is an integer; every SELECT request will only contain one predicate. (Hint: The “calculator” tutorial in the P4 VM provides useful guidance on this behavior.)
        - **Milestone 1:** Implement an in-network key/value store in a single P4 programmable switch. (Suggested timeline: 3 weeks, September 17--October 8)
        - The server/switch should support the key range [0--1024], with all keys being integers, and the values could be any int32 value. The initial state of the key/value store can be initialized statically using your

        own synthetic database. Your control plane script will pre-configure the database to the switch when the system boots.

    - It should support GET, PUT, RANGE, and SELECT requests. In order to do this, your P4 program needs to
        - 1. recognize a special header field that encodes which type of query the packet represents; such a header field could appear anywhere in the packet, e.g., between Ethernet and IP fields, using a bit<2> variable.
        - 2. use match/action tables to distinguish the header types and trigger different types of processing depending on the request types.
        - 3. implement algorithms for GET, PUT, RANGE, and SELECT on your database.
        - 4. in the deparser, serialize the response packets and send them back to the requesting client.
        - 5. you may find it necessary to use the “recirculate” primitive in P4 to emulate loop behaviors in the switch. This is a heavyweight primitive in real hardware switches so should be invoked sparingly.
    - In terms of the clients, you can reuse the Scapy-based Python scripts in the homework set. They’ll need to be modified to send PUT/GET/RANGE/SELECT requests in a format that is consistent with your P4 program.
    - Multi-version requests; 5% extra credit.
        - 6. the version value could be stored with the key/value pair, or it could be stored in a separate match/action table. you can determine which implementation is more suitable for your system.
        - 7. the GET, RANGE, and SELECT requests will be extended with a version number, which specifies the correct version to fetch from the key/value store. The match will have to take into account the requested version number, too. The PUT request does not have a version number; rather, by default it will update the key/value pair with the highest version number for that key. In most key/value stores, GET/RANGE/SELECT requests, if they do not have version numbers, will by default process the highest versions---however, for simplicity this “default-to-highest-version” feature is not required in the project.
        - 8. in a real-world key/value store, the version numbers are not restricted by any upperbound. However, for this project, you can assume that the version number is limited to [0-5].

- **Milestone 2:** Key/value store partitions, load balancing, fault tolerance.
    - In a real-world key/value store, there will be more than one servers for load balancing and fault tolerance. In our setting, this means that

multiple switches will work together as the key/value service. The key range is partitioned across switches, and the requests will be sent to different partitions depending on the key range for load balancing. (Suggested timeline: 3 weeks, October 9 -- 29)

- Implement a two-switch partitioned system and load balance across two switches.
    - You can borrow your HW2 setup, with two switches that maintain the key/value store, and a frontend switch that performs load balancing. As a simple strategy, keys in [0, 512] will reside in switch 1, and those in (512, 1024] will reside in switch 2. Switch 0 will perform load balancing as the frontend based on the key range.
    - For simplicity, you can assume that RANGE requests will not straddle across partition boundaries. They either ask for keys in [0, 512] or (512, 1024]. Ranges like [500, 600] do not need to be supported.
- Implement a “standby” switch, switch 3, which is connected to switch 0 (the frontend load balancer). It stores the entire key range, and gets a copy of all PUT requests (for data consistency). However, it does not respond to any user requests, as it is only a “hot standby”. It only takes over if switch 1 or switch 2 fails.
    - Modify the setup to include this standby switch. The changes will mostly be to the frontend load balancer, which will integrate this third backend partition.
    - The load balancer should be integrated with failure detection logic that monitors switches 1--2 and checks their health online. This can be implemented by a simple protocol “PING/PONG”, where the frontend switch issues a PING periodically to the backend; the backend switch will respond with a PONG. Both are special P4 packets that can be recognized with their respective header values.
    - A real hardware switch has an in-built packet generation mechanism that would allow the switches to generate packets periodically “out of thin air”; these can be the PING/PONG packets. However, in the Mininet setup this is limited. So you can leverage client requests for this “periodicity” behavior. For instance, for every 10-th client request the frontend switch receives, it modifies the request into a PING packet to both backends (in addition to processing the request as it is). Backend switches modify the PING packet into a PONG packet and send it back.
    - If PING/PONG pairs go smoothly, no failure has happened, and the frontend load balancer will continue to spread load evenly between switches 1--2. This check can be done by, say, keeping track of the number of PINGs sent and PONGs received. Since there is a natural latency between

PING/PONG requests, it's possible that the numbers are not exactly the same at any particular moment. Enforce a bound as failure signal. This bound can be checked periodically by client requests (say, every 15-th packet).

        - Extra credit (5%) Failure injection + handling:
            - Simulate a switch failure of switch 1, and validate that your system will correct reroute requests to the standby switch 3.
            - Do not kill the switch process, as this may result in disruption of your setup. Simply use your control plane script to configure the match/action table that governs the PING/PONG packets, and insert a match/action entry that matches all PING packets and drop them. This will stop switch 1 from responding PONG packets to switch 0, the load balancer.
            - Plot the number of requests per time unit that switch 3 receives before and after the simulated failure. X-axis: simulation time; Y-axis: number of requests received at switch 3.

- **Milestone 3:** Secure cloud storage via access control lists.
    - Cloud storage needs to be secure. This means, to start with, every client should first authenticate with the server before it is allowed in. But for simplicity we'll assume that this authentication has already been performed "out of band", and that every client will have a unique client ID. Implement a commonly used mechanism called ACL (Access Control Lists) on the frontend switch to check whether or not a client ID has access to a particular part of the key range. (Suggested timeline: 3 weeks: Oct. 29 -- Nov. 17).
    - Assume there are two clients, Alice and Bob, with IDs 0 and 1. We will implement the following ACL behavior:
        - Alice has read access to all key ranges [0--1024], but she only has write access to key ranges [0, 512]. She is not allowed to modify keys in (512, 1024].
        - Bob has read/write access to [0, 256], and no access to any other key range.
        - These ACL rules can be implemented in a new match/action table in the frontend switch. A new header based on client ID will be used.
    - Bob has a higher rate limit, and Alice has a lower rate limit, for prevention of denial of service attacks.
        - Keep track of how many requests Bob and Alice have sent and deny all extra requests once their rate limits have been reached.
        - In real-world systems, this will require a "sliding window" behavior that keeps track of time, request per second, and timing related features. However, for simplicity we'll skip the timing-based behavior. The rate limit for Bob and Alice can be set statically, say, 100 for Bob and 50 for Alice, for the entire run. Once the limit has been reached, further requests will not be processed.

- **Extra credit beyond these milestones above:** 5%

    - Implement Milestone 1 on a real hardware switch.
    - Hardware emulator arrangements -- email myself and TAs for setup.

- **Notes**: For each milestone, use a separate folder so that it can be graded in a standalone fashion. “MS1/*” contains your code and configurations and scripts for the first milestone. “MS2/*” contains all that is necessary for grading your second milestone, and it can contain duplicate code from “MS1/*”; please make sure it is standalone from the code in “MS1/*”. Similarly for MS3. Also include a six-page (or so) project report that describes your design, implementation, testing, and any figures in PDF. Submit via email (do not include intermediate data files, such as .pcap files in your submission). Email myself and all TAs.

- **READMEs and tests**: Please include READMEs for each milestone/folder. Write comments in your code whenever appropriate. Your project will be graded by code clarity and style, too. In each README, please include the following information:
    - Main files that you have created: P4 programs, configuration files, and your Python scripts for the control plane.
    - Test cases and scripts that you have used: Your test scripts, which will likely be part of your Python-based scapy clients. The scapy script should document the tests and the intended outcomes for each test.
    - Project report.

- **Groups:** Three students per group. All students will receive the same score on the project component of the course. In exceptional situations, groups of four are possible (**upon confirmation with instructor**) if groups of three cannot be formed; in this case, any two out of the three extra credits (MS1, MS2, the Hardware component) will be required components for such a group to receive full scores on the project. The third extra credit component will be extra credit for a 4-student group.