Script-Based Approach to P4 Data Plane Programming

Pascal Bast, Thomas Scheffler
Hochschule für Technik und Wirtschaft Berlin, Germany
pascal.bast@googlemail.com, thomas.scheffler@htw-berlin.de

***Abstract*—This paper presents a script-based approach to P4 code-generation and device configuration for a Tofino P4 switch. We developed this approach for a prototypical implementation of a DDoS stresser application that should be used quickly and easily by network security personnel without a deeper understanding of P4 programming or switch operation. Using a templating engine for the adaptation of P4 code and switch configuration fulfilled this requirement, while allowing further development of the code base.**

I. Introduction

When P4 [1] was first introduced to the network community, it created a lot of interest and researchers immediately started to assess the benefits and potentials of this new technology.

P4 has been a breath of fresh air for network research and development teams, that were otherwise restricted by fixed-function network devices and vendor road maps. P4 became a new tool for protocol research and network experiments at a competitive and immediately usable performance level. A 2023 study conducted by Hauser et al. analyzed 245 research papers on applied P4 research [2] in areas such as advanced networking, monitoring, traffic management, routing, network security and others. Uptake of P4 in production networks, however, has not been equally impressive.

While large *hyperscalers*, with their software-driven approach to networking benefited from deploying P4 in their data centers, the picture looks entirely different in the world of network operators and enterprise networks. Here P4 is met with a lot more resistance and distrust.

Traditional network gear mostly comes with pre-loaded firmware and is ready to be installed, configured and used out of the box. P4-programmable switches offer much more flexibility, as they can be used for multiple roles in the network, but require development tool chains and the writing of dataplane code before they become useful tools.

Creating effective and bug-free P4 programs requires a much deeper understanding of network protocol behavior than is normally required for network administration. Additionally, while P4 tries to abstract some of the platform hardware architecture, it is still a very low-level language that closely interacts with the hardware functions and requires a deep understanding of possible restrictions and approaches.

II. Idea

We wanted to use an Intel Tofino P4 switch as a flexible network research platform that can be used for different experiments and supports network tests for different protocols.

For our specific use-case it became necessary to flexibly adjust certain parameters in the P4 code and reconfigure the Tofino traffic generator, sometimes even between individual test runs. We found that P4 applications can be very flexible and cover very specific use cases, but they sometimes require the adaptation or rewrite of some part of the code. Manual adaptation of source code and device setup is not only labor intensive and repetitive, it also opens the possibility to introduce bugs in the code and hinders reproducibility.

We came up with the idea to use P4 code-templates that will be integrated in a script-based environment, so that the test operator did not have to manipulate any P4 code and needed only a basic understanding of network and device setup.

We used the *Jinja*[1] templating engine to create several P4 template files that are parameterized from a Python script before each test run. This allowed us to keep the P4 programs small and dedicated to a particular task, which also helps in switch resource management and improves development time by reducing complexity and code interdependencies. Small programs can usually be developed, tested, and deployed faster than large ones.

Our approach uses a hybrid configuration model: structural parameters affecting packet processing logic (such as protocol header definitions, parser states, and match-action pipeline structures) require template-based P4 code generation, while runtime parameters (such as IP ranges, packet data and port configurations) are configured via the data plane API.

III. Implementation

We implemented a DDoS stresser application in P4 that is able to generate high-flow packet streams which mimic the characteristic of typical DDoS attacks, such as TCP SYN, UDP or ICMP packet floods.[2]

Figure 1 shows our management system. It is used for the definition of test packets and traffic patterns, as well as the parameterization and execution of the tests. The management

[1] https://palletsprojects.com/projects/jinja/
[2] https://github.com/pascalb97/tofino-traffic-generator

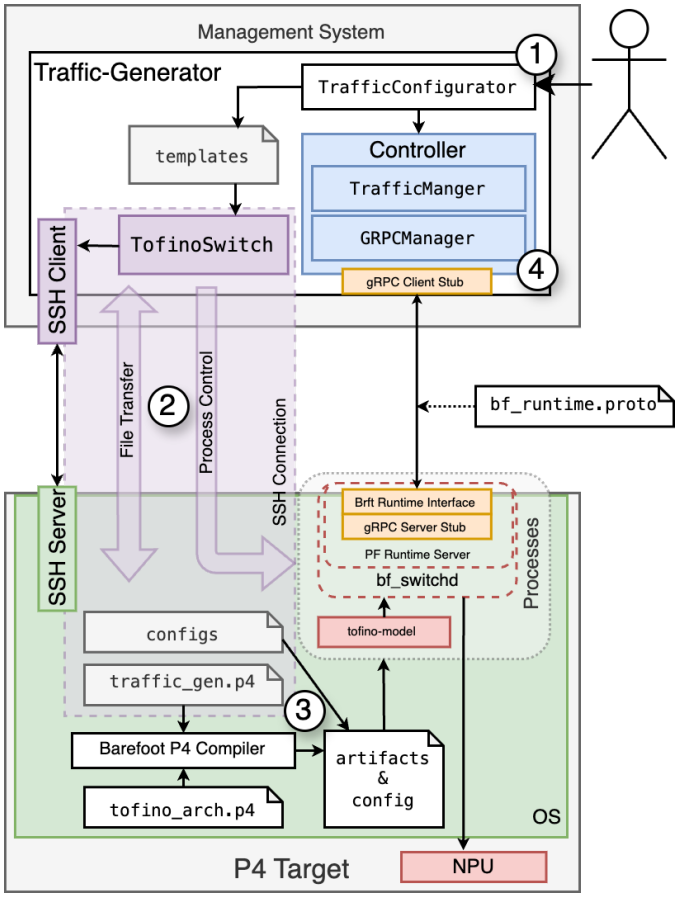

Figure 1. System Architecture of the DDoS Stresser

system is implemented as a set of Python scripts on the operators system.

The operator specifies the necessary test parameters (test duration, IP addresses, switch port configurations, etc.) and the *TrafficConfigurator* generates the necessary P4 code based on pre-defined code templates (*Step 1* in Figure 1). This script also configures the controller to orchestrate the test-setup on the switch. It uses *scapy*[3] for flexible IP-packet generation.

The generated P4 source code and configuration parameters are then transferred to the switch (*Step 2*) and compiled into an executable data-plane binary together with the necessary Bft Runtime Interface and gRPC protocol description (*Step 3*).

In (*Step 4*), the *Controller* initializes and executes the test via the data plane API. We use the `bfrt_helper` library[4] for this. The system gathers relevant information from the P4 switch, such as the total number of packets generated, the packet rate in packets per second (Mpps) and throughput in bits per second (Mbps).

The initial setup time for a single test is dominated by the P4 compilation phase (varying based on the P4 target hardware). Subsequent script-based configuration parameter changes execute in seconds since they do not require recompilation.

[3]https://scapy.net/
[4]https://github.com/APS-Networks/bfrt-helper

A. P4 code adaptation

The use of templates for P4 data plane programming offers several key advantages that make the script-based approach via Python both practical and powerful.

Abstraction of P4 Complexity: Users can work at a higher level of abstraction by defining their requirements in a Python script. Python variables control the generation of data packet structures using *scapy*:

```
traffic_conf.add_packet_data(
    source_cidr="192.168.178.1/24",
    destination_cidr="192.168.178.25/32",
    protocols=["tcp", "ipv4"],
    pkt_len=500
)
```

Listing 1: High-level packet configuration using *scapy*

The same variables are also used to generate the appropriate P4 code that processes these packets by using templates:

```
state parse_ethernet {
    pkt.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
        {% if 'ipv4' in protocols %}
        TYPE_IPV4: parse_ipv4;
        {% else %}
        default: accept;
        {% endif %}
    }
}
```

Listing 2: Generated P4 parser state from Jinja template

Automated Header Processing: While P4 requires explicit header definitions and parsing rules, network administrators generally think in terms of protocols and packet structures. By using Jinja templates, we can automatically generate the necessary P4 parsing logic based on the packet structure defined in Python. This means:

- Header definitions are generated only for protocols actually used
- Parser states are created dynamically, based on the packet requirements
- Complex header dependencies are handled automatically

B. Simplified switch configuration

Simplified Deployment: The use of templates enables:

- Consistent code generation across different deployments
- Automatic adaptation to different switch configurations
- Reduced chance of human error in P4 programming
- Quick modifications without deep P4 knowledge

Automated Switch Setup: The template-based approach can also be used to generate switch configurations, allowing automated setup via the management plane. Instead of manual CLI configuration, device parameters are defined in Python and automatically deployed:

```
#Port configuration settings
traffic_conf.add_physical_output_port(
    output_physical_port=1,
    port_speed="25G")
...
generate_port_config(self.output_physical_port,
↪  self.port_speed)
...
def generate_port_config(port, port_bw):
    template_data = {"port": port, "port_bw": port_bw}
    template_to_file(
        "templates/port_config_template.txt.j2",
        ↪  "src/port_config.txt", template_data)
```

Listing 3: Example of port configuration code

Listing 3 generates the switch port configuration file `port_config.txt` by replacing the placeholder in the template with values from the Python script:

```
port-del {{ port }}/-
port-add {{ port }}/- {{ port_bw }} NONE
port-enb {{ port }}/-
an-set {{ port }}/- 2
```

The port configuration file is then uploaded via `scp` to the switch and executed using `bfshell` (Listing 4):

```
scp_put(self.ssh_client, "src/port_config.txt",
↪  "/tmp/port_config.txt")
stdin, stdout, stderr = ssh_exec(
    self.ssh_client,
    f". .profile > /dev/null ; /bin/bash
    ↪  {remote_env_vars['SDE']}/run_bfshell.sh -f
    ↪  /tmp/port_config.txt")
```

Listing 4: Switch port configuration via Python script

C. Integration with Barefoot Runtime Interface

Our script-based approach is further enhanced by integrating the Barefoot Runtime (BfRt) gRPC interface, which provides programmatic access to the P4 data plane. This integration is crucial for runtime configuration, table management, and performance monitoring. We abstract the complexity of the gRPC interface through Python bindings, while maintaining the full functionality of the P4 switch.

This abstraction layer provides several advantages:

Unified Configuration Interface: By wrapping the BfRt gRPC interface in Python, we create a cohesive programming model that spans from packet generation to runtime management. This reduces the cognitive load on operators and researchers by presenting a single, consistent interface for all switch interactions.

Real-time Performance Monitoring: The integration enables programmatic access to switch metrics and could be used to generate reports or integrate the switch into existing monitoring solutions.

```
# Define the IP ranges and packet template
traffic_conf.add_packet_data(
    source_cidr="192.168.178.1/24",
    destination_cidr="192.168.178.25/32",
    ...
)
...
    self.ip_src, self.source_mask = source_cidr.split("/")
    self.source_mask = self.cidr_to_netmask(source_cidr)
...
def configure_egress_table(
    self, source_mask, destination_mask, src_mac, dst_mac,
    ↪  output_port, program_name):
    egress_table =
    ↪  self.bfrt_info.get_table("pipe.SwitchEgress.egress_table")
    status["clean_table"] = self.clean_table(program_name,
    ↪  egress_table.name)
    request = self.bfrt_helper.create_table_write(
        program_name, egress_table.name,
        {"eg_intr_md.egress_port": Exact(PortId(output_port))},
        action_name="SwitchEgress.replace_ip_address",
        action_params={
            "s_mask": IntField(ip2int(source_mask)),
            "d_mask": IntField(ip2int(destination_mask)),
            ...
        },
        update_type=bfruntime_pb2.Update.Type.INSERT,
    )
    response = self.grpc_client.Write(request)
```

Listing 5: Configuration of P4 table entries using the BfRt gRPC interface

Dynamic Table Management: P4 programs make extensive use of P4 tables for control plane interaction. Listing 5 shows how tables can be manipulated directly from Python code.

IV. Conclusion

We used the Jinja templating engine for quick adaptation of P4 code and switch configuration. We also managed the switch operation through Python bindings for the gRPC interface. This made the use of a P4 switch very easy for the operator. No special programming knowledge of P4 and only a basic understanding of the hardware setup, IP networking and network security was needed to use the switch and adapt it to specific test cases.

The emphasis on simplicity and automation makes this approach particularly suitable for network operators and enterprise customers who lack deep knowledge of P4 programming, but still want to benefit from the impressive capabilities of these systems.

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2656877.2656890

[2] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with p4: Fundamentals, advances, and applied research," *Journal of Network and Computer Applications*, vol. 212, p. 103561, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804522002028