Low Impact Tenant Code Updates on Multi-tenant Programmable Switches

Timo Geier and Sebastian Rieger
Department of Applied Computer Science, Fulda University of Applied Sciences, Germany
Email: {timo.geier, sebastian.rieger}@cs.hs-fulda.de

***Abstract*—Software-defined Networking (SDN) and Programming Protocol-Independent Packet Processors (P4) introduced data processing within the network data plane. To offer multiple tenants to deploy individual code in programmable switches and network devices, code updates must ensure proper tenant isolation and minimal negative cross-tenant impact during code updates. Thus, this paper presents a code deployment pipeline, primarily implements an orchestrator allowing network device discovery, code verification, compilation and low impact deployment on multi-tenant programmable switches. Critical time windows and their durations for the code update are evaluated using hardware switches (Intel Tofino). Performance impacts of code updates on tenants using these switches are evaluated and discussed based on bandwidth tests considering different code deployment options. The architecture proposes a framework to enable seamless updates with minimal to no service interruptions, e.g., using gradual code updates of redundant links between programmable switches in data centers, internet service providers, and mobile networks used by various customers as tenants. Besides P4, common Kubernetes and continuous delivery solutions were used for the presented implementation that is offered as Open Source for further adaption and development.**

***Index Terms*—Network Functions, Container, Multi-tenancy, Network Programmability, Hardware-accelerated Virtualization**

I. Introduction

Recent progress in the area of Software-defined Networking (SDN) and Programmable Data Plane (PDP), esp. using Programming Protocol-Independent Packet Processors (P4) as an advanced SDN approach also supports programmable switches that run code of multiple tenants to individually process their packets. Such features enable these users to benefit from network automation and programmability techniques leveraging in-network processing and computing, e.g., enabling fine-grained traffic engineering, network attack detection and mitigation or data preprocessing, e.g., for machine learning, directly in the data plane (as, e.g., described in [1]). Examples for multi-tenant approaches for P4-based switches as well as the associated control plane were given in [2] [3] [4]. However, as tenant code directly runs in and influences the data plane, multi-tenant PDP code updates need to ensure the least possible negative impacts for other tenants' packets (packet loss, link/path outage) during code changes. This is especially relevant during a short downtime inherently stemming from the data plane code swap and hence being particularly critical. In this paper an approach to evaluate and reduce this negative impact of code updates in multi-tenant PDP switches is presented using a developed orchestration architecture and pipeline, based on Continuous Integration / Continuous Delivery (CI/CD) concepts for the deployment of PDP tenant code fragments. The presented Orchestrator for Multi-tenant PDP Code Updates (OMuProCU) introduces a novel lifecycle management of tenant code for accelerated Containerized Network Functions (CNFs) within the data plane ranging from deployment, over updates up to decommissioning, mitigating their negative impact on other users (i.e., packet loss, latency, bitrate). While [2] introduced the overall concept including Network Operating System (NOS) and control plane, the focus in this paper lies on a robust in-network deployment process with a low impact. Effectiveness of the proposed architecture is evaluated using network bandwidth and latency tests while rolling out PDP tenant code updates. An equal cost multi-pathing (ECMP) testbed using Intel Tofino based PDP hardware is leveraged to measure the impact of multi-tenant code updates. The developed orchestration pipeline code is provided as an open-source repository[1] together with evaluation data and steps used in this paper. The orchestrator can be used to implement multi-tenant programmable networks (e.g., in 5G/6G) with minimal negative impact and without critical downtime windows linked to the programmability.

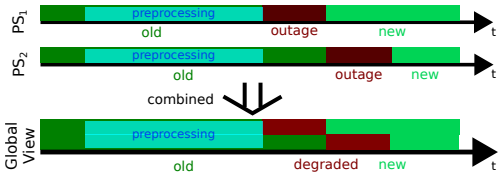


Fig. 1: In-Network Code Update Timeline

Figure 1 shows the process of gradual code roll out using the implemented orchestration on a 2-way ECMP path. While the overall duration of the code update on the entire ECMP path combination (*Global View*) in this example takes twice as long compared to updating a single programmable switch (c.f., PS_x), outages during the critical time window to swap the PDP code are prevented and only pose a degraded performance period. Remaining sections of the paper present and discuss the orchestration pipeline and deployment step strategies for multi-tenant code updates. The next section, presents related work and distinguishing features of our contribution. The paper concludes with an evaluation of the network performance for multiple tenants using the proposed orchestrator also discussing beneficial use-cases.

[1] https://github.com/tiritor/OMuProCU

978-3-903176-59-1

II. Related Work

Deep programmability is a topic that holds the potential to change network design approaches. According to [1], new methods to design new system approaches should be developed, and a network be thought of as a large, programmable and decentralized system. The authors express that this can be important for example in 5G infrastructures. In [5], use cases for PDP with P4 in a 5G edge computing scenario are discussed and an architecture similar to the one that is used in this publication is presented. The paper demonstrates the benefit of the architecture by deploying virtualized switches running on bare metal or in Docker containers with and without P4 acceleration. In [6], a framework to offload network functions in 5G Cloud Computing environments is presented. Furthermore, the implementation of offloading PDP functions as part of accelerated virtual network functions in a multi-tenant cloud infrastructure is proposed in [7]. To get multi-tenancy in the control plane of P4 environments, [3] enhanced the existing P4Runtime and proposed a control plane architecture to be usable in four international P4 Experimental Networks (i-P4EN). Furthermore, the continuation in [4] proposes an architecture to use in-network multi-tenancy in a P4 switch environment which was tested and evaluated using in software as well as on a Tofino-based hardware switch. They introduced a role ID to isolate different tenants in the in-network layer. However, in contrast to this paper, they focused on the implementation without considering an optimal in-network code update strategy. A proposal to optimally place accelerated service function chains (SFC) written in P4 in a topology with the utilization of programmable entities like CPU, NPU, FPGA or programmable ASICs is presented by [8]. Orchestration of serverless applications in edge computing environments using network programmability is proposed by [9]. AWS Greengrass is used to build an IoT cloud and edge computing environment. Also, different approaches utilize the flexibility of a PDP to offload parts of multi-tenant network functions into the data plane. In [10], an architecture for communication between containers in a Kubernetes environment is proposed. This approach only works for the container network infrastructure (CNI) framework flannel by using VXLAN.
While all related work described in this section also focused on accelerated CNF in multi-tenant environments, previous papers did not contain an approach to orchestrate and deploy these accelerated CNFs on PDP switches. This paper presents such an approach and also ensures a low impact on current traffic during code updates.

III. Combining Controlled Orchestration of Accelerated CNF with low impact

Accelerated data processing and in-network computing in virtualized network infrastructures (e.g., CNFs) and their isolation in a multi-tenant form pose a dilemma, since giving tenants direct access to hardware inherently reduces isolation. To circumvent this issue and reach an appropriate tenant isolation while still offering acceleration, tenant control and separation has to be implemented in the data plane (P4 switch) as well as the control plane (NOS). Isolation in NOS and in-network layer already received some research, however existing publications did not consider an orchestration of accelerated multi-tenant CNFs yet. Consequently, existing architectures cannot swap code without downtimes. To achieve a controlled orchestration of accelerated CNF for the deployment, a manifest called Tenant Description Code (TDC) was introduced in the previous work [2]. This is split into three different parts: Tenant Container Description, Tenant Isolation Logic and In-Network Code. The Tenant Container Description (TCD) contains the Kubernetes deployment description for the CNF deployment in the NOS and the acceleration type to be used, while Tenant Isolation Logic (TIL) consists of the export rules and the runtime rules that should be applied after the deployment. Export rules represent a list of VNIs being used by a tenant for this TDC. The In-Network Code (INC) includes the custom accelerated function which should be embedded in the previously defined accelerator template code called Tenant INC Framework (TIF).

To reach this goal, the proposed OMuProCU must meet the following requirements. On the one hand, the deployment process needs to be as resilient as possible by running all non-critical tasks before the critical task. On the other hand, it needs to ensure that the TDCs submitted by the tenants are valid and can be deployed safely without affecting other tenants, especially in the in-network layer, where the NOS can be controlled by using more sophisticated multi-tenant containerization constraints, such as quotas. Also, the commissioning process must be done in an optimal time window depending on the underlying network infrastructure. In addition, the preprocessing tasks of the in-network layer are processed in the NOS and should also not have a large impact on the critical part of the deployment process.

IV. Scheduling of hardware-accelerated CNF

The requirements described in Section III are addressed by our OMuProCU. For this, the architecture of [2] is improved and adapted to use the TNA Architecture instead of the software-switch BMv2. Also, the improved architecture covers multiple purposes, and is not limited to a single use case as in [2]. The microservice architecture in the orchestrator itself is kept to have single purpose components interacting with each other. Though the OMuProCU evaluation in this paper currently manages and orchestrates only one PS, the orchestrator can use multiple switches within network infrastructures.
To gain a more secure and validated deployment of offloaded code parts of multiple tenants without a large negative impact on the overall latency or on live packet processing, a proper deployment process is needed. Therefore, the presented deployment process follows common CI/CD paradigms proposing a pipeline concept with a reliable rollout mechanism. In general, the order of the deployment steps is important since the critical part must be kept as short as possible while all other, dependent, non-critical steps must be processed before this critical time window. Consequently, the tenant code change

operations (create/update/remove OTFs) must be performed as fast as possible, keeping the downtime window to a minimum. On the other hand, this makes the process rather static and offers a big time improvement, as the critical part is already isolated in the presented implementation, as can be seen in the structure of the deployment pipeline described in Section IV-B.

A. Validation of TDC manifests

The main issue which should be validated in this approach is to prevent security issues when using common accelerator chips in programmable hardware for multi-tenant use cases like access to not allowed areas, e.g., tables/registers of other tenants. In the TCD part, the Kubernetes deployment description is mainly validated. Also, the usage of the correct and allowed associated hardware is verified. TIL is validated and checked for access validation to other tenants' resources or functions to ensure isolation. Especially, rules must be checked such that they are considered for the corresponding accelerated tenant function, e.g., the rules meant to be deployed in the OTF must be matched to the corresponding table while the data export rules must point to the correct corresponding tenant CNF. Similar to the TIL check, the INC must be checked for the same access violation criteria. Furthermore, the code must be checked for table or register accesses which are implemented in other OTFs. Also, include and extern statements must be checked for access violations and access to other OTFs in general needs to be limited.

B. Deployment Process of TDC manifests

The proposed orchestrator is based on previous work presented in [2] and offers a more detailed architecture as well as an improved algorithm for the deployment process. Also, it is adapted to use TNA as hardware-based acceleration architecture and implements a proper validation. In this adaption process, the previous introduced TIF structure is generalized to cover different use cases instead of a single one. As described before, the update process must be optimized to ensure that all steps are successfully done before the critical part begins. The proposed pipeline model is visualized in Figure 2.

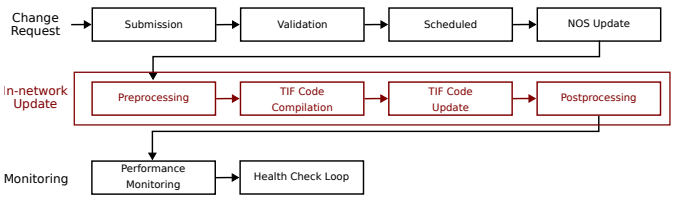


Fig. 2: OMuProCU Orchestrator Pipeline Steps

At the beginning of the OMuProCU pipeline, a TDC manifest is submitted by a client. As next step, the validation of this submitted manifest will be processed as described in Section IV-A. If the validation was successful, the deployment of the submitted manifest is scheduled for the next maintenance time window. Otherwise, the submission is denied, and an error message sent back to the change initiator. The size of the schedule time window is a configuration aspect and depends on the deployment process duration and the used infrastructure. More conservative networks would only allow scheduling in big time windows while agile infrastructures could deploy scheduled deployments in shorter intervals. Once this time window is reached, the deployment is triggered. If the orchestrator is in the deployment state, the submission of new deployments is locked. Respectively, any incoming submission will be denied. Also, no runtime rule updates for any deployed manifest are possible in the deployment process. First, the tenant CNF part in the NOS will be updated. After the successful deployment in the NOS deployment, the in-network update part is processed, which starts with a preprocessing of the TDCs by embedding the in-network code of the manifest into the TIF template for the specified accelerator and compiling the code for the accelerators afterwards. The compiled TIF code is subsequently applied to each accelerator. As last step of this part, post-processing steps like hardware initialization and application of pre-existing and newly added, removed or updated runtime rules are executed. Final step before entering the health check loop for deployed TDCs is to monitor the performance of the applied TIF to measure the impact of the new TIF against the previous one. This was implemented to explicitly prevent the new code of the tenant from causing significant performance degradation on all other tenants compared to the old version.
As highlighted in Figure 2, the most critical steps are the in-network update steps, because they cannot be done in parallel to other deployment steps and can cause a failure state instead of a degraded one if one of these step fails by an incorrect INC.

C. Implementation Details

The proposed framework is mainly developed in Python, while microservice components are split into modules. Commonly used modules like the orchestrator client and the validation are packaged in separate modules to provide an interface to the experiment scripts in the testbed needed for the evaluation. gRPC is used for the communication between components of the orchestrator and the Tofino chip. More precisely, the communication with the Tofino chip is carried out using the BarefootRuntime-API of the Open-Tofino framework released by Barefoot Networks/Intel [11]. The presented TIF is implemented in P4 and extended by using the Tofino framework. The testbed consists of two hosts and four APS BF2556X-1T switches connected to the hosts via 10 Gbit/s interfaces. Each host has 4 CPU cores, 4 GB RAM and uses a Mellanox 10 Gbit/s network card for the network connection between each host and the PS. The testbed setup is shown in Figure 3. To ensure a multi-tenant network environment in the testbed, VXLAN point-to-point connections are established between the hosts.

V. Evaluation

For all following experiments, the same testbed is used as described before and the evaluation is done on one switch in the testbed while the other switches are programmed as repeater to ensure minimal interference of the code swap on the connected links in the presented setup in Section IV-C.

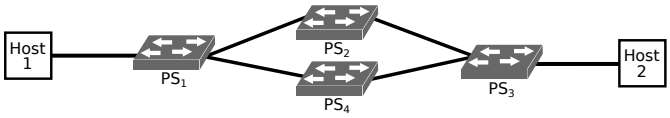
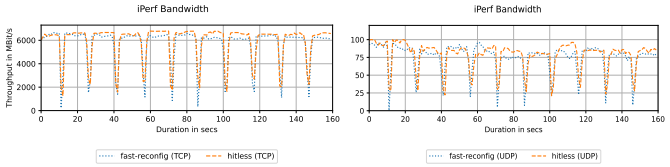


Fig. 3: OMuProCU Testbed Architecture

A. Code Swap Impact Evaluation

To be able to discuss the significance of this problem, a simple P4 program implementing a repeater between two links is deployed 10 times in a row over a time interval of 15 seconds. The BF-Runtime interface provides two different modes to update the PDP code: fast-reconfig and hitless. fast-reconfig keeps the time window for code swap smaller, but packets transmitted in this window are definitely lost. hitless mode tries to keep the packet loss as low as possible, but takes more time to perform the actual code change. [11]



(a) TCP (b) UDP

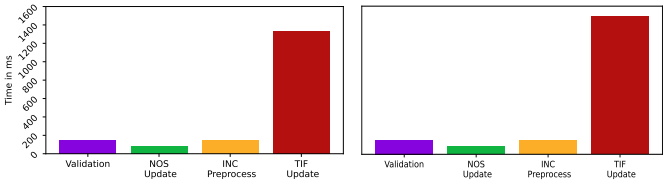
Fig. 4: Bandwidth measurement for available device modes

The bandwidth test traffic is performed using the common iPerf3. As transport protocols, TCP and UDP are used. Running normal bandwidth tests between the hosts achieves about *9.33 Gbit/s* utilization in our testbed. Since the framework should be used in multi-tenant environments, VXLAN is utilized as it is typically used as layer 2 isolation in common data center and cloud infrastructures. Implementing VXLAN in the testbed however causes a significantly lower mean bitrate with peaks at ca. *6.543 Gbit/s*, due to additional load on the hosts caused by the tunneling encapsulation. The cause of this is the VXLAN kernel module implementation of the Linux kernel used at the host endpoints since there is no offloading available and the CPU must handle the encapsulation and decapsulation process. VXLAN could also be handled on edge switches, but the evaluation primarily focuses experienced downtime and relative service degradation instead of bandwidth. Figure 4 shows the result of this experiment.

For both modes and transport protocols, the bandwidth drops if the code update is in progress and therefore confirms, that the programmable switch will be in a degraded state during code update. On the other hand hitless offers a significantly better bitrate recovery after the code update as seen in Figure 4. Also, the spike in this mode is not as deep as in fast-reconfig mode. So, the time window for the code update is critical and must be kept as short as possible in all cases. Throughput for UDP needs to be specified as it obviously does not include flow and congestion control. Though iPerf3 can be tuned to offer higher bandwidths for UDP using bursts, a bitrate of 100 Mbit/s was chosen to circumvent iPerf3's UDP bandwidth capping and as only the relative impact of the code update on the throughput is evaluated and discussed.

B. Time Measurement of the Orchestrator

The deployment process can be time-consuming as all steps which are described in Figure 2 must be executed. The evaluation uses fast-reconfig mode, but results are similar in hitless mode, though it takes slightly longer, as shown in Section V-C. According to our tests, the deployment duration took in mean *45.01 seconds* including scheduling time. Therefore, to classify the critical time window size in relation to other steps of the deployment process for the implemented framework, the duration for each step is measured. Some steps are not relevant and vary based on the hardware the developed orchestrator runs on, e.g., the compilation time depends on the used accelerator and its compilation framework. As another example: The scheduled time depends on the network structure and its configuration, which means that critical network infrastructures have bigger scheduling intervals as non-critical network infrastructure parts. In our testbed, the schedule time window was set to *10 seconds*. Also, the compilation duration for the TIF took in mean *36.886 seconds* on our hardware. Figure 5 shows the duration plots for different influenceable and relevant steps of the orchestrator in relation to the critical TIF update step in both modes. Since the results for both transport protocols were similar in both modes, the results for TCP are shown and explained as an example.



(a) fast-reconfig mode (TCP) (b) hitless mode (TCP)

Fig. 5: Time measurement for influenceable OMuProCU steps

The critical time window is the TIF update time. This includes the hardware initialization of after the update proceeded. The biggest time period of all steps is the critical time window of in mean *1.524 seconds* in our tests which is significantly longer than the other steps which can and must be processed before. From this follows that the TIF update is the most time-consuming part in the in-network deployment process and highlights the importance of a good scheduling and processing algorithm. Otherwise, if for example the preprocessing was not carried out correctly, additional code updates are necessary to revert the change again imposing the impact of additional critical TIF update durations.

C. Time Measurement of TIF Deployment

As described in Section V-B, the duration of the code swap and chip initialization consumes the most time compared to all other influenceable and self-implemented steps. This step should simultaneously be the one to be kept as short as possible to get the shortest downtime interval. As Tofino chips are common standard for programmable switch hardware, designing a new architecture for better code update mechanism to solve this problem is beyond the scope of this paper. On the contrary, we intentionally developed an architecture that tries

to limit the negative impact of multi-tenant code updates on this commonly available reference architecture and hardware. Thus, we evaluated and analyzed the possibilities to optimize the code update using Open-Tofino as follows. Similar to the evaluation in Section V-A, the duration of this step for the developed TIF without any custom tenant function is measured by performing 10 subsequent code swaps in a row for each device initialization mode and transport protocol. The mean of aggregated duration of these code swaps are presented in Table I. Since both used transport protocols provide similar results, the results for TCP are shown.

device init mode	protocol	code swap	initialization	total
fast-reconfig	TCP	1.305000	0.066500	**1.371500**
hitless	TCP	1.580700	0.066000	**1.646700**

TABLE I: Time measurement of TIF code updates (in secs)

If using fast-reconfig mode, the swap performs faster than using the hitless mode for both protocols. The reason for that stems from the hitless mode trying a hitless code update which results in a longer duration. However, this result was surprising, since there should be no performance drop at all. Nonetheless, it is plausible, as at some point in time packets going through the updated P4 pipeline in the Tofino-based switch are impacted by the update. Though, if the code update is performed in hitless mode, the bandwidth loss and its recovery perform better. The reason for this effect is that hitless mode is more cautious while replacing code fragments.

VI. Discussion

As seen in Section V-A, the used hardware switch causes packet loss while commissioning provided code regardless of the method used to swap the code. Also, the largest duration of all influenceable orchestrator steps is caused by the TIF Update process. The PDP code update mode hitless can reduce the commissioning time discussed in Section V-C, but this mode cannot compensate for most packet loss as observable in Section V-A. This highlights the importance to find and build a proper solution for deploying accelerated CNFs with as little negative impact as possible on the data plane processing which was tackled by implementing OMuProCU as proposed in this paper. By introducing the OMuProCU pipeline, scheduling is improved because all steps that can fail and trigger a rollback with another PDP code swap are carried out beforehand. The deployment process is optimized to keep the critical time window as small as possible. Using the standard Tofino pipeline, there is no space for further optimization regarding the duration of these steps. Even if the deployment was successful, the performance of the framework should be monitored to check for performance issues in the new provided tenant code and keep the possibility to trigger a rollback to the previous provided code if necessary.
Furthermore, the timing to trigger the deployment of the submitted and scheduled manifests is done in OMuProCU by using a schedule time interval. As described before, this strongly depends on the used network infrastructure (cf. more agile or more conservative network architecture design). This can be further improved by using trigger thresholds, e.g., if the bitrate is under a specific threshold and hence the network load is not critical. The threshold can again be defined based on the network infrastructure. Currently, the OMuProCU covers only the control plane of one PS. Each switch in the infrastructure would run its own OMuProCU and there is no connection between them at the moment. However, the orchestrator can be extended to cover multiple programmable switches or network devices in an entire infrastructure. This would highlight the benefit of the provided and evaluated implementation for example in a service or cloud provider. In such environments typically multiple tenants use the same network, e.g., being resellers or overlay providers offering value-added services. Using the orchestrator, these customers of the provider can individually place parts of their services and applications in the data plane without interfering with each other.

VII. Conclusion and Future Work

In this paper, we presented the framework OMuProCU, which ensures a lifecycle management of accelerated CNFs in the data plane from deploying over updating to decommissioning with low impact on the used network infrastructure. Also, we demonstrated the impact of the code update process which occurs using state-of-the-art hardware and its importance in the deployment process of accelerated CNF. As next steps, the OMuProCU will be extended to run in a distributed global control plane over several switches to get a global view using update mechanism for an entire infrastructure. Also, the orchestration of tenant functions on specific acceleration hardware across network infrastructures managed by the framework will be implemented. Moreover, we currently consider the evaluation of new isolation techniques for the CNF like containerless techniques (WebAssembly) and microVMs.

References

[1] N. Foster *et al.*, “Using deep programmability to put network owners in control”, vol. 50, no. 4, pp. 82–88.

[2] T. Geier and S. Rieger, “Improving the deployment of multi-tenant containerized network function acceleration”, in *2022 5th International Conference on Advanced Communication Technologies and Networking (CommNet)*. IEEE, 2022, pp. 1–7.

[3] B. Chung *et al.*, “P4mt: Multi-tenant support prototype for international p4 testbed”, pp. 3–4, publisher: IEEE ISBN: 9781728143873.

[4] ——, “P4mt: Designing and evaluating multi-tenant services for p4 switches”, in *2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, pp. 267–272.

[5] F. Paolucci *et al.*, “Enhancing 5G SDN/NFV Edge with P4 Data Plane Programmability”, *IEEE Network*, no. June, pp. 154–160, 2021.

[6] T. Osinski *et al.*, “DPPx: A P4-based Data Plane Programmability and Exposure framework to enhance NFV services”, *Proceedings of the 2019 IEEE Conference on Network Softwarization: Unleashing the Power of Network Softwarization, NetSoft 2019*, no. 1, pp. 296–300, 2019.

[7] ——, “Offloading data plane functions to the multi-tenant Cloud Infrastructure using P4”, *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2019*, 2019.

[8] H. Harkous *et al.*, “Performance-aware orchestration of p4-based heterogeneous cloud environments”, pp. 1–1, 2023.

[9] I. Pelle *et al.*, “P4-assisted seamless migration of serverless applications towards the edge continuum”, vol. 146, pp. 122–138.

[10] D. Scano *et al.*, “Enabling p4 network telemetry in edge micro data centers with kubernetes orchestration”, *IEEE Access*, vol. 11, pp. 22 637–22 653, 2023.

[11] BareFoot Networks, “Open-Tofino repository”. [Online]. Available: https://github.com/barefootnetworks/Open-Tofino