HTBQueue: A Hierarchical Token Bucket Implementation for the OMNeT++/INET Framework

Marcin Bosk[1,*], Marija Gajić[2,*], Susanna Schwarzmann[3], Stanislav Lange[2], and Thomas Zinner[2]

[1] Technical University of Munich, Department of Informatics, Chair of Connected Mobility
[2] NTNU - Norwegian University of Science and Technology, Department of Information Security and Communication Technology
[3] Technische Universität Berlin, Chair of Internet Network Architectures

Abstract

The hierarchical token bucket (HTB) algorithm allows to specify per-flow bitrate guarantees and enables excess bandwidth sharing between flows of the same class. Additionally, it provides capabilities to prioritize the traffic of specific flows, potentially considering their delay demands. HTB hence constitutes a powerful mechanism to enforce QoS requirements hierarchically and on a fine granular per-flow level, making it an appropriate choice in numerous use-cases. In this paper, we present *HTBQueue*, our implementation of a compound module for HTB support in the discrete event simulator OMNeT++. We validate *HTBQueue*'s functionality in terms of rate conformance and fair bandwidth sharing behavior between competing flows. We furthermore demonstrate its support for flow prioritization.

1   Introduction

The Hierarchical Token Bucket (HTB) algorithm is one of the most widely used mechanisms for rate limiting. Implemented in the Linux traffic control tool `tc`, it is used for network control in testbed setups for research experiments [1, 2, 3] or in general to enforce per-flow Quality of Service (QoS) policies [4, 5, 6]. HTB allows to classify various types of traffic into different queues, according to properties such as the service type or IP address. The different queues into which traffic is classified can be configured with a priority level and a rate limit, allowing to schedule the packets and shape the traffic accordingly. HTB relies on two token buckets for controlling the bandwidth usage of a link. The tokens are generated with the desired per-flow rate and packets can only be dequeued if enough tokens are available in the bucket. The unique feature of HTB is the bandwidth sharing mechanism. A flow which does not fully exploit its *assured rate* consequently does not consume all of its available tokens. These excess tokens (representing the excess bandwidth) can be borrowed by other flows sharing the same parent in the HTB structure. The amount of bandwidth a flow can borrow is limited by its *ceiling rate*.

Although the concept of HTB and its features of setting assured and ceiling rate are relatively old, the same mechanisms are still proposed in modern networking architectures. The 3GPP [7] standard for the Next Generation Radio Access Network (NG-RAN) in 5G defines a guaranteed bitrate (GBR) as well as a maximum bitrate (MBR) for each flow. While the C++ based discrete event simulator OMNeT++ [8] implements a generic module for token buckets[1], a module providing the full capabilities of HTB, including per-flow assured and ceiling rate, is still missing. In this paper, we propose a ready-to-use OMNeT++ module[2] for the

---

[*]Marcin Bosk and Marija Gajić contributed equally to this paper.
[1]https://inet.omnetpp.org/docs/tutorials/queueing/doc/TokenBucket.html
[2]Implementation available at: https://github.com/fg-inet/omnet_htb

INET Framework [9] that supports two-level bitrate guarantees. Our compound module, called *HTBQueue* implements HTB's classful queueing approach based on the Linux HTB implementation [10]. It allows to set priorities and rate limits for an arbitrary number of flows.[3] With the presented tool, we can define a maximum rate for the root and include several intermediate levels that define how the overall capacity is allocated among the different classes and leaf nodes and to which extent leaf nodes can borrow from each other.

The remainder of the paper is structured as follows. Section 2 presents related work. Section 3 gives a brief and general overview on the working principles of HTB, while Section 4 details on our specific implementation. We validate the functionality of our compound module in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

According to [10], the main concept and idea of the HTB originate from [11] where a hierarchical model for link sharing and a resource management model were presented. In [4], the HTB was used to enhance the QoS guaranteed in wireless local networks. It was shown that very low standard deviation and sustainable throughput in such networks are achievable via application of the HTB traffic control in the MAC layer. In addition, this paper shows that HTB introduces hierarchical classes. Such classes are highly desired by the operators due to the fact that they allow for more efficient and scalable network management compared with the per-flow traffic control. High rate conformance of the HTB was further tested and confirmed in [12] and [13]. In [14], the authors show the potential of the wide range of the actual Linux-based HTB implementations (including the CLI method, Linux Network Service, text configuration files, and the Web interface) used for satisfying complex QoS requirements. The Linux implementation on which we based our compound OMNeT++ module was presented in [10] and [15]. In our recent work [16], one of the use cases of the proposed HTB OMNeT++ module was presented. We investigated the potentials of using QoS Flows and slicing to achieve high customer QoE and better utilization of the available resources. Our implementation of the HTB in OMNeT++ was used to emulate and evaluate slice-like traffic isolation.

Another area where traffic shaping plays a key role in meeting performance guarantees is time-sensitive networking (TSN). In this context, various shaping mechanisms such as the credit-based shaper are standardized and have been implemented as OMNeT++ modules within the NeSTiNg [17] framework. However, in contrast to the HTB presented in this work, TSN-related shapers usually do not provide means of two-level shaping and borrowing via guaranteed and maximum bitrate settings.

3 HTB Overview

Traffic control is an essential part of today's network management, supporting a myriad of co-existing heterogeneous applications. According to the 3GPP standards for the Next Generation Radio Access Networks (NG-RAN) in 5G [7], each flow should have its *guaranteed (GBR)* and *maximum bitrate (MBR)*. HTB is a traffic shaper and policer which allows such two-level bitrate guarantees and is therefore very useful to mobile network operators. In the HTB terminology, the GBR and MBR precisely translate to the *assured bitrate* and *ceiling bitrate*.

HTB is a type of classful *token bucket* algorithm. The rate of all the *classes* in the HTB hierarchy is controlled with two nested token buckets governed respectively by *tokens* and

---

[3]Due to the simulative nature our implementation does not have scalability issues, while Linux HTB does.

ctokens. The *tokens* are for sending at the assured rate, and the *ctokens* are for sending at the ceiling rate. By a *class*, we refer to a type of node in the HTB tree structure.
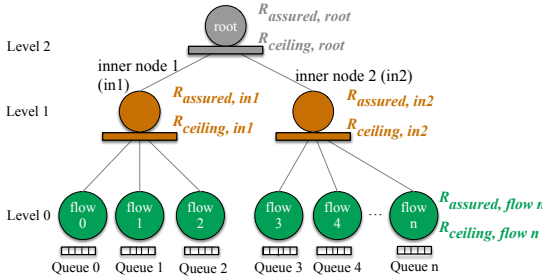

Figure 1: Examplary HTB tree structure.

The key structure for the HTB hierarchy is the HTB tree. It consists of three types of classes (or tree nodes): root, intermediate/inner nodes, and leaves. The root node has no parent node, while the leaves have no child nodes. Inner nodes are the intermediate nodes between the root and the leaves. An example of such a tree structure with 3 levels, $L_{max} = 2$, is shown in Figure 1. The leaves are on the lowest level $L = 0$, and the root is on the highest level $L = 2$. In the figure, we have two inner nodes constituting the middle level $L = 1$. One leaf can be thought of as one flow. In general, only the leaves have queues. Each leaf has an associated priority level - the lower the value, the higher the priority.

All three HTB classes have the following main parameters: the level to which they belong, their assured rate $R_{assured}$, their ceiling rate $R_{ceiling}$ which represents the maximum achievable rate, and a quantum Q_{class} that defines the maximum size of data that can be dequeued in one round. A prerequisite for the rates is that the sum of the assured rates of one node's children has to be less than or equal to the assured rate of that node. The root's assured and ceiling rate are usually set equal to the link bandwidth. Depending on a node's current rate $R_{current}$, at every instance of time, the state of the node is determined with the following three values:

- 0 - *can send*: $R_{current} \leq R_{assured}$,
- 1 - *may borrow*: $R_{assured} < R_{current} < R_{ceiling}$, and
- 2 - *can't send*: $R_{current} \geq R_{ceiling}$.

In addition, each class on level $L > 0$ keeps a list D_{class} of its descendants that are in *may borrow* mode, including both leaves and intermediate nodes (cf. Figure 1). Descendants in this list can, the same as in Linux HTB, have different priority levels $P \in \mathbb{Z}^+, P \leq 7$.

The fundamental working principle of the HTB is to first satisfy the assured rate of all active classes on all priority levels, and then share the parent's excess bandwidth fairly among all the child nodes that are in *may borrow* mode. Among the nodes in this list, those with the highest priority are served first. The portion of the excess bandwidth that each descendant class gets from its parent is *the borrowed bandwidth* - B_{class}. The same principle is then recursively applied for lower priority levels. This link sharing principle is more precisely described with the following equations[4]:

$$R_{class} = \min(R_{ceiling}, R_{assured} + B_{class}) \tag{1}$$

$$B_{class} = \begin{cases} \frac{Q_{class} \cdot R_{parent}}{\sum_{i \in \{x \in D_{parent} | P_x = P_{class}\}} Q_i}, & \text{if } \min(\{P_x | x \in D_{parent}\}) \geq P_{class} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

where R_{parent} is the current rate utilized by the parent, Q_P are quantums of the descendant nodes in D_{parent}, which are on the same priority level as the class of interest ($P = P_{class}$).

---

[4]Based on equations from: http://luxik.cdi.cz/~devik/qos/htb/manual/theory.htm, accessed on 09.06.21
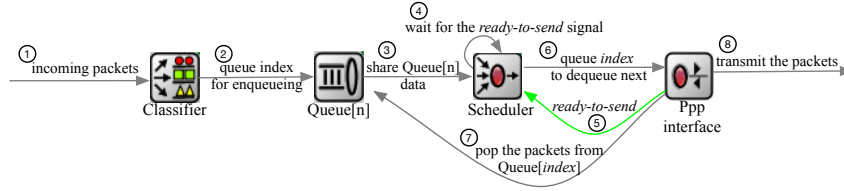
Figure 2: Structure of the HTBQueue OMNeT++ module.

The HTB based scheduler applies these equations always starting from the active classes at the lowest tree level ($L = 0$).

From Equations 1 and 2, we can see that formulas are recursive (R_{parent} is determined via Equation 1), as well as that prioritized applications (i.e., the ones with lower P values) are being served first. In the cases when there are several descendant nodes borrowing from a common parent and having the same priority level, Deficit Round Robin (DRR) is applied to prevent potential starvation of the classes [18]. The amount of bytes sent per node and DRR round is controlled by the quantum.
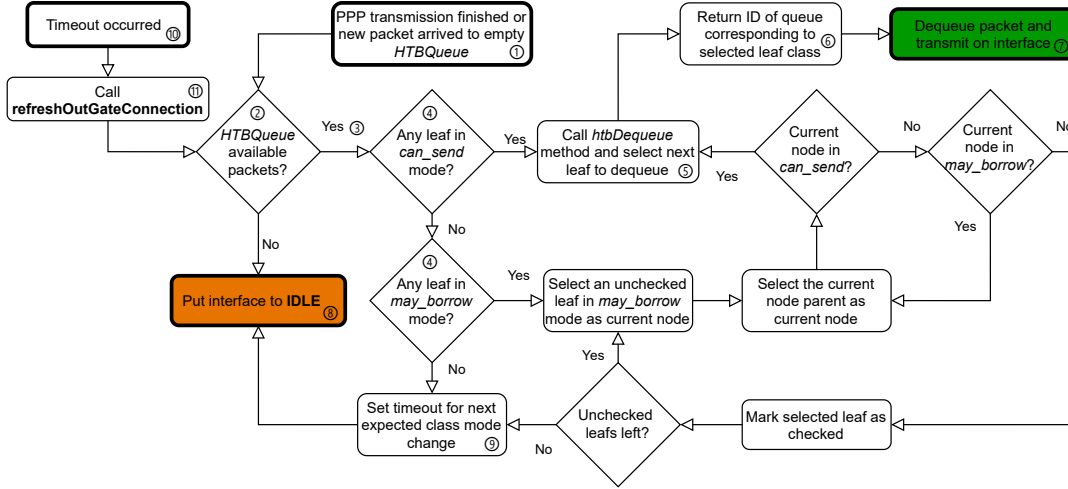
4   Implementation

We base our implementation on existing queueing compound modules from the OMNeT++ INET framework. More specifically, we extend the *PriorityQueue* to build our compound module. *HTBQueue* consists of two main modules: a classifier (*HTBClassifier*) and a scheduler (*HTBScheduler*). Packets enqueued by the HTB are stored within multiple queuing modules. Any queue from the INET framework can be used with the default being the generic *PacketQueue*. As the implementation requires link-layer specific knowledge and a slight adjustment to the utilized interface module (see Section 4.3), currently, the *HTBQueue* module can be used only with the Point-to-Point (PPP) interface available within the INET framework. We use OMNeT++ version 5.5.1[5] and INET Framework version 4.2.0[6].

4.1   *HTBQueue* Compound Module Functional Overview

Figure 2 presents the functional flow of the *HTBQueue* module operation from packet arrival until packet dequeue. Upon arrival ①, the classifier classifies all incoming packets according to its filters. These packets are placed ② into a respective queue that directly corresponds to a leaf in the HTB class structure. The classifier also informs ③ the scheduler into which queue a packet is placed. It allows the *HTBScheduler* to activate the respective classes. This way the scheduler knows that there are packets present in the queue of the respective leaf class and it can include these classes in the dequeue operation. The packets stay in their queues until the PPP interface sends ④ a *ready-to-send* signal. Upon reception of that signal ⑤, the scheduler determines the index of the next leaf (and in turn the index of the packet queue) to dequeue. The queue index is ⑥ determined based on the information that the scheduler has about the queues, their rate boundaries, class states, and the DRR resource sharing principle. Lastly, the packets are popped ⑦ from the chosen queue and sent ⑧ over the PPP interface.

---

[5]https://github.com/omnetpp/omnetpp/releases/tag/omnetpp-5.5.1, last accessed on 15th of June 2021
[6]https://github.com/inet-framework/inet/releases/tag/v4.2.0, last accessed on 15th of June 2021

Figure 3: Flow chart of the *HTBScheduler* functionality

4.2   *HTBClassifier* Module

The *HTBClassifier* module is an adaptation of the already existing *ContentBasedClassifier*.[7] The functionality of the *ContentBasedClassifier* is fully preserved and extended to allow for cooperation with the *HTBScheduler* module. With the *HTBClassifier*, filters that direct flows into desired leaf classes of the HTB structure can be specified. The packets from these flows are subsequently placed in the respective packet queues within the *HTBQueue* module.

The additional functionality compared to the *ContentBasedClassifier* allows the *HTBClassifier* to inform the scheduler into which queue (i.e., leaf class) an incoming packet is placed. This involves calling the `htbEnqueue` method of the scheduler which activates the respective leaf class if it is not already active, and thus makes sure that the leaf is considered for the dequeue operation.

4.3   *HTBScheduler* Module

We implement the core functionality of the HTB queueing discipline in the *HTBScheduler* module. The module is essentially a port of the Linux HTB source code[8]. In its majority, the *HTBScheduler* is a translation of the `C`-based Linux HTB implementation (including all the main functionalities and constructs) into `C++`/OMNeT++ compatible code. The *HTBScheduler* does not implement any actual queueing, instead it only keeps track of the states of the packet queues. It includes the actual tree structure of the HTB and implements key functions of the HTB responsible for tracking the packet enqueues, keeping the current state in each leaf queue, and selecting a leaf class queue to transmit next. The dequeue occurs when the PPP interface finished transmission, or is idle and a new packet becomes available for dequeuing.

The functionality of the *HTBScheduler* is outlined in Figure 3. After a finished transmission ①, the *PPP* module invokes the *HTBQueue* and checks for packets to dequeue ②. If packets are available ③, and any class related to any packet queue with available packets is not

---

[7]For more details, see: `https://doc.omnetpp.org/inet/api-current/neddoc/inet.queueing.classifier.ContentBasedClassifier.html`, last accessed on 13th of July 2021

[8]`https://github.com/torvalds/linux/blob/master/net/sched/sch_htb.c`, last accessed on 13.07.21

in *can't send* mode ④, the `htbDequeue` method is called ⑤ and a packet queue index is returned to the interface ⑥. Then, the packet is dequeued and transmitted ⑦. If packets are available and associated classes are in *can't send* mode, the PPP interface is put to idle ⑧ and a timeout is prepared ⑨ that informs the *PPP* module at the moment a packet is ready for dequeue. The same operations are performed when the *HTBQueue* is empty and a new packet arrives ①. The functionality of the *HTBScheduler* also required to make the `refreshOutGateConnection` method of the *PPP* module `public`. The method is called ⑪ from the *HTBScheduler* module in case there is a packet to dequeue, the timeout occurred, and the interface was already idle at that point ⑩.

The scheduler can be configured using an XML file that defines the HTB class hierarchy (as shown in Figure 1) along with the class-specific settings. Each class is represented as a separate element in the XML file, exemplarily shown for a generic class representation in Listing 1. Detailed instructions and examples for creation of such XML documents can be found in the GitHub repository.

```
<class id="className">
        <parentId> classParent</parentId>
        <rate type="int">R_assured, className</rate>
        <ceil type="int">R_ceiling, className</ceil>
        <burst type="int">burst</burst>
        <cburst type="int">cburst</cburst>
        <level type="int">classLevel</level>
        <quantum type="int">quantum</quantum>
        <mbuffer type="int">mbuffer</mbuffer>
        <priority>priority</priority>
        <queueNum type="int">n</queueNum>
</class>
```

Listing 1: XML representation of a class.

The per class settings include: assured bit-`rate` ($R_{assured}$), `ceil`-ing bitrate ($R_{ceiling}$), `burst` (Bytes that can be burst at MBR), `cburst` (Bytes that can be burst at link bitrate), `parentId` (parent in HTB tree hierarchy, `NULL` for the `root` class), `level` (level in the HTB tree structure), `quantum` (Bytes that can be sent in one DRR round), `mbuffer` (penalty time for big burst events), `priority` (only for the leaf classes therefore green in Listing 1.), and `queueNum` (corresponding packet queue index, only for the leaf class). Additionally, each class must have a unique name, the top-most class needs to be called "root", and inner as well as leaf classes must have "inner" or "leaf" in their names respectively.

5 Validation

We evaluate the functionality of our HTB implementation via three scenarios: Scenario 1 considers five UDP flows without inner classes, Scenario 2 considers five flows with inner classes, and Scenario 3 considers two flows of different priority. The simulation setup consists of two hosts, directly connected via a link having 50Mbit/s capacity. The queueing module of the PPP interface on each host is replaced with the *HTBQueue* module. The first flow starts at second 0, the subsequent flows start with an offset of 10s each. Each client sends 1500 Byte (physical layer size) packets each $100\mu s$, corresponding to a bitrate of 120 Mbit/s, and terminates after 100s. Each flow is assigned to its own leaf class in the HTB hierarchy. The configurations of the HTB and of the rates in the different scenarios are shown in Table 1. In scenario 2, class Inner 1 is a parent for flows 0, 1, and 2 and Inner 2 is the parent of flows 3 and 4. Within scenario 3, flow 0 is prioritized.

Figure 4a shows the results for Scenario 1. The plot shows the throughput of each flow on the y-axis and the x-axis represents the experiment time. As expected, each flow receives at least its $R_{assured}$ and never exceeds its $R_{ceiling}$. The remaining bandwidth is also evenly divided between flows competing for it. In more detail, we observe that in the first 10 seconds of simulation, the only active flow 0 achieves a throughput equal to its $R_{ceiling}$, that is 20 Mbit/s. Flow 1, upon arrival, is able to utilize its full $R_{ceiling}$ of 25 Mbit/s as well since the

Table 1: Validation scenarios leaf/inner class $R_{assured}$ (R_a) and $R_{ceiling}$ (R_c) in Mbit/s

	Flow 0		Flow 1		Flow 2		Flow 3		Flow 4		Inner 1		Inner 2	
	R_a	R_c	R_a	R_c	R_a	R_c	R_a	R_c	R_a	R_c	R_a	R_c	R_a	R_c
Scenario 1	3	20	6	25	9	30	12	35	15	40	—		—	
Scenario 2	3	20	6	25	9	30	12	35	15	40	20	40	30	40
Scenario 3	5	30	5	30	—		—		—		—		—	
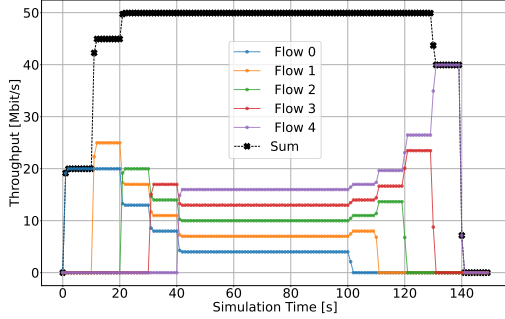
(a) Scenario 1 - no inner nodes.
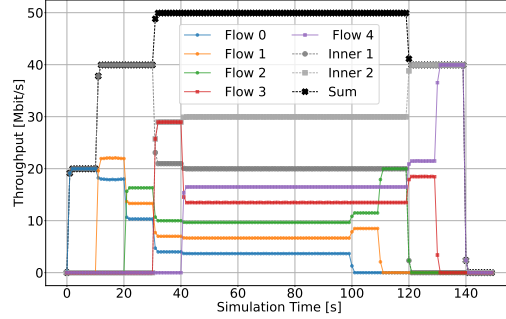
(b) Scenario 2 - two inner nodes.

Figure 4: Scenarios with 5 UDP flows.

total available bandwidth is not yet fully utilized. With the arrival of flow 2, each flow still receives its $R_{assured}$ and the remaining bandwidth is shared equally as expected. That is, flows 0 through 2 receive their $R_{assured}$ of 3, 6 and 9 Mbit/s, respectively and each flow additionally gets an even share of the remaining $50 - 3 - 6 - 9 = 32$ Mbit/s for a total of 13.66, 16.66, and 19.66 Mbit/s respectively. The same principle holds until the end of the simulation.

Figure 4b shows the results for Scenario 2. For the first 10 seconds, we see the same behavior as in Scenario 1. Between seconds 10 and 30, we see a slightly different sharing behavior, since flows 0, 1, and 2 are limited by their parent inner class. The total bandwidth share of flows 0, 1, and 2 now cannot exceed 40 Mbit/s (the $R_{ceiling}$ of their parent). This results in only 22 Mbit/s (instead of 32) being available for sharing and allows the flows to get 10.33, 13.33, and 16.33 Mbit/s, respectively. Upon arrival, flow 3, having a different parent, is able to utilize
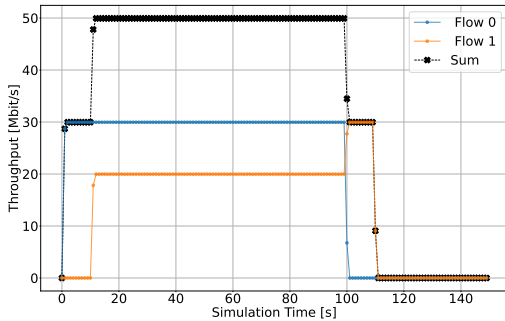

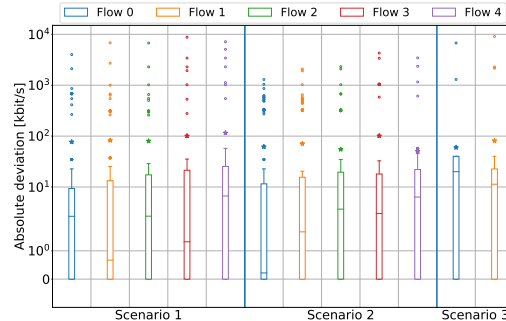Figure 5: Scenario 3 - 2 flows with priority for flow 0.

Figure 6: Absolute deviation of the obtained throughput from the expected throughput.

29 Mbit/s with the rates of flows 0 to 2 dropping to their share of the remaining 21 Mbit/s. With flow 4 arriving 10 seconds later, flows 3 and 4 utilize the 30 Mbit/s ($R_{assured}$ of inner 2), each getting their $R_{assured}$ and evenly splitting the remaining bandwidth to obtain 13.5 and 16.5 Mbit/s, respectively. Similarly, flows 0 to 2 fully utilize and share the remaining 20 Mbit/s ($R_{assured}$ of inner 1). Similarly, the sharing continues until the end of the experiment.

Figure 5 shows the results for Scenario 3. The prioritized flow 0 is able to continuously obtain its $R_{ceiling}$ and flow 1 obtains the remaining rate until flow 0 is switched off. The second flow is then also able to obtain its $R_{ceiling}$. This behavior of flow 0 is in line with the expectation, as the HTB will always first satisfy the $R_{assured}$ of leaves and then satisfy the $R_{ceiling}$ of higher priority leaves. Additionally, the end-to-end delay of flow 0 remains constant throughout the experiment at 220 ms. Flow 1 starts with an end-to-end delay of 320 ms and subsequently drops to 220 ms when flow 0 ends its transmission (the delays are not shown on the figures for the sake of clarity). This is different from the other scenarios, where we see the delay changing at every arrival or departure of a flow. Hence, priorities in the HTB can help to control the delay. Equivalent tests have been run using a TCP-based client and the corresponding results confirmed that the bandwidth is allocated as expected for TCP as well.

Several studies, as indicated in Section 2, highlight the high rate conformance of the HTB as one of its key properties. Therefore, we also evaluated the rate conformance obtained in the previously described scenarios. Figure 6 shows the absolute deviation of the per-second throughput from the theoretically expected value, for each of the flows. For better visibility, the deviation is presented with a logarithmically scaled y-axis. A star symbol represents the mean of the absolute deviation for the corresponding flow. The horizontal lines represent the median, the boxes denote the 25th (Q1) to 75th (Q3) percentile, and the whiskers denote Q1 and Q3 plus the 1.5-fold of the inter-quartile range (Q3-Q1). For example, for Scenario 1, the mean deviations for the five flows are (76.89, 82.56, 80.03, 100.99, 115.45) kbps, respectively. Relative to the assured rates of the flows, the average deviations are equal to $2.6\%, 1.38\%, 0.89\%, 0.84\%$, and 0.77%, respectively. However, there are several outliers, as we can see from the figure. These outliers occur in the transient phases, i.e., upon flows arriving or leaving the system. Furthermore, HTB guarantees the rates *on average*, so reaching a steady state takes longer than the 1s scale on which the measurements were done. The median deviations are below 10 kbps for all scenarios, except for those including priorities. The average deviation is around 100 kbps on maximum. Therefore, we can conclude that our *HTBQueue* implementation has a high rate conformance, comparable to existing HTB implementations.

6   Conclusion

The hierarchical design of HTB allows to assign each traffic flow two bandwidth parameters: an assured rate and a ceiling rate. While the first one is the minimum guaranteed rate, the second one is an upper limit until which the flow can borrow excess bandwidth. So far, OMNeT++ provided no possibility to configure such bitrate guarantees on a per-flow or per-class level. To fill this gap, we implemented the full HTB functionality as an OMNeT++ compound module, based on the implementation in the Linux traffic control (tc). It allows to classify flows and to hierarchically configure two-level bitrate guarantees on a per-class and per-flow granularity by making use of token and ctoken buckets. Our experimental validation shows the high conformance of assured and ceiling rates, as well as the fairness in sharing excess bandwidth between competing flows. We furthermore show that different priority levels allow the *HTBQueue* to interfere in the fair excess bandwidth sharing mechanism, in favor of the higher priority flow.

Acknowledgment

References

[1] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planet-lab: an overlay testbed for broad-coverage services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.

[2] H. Li, H. Zhou, H. Zhang, B. Feng, and W. Shi, "Emustack: An openstack-based dtn network emulation platform (extended version)," *Mobile Information Systems*, vol. 2016, 2016.

[3] J. Yan and D. Jin, "Vt-mininet: Virtual-time-enabled mininet for scalable and accurate software-define network emulation," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015, pp. 1–7.

[4] J. L. Valenzuela, A. Monleon, I. San Esteban, M. Portoles, and O. Sallent, "A hierarchical token bucket algorithm to enhance qos in ieee 802.11: proposal, implementation and evaluation," in *IEEE 60th Vehicular Technology Conference (VTC)*, vol. 4, 2004, pp. 2659–2662 Vol. 4.

[5] J. Vestin and A. Kassler, "Qos enabled wifi mac layer processing as an example of a nfv service," in *Proceedings of the 1st Conference on Network Softwarization (NetSoft)*. IEEE, 2015, pp. 1–9.

[6] S. Ren, Q. Feng, and W. Dou, "An end-to-end qos routing on software defined network based on hierarchical token bucket queuing discipline," in *Proceedings of the 2017 International Conference on Data Mining, Communications and Information Technology*, 2017, pp. 1–5.

[7] 3rd Generation Partnership Project (3GPP), "NR; NR and NG-RAN Overall description; Stage-2," Technical Specification (TS) 38.300, Mar. 2021, release 16.5.0.

[8] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," 01 2008, p. 60.

[9] "Inet framework - what is inet framework?" https://inet.omnetpp.org/Introduction.html, [Online; accessed 14-July-2021].

[10] M. Devera, "Hierarchical token bucket," http://luxik.cdi.cz/~devik/qos/htb/, February 2002, [Online; accessed 16-March-2021].

[11] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Transactions on Networking*, vol. 3, no. 4, pp. 365–386, 1995.

[12] A. Saeed, N. Dukkipati, V. Valancius, T. Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable traffic shaping at end-hosts," in *ACM SIGCOMM 2017*, 2017.

[13] D. Ivancic, N. Hadjina, and D. Basch, "Analysis of precision of the htb packet scheduler," in *2005 18th International Conference on Applied Electromagnetics and Communications*, 2005, pp. 1–4.

[14] D. G. Balan and D. A. Potorac, "Linux htb queuing discipline implementations," in *2009 First International Conference on Networked Digital Technologies*, 2009, pp. 122–126.

[15] M. A. Brown. [Online]. Available: ftp://ftp.wayne.edu/ldp/en/Traffic-Control-tcng-HTB-HOWTO/Traffic-Control-tcng-HTB-HOWTO.pdf

[16] M. Bosk, M. Gajić, S. Schwarzmann, S. Lange, R. Trivisonno, C. Marquezan, and T. Zinner, "Using 5g qos mechanisms to achieve qoe-aware resource allocation," in *2021 17th International Conference on Network and Service Management (CNSM)*. IEEE, 2021.

[17] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Dürr, S. Kehrer, and K. Rothermel, "NeSTiNg: Simulating IEEE time-sensitive networking (TSN) in OMNeT++," in *International Conference on Networked Systems (NetSys)*, 2019.

[18] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Transactions on networking*, vol. 4, no. 3, pp. 375–385, 1996.