Toward Scalable Docker-Based Emulations of Blockchain Networks

Diego Pennino[1,2], Maurizio Pizzonia[2]

[1]Università degli Studi della Tuscia, Dipartimento di Economia, Ingegneria, Società e Impresa, Via del paradiso 47, 01100 Viterbo, Italy;

[2]Università degli Studi Roma Tre, Dipartimento di Ingegneria Civile, Informatica e delle Tecnologie Aeronautiche, Via della Vasca Navale 79, 00146 Rome, Italy;

Abstract

Blockchain, like any other technology, needs a strong testing methodology to support its evolution. Setting up meaningful blockchain tests is a notoriously complex task for several reasons: software is complex, large number of nodes are involved, network is non ideal, etc. Developers usually adopts small virtual laboratories or costly real devnets, based on real software. Researchers usually prefer simulations of a large number of nodes, based on simplified models.

In this paper, we aim to obtain the advantages of both approaches, i.e., performing large, realistic, unexpensive, and flexible experiments, using real blockchain software within a virtual environment. To do that, we tackle the challenge of running large blockchain networks in a single physical machine, leveraging Linux and Docker. We analyze a number of problems that arise when large blockchain networks are emulated and we provide technical solutions for all of them. Finally, we describe our experience of emulating a fairly large blockchain network, comprising more than 3000 containers, for research purposes.

1. Introduction

Performing realistic experiments for blockchain networks is notoriously hard. However, reproducing a realistic blockchain network is desirable for both research and development purposes. Researchers may aim to test new protocols and make realistic measurements in laboratory. Developers of blockchain software would like to test new software versions before distribution and deployment and possibly perform what-if analysis without disrupting costly testnets.

The complexity of setting up realistic experiments stems from several factors.

1. A real blockchain network may encompass a very large number of nodes.
2. The software run by nodes is usually quite complex.
3. Communications among nodes are affected by typical properties of transport protocols (e.g., the *slow start* of TCP).

DLT'23: 5th Distributed Ledger Technology Workshop, May 25-26, 2023, Bologna, Italy
diego.pennino@unitus.it (D. Pennino); pizzonia@ing.uniroma3.it (M. Pizzonia)
0000-0001-5339-4531 (D. Pennino); 0000-0001-8758-3437 (M. Pizzonia)

4. Nodes are usually spread over the internet, which implies that any communication between them is affected by all wanted and unwanted properties of the real internet, prominently delay and packet loss.

Item 1, naturally lead us toward the adoption of a simplified and clean simulation. On the contrary, Items 2, 3, and 4 may be better reproduced by emulation environments that leverage the very same technology and software of a real production environment.

Currently, developers adopt development networks that mimic to some extent production environments in the sense that they are made of a substantial number of real nodes, are spread over the internet, and run the real software. However, this approach is costly (because many machines are dedicated to this task), time-consuming (since machines have to be managed), and unhandy (because when a new version has to be tested all machines have to be updated making it potentially unusable for other purposes). Further, sharing the network among tests of several software versions can lead to results that are hard to interpret. On the other hand, researchers mostly limit themselves to simplified simulations to reduce costs.

In this paper, we show how it is possible to run several thousands of distinct blockchain nodes on a single machine, all running realistic software and adopting real TCP/IP stack. We also show how to inject realistic delays into the emulated network.

Our approach is simple in principle but not so easy to be applied in practice. We just create one Docker [1] container for each node, each with its own ip address and let them talk each other. Practically applying this natural approach has a number of complex aspects.

Surprisingly, the resource limitation was not the main concern in our work. We were able to run more than 3000 containers, running python-based software (plainly interpreted) for blockchain nodes, occupying 350GB of RAM. We explain in the paper how CPU is not a problem provided that we accept some inflation of the execution time.

The main contribution of our work is a list of technical problems we encountered and recipes we suggest to solve them. In particular, we analyzed and addressed the limits of the Linux kernel related to launching and connecting a large number of containers. We dealt with the detrimental effect of the ARP protocol realizing a solution that completely remove this kind of traffic without resorting to a quadratic number of static ARP cache entries. We showed how to configure realistic, internet-like, delays among all pairs of nodes without inserting a quadratic number of firewalling rules. We showed how it is possible to arbitrarily reduce CPU consumption of our experimentation by inflating all the time-related parameters involved in the emulation, comprising TCP retransmission timeouts, which are hardcoded in the kernel.

The rest of this paper is structured as follows. In Section 2, we review the state of the art. In Section 3, we describe the experimentation context for which we undertaken our work. In Section 4, we show how to launch a large number of blockchain nodes on a single machine. In Section 5, we show how to connect them. In Section 6, we describe the ARP traffic problem and our AutoARPD tool to solve it. In Section 7, we describe the configurations to emulate internet-like delays among nodes. In Section 8, we describe how to arbitrarily lower CPU load. In Section 9, we discuss a practical application of the described techniques. In Section 10, we draw the conclusions.

2. State of the Art

As stated in the introduction, currently blockchain developers adopt real networks, usually called *devnets*, dedicated to experiment with new releases. The drawbacks of this approach were listed in the introduction. Before deployment on a devnet, it is likely that developers perform some small scale test of the software in a small laboratory environment, possibly using some form of virtualization. In this paper, we essentially discuss how to scale this last approach to a large number of nodes. Researchers mostly use *simulation*, that is ad-hoc software based on simplified models of the elements of the blockchain system that computes the "evolution" over time of the blockchain system. A large number of simulation systems are described in literature or freely available for download over the internet, see for example, [2, 3, 4, 5, 6, 7, 8]. The work in [9] provides a framework to evaluate private blockchains technologies based on six-layers: application, contract, incentive, consensus, node/data, network. One of the most relevant and accurate emulation/simulation approach is called *BlockPerf* and it is proposed in [10]. It emulates the network layer by taking advantage of geographically sparse real nodes and simulates the remaining layers, trying to cover as much as possible the layers mentioned above. This approach has the disadvantage of relying on many geographically distributed machines which is one of the problems of the devnet approach.

Further, it requires careful planning and deployment of nodes. This problem is even more relevant for IoT-targeted technologies like the *tangle* [11, 12] in IOTA, where emulation of a very large number of devices might be required for a realistic experimentation. The difficulty of blockchain emulation is also remarked in [13]. The authors of that survey, point out that this kind of emulation is extremely demanding in terms of resources and that no general tool exists to support it. Further, works dealing with blockchain emulation are targeted to specific technologies (usually permissioned ones). In our work, we aim at providing approaches for general blockchain emulations that help overcoming these difficulties.

Regarding peer-to-peer networks, mostly targeted to file/content storage and distribution, many simulation approaches were proposed (several are reported here [14, 15]) and rarely emulation [16].

Further, the work in [17] proposes a model for generating large realistic internet delay matrixes with the intent to support simulation of large geographically distributed systems. We use these results in Section 7.

3. Our Experimental Context

In this section, we describe the research context in which we have found the need for large scale blockchain emulation.

We tackled our large scale blockchain emulation problem in the context of researching new storage approaches for blockchain. It is well known that full nodes in a blockchain have to store the entire blockchain state. This is both a scalability problem and makes clumsy to add new full nodes to the blockchain. The work by Bernardini et al. [18] proposes to keep the blockchain state in an Authenticated Data Structure (ADS, [19]) stored in a Distributed Hash Table inspired to Kademlia [20]. The chosen ADS is a variation of Merkle Hash Tree (MHT, [21, 22]) and in particular it is a binary prefix tree equipped with the same hash linkage of a MHT. In the model adopted by [18], the blockchain state is made of the value of the accounts (one value for each address) and are conceptually stored at the leaves of this ADS (indexed by address).

Regarding the relation with the DHT, in [18] each blockchain node is also a Kademlia [20] node that stores a pruned version of the ADS, called *pADS*. As in the regular Kademlia, nodes have a (random) identifier extracted from the same space of the keys of the data. Here the keys are the addresses of the accounts. Nodes receive new blocks in broadcast. From each block, they obtain update information for all addresses updated by that block. A node *retains* only those data that are *close* to its node identifier. Here closeness is intended in Kademlia sense, that is according to the well known Kademlia xor metric [20]. Each node stores in its pADS the paths of the leaves related to addresses retained by the node, up to the root. The pruned parts are those not retained or not used by any address. For those pruned part, a node just keeps the *root hash* of the subtree (see [18]). The result is a network that keeps the blockchain state as Kademlia does (with replication) but whose response can be checked against a trusted root hash as for regular MHTs. All nodes get from the "last block" the trusted root hash to use for checking that the replies of the Kademlia network are genuine. This check involves obtaing a *Merkle proof* (i.e., the siblings of the nodes form the leaf to the root in the ADS) for the required value.

We call *block producers* the nodes that create a new block. The considered model does not constrain the kind of consensus. In this model, a block producer can perform its task without keeping any state. This is made possible by the way in which transactions are created. Who creates a transaction t asks to the Kademlia network the value and the Merkle proofs for all the addresses that t is going to change. These Merkle proofs are attached to the t. No signature is required for them so they do not need to be stored in the blockchain. Merkel proofs are taken by each block producer instead of accessing a local copy of the state or instead of asking it to the Kademlia network. With that Merkle proofs block producers can also reconstruct the part of the ADS that is affected by each block and compute the next root hash of the ADS and include it in the produced block.

This machinery is further complicated by the fact that, due to network delays, not all Kademlia nodes store the blockchain state updated to the same block. To overcome this problem all nodes that need to handle and check Merkle proofs stores a queue of the last l blocks from which to take the right root hash against which to check the replies of the Kademlia network.

In general, transactions arrive to block producers with Merkle proofs that are late by a certain number of blocks. Block producers have to update the Merkle proofs to match the root hash declared in the last block before they can be merged in a pADS. It can be proven that this can always be done securely, even if the actual algorithm may be tricky.

The primary objectives of our experimentation were to develop a realistic methodology to estimate the value l and to check in practice that all the tricky details of the model actually worked, comprising the Merkle proof updating procedure.

Provided the complexity of the methodology, realizing it in a simulator seemed not viable to us. Further, realistic delays were needed for the estimation of l. This forced us to head toward emulation instead of simulation.

4. Launching a Large Number of Nodes

To emulate a large blockchain network, for example having one Docker container for each node, we need to be able to launch a very large number of processes. Linux kernel adopts a defensive approach regarding resource usage. It has a system of limits that protects the whole system from predatory behavior of certain users or processes and that can make the whole system unusable. While this is clearly a must-have feature for multiuser systems to protect against denial of service attacks, in our case there is no reason to adopt this kind of limits. We are supposing the machine that we use to run our experimentation is just dedicated to this purpose.

The first two parameters that limit how many Docker containers we could launch are: (1) the number of open files and (2) the maximum number of processes. In Unix, we have these two limits for each user. Note that, the Docker hypervisor runs as root, as well as all the processes that are launched within the container. So we need to adjust these limits for the root user. These limits are controlled by the *ulimit* (*user limit*) settings. There exist two types of ulimit, namely *hard* and *soft* limits. This distinction makes sense in an environment in which a user may need to rise its limit temporarily and autonomously. In our case, we just rise both kinds of limits. We can permanently change both limits by editing the file `/etc/security/limits.conf`. This file is read by the PAM module pam_limits and applied upon login. Note that, in general, each blockhain node has multiple open files and user processes. The actual amount depends on the specific blockchain software that we intend to run. On the other hand, if we work on a dedicated machine there is no reason to keep these limits low. In our case, we add the lines `root hard nofile 1574415` and `root soft nofile 1574415` to increase the hard and soft limits concerning the number of open files allowed for the root user. We add the lines `root hard nproc 1574415` and `root soft nproc 1574415` to increase the hard and soft limits concerning the maximum number of processes available for the root user.

A number of kernel parameters may also affect the scalability of our experiments. These are kernel level parameters and affect the system as a whole. Again the main reasons for them is to protect the whole system from undesirable resource depletion, and many limits can be arbitrarily risen to ease our experimentation.

While the relevant parameters my depend on the specific needs of each experimentation, in Table 1 of the Appendix, we describe a list of parameters suitable for our use case. Parameter values are changed by editing the file `/etc/sysctl.conf` (applied at boot time), or changed and inspected at run time using the `sysctl` command.

5. Connecting a Large Number of Nodes

To connect many blockchain nodes among them, the natural approach is to attach each container to a virtual Linux bridge. On of the first problems encountered toward scaling to a large number of blockchain nodes is the fact that Linux bridges support a limited numbers of *ports* (i.e. the virtual interfaces among which packets are switched).

In the following. we refer to version v4.19.208 of the Linux kernel. The maximum number of ports allowed on a Linux bridge is 1024. This value is controlled by a hard coded parameter in the kernel named BR_MAX_PORTS, which is defined in the file `net/bridge/br_private.h`, as

`#define BR_MAX_PORTS (1 << BR_PORT_BITS)`, with BR_PORT_BITS immediately defined one line above as `#define BR_PORT_BITS 10`. This means that a default Linux bridge is limited to $2^{10} = 1024$ ports. If we need to connect $N > 1024$ containers, one possibility is to increase BR_PORT_BITS to obtain at least $N \leq 2^{\text{BR\_PORT\_BITS}}$ (i.e., $N \leq$ BR_MAX_PORTS). After this, we need to recompile the kernel to make the change effective. In our case, we set BR_PORT_BITS=17.

A second possibility, to get more ports without recompiling the kernel, may be to use multiple bridges connected to each other. We do not recommend this approach for two reasons: it introduces an additional emulation-level network topology which might be unhandy to manage and it requires the kernel to perform switching more than once for each packet sent among nodes.

Finally, we mention an unexpected behavior. Normally, bridges update their forwarding database and forward traffic only to known destination. This database essentially contains a list of $\langle \text{MAC}, \text{VETH} \rangle$ pairs which identify the VETH interface to be used to reach the container with that MAC address. However, when they are used with Docker, Linux bridges never update their forwareding database and packets are always broadcasted. We were not able to find documentation about this behavior, however, this is clearly a big waste of CPU. To solve this problem, we insert the specific entry for each container manually (also known as *static entry*). We can do that running, on the host, the following command for each container of our experiment, `bridge fdb add <MAC> dev <VETH> master static`. After this configuration each packet is regularly forwarded only to the correct virtual interface.

6. Addressing the ARP Broadcast Problem

When a blockchain experimentation involving a large number of nodes is started, each node starts to talk to each other through the bridge that connects all of them. To send IP packets to the right destination (which is known by its IP address), they have to be encapsulated into level two frames whose destination address are so called *MAC addresses*. Remember that we need IP (and all the above network stack) to perform a realistic emulation, hence we cannot just avoid IP and rely directly on a level-two protocol.

All regular IP networks adopt the *Address Resolution Protocol* (ARP) to obtain a corresponding MAC address when they have to send a packet to a certain IP address. ARP performs a level-two broadcast to ask who owns a certain IP address (this is called *ARP request*). The owner of the IP address sends an unicast *ARP reply*. This occurs the first time a node has to send an IP packet to a certain destination on the bridged network. The obtained association, called *ARP entry*, is stored in an *ARP cache* and it is used for the subsequent packets to be sent to the same destination. Actually, each ARP entry has a further field called *NUD* which will be explained later.

Observe that, when the experimentation starts, each node starts talking with a number of other nodes. Each of this communication gives rise to a broadcast ARP request. The kernel duplicates all these requests for all nodes connected to the bridge (i.e., $O(N^2)$ communications) and processes each of them when they reach destination. This easily increases CPU load to the point that the whole process is greatly delayed. If some sort of timeouts is in place (e.g., TCP connection timeouts or application specific timeouts), nodes can easily fail to contact each other.

In real networks ARP is fundamental to make management of these network easy, by dynamically finding MAC address corresponding to IP addresses when needed. In principle, the same result could be obtained by configuring static ARP entries (which could be configured by using the `arp` command). A static configuration would completely solve the broadcast ARP problem. However, we are now faced with the following two additional problems: (1) it is not obvious which node will contact which other, since this might be decided at running time and possibly on a random basis, (2) inserting all possible static ARP entries replicated for each node means to force the kernel to store a quadratic number of ARP entries.

Regarding the second aspect, the kernel is tuned for small ARP caches and hence their size is limited. For this reason the standard tools that ship with a Linux system provide a well-known user space solution named *arpd* (whose source code can be found in [23]). It is a user-space daemon that helps the kernel in keeping a large ARP cache on the local disk. It is clearly less efficient than a kernel-only solution but it scales well when a very large number of ARP entries is needed. We will show below that arpd turns out not to be a good solution to our problem. However, since our proposal is derived from arpd, it is useful to understand its design.

Arpd interacts with the kernel using the *NETLINK* protocol [24] as follows. First the kernel is instructed not to perform an ARP request[1] but to ask to the arpd process[2]. When a process (i.e., in our case the process implementing a blockchain node running in a Docker container) intends to send a message to a certain IP address for which no entry is present in the ARP cache, the kernel sends a request for that IP address to the arpd daemon (which should be running in the container). It searches that address in its database and, if it is present, sends back a response to the kernel, which can be modeled as a triple $\langle \text{IP addr}, \text{MAC addr}, \text{NUD} \rangle$ and can be put in the ARP cache by the kernel. The term NUD stands for *Neighbour Unreachability Detection* and identifies the state of a neighbor entry in the ARP cache. It can assume many values, comprising the following two (we omit details that are irrelevant for our context[3]).

NUD_REACHABLE It means that the entry was recently used to send an IP packet.

NUD_STALE It means that the entry was not used recently, hence before being used again a *reachability confirmation* procedure should be performed.

A first solution to our ARP broadcast problem is to initialize the arpd database with all the entries for our network. However, due to the peculiar way arpd works, this still leave in the network a large number of unicast ARP requests. In fact, unfortunately, the arpd daemon instructs the kernel to insert an ARP entry labelled NUD_STALE. In this case, the kernel has to perform a reachability confirmation. This means an unicast ARP request has to be performed before using that entry. If successful, the entry is labeled as NUD_REACHABLE and can be actually used.

[1]This is obtained by configuring the following parameter for a specific network interface: mcast_solicit = 0
[2]This is obtained by configuring the following parameter for a specific network interface: app_solicit = 1
[3]A complete description of the “neighbouring subsystem” in Linux can be found at [25]

Note that, for the vast majority of blockchain experiments the actual MAC addresses are not relevant. One may wonder if coordinating MAC addresses with IP addresses can help the translation in some way.

First note that Docker allows the user to customize the MAC address of virtual network interfaces. Further, it also has a peculiar approach to set default mac addresses when the IP address of the interface is known at the container start up. For a virtual interface whose IP address is $b_1.b_2.b_3.b_4$, with b_i in $0 \ldots 255$ ($0 \ldots$ FF hexadecimal), the corresponding default mac address for the interface is 02:42:b_1:b_2:b_3:b_4[4]. In this context, it is possible, in principle, to provide a MAC to IP translation without the need to rely to the ARP protocol.

We developed a daemon named AutoARPD, whose source code is freely available on-line [26] that interacts with the Linux kernel as arpd does, but instead of querying its database (or performing ARP requests by its own), it locally computes the corresponding MAC address from the IP address according to a configurable pattern. In this way, we get rid of all the ARP traffic and at the same time we do not need to have any special support hardcoded in the blockchain node software.

More in detail, when a node needs to contact a certain IP $b_1.b_2.b_3.b_4$ address that has no ARP entry in the ARP cache, the kernel sends a NETLINK requests for that IP address to the daemon AutoARPD daemon, which instructs the kernel to configure the following ARP cache entry

$$\langle b_1.b_2.b_3.b_4, \text{02:42:}b_1\text{:}b_2\text{:}b_3\text{:}b_4, \text{NUD\_REACHABLE} \rangle \, .$$

Note that AutoARPD labels the entry as NUD_REACHABLE so that the kernel will trust and use that entry without performing any reachability confirmation, which is not needed in our controlled environment.

The kernel may switch that entry to NUD_STALE under low usage conditions, which might lead again to an unuseful reachability confirmation procedure. To avoid this case, we can increase a related threshold to exceed the duration of our experiment
(e.g., `base_reachable_time_ms = 72000000`, an interface specific parameter).

To enable AutoARPD, the procedure is the same of enabling arpd. Clearly AutoARPD should run in each container alongside the blockchain node software. Further, the following two kernel parameters should be set for each container:
`net.ipv4.neigh.`*`interface_name`*`.mcast_solicit = 0` and
`net.ipv4.neigh.`*`interface_name`*`.app_solicit = 1`.
This can be done either during creation, by using the proper Docker option (`--sysctl`), or after the start up of the container (either by directly issuing sysctl commands or by asking AutoARPD to do that). To perform this change after startup, the container should run in priviledged mode.

7. Emulating Realistic Internet Delays

To set up a realistic experiment, it is paramount to also emulate non-ideal aspects of the internet. In particular, for our experimentation context, described in Section 3, it is important to reproduce

[4]Contrary from regular MAC addresses the prefix is not linked with a vendor. Regarding 02 (hexadecimal), the second bit set to 1 means a *locally administered address*, i.e., no ethernet card has by default a MAC address with this bit set to 1.

realistic network delays. In this section, we describe our approach to emulating realistic delays for large blockchain experiments.

Zhang et al. [17], recognized the "internet delay space" as an important aspect in the design of global-scale distributed systems. In their work, they analyze delay measured among thousands of Internet edge networks. From these observations, they designed *Delay Space Synthesizer* (DS^2). In the DS^2 project page [27], along with the software, they also published two matrices that represents realistic end-to-end internet delays. In our experiment, we used *Matrix1* (size: 3997x3997, unit: ms) as input to our system that emulates internet delays. Entries of that matrix represent one-way delays.

First, we remark that the network connecting the nodes of our blockchain is a virtual and very simple one. Packets are switched among nodes by the kernel. For this reason we can take advantage of well known tools provided by the Linux world for modifying the network behavior. We used the *Traffic Control* (*TC*) subsystem that helps in policing, classifying, shaping, and scheduling network traffic, and the *NFTables* (*NFT*) subsystem that provides filtering and classification of network packets.

Essentially, NFT is used to create a configuration that marks packets with a *class* (this is a mark for kernel use only, it does not affect what is received by the destination) and then TC is used to create a configuration to queue packets per class and to assign each queue the right delay.

Suppose to have 3000 nodes, the delay matrix has 9M entries. Even supposing to have symmetric delays (as the DS^2 matrices are) handling millions of classification rules in the kernel is clearly not feasible. We proceed as follows. First, we approximated delays quantizing them at 10ms steps obtaining 184 different latency values (delays ranges from zero to about 2 seconds, but extreme cases are sparse) and hence 184 distinct classes. For each class, we configure a NFT classification rule as follows. We select all pairs of nodes that are associated with that class and create a *set* of *unordered pairs* of IP addresses. NFT sets is a kind of data structure supported by the Linux netfilter module that allows to efficiently match a packet against a large set of addresses, or pair of addresses as in our case. The *Classification* procedure is detailed in Algorithm 1 of the Appendix.

Each class of packets has to be treated differently, and in particular their packets have to wait in a queue for the delay time associated with their class. We can do this by using TC. TC involves the use of *qdiscs* (*queuing discipline*) and *filters* to create a tree-like structure, where the root is the queue where *incoming* or *outgoing* packets are put. Afterwards, filters sort the packets into the various branches until they reach the leaves of the structure, where a final queue represents the final destination queue of the packet. In our experiment, we (arbitrarily) decided to delay packets outgoing from the bridge, adopting the same filtering/qdisc structure for all network interfaces, which is depicted in Figure 1. The semantic of the elements are as follows.

prio is a *classful* qdisc that is able to dispatch packets into an array of *band* (a sort of "channels", at most 16 bands are available). Which bands is selected for each packet depends on the *filter* attached to each band. When a packet is dispatched to that band it can be processed by another qdisc. The original purpose of the prio qdisc is to prioritize traffic. However, we essentially use it to provide a first level of sorting on the basis of the delay class of the

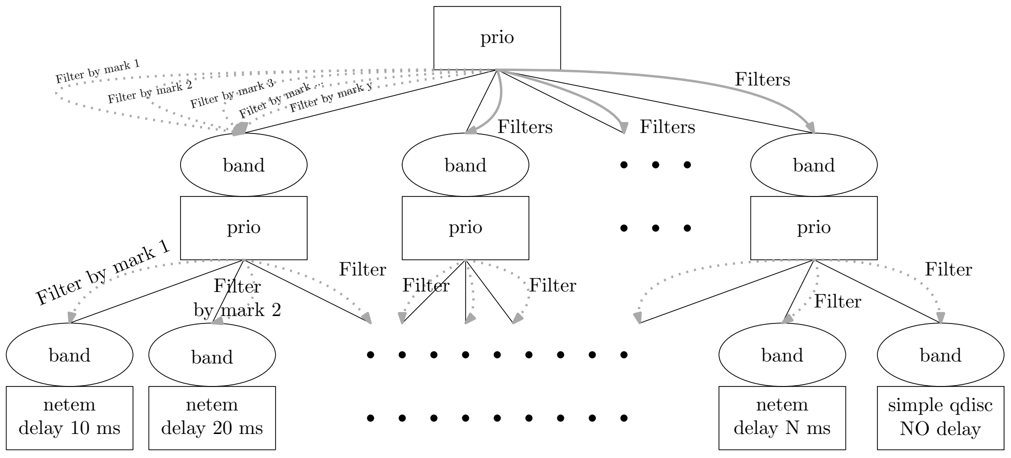

Figure 1: TC tree with two levels of prio qdisc.

packet. We use them as internal nodes of the tree.

filter is a rule attached to a band of a classful qdisc to determine if a packet has to be dispatched into that band. We use them to select the packet to the correct child of a prio qdisc at each step on the basis of the delay class of the packet.

netem is a classless qdisc that is used to add delay, packet loss, duplication and other characteristics to packets. We use them as leaves of the tree. In our experiments, we used only the delay feature.

The main idea is to define a tree with a number of leaves equal to the number of delays classes to be emulated plus one (no delay).

Since we have more than 16 classes, we have to perform two sorting levels (see Figure 1). In the first level, filters match ranges of classes. In the second level, filters match at a finer granularity only within the range selected in the first level and select the netem qdisc that will apply the correct delay. With this scheme, we can support up to 255 delay classes. Algorithm 2 of the Appendix shows the procedure *Delays* to produce the above described tree structure.

We note that, by introducing artificial delays we also affect CPU load. For an experiment with no artificial delays configured, broadcasts are ideally propagated instantaneously and CPU burst for processing propagated transactions or blocks are all close together. In this case, the propagation speed is dominated by the speed of the CPU, which is the bottleneck of the system in that specific instant of time. On the contrary, in an experiment with artificial delays, the CPU load related to a block or a transaction is more evenly distributed over time and the system is not hindered by a CPU limit, but only by the configured delays. We artificially exploit and tune this phenomenon in Section 8.

8. Time Inflation

Every physical machine has limited resources. Essentially, the bottleneck that limits the scalability of our experimentation can either be the CPU or the amount of RAM. In this section, we show a technique, which we call *time inflation*, which is extremely useful for working around CPU limits at the price of an increased duration of the experiment.

The main idea is to inflate by a factor x all the delays involved in our experimentation. The net effect is to increase by a factor x also the duration of the experiment. On the other hand, this has also the effect to let the experiment run as if the CPU power was also increased by a factor x.

To correctly apply this technique, we have to perform the following three kinds of inflation.

1. Network delays. In our case, this is very simple: we just increase by a factor x all the delays in *Matrix1* (see Section 7).

2. Timers involved in the execution of the blockchain node software. For a blockchain, the most important of them is likely to be the block time. This was easy in our case, since the software of the node was written for the experiment. When using production software, timers are likely to be configurable by command line parameters.

3. Timers involved in the generation of the load. If the objective of the experiment is to show the possibility to process a certain transaction load, this should be also spread over time by a factor x.

4. Timers involved in the execution of protocols that are under the control of the kernel. These are essentially timers related to TCP.

While for the first three items there is not very much to discuss, the last aspect is quite critical. In fact, increasing the delays of the network, TCP packets can exceed the TCP *retransmission timeout* (*RTO*). This means that the sender does not receive the ACK regarding the sent packet before the timeout expires, even if the packet is not lost and it is correctly delivered. Hence, the sender mistakenly detect a packet loss and retransmits the packet, causing an artificial increase of the network load (and hence of the CPU load).

Retransmission timeout are dynamically managed[5] by the kernel on a per-connection basis. Unfortunately, the initial RTO timeout is a value hard coded in the kernel, 1 second in the kernel version we used, which implies that during the TCP three-way handshake spourious retransmission occour if one way delays are larger the 0.5 seconds. The easiest way to change this initial RTO value is to recompile the kernel. However, in an experimental setting, it is quite unhandy to recompile the kernel every time we want to increase or decrease this parameter to match the time inflation.

We have adopted a more comfortable solution to modify the initial RTO in a flexible way based on the *Berkeley Packet Filters* (*BPF*) facility of the Linux kernel. This tool allows us to write code that is “attached” to a designated code path in the kernel. When the code path is traversed in the processing of the packet, the attached BPF program is executed. The BPF is

[5]Exponentially increasing at each retransmission.

```
1   #include <linux/bpf.h>
2
3   #ifndef __section
4   # define __section(NAME)     \
5   __attribute__((section(NAME), used))
6   #endif
7
8
9   __section("sockops")
10  int set_initial_rto(struct bpf_sock_ops *skops)
11  {
12          const int timeout = 3; // initial RTO timout in seconds
13          const int hz = 250;  // this value has to match the HZ value of the system
14
15          int op = (int) skops->op;
16          if (op == BPF_SOCK_OPS_TIMEOUT_INIT) {
17                  skops->reply =  timeout * hz;
18                  return 1;
19          }
20
21          return 1;
22  }
23
24  char _license[] __section("license") = "GPL";
```

Figure 2: BPF code to provide a custom initialization for TCP retransmission timeout. This allows us to correctly handle TCP connection setup in large scale blockchain experimentation with inflated time without recompiling the kernel.

quite efficient, since code is compiled. Further, in our case, the BPF code is executed only during connection establishment to correctly set the initial value of the RTO of the new connection. In the following, we provide some details on how to realize this solution.

The initial RTO value is defined in the file `include/net/tcp.h` of the Linux kernel source code. It is defined as `#define TCP_TIMEOUT_INIT ((unsigned)(1*HZ))`, where `HZ` is a constant stating the quantity of certain interrupts that the kernel performs per second[6] [28, 29]. Hence, `1*HZ` means 1 second. The default RTO value is computed by a specific Linux kernel function [30], which provides a hook for optional BPF code, and defaults to `TCP_TIMEOUT_INIT`. The BPF code in Figure 2 exploits that hook. This is a C code which should be compiled with specific compiler and options to produce BPF-compatible bytecode. Then, the bytecode should be loaded into the kernel. The bytecode will be compiled in native machine language by the kernel itself. The details are provided in Appendix 11.2.

While the BPF technology imposes quite a lot of boilerplate code, the important part of the code shown in Figure 2 is quite simple (Lines 11-22). Line 13 should be changed to match the HZ of the system and Line 12 to match our desired timeout in seconds (e.g. 3 seconds in the example).

[6]For our system, HZ=250. It can be found with the following command: `grep 'CONFIG_HZ='/boot/config-$(uname -r)`

9. Experimental Results

By adopting the techniques shown in this paper, we were able to run a blockchain experiment of with 3500 Docker containers, with realistic network delays. The objectives and scientific context of experimentation was described in Section 3. Each of our containers runs both the node software and the AutoARPD software described in Section 6.

The whole experiment runs in a single virtual machine configured with 40 cores and 400GB of RAM running Linux (Debian 11, kernel ver. 4.19.208) as guest operating system. The underlying hardware was equipped with two Intel(R) Xeon(R) Gold 6238R CPU @ 2.20GHz for a total of 112 cores and 1TB of RAM, with Windows operating system and HyperV as virtualization software. We were not able to reserve more than 400GB of RAM to our virtual machine due to limits of HyperV. We instructed HyperV to reserve all the RAM of our virtual machine since the beginning so that no delay is introduced by HyperV when large amount of RAM is requested.

In our specific experiment, containers are not one-to-one with nodes and nodes may have several different roles. Essentially, each node may take from two to four containers. We do not get into further details that are not very relevant for the purpose of this paper.

Overlay network topology was randomly generated. Our experiment implies creation of inter-node connections: the TCP-based are 8000 (for block propagation) and the UDP-based are 64000 (for transaction propagation and the Kademlia protocol). We have a block time of 5 seconds and a load of 20 transactions per block on average. We set the time inflation factor at x4.

We run one of our experiments for about 12 minutes of wall-clock time corresponding to 3 minute of emulated time. The occupied RAM was about 350GB which is close to the limits of our virtual machine, while CPU was below 50% on average. Hence, in our setting, the bottleneck was the RAM while without adopting time inflation, it would have been the CPU.

Starting up the whole experiment, takes quite some time: about 3.5 hours. To startup all the needed containers, it takes about 1 hour. Our node software starts waiting for a Unix signal before doing anything. In fact, to complete the setup, all the containers have to be up, but it is meaningless to start the experimentation before the setup is competed. For this reason, containers have to be stopped or put in a sleeping state in some way[7].

To setup configurations for delay emulation and bridge configuration, we need to gather information about interfaces of each container. This information gathering takes about 30 minutes. Setting up static entries for bridge forwarding database for all containers takes also about 30 minutes. We think that these very large times are mostly due to locking of involved data structures which essentially imposes serialized access.

Computing and setting of delay-related configuration, according to what we have described in Section 7, takes about 40 minutes.

In our experiment, we use two kinds of overlay networks. In both cases, we do not leave nodes to perform node discovery autonomously, but we configure artificially created overlay network routing tables. At this point, we trigger the nodes to contact neighbors, which involves setting up about 8000 TCP-based connections and about 64000 UDP-based connections. This

[7] Actually, a different approach would be to start containers with a simple shell, or other waiting process, and then start the useful processes afterward.

takes about 50 minutes. We were surprised to observe that this phase is quite demanding in terms of CPU. We were able to perform it by sequentially triggering each container to set up these connections, delaying each trigger of about half second.

After this, we trigger the nodes to start working. In particular, certain nodes are dedicated to create the transaction load of the network. These nodes are started after that other nodes are triggered to accept transactions. This part takes negligible time.

To conduct our experiment, we did not used any special tool. We used a makefile with *targets* dedicated to startup the containers, to shut them down at the end of the experiment, to invoke an external python script for the more complex settings (see below), to trigger the nodes to start working (using Unix signals), to activate some logging (for debugging purposes), to stop the activity of the nodes, and to trigger the nodes to dump specific information needed for the experimentation. Regarding the external python script, it takes care of all the settings concerning the virtual and overlay networks, i.e., configuration of static entries on the bridge, creation of the random topologies for overlay networks, communicating them to the nodes, computation of the delays for the virtual network, and application of delay-related network configurations.

10. Conclusions and Future Works

We showed an array of techniques targeted to solve several technical and methodological problems with the aim to enable scalable and realistic emulation of blockchain networks.

We described our experience of adopting these techniques in a research experimentation. While the machinery is quite complicated, our first experience is promising. We were able to run more than 3000 containers running quite complex software on a single (virtual) machine.

There are a number of possible future improvements. First, a system that simplify the setup could be very helpful for giving the possibility to other research groups to reproduce large scale emulations of this kind. Second, currently, our approach is limited to a single host. We intend to explore the possibility to distribute containers among several machines adopting technologies like Kubernates. Third, we experimented only with a software written for research purposes. We would like to experiment with real software of a blockchain node. Fourth, usually blockchain experiments require to create a transaction load. A library supporting this would be desirable. Fifth, all blockchain experiments need to setup a (possibly randomly generated) overlay topology. A library supporting this would be desirable. Sixth, performing an experiment usually involves also gathering data about the experiment itself. A library to help with this would be desirable.

References

[1] Docker Inc., Docker: Accelerated, containerized application development, https://www.docker.com/, (Accessed on 20/02/2023).

[2] I13 Blockchain - TU Munich, Dlsf: Distributed ledger simulation framework, https://github.com/i13-msrg/dlsf, (Accessed on 16/02/2023).

[3] N. Agrawal, R. Prashanthi, O. Biçer, A. Küpçü, Blocksim-net: A network based blockchain simulator, arXiv preprint arXiv:2011.03241 (2020).

[4] I13 Blockchain - TU Munich, Github - i13-msrg/evibes-plasma: Ethereum plasma proof of concept and simulator, https://github.com/i13-msrg/evibes-plasma, (Accessed on 16/02/2023).

[5] S. M. Fattahi, A. Makanju, A. M. Fard, Simba: An efficient simulator for blockchain applications, in: 2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S), IEEE, 2020, pp. 51–52.

[6] Y. Aoki, K. Otsuki, T. Kaneko, R. Banno, K. Shudo, Simblock: A blockchain network simulator, in: IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), IEEE, 2019, pp. 325–329.

[7] M. R. A. Lathif, P. Nasirifard, H.-A. Jacobsen, Cidds: A configurable and distributed dag-based distributed ledger simulation framework, in: Proceedings of the 19th International Middleware Conference (Posters), 2018, pp. 7–8.

[8] ethereum, Github - ethereum/hive: Ethereum end-to-end test harness, https://github.com/ethereum/hive, (Accessed on 16/02/2023).

[9] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, K.-L. Tan, Blockbench: A framework for analyzing private blockchains, in: Proceedings of the 2017 ACM international conference on management of data, 2017, pp. 1085–1100.

[10] J. Polge, S. Ghatpande, S. Kubler, J. Robert, Y. Le Traon, Blockperf: A hybrid blockchain emulator/simulator framework, IEEE Access 9 (2021) 107858–107872.

[11] S. Popov, The tangle, White paper 1 (2018) 30.

[12] S. Müller, A. Penzkofer, N. Polyanskii, J. Theis, W. Sanders, H. Moog, Tangle 2.0 leaderless nakamoto consensus on the heaviest dag, IEEE Access 10 (2022) 105807–105842. doi:10.1109/ACCESS.2022.3211422.

[13] S. Smetanin, A. Ometov, M. Komarov, P. Masek, Y. Koucheryavy, Blockchain evaluation approaches: State-of-the-art and future perspective, Sensors 20 (2020). URL: https://www.mdpi.com/1424-8220/20/12/3358. doi:10.3390/s20123358.

[14] A. Basu, S. Fleming, J. Stanier, S. Naicken, I. Wakeman, V. K. Gurbani, The state of peer-to-peer network simulators, ACM Computing Surveys (CSUR) 45 (2013) 1–25.

[15] M. Ebrahim, S. Khan, S. S. U. H. Mohani, Peer-to-peer network simulators: an analytical review, arXiv preprint arXiv:1405.0400 (2014).

[16] L. Nussbaum, O. Richard, Lightweight emulation to study peer-to-peer systems, in: Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, IEEE, 2006, pp. 8–pp.

[17] B. Zhang, T. S. E. Ng, A. Nandi, R. H. Riedi, P. Druschel, G. Wang, Measurement-based analysis, modeling, and synthesis of the internet delay space, IEEE/ACM Transactions on Networking 18 (2010) 229–242. doi:10.1109/TNET.2009.2024083.

[18] M. Bernardini, D. Pennino, M. Pizzonia, Blockchains meet distributed hash tables: Decoupling validation from state storage, in: P. Mori, M. Bartoletti, S. Bistarelli (Eds.), Distributed Ledger Technology Workshop (DLT 2019), volume 2334, CEUR-WS, 2019, pp. 43–55. URL: http://ceur-ws.org/Vol-2334/.

[19] R. Tamassia, Authenticated data structures, in: Algorithms-ESA 2003: 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003. Proceedings 11, Springer,

2003, pp. 2–5.
[20] P. Maymounkov, D. Mazieres, Kademlia: A peer-to-peer information system based on the xor metric, in: Peer-to-Peer Systems: First InternationalWorkshop, IPTPS 2002 Cambridge, MA, USA, March 7–8, 2002 Revised Papers, Springer, 2002, pp. 53–65.
[21] R. C. Merkle, A certified digital signature, in: Advances in cryptology—CRYPTO'89 proceedings, Springer, 2001, pp. 218–238.
[22] R. C. Merkle, Protocols for public key cryptosystems, in: Secure communications and asymmetric cryptosystems, Routledge, 2019, pp. 73–104.
[23] S. Hemminger, Github - shemminger/iproute2: Linux routing utilities, https://github.com/shemminger/iproute2, (Accessed on 22/02/2023).
[24] The Linux Kernel Organization, Introduction to netlink — the linux kernel documentation, https://www.kernel.org/doc/html/latest/userspace-api/netlink/intro.html, (Accessed on 22/02/2023).
[25] C. Benvenuti, Understanding Linux network internals, O'Reilly Media, Inc., 2006.
[26] L. Rossicone, D. Pennino, Autoarpd · gitlab, https://gitlab.com/uniroma3/compunet/networks/AutoARPD, (Accessed on 23/01/2023).
[27] Rice University, Internet delay space synthesizer, https://www.cs.rice.edu/~eugeneng/research/ds2/, (Accessed on 20/12/2022).
[28] M. Kerrisk, time(7) - linux manual page, https://man7.org/linux/man-pages/man7/time.7.html, (Accessed on 27/02/2023).
[29] M. Kerrisk, Linux man pages online, https://man7.org/linux/man-pages/index.html, (Accessed on 27/02/2023).
[30] Bootlin, include/net/tcp.h - linux source code (v4.19.208), https://elixir.bootlin.com/linux/v4.19.208/source/include/net/tcp.h#L2173, (Accessed on 03/03/2023).
[31] J. Bainbridge, J. Maxwell, Red hat enterprise linux network performance tuning guide, https://access.redhat.com/sites/default/files/attachments/20150325_network_performance_tuning.pdf, (Accessed on 30/11/2022).

11. Appendix

11.1. Kernel Parameters

Table 1 shows some kernel parameters that are (or may be) relevant for scaling experimentation with the approach described in this paper. The table shows the name of the parameter, a small description, a default value (based on Debian 11 distribution) and an example line to add to the `/etc/sysctl.conf` configuration file. We want to point out that this list is not intended to be complete. Other parameters may need to be changed, because for examples we have a different default value and/or we have a huge number of connections and/or if the messages exchanged are very large. For more details concerning Linux network performance tuning, we suggest reading the Red Hat performance tuning Guide [31].

Algorithm 1 Procedure Classify to create netfilter rules to mark each packet with its delay class.

```
Input buckets: key-value data structure ordered by key
              key = delay/latency
              value = list of pairs (source and destination)
       latency: delay to parse
Output mark (string): the latency index in the set of bucket keys, plus one.

function LATENCY2MARK(buckets, latency)
    return string(buckets.keys().index(latency) + 1)
end function


Input buckets: key-value data structure

procedure CLASSIFICATION(buckets)
    ▷ create a table with name latem with address family ip
    nft add table ip latem
    ▷ add a chain with name latem_chain in latem
    nft add chain latem latem_chain { type filter hook forward priority 0 \; }
    for all pairs $(k, v)$ in buckets do
        ▷ get the marker
        $m$ ← LATENCY2MARK(buckets, k)
        ▷ create a set of pairs (ipv4_addr, ipv4_addr) with name nodes_<m>
        nft add set nodes_<m> { type ipv4_addr . ipv4_addr \; }
        $N \leftarrow \emptyset$
        for all $(s, d)$ in $v$ do
            Add ("$s$ . $d$") in $N$
            Add ("$d$ . $s$") in $N$
        end for
        ▷ add all element to the set
        nft add element latem nodes_<m> { <N> }
        ▷ add a rule to mark with the identifier <m> a package with the source-destination
          pair present in the set nodes_<m>
        nft add rule latem latem_chain ip saddr . ip daddr nodes_<m> meta mark set <m>
    end for
end procedure
```

11.2. Detailed Procedure to Load the BPF Code into the Kernel

The following is the procedure to compile, and load into the kernel, the code in Figure 2. Suppose the code is in the file `tcp-rto.c`.

1. To compile the code, use the following command

Algorithm 2 Procedure Delays for building the tree of qdisc and filters to apply the delay at each packet according to its class.

```
Input buckets: key-value data structure
      $v$: a virtual interface
      $b$: number of bands in each prio qdisc

procedure DELAYS(buckets, $v$, $b$)
    ▷ create root qdisc for outgoing packages
    tc qdisc add dev <$v$> root handle 1: prio bands <$b$>
    ▷ create second level of prio discks
    for all $i$ form 1 to $b$ do
        $e \leftarrow \text{hex}(i)$        ▷ hexadecimal transformation of $i$ without the 0x prefix
        tc qdisc add dev <$v$> parent 1:<$e$> handle 1<$e$>: prio bands <$b$>
    end for
    ▷ create default filters
    tc filter add dev <$v$> protocol all parent 1:
              prio 20 matchall classid 1:<hex($b$)>
    tc filter add dev <$v$> protocol all parent 1<hex($b$)>:
              prio 20 matchall classid 1<hex($b$)>:<hex($b$)>
    ▷ create all qdiscs with delay and their filters
    for all $k$ in buckets.key() do
        ▷ get the marker
        $m \leftarrow$ LATENCY2MARK(buckets, k)
        ▷ identify the correct branches at depths one and two
        $f \leftarrow \frac{m}{b}$
        $s \leftarrow m \pmod{b}$
        if $s \geq 1$ then
            $f \leftarrow f + 1$
        else
            $s \leftarrow b$
        end if
        ▷ create the netem qdisc with the correct delay
        tc qdisc add dev <$v$> parent 1<hex($f$)>:<hex($s$)> netem delay <$k$>ms
        ▷ create a filter for each branch
        tc filter add dev <$v$> protocol ip parent 1:
                  prio 10 handle <$m$> fw classid 1:<hex($f$)>
        tc filter add dev <$v$> protocol ip parent 1<hex($f$)>:
                  prio 10 handle <$m$> fw classid 1<hex($f$)>:<hex($s$)>
    end for
end procedure
```

`clang -O2 -target bpf -c tcp-rto.c -o tcp-rto.o`.

2. To load the code into the kernel, use the following command
   `bpftool prog load tcp-rto.o /sys/fs/bpf/tcp-rto`.

3. Find its *program ID* using the command `bpftool prog show`. We obtain an output like the following
   ```
   ...
   169: sock_ops name set_initial_rto tag e4384b8da577553a gpl
         loaded_at 2021-04-29T15:49:03+0800 uid 0
         xlated 296B jited 186B memlock 4096B
   ```
   where the program ID is the number on the right (169 in this example).

4. The BPF program should be also attached to a cgroup using the following command
   `bpftool cgroup attach /sys/fs/cgroup sock_ops id 169`.

The BPF program can be unloaded by calling the following commands
`rm /sys/fs/bpf/tcp-rto`
`bpftool cgroup detach /sys/fs/cgroup sock_ops id 169`.

Table 1
Table showing some relevant kernel parameters.

Parameter	Description	Default	`Example line`
pty	Maximum number of pseduo-terminals. Each Docker container normally uses one of them.	4096	`kernel.pty.max = 11000`
rmem_max	The maximum receive socket buffer size in bytes.	212992	`net.core.rmem_max=2147483647`
rmem_default	The default setting of the socket receive buffer in bytes.	212992	`net.core.rmem_default=2147483647`
wmem_max	The maximum send socket buffer size in bytes.	212992	`net.core.wmem_max=2147483647`
wmem_default	The default setting of the socket send buffer in bytes.	212992	`net.core.wmem_default=2147483647`
tcp_rmem	Contains three values that represent the minimum, default and maximum size of the TCP socket receive buffer.	4096 131072 6291456	`net.ipv4.tcp_rmem="10240 87380 16777216"`
tcp_wmem	Contains three values that represent the minimum, default and maximum size of the TCP socket send buffer.	4096 16384 4194304	`net.ipv4.tcp_wmem="10240 87380 16777216"`
gc_thresh1	The minimum number of entries to keep in the ARP cache. The garbage collector will not run if there are fewer than this number of entries in the cache.	128	`net.ipv4.neigh.default.gc_thresh1=200000`
gc_thresh2	The soft maximum number of entries to keep in the ARP cache. The garbage collector will allow the number of entries to exceed this for 5 seconds before collection will be performed.	512	`net.ipv4.neigh.default.gc_thresh2=200000`
gc_thresh3	The hard maximum number of entries to keep in the ARP cache. The garbage collector will always run if there are more than this number of entries in the cache.	1024	`net.ipv4.neigh.default.gc_thresh3=200000`