Experience Track

TENSOR: Lightweight BGP Non-Stop Routing

Congcong Miao[1*], Yunming Xiao[2*], Marco Canini[3], Ruiqiang Dai[1], Shengli Zheng[1], Jilong Wang[4,5], Jiwu Bu[1], Aleksandar Kuzmanovic[2], Yachen Wang[1]
[1] Tencent, [2] Northwestern University, [3] KAUST, [4] Tsinghua University, [5] Quancheng Laboratory

ABSTRACT

As the solitary inter-domain protocol, BGP plays an important role in today's Internet. Its failures threaten network stability and will usually result in large-scale packet losses. Thus, the non-stop routing (NSR) capability that protects inter-domain connectivity from being disrupted by various failures, is critical to any Autonomous System (AS) operator. Replicating the BGP and underlying TCP connection status is key to realizing NSR. But existing NSR solutions, which heavily rely on OS kernel modifications, have become impractical due to providers' adoption of virtualized network gateways for better scalability and manageability.

In this paper, we tackle this problem by proposing TENSOR, which incorporates a novel kernel-modification-free replication design and lightweight architecture. More concretely, the kernel-modification-free replication design mitigates the reliance on OS kernel modification and hence allows the virtualization of the network gateway. Meanwhile, lightweight virtualization provides strong performance guarantees and improves system reliability. Moreover, TENSOR provides a solution to the split-brain problem that affects NSR solutions. Through extensive experiments, we show that TENSOR realizes NSR while bearing little overhead compared to open-source BGP implementations. Further, our two-year operational experience on a fleet of 400 servers controlling over 31,000 BGP peering connections demonstrates that TENSOR reduces the development, deployment, and maintenance costs significantly – at least by factors of 20, 5, and 10, respectively, while retaining the same SLA with the NSR-enabled routers.

CCS CONCEPTS

• **Networks → Transport protocols**; • **Computer systems organization → Dependable and fault-tolerant systems and networks**.

KEYWORDS

Border Gateway Protocol; Fault Tolerance; Lightweight Virtualization

ACM Reference Format:
Congcong Miao[1*], Yunming Xiao[2*], Marco Canini[3], Ruiqiang Dai[1], Shengli Zheng[1], Jilong Wang[4,5], Jiwu Bu[1], Aleksandar Kuzmanovic[2], Yachen Wang[1]. 2023. TENSOR: Lightweight BGP Non-Stop Routing. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3603269.3604852

1 INTRODUCTION

The Border Gateway Protocol (BGP) [40] is the solitary inter-domain protocol that stitches tens of thousands of autonomous systems (ASes) up. Thus, managing BGP connections is crucial for AS operators. It is also hard: the number of AS peerings is growing rapidly because of the growing number of ASes [10] and the trend of Internet flattening [15]. Manually configuring thousands of border routers is not practical for any AS operator, nor does it meet the flexibility and availability requirements of modern networks [28]. Instead, many major AS operators have virtualized their network gateway, *i.e.,* introducing a software-defined control plane to scale up and centralize the management of the BGP routing policies [13, 21, 22, 44, 47, 54].

Nevertheless, existing BGP virtualizations leave a critical problem of failure handling unresolved – the non-stop routing (NSR). NSR aims to ensure the application-level BGP status and functionality on both sides of the peering ASes are not affected by single-point failures at the application, router, or network level. For example, TCP connection failures, software/hardware failures of the border routers, etc.

NSR is of critical importance because a border router will consider the link to be broken when any of the low-level failures occur, and withdraw all its BGP routes to the peering AS. That means no packets will be routed through the link before the BGP session re-establishes and all the BGP routes recover. The impact is large during this period of time. For example, the average throughput of Tencent Cloud and its peering ASes is 37 Gbps per link. As it may require several minutes for the BGP to recover even if peers reconnect right away [13, 26], a single point of failure may affect over 1 TB of live traffic. The estimated cost due to BGP failure is at the scale of multi-million dollars each year at Tencent Cloud.

Unfortunately, BGP NSR is not compatible with BGP virtualization because BGP is based on the connection-oriented TCP protocol. More concretely, to replicate incoming/outgoing TCP packets so that the TCP connection can be recovered at any time, most of the existing NSR solutions heavily rely on OS kernel modifications. The dependency on kernel modifications is in conflict with lightweight virtualization techniques such as containerization. Kernel-level solutions introduce high development, deployment, and maintenance costs for the AS operators. An alternative is to adopt replication of virtual machines (VMs) [43]. However, replicating the VMs to support NSR is too costly and not feasible in practice.

To tackle the above problem, we present TENSOR (<u>Te</u>ncent Cloud's <u>N</u>on-<u>S</u>t<u>o</u>p <u>R</u>outing). At the core of TENSOR's approach to BGP NSR is a novel design for primary-backup BGP replication in virtualized containers that builds atop a *kernel-free packet*

[*]Both authors contributed equally to the paper.

replication mechanism. Packet replication is realized using only Linux built-in system hooks in the protocols stack and does not require kernel modifications; it only requires minimal modifications to the existing BGP codebase. Meanwhile, TENSOR improves system reliability by reducing the number of system components compared to existing NSR-enabled routers and by decoupling their inter-dependency.

The downside of kernel-free packet replication is that it introduces a delay to TCP acknowledgment packets, which negatively affects TCP and application performance. To minimize its impact, we leverage the parallelism from lightweight virtualization to distribute the load and hence decrease the per-process overhead of replicating packets without altering BGP semantics. The lightweight virtualization also simplifies the BGP and NSR management. Moreover, we demonstrate that TENSOR provides a natural way to tackle the split-brain problem, which is a notorious source of problems for previous primary-backup NSR solutions. This is because the lightweight virtualization allows us to partition the BGP process into a fine-grained, isolated participants, which offer greater robustness to tolerate single-point failures.

We conduct extensive experiments in a production-level testbed to demonstrate the benefits of TENSOR (§ 3). Our results show that while enabling BGP NSR and BGP virtualization concurrently, the kernel-modification-free ("kernel-free" for short below) packet replication only introduces an overhead of less than one second to receive tens of thousands of routing updates from a peering AS, compared to open-source BGP implementations which do not support NSR. And the lightweight virtualization design of TENSOR keeps the overhead the same regardless of receiving from one or up to hundreds of peering ASes. Meanwhile, TENSOR has barely any performance degradation when generating and sending out routing updates to its peers. While evaluating the failure recovery, we find that the NSR of TENSOR costs less than 10 seconds to migrate the BGP process with zero link downtime for any failure scenarios. The migration time is reduced by a factor of 2 to 25 compared to the link downtime of open-source BGP implementations.

When compared to the NSR-enabled routers, TENSOR achieves the same service-level agreements (SLA) regarding the failure recovery time and link downtime. Meanwhile, TENSOR outperforms the NSR-enabled routers in the development, deployment, and maintenance costs. From our operational experiences, TENSOR shortens the development duration by a factor of 4 and the labor cost by a factor of 20. Further, TENSOR also reduces the deployment costs by a factor of 5 and the maintenance costs by a factor of 10. TENSOR has been deployed at Tencent Cloud and achieved zero link downtime for over two years.

This work does not raise any ethical issues.

2 BACKGROUND

2.1 BGP and Non-Stop Routing

Today's Internet resides at the interconnection of tens of thousands of ASes. BGP is the converged solution to stitch the ASes together. BGP establishes sessions between gateway routers, *i.e.,* routers at the border of the ASes, to communicate with other gateway routers of peering ASes to exchange necessary routing information and build the routing tables for destinations beyond the AS that they belong to.

Nevertheless, BGP sessions are subject to being interrupted by failures. There are mainly three points of failures that may lead to the breaking of the BGP sessions: application, router, and the network. Application failures may occur when a program bug is triggered [18, 58] or when erroneous configurations are imported [23]. To recover the application failure, it may take tens of seconds to restart the BGP application [53]. Next, router failures may occur due to OS-level software bugs, or hardware failures such as RAM or disk failures. The time cost to recover from router failure is undetermined: it may take a few minutes to reboot the router, or it may take days for multiple on-site and vendor engineers to fix the issue. Lastly, network failures are the most common type of failures. They may occur when the physical link is damaged or broken, or when the connection between transceivers and cables is loose or inactive, etc. Some failures are permanent and require on-site engineers to repair or replace the physical devices while others are temporary, e.g., a loose contact of the cable may result in network jitters that last for tens of seconds but will recover on its own.

There are multiple methods to detect a BGP failure. To begin with, each pair of gateway routers running BGP maintains a TCP connection and exchanges routing information when network changes occur and keep-alive messages to maintain the connectivity to the peering router, and hence the peering AS. When keep-alive messages time out, when the TCP connection breaks, or when the accompanying link detection protocols such as Bidirectional Forwarding Detection (BFD) [24] report a link failure, gateway routers consider the connectivity to their peering ASes to be failed, and then withdraw all the routes to the peering AS.

Before the BGP session re-converges, all the packets that are supposed to go through the link will be re-routed or, in the worst case, dropped. This may mean a huge amount of packet drops, especially when the link is busy. For instance, a one-minute one-link downtime will impact 277 GBs of live traffic in Tencent Cloud (see § 4.4). This results in a cost of several million dollars each year for SLA violations.

Therefore, it is essential to keep the gateway router available and avoid the disruption of BGP sessions between peering gateway routers, either proactively, e.g., when a software upgrade is needed and routers need to be rebooted, or passively, e.g., single-point failure of gateway routers [53]. The first scenario is easy to handle because of its predictable nature. BGP includes a mechanism named `graceful restart` [42] that proactively notifies the peer to keep the routing policies unchanged during the BGP connection downtime. This is also referred to as non-stop forwarding (NSF). Nevertheless, the latter scenario is much harder to handle because it is caused by unpredictable failures – the BGP process cannot notify its peer before the failure happens. The ability to keep the BGP session from being disrupted by unpredictable failures is referred to as non-stop routing (NSR). In this paper, we focus on the BGP NSR.

2.2 NSR Challenges

Previous studies [29, 51, 56] have found that replication is key to realizing BGP non-stop routing. Figure 1 presents a high-level

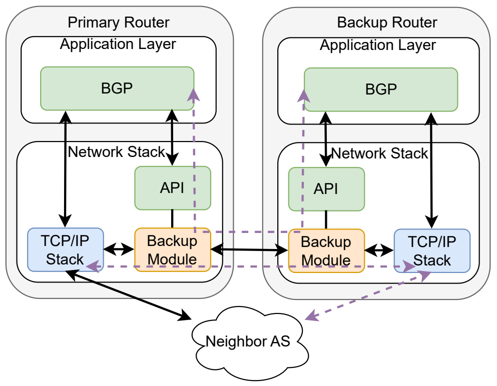


Figure 1: Overview of the generalization of existing NSR solutions.

overview of common NSR solutions. Typically, when the primary router exchanges BGP messages with its peers, it needs to back up both the application-layer state, *i.e.,* BGP session information, and the transport-layer state, *i.e.,* the TCP connection state. The primary router sends both application-layer and transport-layer states to the backup router through a (highly available) backup module. The common practice is to implement the backup module as a kernel module. The backup router has a symmetric architecture as the primary router, where its backup module receives the data and feeds it into the TCP/IP stack and BGP process for replicating the state. If the primary router experiences a failure, the backup router takes over the BGP session after a failure detector triggers. Given that the state of the TCP/IP stack and BGP session is replicated, the process is transparent to the remote BGP peer.

Replicating data adds overheads. Early works proposed to modify the TCP protocol [48–50], which however involves lengthy standardization processes and is not practical. In contrast, solutions that ensure the *transparency* to the peering routers are more favored.

A line of work [16, 17, 29, 51, 56] is to seamlessly migrate the TCP connection by introducing wrappers at the TCP/IP stack, which corresponds to the backup module in Figure 1. The wrappers replicate every egress packet before sending it out and every ingress packet before passing it to the application layer. However, the (synchronous) replication latency results in a *delayed acknowledgment*, which negatively affects the performance. Existing work realizes the wrappers as a loadable OS kernel module. Wrappers at the TCP/IP stack are also favored by hardware vendors, e.g., Comware router [1].

Another line of work supports non-stop routing through virtual machine replication [19, 20, 53]. While these solutions provide better scalability than packet replication, they are less performant because of the large amount of state to be replicated and the associated overheads.

Moreover, existing solutions have unresolved issues and encompass complicated designs that lower the system reliability. Below, we summarize the challenges lying ahead of realizing a practical and effective BGP non-stop routing system in today's network environments.

System reliability. Existing BGP NSR solutions often require modification of the OS kernel and encompass designs that couple TCP/IP stack, BGP process, device management, backup modules, and other components. The numerous components inevitably result in a lower level of system reliability.

Virtualization support. Following the heavy dependence on kernel modifications or insertions of kernel modules, existing BGP NSR solutions are left behind in today's network evolution — a contrast with a trend towards network virtualization [21]. For instance, Tencent has virtualized the cloud gateway with the disaggregated software-defined router (DSR), which decouples the control plane and data plane [47]. Meta and Google, among other cloud providers, have also taken similar approaches [13, 21, 44, 54]. However, virtualization is hard to achieve for most of the BGP NSR solutions because kernel modifications are involved.

Performance. BGP NSR requires replicating a large amount of states. But the availability guarantees of existing solutions are often at the cost of performance. For a major cloud provider such as Tencent, the number of peering ASes is over 6,000 and the total BGP table size is over a million. Performance – in particular throughput – is hence critical to ensure the functioning of the cloud gateway.

Split-brain problem. Split-brain refers to an error state where the primary and backup servers (routers in BGP NSR) mistakenly consider that the other server is down and they both execute as the primary server. This is often due to misconfigurations, erroneous communication, or asynchronous communication between the two routers. The split-brain issue results in an availability inconsistency. In the BGP scenarios, it translates to disruption of BGP connections and in turn, packet losses. The split-brain issue exists for any replicated primary-backup system design, e.g., two routers in Figure 1; replicated state machine approaches [35] address it via a consensus protocol at the cost of a larger replication group (*i.e.,* $2f+1$ for f crash failures).

3 SYSTEM DESIGN

3.1 Kernel-Free Packet Replication

3.1.1 Design Overview. To better satisfy the virtualization requirements [44, 47, 54], we propose a new method for BGP NSR based on a kernel-free[1] packet replication mechanism while minimizing the data to be replicated. Our method is transparent to the remote endpoint. Figure 2 illustrates the overview of our proposal.

Conceptually, our proposal is straightforward. When a message is received by the primary BGP process at the application layer, we replicate the message to a highly-available distributed database — Redis [9] is used in our case. The backup BGP process reads and processes the replicated messages to stay up to date with the primary. Meanwhile, the primary also performs the regular processing of BGP messages and tracks TCP connection state. However, since the primary receives messages via the regular TCP/IP stack, this entails normal packet processing, which generates TCP ACK packets; the design must deal with this aspect in a careful way to avoid replication inconsistencies.

An inconsistency would occur if the primary crashes after the TCP ACK packets are sent out but before BGP messages have been correctly replicated. This is because the remote endpoint will clear its TCP sending buffer once it receives the TCP ACK packets. Thus, the backup router will not be able to observe the messages as they

[1]In the sense that it requires no kernel modifications.

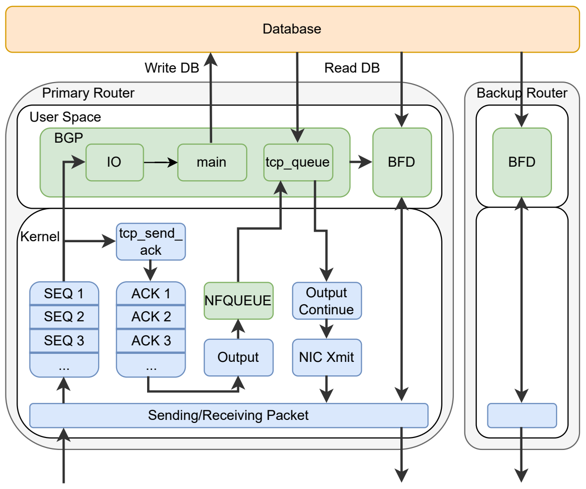


Figure 2: Overview of kernel-free packet replication. To avoid kernel modifications, we leverage the existing hooks of the Netfilter Linux kernel module.

are acknowledged by the primary but not properly replicated, and it will cause a BGP recovery failure.

To avoid such failures, we propose to use a Linux built-in system module called Netfilter to intercept the outgoing TCP ACK packets and delay their transmission until the messages are known to have been replicated. Hence, this effectively establishes a synchronous replication channel. Synchronous replication ensures consistency and correctness between the primary router and the database, and by extension, between the primary and the backup router.

Importantly, our proposal avoids any modifications to the kernel. The only changes are at the application level: the BGP process needs to (i) replicate the messages to the distributed database, and (ii) delay the TCP ACK packets to ensure consistency. This kernel-free design thus allows to easily virtualize the BGP NSR system.

3.1.2 Technical Details. **Intercepting packets**. To intercept the TCP ACK packets, we rely on the Netfilter's OUTPUT hook [2], which is triggered by any locally created egress packet hitting the network stack. We re-route the intercepted packets to an NFQUEUE target delegating for `tcp_queue`, a thread of the primary BGP process at the application layer.

User-space primary BGP. The common practice for BGP implementation spawns three threads: a main thread, an IO thread, and a keepalive thread [5]. The main thread handles the high-level processing of BGP, such as maintaining and updating the BGP routing tables, generating routing information updates, etc. In particular, the main thread may maintain multiple BGP routing tables using the virtual routing and forwarding (VRF) technique [41], where each VRF usually corresponds to a peering AS (we discuss more complicated settings in § 3.2.4). The IO thread is responsible for receiving and sending messages for the main thread and the keepalive thread, whereas the keepalive thread is responsible for processing BGP keepalive messages. In fact, there are 5 types of BGP messages: open, update, notification, keepalive, and route-refresh [40]. All the types of BGP messages except keepalive are executed in the main thread. Keepalive is implemented as a separate thread from the main thread to ensure that the keepalive messages are not blocked by other messages or routing table operations, which may result in connection failures.

Besides the existing three threads, TENSOR introduces another thread named `tcp_queue`. This thread accepts the TCP ACK packets re-routed by Netfilter and holds them in a FIFO queue until it confirms that the messages are properly replicated.

Matching ACK numbers. `tcp_queue` releases any held-up TCP ACK packet, *i.e.,* allowing it to be put on egress, whenever the corresponding message has been properly replicated in the database. The corresponding message to a TCP ACK packet refers to the BGP message that caused that TCP ACK packet.

The technical challenge is to establish a mapping between a TCP ACK packet and the BGP message that generated it. Since the main thread only receives BGP messages via the TCP byte stream from the Linux socket interface, it has no visibility into the TCP header information. We infer TCP-level information as follows. At the start of a BGP connection, *i.e.,* when the socket connects successfully, we use the TCP_REPAIR option to obtain the initial SEQ and ACK numbers along with other necessary information including the values of protocol options using TCP_INFO option. When receiving a message, the main thread infers the current ACK number by adding the initial SEQ number and the cumulative size of all the previously received messages. When the main thread replicates the received BGP message, it also writes its inferred ACK number to the database, which allows `tcp_queue` to match the corresponding ACK packets.

Outgoing BGP messages. Similar to the case of incoming BGP messages, we replicate all the outgoing BGP messages generated by both the main and keepalive threads and the corresponding TCP status in the database before sending them. This step is needed to ensure that the backup router can recover the TCP sender buffer once it takes over the connection.

Delayed sending is simpler than delayed acknowledgment: the main and keepalive threads execute a database write operation before handing over any message to the IO thread. Unlike with the `tcp_queue` thread, no database read operations are needed.

Another difference between incoming and outgoing message replication is that while in the former case only the main thread writes messages to the database, in the latter case both the main and keepalive threads write to the database. Race conditions may occur when both threads write to the database at the same time. One solution is to merge the keepalive thread to the main thread. However, this might lead to a situation where a keepalive timeout could happen and the BGP is disconnected because the keepalive is hindered by too many other types of messages. To avoid such BGP connection failures and to follow the common practice of BGP multi-threading, we choose to implement a per-message lock to support multi-threading read and write from/to the database. Note that the ordering of the database operations is only required for messages within a BGP connection but not required for messages across different BGP connections.

Storage overhead. An additional problem arises in that the volume of replicated messages for each BGP connection will constantly grow. Since BGP connections are usually long-lived, the storage overhead will become a burden for TENSOR. One observation is that when the BGP process has received a complete BGP message

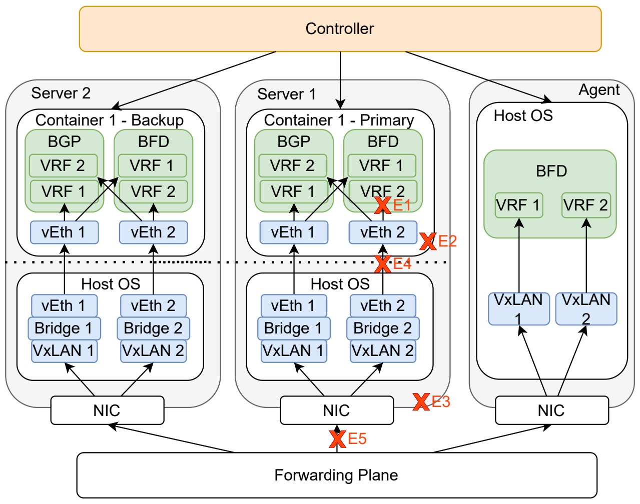


Figure 3: Overview of TENSOR's virtualized architecture.

and has updated the BGP routing table based on it, the processed messages are no longer needed. Following this observation, we remove the replicated messages that have been applied to routing tables from the database to avoid the steadily increasing storage burden. Hence, the total size of messages replicated in the database will not be more than what is needed by one complete BGP message. In general, it is less than 64KB per BGP connection.

BGP routing tables. Since BGP messages are now discarded from the database, in order to restore the BGP routing tables at the backup router, the main thread backs up in the database the BGP routing tables whenever they are updated. This step also has the advantage that the backup BGP router does not need to replay all previous BGP messages to reconstruct the state of the BGP routing tables.

3.2 A Lightweight Virtualized Approach

One drawback of the packet replication design is performance – the TCP throughput is impacted because the delayed acknowledgment inevitably increases the network latency. In Tencent Cloud, one gateway router is expected to support over one thousand BGP connections. Thus, synchronously replicating all messages will incur non-negligible overhead to the system. In this section, we present our containerized approach to mitigate this issue.

3.2.1 Design Overview. To minimize the performance degradation, we propose to spread out the BGP connections using containers – a lighter-weight virtualization technique. More concretely, we include one BGP process in one container where one BGP process can support a few peers using VRF [41] – a technique that allows multiple instances of a routing table to co-exist. Each BGP process should be running in a pair of containers on different host machines – one serves as the primary router and the other serves as the backup router. In this design, each container has much less data – only for one BGP process – to backup. In this way, we naturally leverage the multi-threading parallelism provided by containerization without the need to carefully tweak the BGP program which takes extra effort.

Figure 3 illustrates an example of our containerized BGP design. Specifically, container 1 has two instances distributed at server 1 and server 2, respectively. Its instance on server 1 serves as the primary router whereas the other instance on server 2 serves as the backup router. Inside container 1, there are two processes – BGP and BFD. Both BGP and BFD processes include two VRFs (VRF 1 and VRF 2) which correspond to two ASes, respectively. When the primary container fails, the backup container needs to take over the conversation with the peering AS for both BGP and BFD processes without letting the remote end-host acknowledge the switch. Figure 3 also includes another "agent" server, for which we will include more details in § 3.3.

Additionally, dispersing the BGP connections with containerization will reduce the scale of impacts from exceptions. Under the traditional BGP setup, all the other running BGP connections on the same machine will be impacted if one of them fails, e.g., when we need to upgrade the software version, when a BGP connection crashes because of a configuration change, etc. This is because, in essence, all of the BGP connections – regardless of whether VRF is used or not – are running within the same process context. But in our proposal, the impact of such failure is limited to the BGP connections inside the same container.

Last but not least, containerization also helps to speed up the system booting. In Tencent's cloud gateway, the number of configurations, e.g., existing routing tables, on a gateway router will often reach the scale of ∼10K or ∼100K. When the system boots or reboots, it needs to load all the configurations into memory. This may take up to ∼20 minutes. Some optimizations are proposed to speed up the configuration loading, e.g., converting the configurations into loadable binary files or implementing a parallel reading spawning multiple threads. Yet, they take a lot of engineering efforts and complicate the BGP program. But with the containerized approach, the number of BGP connections is much smaller in each container, and so is the number of configurations. With the parallel nature of containers, we achieve a faster system booting without any optimizations mentioned earlier –the loading time is shortened from ∼20 minutes to ∼20 seconds.

3.2.2 Controller Design. With containerization, the minimum operation unit becomes a container instead of a physical router. A new controller is needed to organize the resources and manage the containers on all the servers.

As shown in Figure 3, the controller directly manages the containers on all the servers. We implement the controller based on Tencent Kubernetes Engine (TKE) [11]. To ensure robustness, the controller is logically centralized but physically distributed. The controller connects to the containers using gRPC [7]. The controller is responsible for not only the container orchestration, e.g., container provisioning, deployment, scaling, etc., but also the application-layer management, *i.e.,* mapping the BGP connections to the containers and monitoring the health of BGP processes. In contrast, the host server does not acknowledge the mappings between the BGP connections and the containers but only focuses on local routing control (VXLAN) and failure handling (see § 3.3).

3.2.3 Underlay Network Design. On a physical router, each VRF routing table corresponds to a VXLAN. Thus it seems to be a natural choice to include the VXLAN in the same container as the VRF's parent BGP process. Nevertheless, we choose not to do so but leave the VXLAN on the host, as illustrated in Figure 3. This is because if

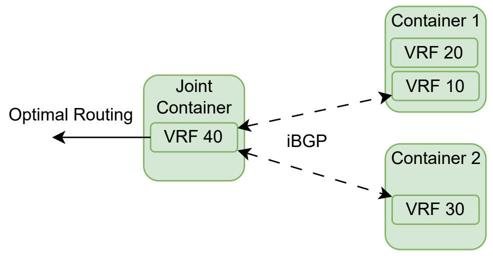


Figure 4: Joint BGP container for global optimal routing.

the VXLAN is containerized, it would require the forwarding plane – composed of commodity switches – to learn the mappings among the physical servers, the container instances, and the VXLAN before being able to forward the packets accordingly. However, this messes up the dependencies between different components of the cloud gateway and contradicts the design principle of separation of forwarding and control planes in DSR. Instead, we bind each VRF to a pair of virtual Ethernet interfaces (vEth) – one inside the container and one on the host – and use a bridge to connect the VXLAN to the vEth on the host. In this way, the VRF is bound to the VXLAN, and the containerization of the VRF is transparent to any network components or middlewares outside the host.

3.2.4 Splitting the BGP. The conventional approach for border routers involves each border router connecting to multiple ASes to establish connectivity and exchange routing updates, enabling the dissemination of network reachability information. Further, the internal BGP (iBGP) is used to synchronize information between different border routers. However, TENSOR revolutionizes this setup by splitting the BGP routing of one border router into multiple containers, where each container hosts only one BGP process and supports the minimum number of BGP connections necessary. Such BGP splitting may impede optimal routing decisions, as they often depend on information from multiple border routers or a larger number of BGP containers.

To achieve the desired BGP routing split while retaining the optimal routing decisions, we rely on a splitting strategy based on ASes and clients and the creation of joint containers. As a general rule, each BGP container is divided in such a way that it handles one AS or one client, ensuring a clear separation. However, there are exceptional cases where certain global information needs to be shared between two separate BGP containers. In such scenarios, we introduce an additional joint BGP container that synchronizes the required information between these separate containers with the iBGP protocol. Figure 4 illustrates this arrangement, which creates dependencies among the containers. Whenever any of the dependent BGP containers is updated, the joint BGP container must be updated accordingly, and vice versa. This enables the joint BGP container to make optimal routing decisions by leveraging the shared global information.

3.3 Addressing the Split-Brain Problem

With the combination of kernel-free packet replication and containerization, TENSOR is able to mitigate the three challenges of BGP NSR systems – system reliability, virtualization support, and performance (see § 2.2). It still leaves a critical problem that needs to be tackled – the split-brain problem.

3.3.1 Design Overview. The split-brain problem has been a pain for the existing NSR solutions because their design only involves the primary and the backup routers – a two-node system will always suffer from split-brain because a majority consensus cannot be reached when either node incurs a problem [35]. Solving the split-brain problem requires adding at least a third "witness" node. Yet, this adds cost and complexity to the system to maintain the witness node.

It turns out that, unexpectedly yet reasonably, containerization is also a cure for the split-brain problem. It naturally introduces additional nodes to the system, e.g., the controller. Below, we first explain the liveness probe in our system as the basis for any failure localization and handling. Next, we discuss in detail the failure handling in various scenarios.

3.3.2 Liveness Probe. **Between the peering ASes**. Bidirectional Forwarding Detection (BFD) is a lightweight protocol designed for fast link failure detection [24]. Implementing BFD for BGP brings a significantly faster reconvergence time and is the common practice for liveness probes between peering ASes [25], and we follow this practice. Each BGP process connection is associated with a BFD process. In TENSOR, it means that each container runs one BFD process. BFD also supports VRF where its VRFs are one-to-one mapped to the VRFs in the BGP process.

The BFD process will report the link failure (of the corresponding VRF) to the BGP process through inter-process communication (IPC). In TENSOR, a link failure will be detected if any of (*i*) the container, (*ii*) the host machine, or (*iii*) the link to the peering AS fails. While failure (*iii*) indeed suggests that a restart of the BGP is needed, the other two failures in addition to (*iv*) the failure of the BFD process, should not be acknowledged by the BGP router in the peering AS under NSR. That is to say, TENSOR needs to keep sending the BFD keepalive messages either during the rebooting of the primary BGP process/BFD process/container (see § 3.3.3) or during the BGP migration to the backup container.

A strawman solution is to keep both the primary and backup containers alive. Because the BFD keepalive messages are simple and stateless, it is viable for both the primary and backup containers to receive and reply to the incoming BFD packets. In this way, the BFD process of the remote end-point will not detect a failure caused by the primary BFD process/container/host machine errors.

Nevertheless, there are two major drawbacks of this approach. First, it requires additional management of the BGP process in the backup container. This is because while the BFD allows both primary and backup processes to receive and reply to messages concurrently, the BGP does not since it exchanges stateful packets and it is based on TCP: the TCP connection will be interrupted if both the primary and backup processes send packets to the remote end-point concurrently. Therefore, the controller needs to micro-manage the processes inside each container to ensure that the BGP process in the backup container is not active when the primary container works.

An alternative solution is to configure the forwarding plane to drop the packets sent from the backup container. In this way, the remote end-host receives only one reply while the backup container believes that it is talking to the peering AS. A benefit of this design is that the backup container can resume the BGP connection immediately after the primary container fails. However, this solution requires the forwarding plane to identify which is the backup container and the status of all containers. This disrupts the dependencies between different components and defies the design principle of DSR [47].

Either way, it requires the backup container to be alive at all times, and this consumes computing and memory resources. We instead propose an energy-saving BFD relay solution that supports a "cold-start" of backup containers, *i.e.,* the backup container needs to be created and booted, or a "preheat start" of backup containers, *i.e.,* a few backup containers (fewer than primary containers) are kept alive but they need to download the BGP and TCP status from the database.

Our design builds based on the fact that the BGP connections can survive not sending/receiving packets for a short period of time with the help of the TCP retransmission mechanism [39]. On the other hand, BFD will not survive this pause because its timeout interval is usually less than 1 second [14] – 100 ms ×3 is adopted in Tencent's cloud gateway. To avoid this problem, we introduce a third node – an agent server – to be the BFD relay during the short rebooting/migration interval. As shown in Figure 3, the agent server runs duplicate BFD processes for all the containers on other machines. The agent server replaces the role of the backup container for sending the BFD packets concurrently with the primary container. When the primary container is down, the agent keeps sending BFD keepalive messages so the remote end-host does not acknowledge the local failures.

Since the task on the agent server is simple and lightweight, we do not containerize its BFD processes. At a high level, the agent is weakly coupled with the other components of the system including the primary/backup containers and the controller: in normal times, the failure of the agent, e.g., server downtime, will not affect the normal TENSOR functioning. The only failure that could affect the BGP NSR is when both the primary container and the agent are down at the same time while the backup container is not pulled up yet. The chance of such failure is minimal. And it cannot be handled by any existing NSR solutions where only single-point failure is considered.

Inter-component measurements. In TENSOR, we implement various measurements between the system components. First, the controller will set up gRPC channels to all the containers, their host machines, and the agent server. The gRPC channels will send gRPC heartbeats for health monitoring. Next, the agent server will send Internet protocol service level agreement (IP SLA) probes to the containers and their host machines. Further, the host machines will also send IP SLA probes to each other to monitor the inter-connectivity. The agent server and the host machines will report their measurement results to the controller through the gRPC channels.

3.3.3 Failure Location and Handling. **Application failures**. The BGP/BFD application process may fail due to misconfigurations, bugs, insufficient resources, etc. This corresponds to E1 in Figure 3. To locate such failures, we conduct continuous monitoring and periodical examinations of the running status of the BGP/BFD application process inside the container. When an application failure is detected in the primary container, the incident will be reported to the controller through gRPC. The controller will reboot the container or initiate an NSR migration.

Container failures. As identified by E2 in Figure 3, containers might fail due to misconfigurations, over-commitment of resources, etc. Such failures can be detected by (*i*) process monitors on the host machine, such as Docker Daemon, (*ii*) gRPC health check from the controller, and (*iii*) IP SLA probes from either the host machine or the agent server. After detecting container failures, the controller begins the BGP NSR migration, which includes booting the backup container, starting the BGP and BFD processes, recovering TCP and BGP status from the database, and resuming the connections.

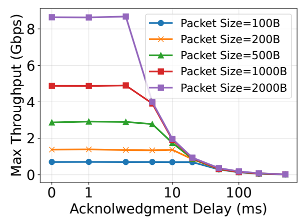
Host machine failures. Another possible failure point is the host machine (as a replacement of the router in § 2). Possible causes include hardware failures, loss of electricity, etc. As illustrated by E3 in Figure 3, all the containers as well as the network configurations will be lost when a host machine fails and all of them need to be recovered. Given the large scale of impacts that a host machine failure involves, we take multiple measurements to verify it and avoid false positives. We only take action to isolate the host machine and recover everything on it when all the failure measurements return positive results, *i.e.,* that the host machine indeed fails. These measurements include (*i*) gRPC heartbeat from the controller, (*ii*) IP SLA to *all* the containers on this host machine, and (*iii*) IP SLA between servers in the cluster. Moreover, a 3-second timer will be given before we begin the recovery to avoid false positives of other failures – we will discuss below in "network failures".

Network failures. Figure 3 illustrates two possible network failures: one is that the (virtual) network of the primary container fails (E4), and the other is that the network of the host machine fails (E5). The location of the virtual network failure of the primary container is similar to that of the container failures. The only difference is that the process monitor on the host machine will not report an error while all the network probes will fail. After locating the failure, the controller will kill the primary container through TKE while starting the BGP NSR migration to the backup container.

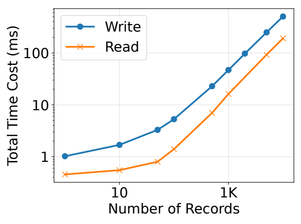
On the other hand, the network failure of the host machine is similar to that of the host machine failures: we take similar measurements to locate the failure and similar actions for recovery. Nevertheless, one critical difference is that the network failure might not be permanent, e.g., network jitter is only temporary. Given the scale of impacts from failure handling similar to host machine failures, we take extra caution to distinguish permanent network failures from temporary ones. Our solution is setting an extra timer of 3 seconds. If *all* the measurements still fail after 3 seconds, then we consider this failure to be permanent and migrate all the containers. Note that once we decide to migrate, the original server will not be re-used before a manual reset – even if it goes back online before that – to avoid split-brain issues or oscillations.

4 EVALUATION

We evaluate the performance and benefits of TENSOR below. We first focus on evaluating the impact on TCP performance from delayed acknowledgment and the delay we introduce in our design (§ 4.1). Next, we show the integral performance of TENSOR (§ 4.2). We then analyze the failure recovery and demonstrate the key



(a) Maximum throughput.



(b) Time costs of DB operations.

Figure 5: Evaluation of the delayed acknowledgment. (a) TCP maximum throughput as a function of the acknowledgment delay; (b) Total time costs of database read operation as functions of the number of records.

benefits of TENSOR compared to existing solutions (§ 4.3). Last, we talk about our operational experience of TENSOR over the past two years (§ 4.4).

For all the following evaluations, we use the same setup as the actual production environment in Tencent Cloud. In particular, each machine is equipped with a 96-core Intel Xeon CPU with 400 GB RAM. On the forwarding plane, the machines are connected via 100 Gbps Ethernet.

4.1 Impact of Delayed ACKs

A key innovation of TENSOR is the kernel-free packet replication which enhances the system reliability and, most importantly, enables the virtualization of BGP NSR. Nevertheless, the benefits come at the cost of performance as we introduce delayed acknowledgment. Below, we aim to understand the performance impact of the delayed acknowledgment.

Transport layer. We first focus on the transport layer to understand the applicability of delayed acknowledgment to general applications. We set up two machines connected via a 100 Gbps Ethernet. Then, we run `iperf` between two machines, where we implement the kernel-free delayed TCP acknowledgment on one machine representing the machine in our cloud gateway while keeping the other machine as normal representing the peering AS.

Figure 5(a) presents the maximum throughput of the TCPs with and without delayed acknowledgment. the results show that the TCP maximum throughput decreases as the acknowledgment delay increases. Particularly, we notice that a threshold exists: the TCP maximum throughput will stay the same as the TCP without any acknowledgment delay when the introduced delay is less than a threshold.

When testing with different packet sizes, we observe that the threshold, *i.e.,* the maximum delay which does not affect the TCP performance, decreases as the packet size increases. In particular, the maximum delays with no impact on the TCP throughput are 20 ms, 10 ms, 5 ms, 2 ms, and 2 ms for TCP connections with packet sizes of 100B, 200B, 500B, 1000B, and 2000B, respectively. When the acknowledgment delays are larger than these numbers, the maximum throughput will decrease. The pace of throughput decrement also increases as the packet size increases.

The results conform to the design of the TCP congestion control: the optimal congestion window size will be limited and dependent on the delay [37, 38]. In essence, introducing an acknowledgment delay is similar yet slightly different from increasing the end-to-end delay. Figure 5(a) shows that for a given acknowledgment delay, there exists an upper bound for the maximum throughput. When the TCP throughput with a given packet size is smaller than the upper bound, it behaves as normal TCP; otherwise, it will be capped by the upper bound, *i.e.,* performance is degraded.

Application layer. To investigate whether the kernel-free packet replication fits the BGP NSR requirement, we need to understand how much delay will be introduced by the application layer operations, particularly, the database-related operations in TENSOR.

When TENSOR receives a packet, the TCP acknowledgment will be held until two database operations are done: a write operation when its message is received and a read operation by tcp_queue to confirm that the corresponding outgoing acknowledgment packet is properly backed up (See § 3.1). When TENSOR sends a packet, the packet will be held after being generated until a database write is done.

We set up a Redis server on another host machine. Each record to write to or read from the Redis database is a pair of a 90B key and a 4 KB value. This represents the largest BGP message, where each message can carry tens or hundreds of routing updates. Specifically, the key consists of a 16B VRF prefix, a 36B four-tuple identification for IPv6-based TCP connection, and a 38B identification for the peering AS and the client. The value is the whole BGP message which has a maximum size limit of 4 KB [40].

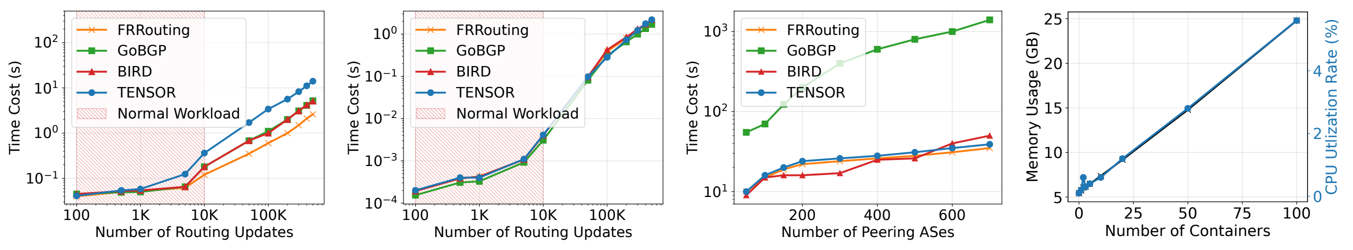
Note that our Redis server will not store data on disk but only in RAM. This setting provides better performance, *i.e.,* faster read and write operations, while satisfying our fault-tolerance needs. Specifically, TENSOR targets providing BGP NSR with respect to single-point failures. When either the database or the BGP container fails, TENSOR can be recovered by simply rebooting the failed service and re-synchronize all the data. The scenarios where both the database (a fault-tolerant service by itself) and the BGP container fail are multi-point failures and hence are out of scope.

Figure 5(b) presents the total time costs of database read and write operations as functions of the number of records. The time to read one record only takes less than 500 μs, and the time to write one record takes roughly 1 ms. Corresponding to Figure 5(a), this is well within the threshold for the delayed acknowledgment to start affecting the performance.

Overall, the relationships between the time costs and the number of records are very similar for both database read and write operations. The difference is that the write operation takes approximately 2.5x longer than the read operation. When operating on multiple records in a batch, the per-record time will be shortened. It takes less than 1 ms to read roughly 70 records, and 200 ms for up to 10K records. For writing records, it takes less than 2 ms for 10 records, and ~500 ms for 10K packets.

4.2 Integral Performance

Next, we explore the integral performance of TENSOR.For performance comparison, we adopt the three most popular open-source BGP implementations: FRRouting [5], GoBGP [6], and BIRD [12]. Note that these open-source BGP implementations do not support BGP NSR. Despite that, we used them as a reference of comparison because they have very similar performance to our original BGP



(a) Receiving from peer(s). (b) Sending to one peer. (c) Sending to multiple peers. (d) CPU and memory usage.

Figure 6: Evaluation of TENSOR. (a)(b) The time cost as a function of the number of routing updates exchanged; (c) The time cost as a function of the number of peers; (d) The memory usage and CPU utilization as functions of the number of containers.

program without the NSR capability. We set up two machines where one installs TENSOR and the other installs FRRouting to represent the peering AS.

Receiving routing updates. Figure 6(a) illustrates the time cost of receiving and learning from the peering AS with different numbers of routing updates, ranging from 100 to 500K. When there are only 100 updates, all the solutions take around 40 ms to complete. The time costs for all solutions stay low (less than 100 ms) before the number of updates reaches around 10K. Later, the time costs increase almost linearly as the number of routing updates increases, and discrepancies appear: FRRouting has the best performance; GoBGP and BIRD have very similar performance; TENSOR has the lowest performance.

We suspect the performance discrepancy originates from indeterministic behaviors of tcp_queue that needs to read the database multiple times before confirming a packet is backed up properly. Despite slight performance discrepancy, TENSOR boosts other aspects of the system (see § 4.3). Note that the maximum number of 500K routing updates in Figure 6(a) represents an extreme scenario. In normal cases, the number of routing updates exchanged will be less than 10K, where the overhead is tens of milliseconds and is acceptable.

There will be no difference for the IO thread to handle packets from one or multiple ASes. Thus, when receiving and learning from multiple ASes, the time costs of other BGP implementations will conform to Figure 6(a) where the number of routing updates should be equal to the sum of that from all the ASes. For instance, it will take at least 5 seconds for any open-sourced implementation to finish the learning from 50 ASes, where each AS sends 10K updates (thus the sum is 500K updates). But thanks to the containerized approach which naturally enables parallelism, each BGP process in TENSOR only needs to connect to one to several ASes, and hence bears sub-second's overhead. Therefore, the overall performance of TENSOR is acceptable in practice.

Sending routing updates. Figure 6(b) presents the time costs of generating and sending out routing updates to a peering AS. Overall, the pattern is similar to that of receiving and learning updates, *i.e.,* the time costs stay low before the number of updates reaches 5K and then they start to increase linearly as the number of updates increases. The good news is that TENSOR achieves approximately the same performance as the other three implementations. The smaller performance difference between TENSOR and the other implementations is also expected because less delay will be involved when sending packets than receiving packets (see § 4.1).

When sending the updates to multiple peers, the BGP process has different behaviors. Because the BGP update message for many peers will be largely the same except for the header information, it is possible to speed up the process by copying the messages. This is referred to as "update packing" [57]. We have implemented update packing in TENSOR.

We now test sending routing updates to multiple peering ASes. We simulate from 50 up to 700 peering ASes and set a fixed number of updates to send per AS as 100. As shown in Figure 6(c), we observe similar performance for TENSOR, FRRouting, and BIRD, whereas GoBGP costs at least 5x more time than the other implementations. This is because the update packing is not implemented in GoBGP. Moreover, TENSOR outperforms BIRD when the number of peering ASes is greater than 600.

Scalability. Next, we demonstrate the scalability of TENSOR. Figure 6(d) shows that the memory usage and CPU utilization rate increase linearly as the number of containers on one host machine increases. Supporting 100 containers only costs 25 GB of memory and 5.6% of the CPU. The containerization design allows easily scaling up the services on one machine as well as scaling up the number of machines. To date, we have deployed TENSOR on over 400 machines and they cover 100% BGP traffic for all enterprise peers in Tencent Cloud.

In conclusion, the performance of TENSOR is acceptable in practice while it provides critical boosts over the system reliability, virtualization support, and fault tolerance.

4.3 Failure Recovery

We now demonstrate the failure recoveries of TENSOR and other BGP implementations in four failure scenarios: application, container, host machine, and host network. Container failure is unique to TENSOR since no virtualization is used in other BGP implementations. As illustrated in Table 1, the most frequent failure (65%) is the host network failure, which is often caused by cable aging or unstable adapters, whereas the least frequent failure (3%) is application failure which is mostly due to software bugs.

Table 1 presents the detailed time costs for TENSOR (the first number) and other BGP implementations (the second number in the brackets) side by side. It is noteworthy that the numbers are different: the time costs for TENSOR represent how long it takes to operate internally, and they are transparent to the peers, *i.e.,* no

Table 1: Failure recovery comparison between TENSOR (the first bold number outside the brackets) and other BPG implementations (FRRouting/GoBGP/BIRD, the second number in the brackets). The time cost is in second.

Failure Type (Frequency)	Failure Detection	Initiates NSR Migration / Reboot Machine or BGP	TCP Recovery / Reconnection	BGP Application Recovery	Total Time Cost
Application (3%)	**0.01** (~1)	**0.10** (~20)	**1.09** (~1)	**1.06** (~5)	**2.26** (~30)
Container (13%)	**0.31** (N/A)	**0.10** (N/A)	**1.19** (N/A)	**1.01** (N/A)	**2.61** (N/A)
Host Machine (19%)	**3.30** (~15)	**0.20** (~200)	**4.50** (~5)	**1.05** (~10)	**9.05** (~240)
Host Network (65%)	**3.30** (~5)	**0.21** (~5)	**4.45** (~5)	**1.21** (~10)	**9.17** (~25)

link downtime. On the other hand, the time costs for other BGP implementations represent the link downtime duration, where no packets can be routed through.

Overall, we find that TENSOR takes a shorter time to detect failures in most scenarios. After that, other BGP implementations require the engineer to manually reboot the BGP process or the machine, which is very time-consuming. The only exception is the host network failure where they do not reboot but wait for the network to recover and then reconnect, which empirically takes several seconds to tens of seconds. In contrast, TENSOR's controller will initiate NSR migration automatically within hundreds of milliseconds.

The TCP recovery for TENSOR and TCP reconnection for other implementations takes barely the same time costs. For BGP application recovery, TENSOR can complete within ~1 second whereas it takes 5 to 10 seconds for other implementations. Note that we consider an average workload here. In case of high workload, it might take other implementations several minutes to recover [13, 26] whereas the time cost will stay almost the same for TENSOR because it (*i*) does not require reconverging the routing policies and (*ii*) handles the workload in a highly parallel manner.

In general, TENSOR speeds up the failure recovery with no link downtime by 2x to 25x compared to the link downtime of the other BGP implementations, e.g., FRRouting/GoBGP/BIRD.

4.4 Operational Experience

Service level agreement. Table 2 summarizes the SLA guarantees and operational costs of TENSOR and other solutions. In general, TENSOR realizes the same SLA in terms of NSR migration time at the level of seconds as the NSR-enabled routers. The open-source BGP implementations including FRRouting, GoBGP, and BIRD in the previous evaluations (see § 4.2), however, do not realize any NSR guarantees. Hence, it will take tens of seconds to minutes for them to recover the BGP connections and rebuild the routing tables.

Development. TENSOR has alleviated the workload for our development team. While ensuring the same SLA, TENSOR is much simpler than the NSR-enabled routers. The NSR-enabled routers usually require the coordination of multiple modules, e.g., routing, forwarding, high availability (HA) modules, etc. The development of each module requires 2-3 engineers. That means that an NSR-enabled router requires ~10 engineers in total. The development and testing time will take 4 to 5 years before the product is open for sale. That translates to ~500 man-months. In contrast, TENSOR is developed by mainly 2 engineers in Tencent. It took 4 months to complete the prototype and 12 months to bring TENSOR online in the Tencent Cloud. That is a roughly 20x reduction of development costs from NSR-enabled routers.

The development difference is also reflected in the size of the codebase. The source lines of code of the NSR-enabled routers are around 50 thousand in addition to the base BGP program, whose codebase may include from 70K to 400K lines [5, 6, 12]. Meanwhile, TENSOR's source lines of code are less than 10 thousand on top of the base BGP program.

Deployment. TENSOR also helps to reduce deployment costs. Typically, an NSR-enabled router costs at least $15K whilst TENSOR with a $3K server can handle roughly the same amount of workload. Open-source BGP implementations cost similar to TENSOR, yet they are not comparable to TENSOR due to the lack of NSR support.

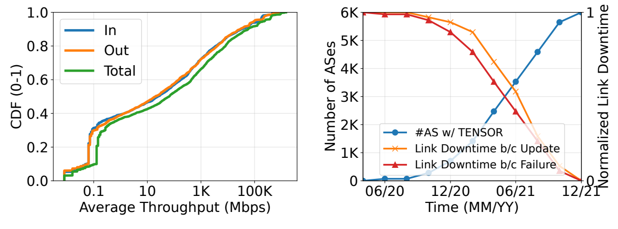
Maintenance. TENSOR has minimized the maintenance costs as well. First, TENSOR allows easier management. The NSR-enabled routers need to be manually pre-configured. In addition, their management requires on-site engineers. In case of a failure, it may take multiple on-site engineers, from both our company and hardware vendors, several days to fix the issue. That translates to over 100 man-hours per month given the failure rate in our operations. In contrast, TENSOR allows us to manage all the BGP processes/containers remotely with the central controller. In our operations, it takes less than 10 man-hours per month to maintain TENSOR.

Another critical help is on the software update. BGP, regardless of the presence or absence of NSR-enabled routers, relies on the graceful restart when the system or software needs to be updated. If the peering AS does not support a graceful restart – which is, unfortunately, the common case for most of our peers – we need to negotiate a specific time window with them or make announcements before performing a system update. And the link will be down during the update. If the peering AS supports the graceful restart, they can keep the current routing policies until the update is done. Yet, no routing policies can be updated during the update. When the network is busy, an outdated routing policy might lead to a massive amount of packet losses. Therefore, we can only perform the update at non-peak hours, e.g., at night. TENSOR, instead, allows transparent system updates at any time.

To quantify the potential losses, we first look at the amount of ongoing traffic in Tencent Cloud. Figure 7(a) is the CDF of the average throughput over 24 hours between Tencent Cloud and its peering ASes. The average and median numbers of the average throughput are over 37 Gbps and 64 Mbps, respectively. Over 30% of the links to peering ASes carry over 1 Gb of data per second. Thus, the impact of link downtime or even not being able to update the routing policies in a timely manner will be significant.

Table 2: Summary of different BGP solutions.

	Failure Recovery	Development Costs		Lines of Source Code	Deployment Cost (US$)	Maintainance Cost (man-hour/month)
		Time	Labor			
FRRouting/GoBGP/ BIRD [5, 6, 12]	(Offline) Tens of Seconds to Minutes	-	-	70K-418K	~3K	~72
NSR-Enabled Router [1]	(Online) Seconds	~50 months	~500 man-months	+50K	>15K	~110
TENSOR	**(Online) Seconds**	**4-12 months**	**~25 man-months**	**+8K**	**~3K**	**<10**



(a) Throughput. (b) TENSOR evolution.

Figure 7: Quantification of operational benefits of TENSOR. (a) The CDF of the average throughput between Tencent Cloud and peering ASes; (b) The adoption of TENSOR and BGP downtime in Tencent Cloud over two years.

Figure 7(b) illustrates the adoption of TENSOR in Tencent Cloud as well as the link downtimes because of updates or failures. Before June 2020, we did not deploy either TENSOR or other NSR solutions because of the complexity and high maintenance costs. At that time, roughly 34 TB of data is impacted every month. We started the initial deployment of TENSOR in June 2020 with 100 ASes. That number stayed for a few months when we thoroughly verified the correctness and robustness of TENSOR. Later, we gradually sped up the deployment process. After a year and a half, we migrated all the enterprise BGP business to TENSOR by the end of 2021.

Until now, TENSOR has been in operation in Tencent Cloud for more than two years, and has supported all the enterprise BGP business for over half a year. We operate TENSOR on a total of more than 400 servers with the same equipment as in our experiments. They support over 3,000 enterprise clients including on-premise datacenters and public cloud and span over 6,000 ASes. With all the clients, TENSOR establishes more than 31,000 BGP connections and exchanges over 1 million routing updates. For the past two years, TENSOR had a link downtime of zero despite that we have tripled the update frequency.

5 LESSONS AND DISCUSSION

Microservice and open-source as the trend for network infrastructure development for cloud providers. In the past, the network infrastructure of different companies heavily relied on mature routing devices provided by hardware vendors. However, this approach no longer meets the requirements of modern cloud providers who prioritize rapid deployment and high flexibility. A new trend has emerged, exemplified by the practice of TENSOR, which involves combining microservices and open-source technologies to revolutionize network infrastructure development.

Microservices, known for their modularity and independent deployability, have gained popularity across various domains. TENSOR demonstrates the extension of this concept to network infrastructure development, offering significant advantages. By adopting microservices, the network infrastructure can isolate and separate different routing services such as BGP and BFD. This ensures that failures in one service do not have a detrimental impact on others. Moreover, microservices enable finer-grained management of each routing service, such as BGP, minimizing the network failure domain. For instance, in the absence of TENSOR, a single gateway router typically handles BGP sessions from multiple clients spanning multiple ASes. A failure in one AS can potentially crash the router and affect other ASes. However, with TENSOR, the BGP programs are split for different ASes, minimizing the scope of the damage.

Additionally, microservices offer the flexibility to scale and adapt to changing demands seamlessly. Open-source technologies complement this trend by breaking down closed product ecosystems created by hardware vendors. They provide cloud providers with a wide range of customizable and community-driven solutions for constructing their network infrastructure. Open-source software also facilitates the rapid development of new network functionalities, allowing providers to stay at the forefront of innovation.

In summary, the combination of microservices and open-source technologies represents a significant paradigm shift in network infrastructure development for cloud providers. It empowers them to address the evolving needs of fast-paced environments while facilitating efficient management and fault tolerance through service isolation and flexibility.

Alternative designs. An alternative to realize the kernel-free packet replication is to run the TCP stack in user space. This approach may be undesirable due to the need for privileged rights to receive all network traffic. We believe it would come with high development and maintenance costs and might also be error-prone due to the complexity of adding a complete TCP stack to our code base. Hence, we retain our design choice of leveraging Linux hooks.

Furthermore, our implementation of TENSOR relies on the Netfilter module because at the time this project started it was a mature technology. We acknowledge that an alternative is to rely on eBPF [4] which has demonstrated better performance over Netfilter [32]. We leave further implementation and comparison as future work.

Dependency loop dilemma. TENSOR relies on a distributed database to back up data including the ongoing TCP packets and routing updates. The distributed database ensures high availability and performance. In our implementation, we adopt Redis and set it up on multiple local servers. One alternative is to use the cloud storage services – most cloud providers including Tencent Cloud have developed multiple cloud products for storage based on Redis or other distributed databases. The cloud storage service may provide better availability and more functionalities than the local setup while minimizing the maintenance cost.

However, relying on cloud storage services may introduce a dependency problem. This is because TENSOR is meant to be the infrastructure *for* the cloud services. In other words, the cloud is depending on TENSOR. If TENSOR relies on the cloud storage services, a dependency loop appears. When an error occurs at either TENSOR or the cloud storage services, the other service will go down as well. Worse, this might result in a cascade failure where all the components or services on the dependency chain will be affected.

Thus, it still needs further studies on wahat to adopt. It is noteworthy that this might apply to all the cloud infrastructure subsystems that could benefit from some cloud services.

Remote replication for disaster recovery. TENSOR could benefit from remote replication for disaster recovery which would help to ensure better data availability. However, the performance will be a problem in realizing synchronized remote replication for TENSOR. As Figure 5(a) shows, the TCP throughput will be impacted significantly if the introduced delay exceeds certain thresholds. And unfortunately, the delay for backing up data at another city or another data center is most likely to exceed the milliseconds-level threshold. An alternative is to back up data in an asynchronous manner. Nevertheless, this would reduce the benefit of availability improvement since TENSOR adopts an application-driven approach (see Section 3.1.2), *i.e.,* only stores the TCP packets that have not been parsed by the BGP application. To summarize, a more detailed study of the trade-off needs to be carried out before implementing the remote replication.

6 RELATED WORK

BGP-related studies. Cloud or content providers have been one of the spots for the BGP research community given their scale and connectivity to multiple ASes. Existing research has explored the (inter-)connectivity of the cloud providers [15, 55], the routing policies of the cloud providers [52], inter-domain routing failures [23], BGP security vulnerabilities and the actions of the network operators [33, 45, 46], and more. Researchers usually rely on popular tools such as peeringDB [8] or data from IXPs such as CAIDA [3] to conduct research from outside the clouds.

On the other hand, cloud or content providers have also contributed to the community. EdgeFabric by Facebook [44] and Espresso by Google [54] share their deployment experiences of running BGP at the edge. Both EdgeFabric and Espresso use an SDN-based approach, *i.e.,* a central controller is responsible for making the decisions. TENSOR and DSR have similar designs. Yet, Facebook [13, 44] and Google [54] handle failures relying on the graceful restart mechanism. There are no discussions with respect to unexpected disruptions such as machine/network failures. In this paper, we focus on BGP non-stop routing for unexpected failures.

BGP non-stop routing. For other routing protocols such as RIP [27], ISIS [36], and OSPF [34], the message exchange with peering routers are connectionless, and hence the non-stop routing support only requires replicating the protocol status, *i.e.,* application-layer status. Nevertheless, BGP relies on the connection-oriented TCP protocol. As a result, non-stop BGP routing requires replicating not only the application-layer status but also the transport-layer status when a failure occurs or migration is needed. It thus follows that TCP packet replication is the key to non-stop BGP routing.

Early proposals propose to extend the TCP protocol so that the peering router can help in the connection migration when needed [48–50]. They however require a consensus of all the gateway routers from all ASes to adopt the same new transport-layer protocol – which is an impractical assumption and is not adopted by the BGP. Another solution involves introducing a proxy server between the peering routers [30, 31]. However, it introduces yet another potential single point of failure.

FT-TCP [56] instead proposed to migrate the TCP *transparently* to its peers by introducing TCP stack wrappers to back up the packets. The downside of this approach is that it introduces *delayed acknowledgment* and negatively affects the performance. Later studies [17, 29, 51] also adopted the packet replication approach. Nevertheless, existing works involve kernel modification, which imposes challenges to network virtualization [47]. In contrast, our approach realizes kernel-free packet replication. In the context of general VM-based fault-tolerance systems [43], our work relies on the deterministic nature of the BGP state machine to avoid high costs from replaying all past inputs/outputs, and does not require a logging channel for state machine operations.

Clark et al. [19] and VROOM [53] propose to replicate the entire virtual machine. Remus [20] advances the virtual machine migration approach by extending the replication at the snapshot granularity and demonstrates better performance. Despite these efforts, performance is still the bottleneck given the much larger volume to be replicated. Instead, our approach only replicates the minimal information that is required to recover the BGP sessions while leveraging lightweight virtualization to provide even better scalability.

7 CONCLUSION

In this paper, we presented TENSOR, a system that supports BGP virtualization and BGP NSR concurrently via a novel kernel-free packet replication design. TENSOR fully leverages the parallelism of lightweight virtualization to improve the performance while also solving the critical split-brain problem. Through experiments in a production-level environment, we demonstrate that TENSOR bares minimal to acceptable performance overhead compared to popular open-source BGP implementations while providing significant benefits in failure recovery. From our two-year operational experience, we find that TENSOR meets our production requirements by achieving the same SLA levels as the NSR-enabled routers, while significantly reducing the development, deployment, and maintenance costs.

ACKNOWLEDGEMENTS

We sincerely thank the anonymous reviewers for their valuable feedback on earlier versions of this paper. We also thank the teams at Tencent for their contributions to the work. Congcong Miao and Jilong Wang are the corresponding authors. This work was supported in part by the National Key Research and Development Program of China under Grant No. 2020YFE0200500.

REFERENCES

[1] Products and Technology - Comware V7 - H3C, 2021. https://www.h3c.com/en/Products_Technology/Operating_System/ComwareV7/.

[2] The netfilter.org "libnetfilter_log" project, 2021. https://netfilter.org/projects/libnetfilter_queue.

[3] CAIDA, 2022. https://www.caida.org/.

[4] eBPF, 2022. https://ebpf.io/.

[5] FRRouting/frr: The FRRouting Protocol Suite, 2022. https://github.com/FRRouting/frr.

[6] GoBGP: BGP implementation in Go, 2022. https://github.com/osrg/gobgp.

[7] gRPC, 2022. https://grpc.io/.

[8] PeeringDB, 2022. https://www.peeringdb.com/.

[9] Redis, 2022. https://redis.io/.

[10] Regional Internet Registries Statistics – RIR Delegations & RIPE NCC Allocations, 2022. https://www-public.imtbs-tsp.eu/~maigron/RIR_Stats/RIR_Delegations/World/ASN-ByNb.html.

[11] Tencent Kubernetes Engine | Tencent Cloud, 2022. https://intl.cloud.tencent.com/products/tke.

[12] The BIRD Internet Routing Daemon Project, 2022. https://bird.network.cz/.

[13] A. Abhashkumar, K. Subramanian, A. Andreyev, H. Kim, N. K. Salem, J. Yang, P. Lapukhov, A. Akella, and H. Zeng. Running BGP in data centers at scale. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 65–81. USENIX Association, 2021.

[14] N. Akiya, M. Binderberger, and G. Mirsky. Common interval support in bidirectional forwarding detection. *RFC*, 7419:1–8, 2014.

[15] T. Arnold, J. He, W. Jiang, M. Calder, Í. Cunha, V. Giotsas, and E. Katz-Bassett. Cloud provider connectivity in the flat internet. In *IMC '20: ACM Internet Measurement Conference, Virtual Event, USA, October 27-29, 2020*, pages 230–246. ACM, 2020.

[16] M. Bernaschi, F. Casadei, and S. Ruco. Migration of secure connections using sockmi. *login Usenix Mag.*, 32(4), 2007.

[17] M. Bernaschi, F. Casadei, and P. Tassotti. Sockmi: a solution for migrating TCP/IP connections. In *15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2007), 7-9 February 2007, Naples, Italy*, pages 221–228. IEEE Computer Society, 2007.

[18] M. Canini, V. Jovanović, D. Venzano, B. Spasojević, O. Crameri, and D. Kostić. Toward Online Testing of Federated and Heterogeneous Distributed Systems. In *The 2011 USENIX Annual Technical Conference (ATC'11)*, 2011.

[19] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings*. USENIX, 2005.

[20] B. Cully, G. Lefebvre, D. T. Meyer, M. Feeley, N. C. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. (best paper). In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, page 161. USENIX Association, 2008.

[21] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 373–387. USENIX Association, 2018.

[22] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 58–72. ACM, 2016.

[23] A. Haeberlen, I. C. Avramopoulos, J. Rexford, and P. Druschel. Netreview: Detecting when interdomain routing goes wrong. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, pages 437–452. USENIX Association, 2009.

[24] D. Katz and D. Ward. Bidirectional forwarding detection (BFD). *RFC*, 5880:1–49, 2010.

[25] D. Katz and D. Ward. Generic application of bidirectional forwarding detection (BFD). *RFC*, 5882:1–17, 2010.

[26] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed internet routing convergence. *IEEE/ACM Trans. Netw.*, 9(3):293–306, 2001.

[27] G. S. Malkin. RIP version 2. *RFC*, 2453:1–39, 1998.

[28] P. Marcos, M. Chiesa, L. Muller, P. Kathiravelu, C. Dietzel, M. Canini, and M. Barcellos. Dynam-IX: a Dynamic Interconnection eXchange. In *CoNEXT*, 2018.

[29] M. Marwah, S. Mishra, and C. Fetzer. TCP server fault tolerance using connection migration to a backup server. In *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*, pages 373–382. IEEE Computer Society, 2003.

[30] M. Marwah, S. Mishra, and C. Fetzer. Fault-tolerant and scalable TCP splice and web server architecture. In *25th IEEE Symposium on Reliable Distributed Systems (SRDS 2006),2-4 October 2006, Leeds, UK*, pages 301–310. IEEE Computer Society, 2006.

[31] M. Marwah, S. Mishra, and C. Fetzer. Enhanced server fault-tolerance for improved user experience. In *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings*, pages 167–176. IEEE Computer Society, 2008.

[32] S. Miano, M. Bertrone, F. Risso, M. V. Bernal, Y. Lu, and J. Pi. Securing linux with a faster and scalable iptables. *Comput. Commun. Rev.*, 49(3):2–17, 2019.

[33] A. Mitseva, A. Panchenko, and T. Engel. The state of affairs in BGP security: A survey of attacks and defenses. *Comput. Commun.*, 124:45–60, 2018.

[34] J. Moy. OSPF version 2. *RFC*, 2328:1–244, 1998.

[35] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014.

[36] D. R. Oran. OSI IS-IS intra-domain routing protocol. *RFC*, 1142:1–517, 1990.

[37] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proceedings of the ACM SIGCOMM 1998 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 31 - September 4, 1998, Vancouver, B.C., Canada*, pages 303–314. ACM, 1998.

[38] N. Parvez, A. Mahanti, and C. L. Williamson. An analytic throughput model for TCP newreno. *IEEE/ACM Trans. Netw.*, 18(2):448–461, 2010.

[39] J. Postel. Transmission control protocol. *RFC*, 793:1–91, 1981.

[40] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (BGP-4). *RFC*, 4271:1–104, 2006.

[41] E. C. Rosen and Y. Rekhter. BGP/MPLS IP virtual private networks (vpns). *RFC*, 4364:1–47, 2006.

[42] S. R. Sangli, E. Chen, R. Fernando, J. G. Scudder, and Y. Rekhter. Graceful restart mechanism for BGP. *RFC*, 4724:1–15, 2007.

[43] D. J. Scales, M. Nelson, and G. Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *ACM SIGOPS Oper. Syst. Rev.*, 44(4):30–39, 2010.

[44] B. Schlinker, H. Kim, T. Cui, E. Katz-Bassett, H. V. Madhyastha, Í. Cunha, J. Quinn, S. Hasan, P. Lapukhov, and H. Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 418–431. ACM, 2017.

[45] P. Sermpezis, V. Kotronis, K. Arakadakis, and A. Vakali. Estimating the impact of BGP prefix hijacking. In *IFIP Networking Conference, IFIP Networking 2021, Espoo and Helsinki, Finland, June 21-24, 2021*, pages 1–10. IEEE, 2021.

[46] P. Sermpezis, V. Kotronis, A. Dainotti, and X. A. Dimitropoulos. A survey among network operators on BGP prefix hijacking. *Comput. Commun. Rev.*, 48(1):64–69, 2018.

[47] H. Shao, X. Wang, Y. Lu, Y. Yu, S. Zheng, and Y. Zhao. Accessing cloud with disaggregated software-defined router. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 1–14. USENIX Association, 2021.

[48] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *3rd USENIX Symposium on Internet Technologies and Systems, USITS'01, San Francisco, California, USA, March 26-28, 2001*, pages 221–232. USENIX, 2001.

[49] F. Sultan, K. Srinivasan, and L. Iftode. Transport layer support for highly-available network services. In *Proceedings of HotOS-VIII: 8th Workshop on Hot Topics in Operating Systems, May 20-23, 2001, Elmau/Oberbayern, Germany*, page 182. IEEE Computer Society, 2001.

[50] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: connection migration for service continuity in the internet. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), Vienna, Austria, July 2-5, 2002*, pages 469–470. IEEE Computer Society, 2002.

[51] R. Surton, K. Birman, and R. van Renesse. Application-driven TCP recovery and non-stop BGP. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*, pages 1–12. IEEE Computer Society, 2013.

[52] H. Svens and L. Hellberg. Analysis of bgp routing for major cloud service providers: A characterization of growth and accessibility, 2022.

[53] Y. Wang, E. Keller, B. Biskeborn, J. E. van der Merwe, and J. Rexford. Virtual routers on the move: live router migration as a network-management primitive. In *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, August 17-22, 2008*, pages 231–242. ACM, 2008.

[54] K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. J. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. M. B. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 432–445. ACM, 2017.

[55] B. Yeganeh, R. Durairajan, R. Rejaie, and W. Willinger. How cloud traffic goes hiding: A study of amazon's peering fabric. In *Proceedings of the Internet Measurement Conference, IMC 2019, Amsterdam, The Netherlands, October 21-23, 2019*, pages 202–216. ACM, 2019.

[56] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud. Engineering fault-tolerant TCP/IP servers using FT-TCP. In *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*, pages 393–402. IEEE Computer Society, 2003.

[57] R. Zhang and M. Bartell. *BGP design and implementation.* Cisco Press, 2003.

[58] Z. Zhou, T. Benson, M. Canini, and B. Chandrasekaran. Tardis: A Fault-Tolerant Design for Network Control Planes. In *SOSR*, 2021.