Bypassing the Load Balancer Without Regrets

Marios Kogias
EPFL

Rishabh Iyer
EPFL

Edouard Bugnion
EPFL

ABSTRACT

Load balancers are a ubiquitous component of cloud deployments and the cornerstone of workload elasticity. Load balancers can significantly affect the end-to-end application latency with their load balancing decisions, and constitute a significant portion of cloud tenant expenses.

We propose CRAB, an alternative L4 load balancing scheme that eliminates latency overheads and scalability bottlenecks while simultaneously enabling the deployment of complex, stateful load balancing policies. A CRAB load balancer only participates in the TCP connection establishment phase and stays off the connection's datapath. Thus, load balancer provisioning depends on the rate of new connections rather than the actual connection bandwidth. CRAB depends on a new TCP option that enables connection redirection. We provide different implementations for a CRAB load balancer on different technologies, *e.g.,* P4, DPDK, and eBPF, showing that a CRAB load balancer does not require many resources to perform well. We introduce the connection redirection option to the Linux kernel with minor modifications, so that it that can be shipped with the VM images offered by the cloud providers. We show how the same functionality can be achieved with a vanilla Linux kernel using a Netfilter module, while we discuss how CRAB can work while clients and servers remain completely agnostic, based on functionality added on the host.

Our evaluation shows that CRAB pushes the IO bottleneck from the load balancer to the servers in cases where vanilla L4 load balancing does not scale and provides end-to-end latencies that are close to direct communication while retaining all the scheduling benefits of stateful L4 load balancing.

ACM Reference Format:
Marios Kogias, Rishabh Iyer, and Edouard Bugnion. 2020. Bypassing the Load Balancer Without Regrets. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421304

1 INTRODUCTION

Load balancing is ubiquitous: nearly all applications today running in datacenters, public clouds, at the edge, or as core internet services rely on some form of load-balancing for both availability and scalability. Load balancing can have different forms, *e.g.,* L4, L7, DNS-based *etc.* and can be implemented in hardware or in software. There has been considerable research on load balancing [3, 9, 16, 24, 35, 42, 43, 47–49] both from academia and industry due to not only the demands for mass deployments, high throughput, and low latency variability, but also the demands to lower provider resources specifically dedicated to it. For instance, Google reports that software-based load balancing can take up to 3-4% of a datacenter's resources [16].

This paper focuses on internal load balancers, which are deployed between clients and servers within the same datacenter or public cloud. Internal load balancers can have a significant impact on the end-to-end latency both due to their load balancing decisions and the intermediate hop, while also constituting a major part of the infrastructure costs for cloud tenants. A common pattern includes the deployment of an internal cloud service, placed behind an internal load balancer, that spawns new service instances according to load requirements and registers them with the load balancer, leading to seamless scalability and elasticity.

Figure 1 illustrates a sample cloud-based, two-tier application. Users using their browsers hit the public IP of the external load balancer and their requests end up being served by the two web servers. Those servers act as internal clients for the backend-servers that are behind the internal load balancer and communicate with a managed database service. This design pattern allows the web tier and the back-end tier to scale independently and remain agnostic to each other due to the use of the two load balancers. Similar examples of such design patterns for services (or microservices) include ML inference to create recommendations, a user authentication microservice [23], generic application servers, and any workload orchestrated in containers such as Kubernetes[39].

Internal load balancers must be able to handle low-latency, high-throughput RPCs, typically implemented on protocols such as gRPC [26], Thrift [55], HTTP, or even custom protocols on top of TCP, *e.g.,* Redis, Memcache. The technical challenge is to spread the load as evenly as possible by leveraging rich, stateful scheduling policies while rapidly adjusting to changes in the service set, adding minimum latency overhead to the application, not creating I/O bottlenecks, and avoiding broken connections. State-of-the-art internal load balancers have benefited from recent innovation in protocol design specifically aimed at improving their scalability, including transport protocols other than TCP [33, 38, 45]. Such approaches though, break backwards compatibility with existing applications, while TCP still remains prevalent both for datacenter [2] and cloud communications. We note that this problem statement is different from that of external load balancers, who must accept and filter standards-based traffic from the Internet, mostly deal with HTTP(S) traffic and might also implement TLS termination.

Our approach *bypasses the load balancer without regret.* Specifically, we remove the load balancer from the critical path as much as possible and offer close to direct communication latencies. At the same time, our design allows for elaborate load balancing policies that improve tail-latency and quickly react to changes in the service set.

We design CRAB, a **C**onnection **R**edirect Lo**A**d **B**alancer. CRAB depends on a new TCP option included in the SYN and SYN-ACK packets that enables traffic redirection. This allows CRAB to only deal with SYN packets and stay off the connection datapath, thus tremendously reducing the load balancing load, while still being able to implement complex load balancing policies that otherwise would require a stateful load balancer implementation.

Our implementation shows that CRAB's datapath can be easily implemented in a programmable switch or in software using kernel-bypass or kernel-based mechanisms. The CRAB implementation in clients and servers requires a modest change; this can be implemented in a kernel module that has no measurable impact on performance or as direct kernel modifications offered as pre-built images to cloud tenants.

Our evaluation demonstrates that CRAB outperforms L4-based load-balancers in terms of added latency overhead, connection throughput, and load balancing policies while being implemented on top of a simple stateless design.

Our contributions are:

- The design of a backward-compatible extension to RFC 791 [50] that enables TCP connection redirection
- The design of a CRAB load balancer that depends on the new connection redirect feature of TCP and supports flexible scheduling policies.
- The implementation of the TCP connection redirection option in the Linux kernel for both clients and servers. Four implementations of the load balancer using P4, DPDK, eBPF and Netfilter.
- A discussion on the caveats, assumptions, and opportunities for CRAB in the public cloud and the integration of CRAB for Kubernetes NodePort load balancing.

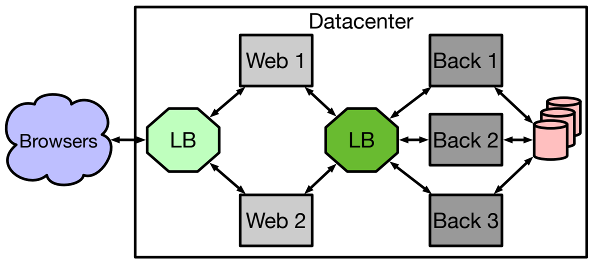


Figure 1: Sample 2-tier cloud application. Web servers handle web traffic coming from users' browsers and act as clients for the back-end servers that run the application logic and communicate with a managed database. The light green octagon is an external load balancer while the dark green one is an internal load balancer.

The end-point CRAB implementation and the source code for the different CRAB load balancers can be found here [1].

2 MOTIVATION AND BACKGROUND

In this section, we first showcase the problem we aim to solve and quantify the potential benefits CRAB can achieve. Then, we provide a comparative description of the state-of-the-art in load balancing that drives our design.

To begin, we run a simple experiment on the public cloud which mimics a scenario that many applications encounter today. We deploy two VMs on Microsoft Azure [5], one acting as a client and the other as a server both configured with accelerated networking [6]. Further, we place the server VM behind an Azure internal load balancer. In this setup, the client VM corresponds to the web tier and the server VM corresponds to the back-end tier from Figure 1. As benchmarks, we run a custom implementation of Netperf's CRR and RR benchmarks [63]. The CRR *(Connect-Request-Response)* benchmark measures the latency to open a connection, send an 8-byte payload, and wait for the server to echo the same payload. On receiving the response, the client closes the connection and starts over. In the RR *(Request-Response)* benchmark clients establish connections once and then use the same connection to send all their requests. RR measures the time between sending an 8-byte request and receiving the echoed back 8-byte response. Both experiments operate in a closed loop with one connection and one outstanding request at a time. Both the client and the server applications run on the vanilla kernel-based networking stack.

Figure 2 illustrates the 99*th* percentile observed latency for the CRR and RR experiments with and without the load balancer. Naturally, the latency for direct communication is

[1] `https://github.com/epfl-dcsl/crab`

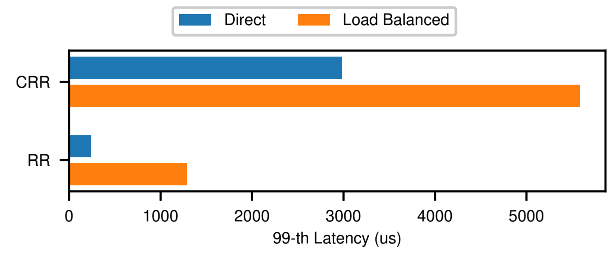


Figure 2: Connection-Request-Response (CRR) and Request-Response (RR) latency benchmarks on Azure with accelerated networking with and without an Azure internal load balancer

lower than the load-balanced scenario. However, the latency overhead introduced by the load balancer is significant for both the RR and the CRR benchmarks. The load balancer adds approximately 1ms and 2ms respectively for the RR and CRR benchmarks; such a large overhead can overshadow the cost of non-balanced RPC.

Given this significant latency overhead associated with internal cloud load balancing, our goal is to minimize it as much as possible to achieve latencies that are close to direct communication. To do so, we need to understand the underlying load balancing mechanisms and policies.

2.1 Load Balancing Flavors

In this section, we categorize and compare the state-of-the-art approaches to load balancing for internal cloud workloads running on top of VMs or containers. Our comparison is based on the following criteria:

- **Load Balancing Policy**: Centralized policies leverage a global view that includes every back-end server while distributed policies make scheduling decisions based only on local state.
- **Persistent Connection Consistency (PCC)**: Can the load-balancer route all packets from the same connection to the same back-end server in the presence of server arrivals and failures?
- **Expected Load**: What is the load balancer load in terms of the packets it has to process for each connection?
- **Latency Overhead**: How much overhead does the load balancer add?
- **Updates**: How quickly does the load balancer take into account scale-up (server-arrival) and scale-down (server-removal) events?

Layer 4 Load Balancing: L4 load balancers operate at the transport layer (TCP/UDP) of the networking stack and remain agnostic to the upper application layers. All public cloud providers offer some form of L4 load balancing, examples include Microsoft's Azure Load Balancer [8], which was used for the experiment in Figure 2, and Amazon's AWS Network Load Balancer [4]

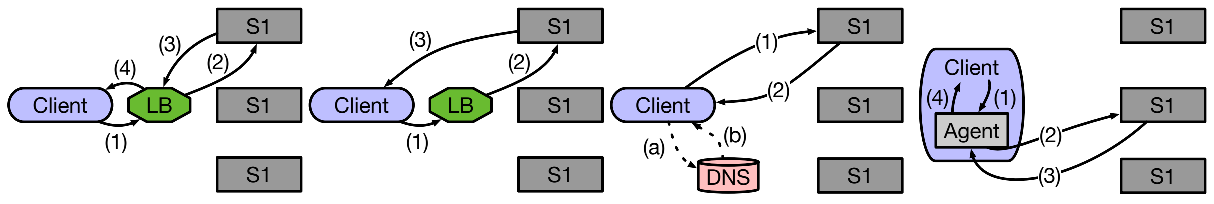
Figure 3a describes the communication between the client, load balancer, and back-end servers for an L4 load balancer. The load balancer listens to a virtual IP (VIP) and the client uses this IP to talk to the service. The service is run on back-end servers that listen to some direct IP (DIP). The load balancer assigns each connection to a particular back-end server and performs address translation. It modifies the destination IP (to the DIP) for packets sent by the client and the source IP (to the VIP) for packets sent by the server. This requires all packets to go through the load balancer adding a latency overhead of 1 RTT to the end-to-end client-server communication and reducing the I/O scalability of the load balancer.

An optimization to the above approach is Direct Server Return (DSR). In this scheme, packets originating at the server can be sent directly to the client without being routed through the load balancer. Servers are aware that they are being load balanced and modify the source IP of outgoing packets to the VIP using address rewriting mechanisms such as `tc` [64]. DSR reduces the load balancer's load since it now only processes client packets and reduces the latency overhead to 0.5 RTT. Figure 3b illustrates an L4 load balancer with DSR enabled.

There has been significant research [3, 9, 16, 24, 35, 42, 43, 47, 47–49] on L4 load balancers. All these approaches can be split into two main categories depending on whether or not they store per-connection state.

Stateless load balancers [3, 47] typically depend on some form of consistent hashing [34] and daisy chaining to ensure that packets with the same 5-tuple will always be forwarded to the same DIP. Relying on hashing to distribute load enables them to eschew per-connection state leading to better performance and scalability. However, this approach has two main caveats. First, load balancing policies are limited to hashing, namely random load balancing; this leads to load imbalances especially when connections are skewed. Second, despite the use of daisy chaining there remain corner cases during server arrival and removal that lead to PCC violations [9].

Stateful load balancers maintain per connection state to correctly route each packet they receive from the client. Further, such load balancers can also maintain state about each back-end server, in order to support more elaborate load balancing policies such as *Join-Shortest-Queue* or Power of two [44]. Such policies cannot be implemented on a stateless load balancer. While per-connection state eliminates PCC violations, the state lookups can become a bottleneck when the number of active connections is large.



(a) L4 Load Balancing **(b) L4 Load Balancing with DSR** **(c) DNS Load Balancing** **(d) Agent-based Load Balancing**

Figure 3: Commonly used load balancing schemes for cloud services based on VMs or containers.

Method\Property	Policy	PCC violations	Expected load	Latency overhead	Updates
L4	Central	Possible*	every packet	1 RTT for every RTT	Fast
L4 w/ DSR	Central	Possible*	one way packets	1/2 RTT for every RTT	Fast
L7	Central	None	every packet	1 RTT for every RTT	Fast
DNS	Central	None	1 RPC every few connections	up to 1RTT per connection	Slow
Local Agent	Distributed	None	every packet	none	Slow
CRAB	Central	None	SYN packets	1/2 RTT for every connection establishment	Fast

* In stateless L4 load balancers

Table 1: Feature comparison between different deployed load balancing schemes and CRAB.

L7 Load Balancing: L7 load balancers or reverse-proxies operate at the application layer. These load balancers terminate client connections and open new connections to the back-end servers. Figure 3a could also describe a L7 load balancing scheme since all the received and transmitted packets have to go through the load balancer. However, for L7 load balancing arrows (1),(4) and (2),(3) would belong to different TCP connections. Popular open-source L7 load balancers include NGINX [46] and HAProxy [27]. Cloud providers also offer such services, *e.g.,* Amazon's AWS ALB [4].

L7 load balancers are typically centralized. Terminating client connections and establishing new ones with back-ends servers, enables them to avoid PCC violations. Further, operating at the application layer allows such load balancers to understand L7 protocols, *e.g.,* HTTP; this enables them to perform fine-grained request-level load balancing as opposed to the more coarse-grained connection level load balancing. However, this results in them depending on complicated software that typically run in userspace. This has the corresponding performance implications, in particular a considerable increase in the latency overhead (we illustrate this in §5.2).

DNS Load Balancing: Another form of load balancing used both in the public internet as well as by container orchestrators such as Docker Swarm [53], and Mesos [30], depends on DNS. DNS load balancing relies on the fact that most clients use the first IP address they receive for a domain after DNS resolution. Typically, the DNS server sends the list of IP addresses in a different order each time it responds to a new client, using the round-robin method. As a result, different clients direct their requests to different servers, effectively distributing the load across the server group. Figure 3c describes the client, server, and DNS server interactions for a DNS load balancing scheme. Steps (a)-(b) can be performed once for several connections (1)-(2).

DNS load balancing, while centralized, is extremely coarse grained, since it only balances the load at a per-client granularity. Further, to avoid the repeated overhead of DNS resolution and reduce the load on the DNS server, clients cache DNS entries; once an entry is in the cache, clients and servers talk directly. Despite its performance benefits, caching can cause severe load imbalance issues. Since clients use the same target IP until the cached entry expires, the system cannot mitigate load imbalances during this period. Also, removing servers from the back-end pool becomes challenging and slow, since administrators have to wait until every possible TTL for the associated entries has expired. DNS load balancing though does not suffer from PCC violations since clients and servers communicate directly.

Local Load-balancing Agent: This load balancing scheme is used in Kubernetes [39]. In a Kubernetes cluster, every node that runs networked containers also runs a local agent responsible for the network configuration. This agent maintains a consistent view of the network by subscribing to state changes in the configuration service (`etcd` [18]). Each service that runs multiple containers and requires load balancing is associated with a specific ClusterIP. The local agent keeps track of the container IPs that run the specific service. Every time a client uses the ClusterIP, the local agent picks one of the target containers and performs address translation for each transmitted and received packet. Thus, while the client believes it is communicating with the ClusterIP, it is actually communicating with a container running the service. Figure 3d describes the above load balancing scheme and the interactions between the client, the server and the local agent.

The main benefit of a local agent is the ease of deployment, since it is integrated into the orchestrator rather than an external service. Placement and scaling decisions automatically update the load balancing decisions performed by this local agent, through pub-sub [19]. However, this approach suffers from three main problems: (1) load balancing decisions are performed in a distributed manner on every server, masking the benefits of smart placement, (2) packet rewriting is on the critical path for every packet sent and received, leading to an increase latency and CPU utilization (3) changes to the pool of target containers take longer to propagate through the use of pub-sub since all machines in the cluster have to receive the update; unlike the case in systems where the load balancing service is standalone and clients and servers are agnostic to it.

Table 1 summarises the above comparison and position CRAB in the design space among the same axis.

2.2 Load Balancing Policies

So far, we have categorized the different load balancing policies used in the above load balancing schemes based on two criteria: (1) whether they are implemented centrally (*e.g.,* L4 load balancers) or in a distributed manner (*e.g.,* load balancing based on a local agent); (2) whether the policy is random (*e.g.,* hashing in stateless L4 load balancers), or richer (*e.g.,* round-robin on stateful L4 load balancers). The performance implications of these policies on application tail latency, though, remains unclear.

To answer the above question, we leverage queuing-theory models and run a discrete event simulation for load balancing policies from each category. Our setup models 16 clients that communicate with 16 servers. Among stateful policies we choose the simplest non-random policy, which is Round-Robin. We simulate both the centralized and distributed versions of this policy. The centralized policy can be thought of as being implemented by a stateful L4 load balancer and the distributed policy as being implemented by Kubernetes-like

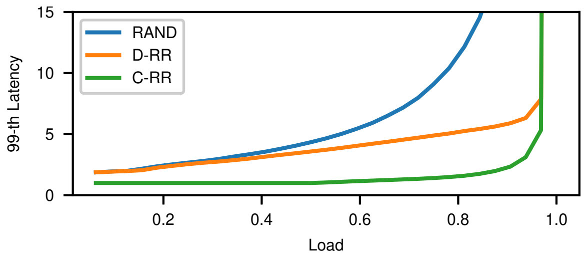


Figure 4: Discrete event simulations of 16 clients sending requests to 16 servers randomly (RAND) or based on a Round-Robin policy in a centralized (C-RR) or distributed manner (D-RR).

local agents. Since stateless load balancing is equivalent to a random assignment of connections to servers, we simulate both centralized and distributed versions of the random policy. We exclude the case of the DNS-based load balancing due to the big impact of TTL both on the required DNS server resources, and the end-to-end latency.

Figure 4 summarizes the simulation results for each policy for a fixed service time distribution and a Poisson interarrival. We observed that the centralized and distributed versions of the random policy displayed identical performance, hence we show them as one. In terms of application tail-latency, the random load balancing has the worst performance. Further, there is a difference between the D-RR (Distributed Round-Robin) policy and the C-RR (Centralized Round-Robin) policies. Due to the randomness of the Poisson inter-arrival, performing Round-Robin on each node leads to worse tail latency, since centralized Round-Robin manages to always pick the least loaded server in this fixed service time experiment. We draw two conclusions from the above results: (1) Decentralized policies are worse at distributing the load than their centralized counterparts leading to worse tail latencies. (2) Rich policies can significantly outperform random load balancing.

However, the choice of load balancing policy is not independent of the load balancer design. For instance, richer policies require stateful implementations that do not scale well, while scalable stateless designs cannot support policies beyond random. This reveals a design trade-off between the scalability of stateless designs and richer load balancing policies of stateful designs.

Our goal is to design a load balancer that combines the best aspects of each design discussed so far, namely: (1) the performance characteristics of DNS-based load balancing (expected load independent of flow size and close to direct communication latencies) (2) the load balancing capabilities of centralized, stateful L4 load balancers and (3) the scalability, flexibility, and PCC violation elimination of stateless L4 load balancers.

3 DESIGN

CRAB is designed to satisfy the following requirements: (1) The load balancer must be able to implement centralized, stateful policies that offer better tail-latency, easier management, and faster updates. (2) The load balancer must not become an I/O and scalability bottleneck. (3) The load balancer must incur the minimal possible latency overhead by eliminating unnecessary network hops. (4) The load balancer must be backwards compatible with existing connection-based transport protocols (specifically TCP). (5) The load balancer must eliminate PCC violations.

The core insight behind CRAB is simple: *Implementing a centralized, stateful load balancing policy at a connection granularity requires the load balancer's involvement only during connection setup, following which the client and server can communicate directly.* Said differently, the load balancer need only map a connection to a back-end server when the connection is being setup, after which it only performs address translations, thus it can be taken off the data path. This eliminates all network hops through the load balancer in the data exchange phase, minimizing the latency overhead and avoiding scenarios where the load balancer becomes the I/O bottleneck, while it completely eliminates the risk of PCC violations. Note, this insight is specific to scenarios in which clients and servers can directly talk to each other and the load balancer is not required to conceal internal infrastructure. This assumption holds for internal load balancers in the public cloud (which is our target deployment) but does not generally hold for load balancers in the public internet.

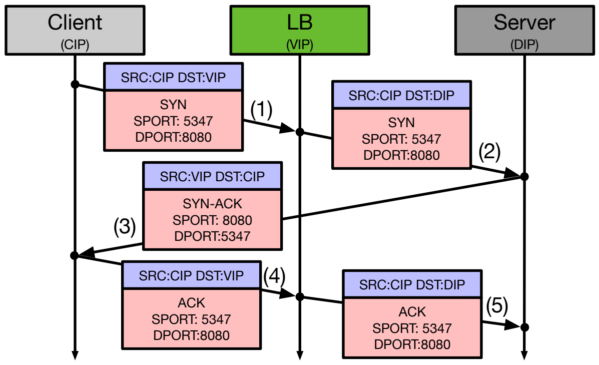
CRAB realizes the above insight for TCP, by extending the traditional 3-way handshake. Figure 5a illustrates this handshake between a client and a server with an L4 DSR-enabled load balancer in the middle for a vanilla TCP implementation. This handshake requires 5 packets to be exchanged. The client first sends a TCP `SYN` packet to the load balancer's VIP (1). The load balancer assigns this connection to a particular back-end server and forwards the `SYN` packet to its DIP (2). Since DSR is enabled, the server replies directly to the client with a `SYN-ACK` packet having VIP as the source IP (3). Finally, the client sends the load balancer an `ACK` packet (4), which the load balancer forwards to the back-end server (5) to finish the connection establishment.

To remove the load balancer from the data path, CRAB leverages what we call *Connection Redirection* (CR). As the name suggests, CR enables redirecting the connection being established to a target IP address that is different from the one initially contacted by the client. To enable CR, we added a new TCP option called `Connection Redirect`. While clients would ordinarily discard `SYN-ACK` packets sent from an IP address they did not send a `SYN` packet to, they now conditionally accept such packets as long as they have the new TCP option. The `Connection Redirect` option includes a 4-byte field that carries the initial destination IP that the client sent the `SYN` packet to. This enables the client to validate that the received `SYN-ACK` is indeed a part of the handshake it initiated and also to find the associated `struct sock`. When a client receives a `SYN-ACK` with a valid `Connection Redirect` option, it changes its internal connection-related data structures and updates them with the new destination IP. Then, it sends the `ACK` to the new destination to finalize the connection establishment. Once this is done, the original destination IP is ignored and the two end-points communicate directly.

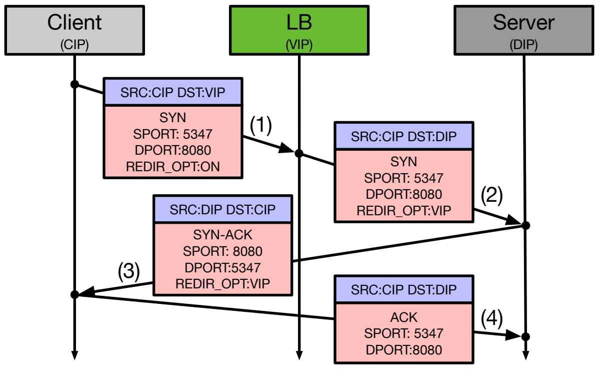
Figure 5b describes the TCP handshake, the associated packets, their IP and TCP headers, and other key fields in the case of connection redirection. As in the vanilla TCP case, the client first sends a TCP `SYN` packet to the load balancer (1). This TCP `SYN` now also indicates whether or not the client supports CR; in this case, we assume it does. The load balancer assigns this connection to a particular back-end server and forwards the `SYN` packet to its DIP (2). In addition, it uses the `Connection Redirect` option to include its VIP in the packet and inform the back-end server that this is a redirected connection. The server then sends the `SYN-ACK` packet directly to the client, with the source IP set to its own, and also echoes the `Connection Redirect` option with load balancer's VIP (3). Finally, the client processes the new TCP option and redirects the connection, resulting in it sending the `ACK` packet directly to the back-end server and bypassing the load balancer (4).

Figure 6 illustrates the complete CRAB load balancing architecture. Steps $a - c$ correspond to the first 3 packets from Figure 5b. All further packets are directly exchanged between the client and the server, completely bypassing the load balancer. The only packets that the CRAB load balancer needs to handle are the TCP `SYN` packets. Once it directs the packet to a particular back-end server (load-balances the connection), it is eliminated from the data path. A direct consequence of this is that CRAB significantly reduces the resources necessary for load balancing. Since a CRAB load balancer handles only new connections, it can be provisioned according to the rate of new connection establishment, as opposed to the receive and transmit throughput of those connections.

In cases where the client's `SYN` packet does not indicate support for connection redirection, the load balancer can



(a) TCP handshake over a L4 load balancer with DST



(b) TCP handshake with connection redirection over CRAB

Figure 5: A load balanced TCP handshake with and without connection redirection. Blue boxes correspond to IP headers, red boxes correspond to TCP headers.

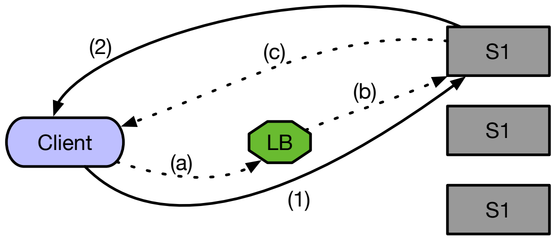


Figure 6: Load Balancing over CRAB using the `Connection Redirect` TCP option. Dashed lines indicate connection establishment. Solid lines indicate data exchange.

fall back to stateless hash-based load balancing, thus remaining compatible with non-CRAB-compliant clients. The load balancer check if the back-end servers are CRAB-compliant through the health probes already sent to make sure servers are up and running.

So, CRAB achieves its design goals as follows: (1) All SYN packets continue to be routed via the load balancer, allowing it to implement the centralized policy of its choice without the limitations of stateless load balancers. (2) Dealing with only SYN packets and not the actual connection payload, ensures that the load balancer is no longer the I/O bottleneck. (3) Removing the load balancer from the data path eliminates all intermediate network hops to it once connection establishment is complete. (4) CRAB is backwards compatible with existing network stacks and falls back to stateless load balancing if the `Connection Redirect` TCP option is not supported. (5) After connection establishment clients talk directly with servers, thus completely eliminating PCC violations.

4 IMPLEMENTATION

CRAB depends on a custom load balancing middlebox and requires changes to the client and the server endpoints. These three components can be implemented using different technologies based on deployment requirements, yet can interoperate independent of the implementation. In this section, we describe the implementations in our current prototype. We discuss alternatives for both implementation and placement of this functionality in §6.

The deployment target for CRAB is a public cloud IaaS provider such as Amazon AWS, Microsoft Azure, or Google Compute Platform. We assume that the provider fully controls the physical infrastructure, but can also control the VM images that cloud tenants use. Unlike other deployment scenarios in which modifying the client endpoints is not feasible, *e.g.,* the internet, clients running on cloud infrastructure can easily integrate new features by running VM images offered by the cloud provider. This approach of specially modified could VM images is not new and already used extensively, *e.g.,* in Azure accelerated networking [7].

4.1 Load Balancing Middlebox

We implement the CRAB middlebox in four different ways keeping in mind the infrastructure IaaS providers use today and the fact that they might need to run several load balancer instances per tenant. We built CRAB middleboxes that rely on, P4 [11], DPDK [14], eBPF [59], and Netfilter modules [62] respectively.

We implemented a CRAB load balancer in ~300 lines of $P4_{14}$ that process TCP SYN packets in the Tofino [10] dataplane. Our DPDK-based CRAB implementation depends on a custom, simple networking stack and the load balancer implementation consists of ~100 lines of C code. Our eBPF-based load balancer leverages XDP [68] and runs natively in the driver context, thus avoiding an extra softIRQ. It processes the incoming packets and sends them out again through the same interface without letting them enter the Linux kernel, while also being able to easily communicate with the user-space through the use of eBPF maps that define the set of DIPs. The eBPF-based implementation consists of ~250 lines of C code. Finally, our Netfilter implementation runs in the context of a `NF_INET_PRE_ROUTING` hook. It is loaded as a kernel module and can communicate with userspace through a character driver to configure the target DIPs and the load balancing policy. The Netfilter implementation is ~200 lines of C code.

Our prototype implementations support two push-based load balancing policies namely random selection and Round-Robin. All implementations currently assume that clients and servers are CRAB compliant; they handle only TCP `SYN` packets and drop all other packets.

4.2 Connection Redirection

Server-side: To enable Connection Redirection, the server must include the `Connection Redirect` option with the load balancer's VIP in the header of the `SYN-ACK` packet it sends to the client. This can be implemented either inside the kernel TCP stack or as part of a header rewriting mechanism before the packet is sent. We provide two implementations for this functionality that display similar performance characteristics. The first is based on a patch to the Linux kernel. It parses the TCP options in the received `SYN` packet and if the `Connection Redirect` option is set, echoes it in the `SYN-ACK` packet. The second implementation leverages Netfilter modules and hooks onto the `NF_INET_LOCAL_OUT` hook. This Netfilter hook modifies the outgoing `SYN-ACK` packets that match a certain source IP and port number and adds the `Connection Redirect` option with a predefined load balancer VIP. The Netfilter implementation totals ~200 lines of C. Given that the server-side implementation of CRAB does not require any kernel data structure modifications it can also be implemented in an eBPF program.

Client-side: The client-side of CRAB is the most intrusive since clients need to modify kernel data structures associated with the connection being redirected. We provide two solutions, one based on patching the kernel and another based on Netfilter modules; both display similar performance. Both implementations use the IP address found in the `Connection Redirect` option to locate the original connection and overwrite the connection's destination IP with the source IP of the received `SYN-ACK`. The Netfilter module totals ~150 lines of C and uses a `NF_INET_PRE_ROUTING` hook to modify the socket structure before the `SYN-ACK` reaches the TCP stack which otherwise would drop the packet due to the source IP being unknown. The kernel patch supporting CRAB is based on Linux 4.19.114 and adds ~200 lines of code.

5 EVALUATION

Our evaluation answers the following questions: (1) How do the different implementations of the CRAB load balancer perform? (2) How does the latency overhead of CRAB compare against existing baselines? (3) How does system throughput scale with CRAB? (4) Can CRAB implement complex scheduling policies that improve the end-to-end application tail latency?

We evaluated CRAB on our infrastructure, rather than the public cloud due to limitations imposed by IaaS providers (*e.g.,* the inability to spoof IPs) which are necessary for the CRAB load balancer. Doing so enables us to be in full control and understand the infrastructure to better reason about the observed performance.

Our experimental setup consists of 10 machines connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. The machines are a mix of Xeon E5-2637 @ 3.5 GHz with 8 cores (16 hyper-threads), and Xeon E5-2650 @ 2.6 GHz with 16 cores (32 hyper-threads). All machines are configured with Intel x520 10GbE NICs (82599EB chipset). The machines configured as clients or servers run either our CRAB-enabled modified Linux kernel or the CRAB client and server Netfilter modules since we did not observe any performance difference between the two implementations. For the experiments on P4 we use a Barefoot Tofino ASIC that runs within an Edgecore Wedge100BF-32X connected to the Quanta switch via a 40Gbps link.

5.1 CRAB Load Balancer Implementations

In this section, we evaluate the different CRAB load balancer implementations based on P4, DPDK, eBPF, and Netfilter modules respectively, in terms of their latency overhead and throughput they can sustain. The goal is not to compare the performance of the four different technologies (P4 vs DPDK vs eBPF vs Netfilter) but rather to provide information on the raw performance each implementation can achieve.

To evaluate latency, we run the same echo benchmarks as in Figure 2 with an 8-byte message size. In the CRR (Connection-Request-Response) benchmark, the client opens a new connection, sends a request, and waits for the response. On receiving the response, it closes the connection. We measure the end-to-end latency from connection establishment until the reply. In the RR (Request-Response) benchmark the client uses a pre-established connection to send requests and get back replies. We measure the end-to-end latency from when the client sends the request until the reply. Both

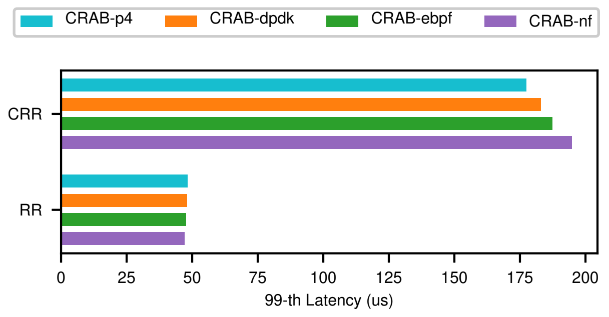


Figure 7: Unloaded tail latency latency measured for the CRR and RR benchmarks for each CRAB load balancer implementation.

experiments run in a closed loop with one client and one server thread, one connection, and one request at a time.

Figure 7 plots the 99-th percentile latency for the CRR and RR benchmarks for each implementation. As expected, we only observe a difference in the CRR experiment since the RR experiment uses a pre-established connection and the load balancer is off the data path. In the CRR experiment, the P4-based implementation has the best performance since it depends on a hardware dataplane. The DPDK-based implementation has the best performance among the software implementations since the load balancer works in a polling mode. The eBPF load balancer operates in native mode in the driver context, thus it performs better than the Netfilter module that runs in the softIRQ context and uses the kernel networking stack.

To evaluate throughput, we need to identify how many SYN packets can each implementation sustain since this is the only traffic that the CRAB load balancer deals with. Unfortunately, we do not possess the resources to run a full setup with TCP clients and servers such that the load balancer is the bottleneck. Instead, we created a DPDK-based client program that bombards the load balancer with SYN packets to stress test the different CRAB implementations. We configured the load balancers to redirect those SYN packets back to the same IP as if the client IP is one of the server DIPs. This enables the client to measure the throughput in terms of SYN packets per second, that each load balancer implementation can sustain.

We are only interested in the achieved throughput of the software-based implementations, since Tofino can achieve line-rate processing. Thus, We configure the three software load balancers to use only one core for packet processing. For the DPDK implementation, we run a single thread, while for the kernel-based implementations we redirected all NIC interrupts to one core, so packet processing would only take place on that core.

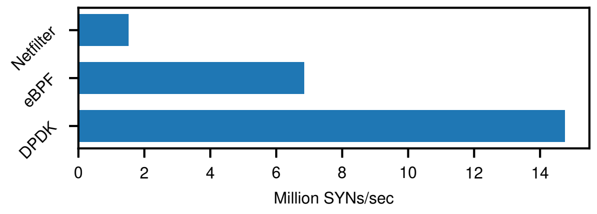


Figure 8: Maximum throughput (new connections per second) achieved by each software-based CRAB load balancer implementation.

Figure 8 summarizes the results and plots the maximum achieved throughput in terms of SYN packets per second. CRAB-DPDK saturates a 10G NIC (~14M SYN packets per second) with a single core. CRAB-EBPF can serve 6.8M SYN/sec, while CRAB-Netfilter can serve 1.5M SYN/sec. As for latency, the performance difference is explained by the different interrupt contexts.

Given the performance results and the ease of deployment that software-based solutions offer, we use only the DPDK-based implementation of CRAB from here onwards as a more realistic candidate to be deployed in the public cloud infrastructure.

5.2 Latency Overhead

We now compare the latency overhead imposed by CRAB against existing load balancer implementations using the same unloaded latency benchmarks. We use the following baselines: (1) A direct configuration where clients and servers communicate directly without a load balancer. This represents the lower bound on latency. (2) NGINX configured as a TCP reverse proxy. This represents an L7 load balancer and (3) An implementation of a stateless L4 load balancer based on the Toeplitz hash [67] which runs on DPDK and is configured with DSR (LBL4-DSR).

Figure 9 plots the observed 99th percentile unloaded latency over 100000 samples for the direct, LBL4-DSR and CRAB load balancers. The NGINX configuration added ~250µs on top of direct for CRR, and ~50µs for RR and hence is not shown in the graph for readability reasons. In the CRR benchmark, the DPDK implementation of CRAB adds ~6µs on top of Direct which corresponds to the half RTT overhead incurred when the SYN packet is routed through the load balancer. In comparison, the LBL4-DSR configuration adds double the overhead (~12µs) since both the SYN and the request packet from the client are routed through the load balancer. In the RR benchmark, CRAB performs the same as Direct, since the load balancer is off the data path. In contrast, the LBL4-DSR load balancer adds 6µs since the request packet from the client is routed through the load balancer.

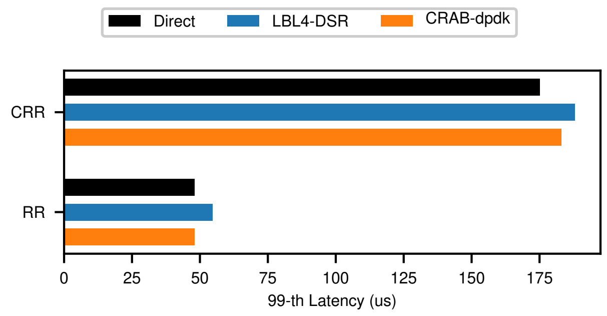


Figure 9: Unloaded tail latency for the CRR and RR benchmarks for setups with no load balancer (Direct), a DPDK-based L4 load balancer with DSR (LBL4-DSR), and the CRAB implementation on DPDK (CRAB-dpdk).

5.3 Throughput Scaling

Here, we illustrate how CRAB's design enables the applications to bypass load balancer bottlenecks and scale their throughput to the I/O capacity of the back-end servers. As mentioned in §2, while DSR alleviates these bottlenecks for applications that are Tx heavy, CRAB seeks to eliminate this bottleneck for a broader range of applications. In this experiment, we target internal cloud services that have a symmetric throughput profile (*e.g.*, storage services, authentication/prediction microservices *etc.*).

To illustrate how CRAB enables throughput scaling, we run the same closed-loop echo benchmark in CRR and RR mode with different message sizes, but measure the maximum goodput (bytes of application payload per unit time) as opposed to the unloaded latency. We use three client and three server machines, and one machine serving as a load balancer, each machine configured with a 10G NIC. We compare CRAB against LBL4-DSR. Both load balancers implement the random load balancing policy.

Figure 10 illustrates the results. For both the RR and CRR workloads, the setup with L4LB-DSR is only able to achieve a maximum goodput of 10G since it is bottlenecked by the single machine that runs the load balancer. On the other hand, we see that CRAB can scale throughput beyond the capacity of the load balancer. In the RR experiment, it can achieve the goodput equal to the I/O capacity of the 3 back-end servers ($3x10 = 30$G) at a payload size of 4096B. In the CRR experiment CRAB and LBL4-DSR perform similarly until a payload size of 8192B; this is due to the experiment being bottlenecked by the cost of new connection establishments. Past 8192B however, LBL4-DSR hits the 10G limit, while CRAB continues to scale.

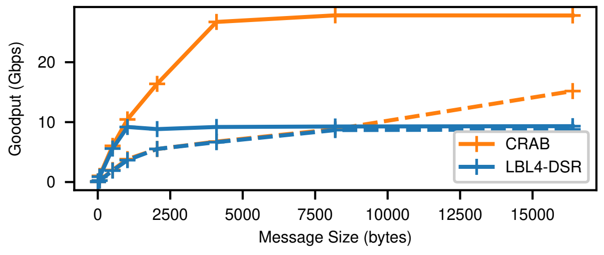


Figure 10: Comparing Max goodput achieved in the CRR (dashed) and RR (solid) benchmarks for CRAB and a DPDK based L4 LB with DSR. I/O capacity of both LBs = 10G. I/O capacity of servers = 30G.

5.4 Load Balancing Policies

So far, we've shown how CRAB, by redirecting connections and bypassing the load balancer incurs a lower latency overhead and enables better throughput scaling. We now evaluate whether CRAB supports elaborate load balancing policies that can significantly improve application tail latency, while still not maintaining per-connection state. Note that CRAB does not propose any new scheduling policies, but must support policies beyond stateless (random) load balancing. For now, we only evaluate push-based policies in which there is no explicit communication between the servers and the load balancer that could help the load balancing decisions. We leave more complicated policies to future work.

To run this experiment on our infrastructure but with a setup that resembles the public cloud, we use 3 servers and configure 16 virtual functions on each server for a total of 48 independent endpoints. We configure each VF with a unique DIP and equally rate limit the VFs so that they take a fair share of the 10G I/O bandwidth of the server. We evaluate the load balancing capabilities of CRAB for two classes of applications — CPU-intensive and I/O-intensive. The CPU bottlenecked application is a synthetic service time server with a fixed service time of 1ms. For the I/O-bottlenecked application, we use NGINX that serves a static file of 8kB over HTTP. For both applications, each request is sent over a new TCP connection. We use Lancet [37] as the load generator. Note, that in both experiments the load balancer is not the bottleneck.

We measure the tail latency as a function of application load for three load balancing policies: (1) Random load balancing with DSR — this represents policies supported by stateless L4 load balancers today (2) A CRAB implementation of random load balancing and (3) A CRAB implementation of Round-Robin load balancing — this represents richer load balancing policies that can only be implemented on stateful load balancers. CRAB, though, can implement Round-Robin without keeping per-connection state at the load balancer.

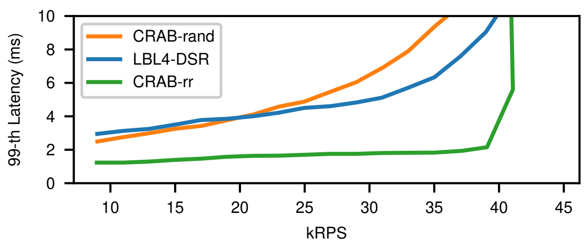


Figure 11: Load Balancing 48 single-core servers running a synthetic service time application with $\bar{S}$ = 1ms

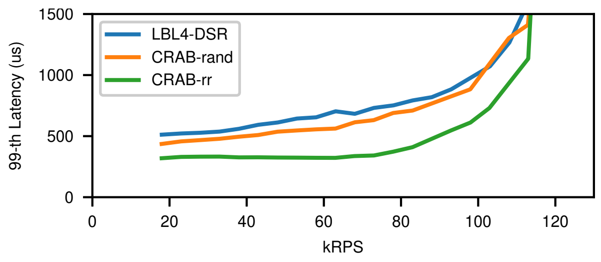


Figure 12: Load Balancing 48 NGINX servers serving an 8 kB static file.

The goal of the experiment is to validate if CRAB can realize the benefits of the elaborate policies as shown in §2.2.

Figures 11, 12 plot the tail latency vs throughput curves for the CPU bound and I/O bound applications respectively. We observe that for both application classes, despite all three policies achieving the same throughput, CRAB Round-Robin achieves significantly lower tail-latency. For application profiles with low service time dispersion, the Round-Robin load balancing policy picks the least loaded server and forward requests to it without requiring explicit communication between the load balancer and the server. Thus, CRAB in addition to eliminating I/O bottlenecks and reducing communication latencies, supports elaborate load balancing policies, truly achieving the best of all worlds.

6 DISCUSSION

Port redirection:: In our existing CRAB implementation we only consider connection redirections based on the target IP assuming that the load balanced service always runs on the same port in the back-end servers. The mechanism can be easily extended to modify the target port, too, in case this is desirable by including both the IP and port in the `Redirection Option`.

Mechanism placement: We implemented connection redirection as part of the Linux kernel assuming the following deployment models: (1) In the case the kernel patch goes upstream, newer kernel version will support it. (2) If not, cloud providers can offer VM images with the modified kernel which cloud tenants can leverage to benefit from CRAB. However, these assumptions are not fundamental to CRAB. We now discuss how CRAB's advantages can be retained with alternative placements of connection redirection that the client and server kernels remain agnostic to.

Cloud providers implement engines either in software [13, 20, 41], or in hardware [21] that accelerate their virtual networking infrastructure. These engines apply address translation rules and encapsulate and decapsulate packets. Connection redirection can be supported by those engines, instead of the guest kernels. On the client side receiving a `SYN-ACK` with the `Connection Redirect` option will create two new rules that will perform Source Network Address Translation (SNAT) for the received packets and Destination Network Address Translation (DNAT) for transmitted packets respectively. The engine will overwrite the DIP with the VIP in the received option for incoming packets, and vice-versa for the transmitted packets. The server-side implementation will create a short-lived rule on receiving a `SYN` packet with the redirection option to echo the option in the outgoing `SYN-ACK`. The downside of such an implementation is that it involves packet modifications on the critical path that can incur performance overheads in a software-based stack. Despite the similarities with the agent-based load balancing in §2, supporting CRAB on the host infrastructure still enables guests to benefit from the centralized load balancing policies and easy and fast updates to the server pool.

Alternative Transports: While we focus only on TCP, here, we discuss how the core ideas behind CRAB apply to other connection-oriented transport protocols, in particular QUIC. QUIC [40] is a low-latency transport protocol designed originally for HTTPS traffic.

While QUIC runs over UDP, it still retains the notion of a connection that is established between a client and a server after a handshake. QUIC also allows a 0-RTT connection establishment for endpoints that have already communicated in the past. After the initial handshake, the connection is associated with a ConnectionID that defines the connection. Load balancers use this ConnectionID to forward packets from the same connection to the correct back-send server [40]. ConnectionIDs also enable seamless connectivity during endpoint migrations (address changes).

We believe CRAB naturally extends to QUIC and can be implemented in two different ways that expose a trade-off between the intrusiveness of the implementation and its performance. First, QUIC can be extended in the same way we extended TCP to support connection redirection during setup, thus benefitting by the simplicity of the proposed CRAB mechanism. QUIC, though, also supports a connection migration functionality that can be used to support CRAB, requiring a more complex middlebox, though. In this case, the 1-RTT handshake has to be performed over the load balancer, since no migration is allowed during connection establishment. Then the server initiates a migration to its DIP instead of the VIP by leveraging the ConnectionID. Following 0-RTT connection establishments can use the DIP directly, thus bypassing the load balancer.

Caveats: Unlike L7 load balancing and DNS-based load balancing that can be deployed by both cloud providers and cloud tenants, L4 load balancing and CRAB require IP spoofing, namely the ability to send an IP packet with a source IP that is different from the local IP. CRAB relies on the load balancer sending packets to the back-end servers with the client IP as source IP. However, IP spoofing is not available for cloud tenants. Thus, CRAB can only be deployed by cloud providers, substituting or complementing their existing L4 load balancing offerings.

CRAB and our proposed connection redirect feature can affect existing mechanisms that perform connection tracking. Such mechanisms can be either implemented in software, such as `conntrack` [57] and Receive Flow Steering (RFS) [66], or in hardware, such as Intel's Application Targeting Routing (ATR) [60]. The goal of such mechanisms is either performance optimization (*e.g.,* interrupt steering for locality) or monitoring (*e.g.,* `conntrack-tools`). In our current implementation, we do not handle such potential violations. In our unloaded latency experiments, we steered all NIC interrupts to the core running the client application while in our throughput experiments all cores were constantly busy. Thus, we did not observe any performance degradation due to hindering of connection tracking mechanisms and specifically ATR.

Other load balancer functionality: Apart from load balancing load balancers today perform other functionalities, such as traffic analysis and firewalling given that all client traffic goes through them. Some of their load balancing decisions might, also, depend on this traffic analysis. CRAB, though, bypasses the load balancer for the datapath, thus it restricts the load balancer visibility on the client traffic. We suggest that any additional to load balancing functionality, such as traffic analysis, should be performed more efficiently in a distributed manner on each machine as opposed to a centralized middlebox whose initial purpose is load balancing. Regarding the load balancing decisions based on traffic analysis, we suggest there is an additional signaling mechanism between the end-hosts and the CRAB load balancer through which servers can notify the load balancer for their availability. The granularity of this mechanism can be either very fine-grained, *e.g.,* for every finished connection, or more coarse-grained *e.g.,* periodically. The existence of such an external control mechanism will enable the deployment of more elaborate load balancing schemes, *e.g.,* CPU-aware ones. In a heterogeneous environment where each server has a different CPU capacity, or in cases where competing workloads affect the available CPU capacity, such a signaling scheme would enable the load balancer to direct the right amount of traffic to each server to avoid overload and SLO violations.

CRAB for Kubernetes NodePort: We only showed the benefits of using connection redirection in load balancing with CRAB when using the central CRAB middlebox. However, connection redirection can have other use cases in a cloud setup, too. One such use-case arises from the Kubernetes ecosystem and the use of NodePort [61].

The NodePort configuration exposes a service running on a Kubernetes cluster on every cluster node independent of whether pods are running this service on the specific node. This way a service can be placed behind an L4 load balancer having all cluster nodes in its server pool. In cases when a node receives traffic for a service that does not run locally, it forwards this traffic to a node that does using kube-proxy. A downside of this approach is that all traffic flows through this intermediate hop, deteriorating the client's perceived latency. To deal with the increased latency, administrators can configure Kubernetes with `externalTrafficPolicy=Local`, so that new connections get rejected if there are no local service instances. This, however, is only a remedy rather than a solution.

Connection redirect can reduce the latency overhead in the above scenario as follows: The initial server that is assigned by the load balancer but does not run the service can add the redirect TCP option before forwarding the SYN packet to the node running the service. That node will then echo back the option as usual, but instead of the client, this option will be intercepted by the stateful load balancer which will then update the target for future packets in this specific connection.

7 RELATED WORK

Connection migration is a similar but different feature to connection redirection that CRAB proposes. The purpose of connection migration implemented in QUIC [40] and proposed for TCP [52] is to serve IP mobility and fault tolerance.

A connection migration can happen at any point in time throughout the connection's lifetime, thus requiring more complicated mechanisms to be implemented. Connection redirection introduced in CRAB can only happen during connection establishment, which enables us to significantly simplify the required mechanism and eliminate its associated overheads, while its purpose is strictly targeting load balancing and connection placement. Although connection migration mechanisms could be abused to work in load balancing scenarios, they would require a much more complicated middlebox logic.

Load balancing is performed at different layers of the networking stack. Network load balancing is below L4 load balancers and orthogonal to our work. Hash-based approaches such as ECMP [51, 54], or more complex approaches such as Conga [1] and alternatives [25, 29, 36, 69, 72] make sure that the multiple paths inside 3-tier datacenter clos topology are equally loaded, thus reducing queuing at the datacenter switches.

CRAB can be thought of as a lightweight L4 load balancer that bypasses the typical L4 load balancing limitations. There are numerous load balancer implementations in software [3, 16, 35, 47, 48], and hardware [9, 24, 42, 43, 49] with stateful or stateless designs. Stateful designs suffer from scalability limits, while stateless designs suffer from suboptimal load balancing policies. To overcome the space limitations in stateful load balancers Kablan et al. [32] suggested using external stores for the load balancing state. CRAB achieves the best of both worlds since it can implement complex load balancing policies and avoid PCC violations while remaining stateless.

The works most closely related to ours, though, are the following. Duchene et al. [15] target a specific to Multi-path TCP (MPTCP) [22] problem and ensure that the different TCP connections within a single MPTCP connection are routed to the same server, using a similar mechanism to connection redirect, in which the back-end server advertises its IP to the client during connection setup. R2P2 [38] similarly to CRAB enables load balancer bypass both on the transmit and receive path from the client perspective, but does so by employing a novel transport layer that exposes individual RPCs. Cheetah [9] exposes an identifier to the clients that is used by the load balancer for forwarding, thus achieving policies equivalent to a stateful load balancer without keeping state at the load balancer. However, cheetah's load balancer is always on the critical path, unlike CRAB. Finally, QUIC's connection migration feature [65] serves as a subset of the proposed connection redirect option.

The above solutions can only be deployed by cloud providers and offered as services to the cloud tenants. Tenants that want to have more control over their infrastructure and how load balancing is performed can deploy L7 or DNS-based load balancing. Examples of open-source software that provides such services is NGINX [46], varnish [70], haproxy [27], envoy [17] *etc.* for L7 and bind [56] and dnsmasq [58] for DNS. L7 load balancers can implement more elaborate load balancing policies than CRAB since they can also do request-level load balancing but incur more significant costs since they run in userspace. Prism [28] and Connection hand-off [31] improve L7 load balancing by temporarily bypassing the load balancer for each request, unlike CRAB that completely bypasses the load balancer for the entire connection. Thus, such approaches still require complicated logic at the middlebox. DNS-based load balancing can display similar performance to CRAB but suffers from the problems associated with caching.

Cloud providers use cluster schedulers such as Borg [71], Kubernetes [12, 39], Mesos [30], Docker Swarm [53] to provision virtual resources for container or VM workloads aiming to maximize utilization without violating customer SLOs. CRAB can be used as part of the above solutions helping their elasticity. CRAB dramatically reduces the resources necessary for load balancing, thus leaving more available resources for the above schedulers to run client workloads on.

8 CONCLUSION

CRAB is a novel design for internal load balancers in the public cloud. It depends on a new TCP option that enables connection redirection; thus, the load balancer participates only in the connection establishment. Unlike traditional L4 load balancers, CRAB does not impact the latency or bandwidth of established connections. CRAB can support the same rich load balancing policies as traditional load balancers, but without keeping per connection state within the load balancer. CRAB is backward compatible with TCP, and easily deployable on the public cloud with minor kernel modifications.

9 ACKNOWLEDGEMENTS

We would like to thank the anonymous PC reviewers for their feedback that helped us improve this paper. This work is part of the Microsoft Swiss JRC TTL-MSR project. Marios Kogias is supported by an IBM PhD Fellowship.

REFERENCES

[1] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: distributed congestion-aware load balancing for datacenters.. In *Proceedings of the ACM SIGCOMM 2014 Conference.* 503–514.

[2] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP).. In *Proceedings of the ACM SIGCOMM 2010 Conference.* 63–74.

[3] João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. 2018. Balancing on the Edge: Transport Affinity without Network State.. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*. 111–124.

[4] aws-lb [n.d.]. AWS Elastic Load Balancing. https://aws.amazon.com/elasticloadbalancing/.

[5] azure [n.d.]. Microsoft Azure Cloud Computing Services. https://azure.microsoft.com.

[6] azure-kernel [n.d.]. Azure Accelerated Networking. https://docs.microsoft.com/en-us/azure/virtual-network/create-vm-accelerated-networking-cli.

[7] azure-kernel [n.d.]. Azure kernel patches for Accelerated Networking. https://github.com/microsoft/azure-linux-kernel/blob/master/4.4.116/README.

[8] azure-lb [n.d.]. Azure Load Balancer. https://docs.microsoft.com/en-us/azure/load-balancer.

[9] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostic, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. 2020. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency.. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*. 667–683.

[10] Barefoot Networks. 2018. Tofino Product Brief. https://barefootnetworks.com/products/brief-tofino/.

[11] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *Computer Communication Review* 44, 3 (2014), 87–95.

[12] Brendan Burns, Brian Grant, David Oppenheimer, Eric A. Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (2016), 50–57.

[13] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization.. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*. 373–387.

[14] dpdk [n.d.]. Data Plane Development Kit. http://www.dpdk.org/.

[15] Fabien Duchene and Olivier Bonaventure. 2017. Making multipath TCP friendlier to load balancers and anycast.. In *Proceedings of the 25th IEEE International Conference on Network Protocols (ICNP)*. 1–10.

[16] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer.. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*. 523–535.

[17] envoy [n.d.]. Envoy Reverse Proxy. https://www.envoyproxy.io/.

[18] etcd [n.d.]. `etcd`: Distributed reliable key-value store for the most critical data of a distributed system. https://github.com/etcd-io/etcd.

[19] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *ACM Comput. Surv.* 35, 2 (2003), 114–131.

[20] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud.. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*. 315–328.

[21] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud.. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*. 51–66.

[22] Alan Ford, Costin Raiciu, Mark Handley, Olivier Bonaventure, and Christoph Paasch. 2020. TCP Extensions for Multipath Operation with Multiple Addresses. *RFC* 8684 (2020), 1–68.

[23] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems.. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*. 3–18.

[24] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. 2014. Duet: cloud scale load balancing with hardware and software.. In *Proceedings of the ACM SIGCOMM 2014 Conference*. 27–38.

[25] Soudeh Ghorbani, Zibin Yang, Philip Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. 2017. DRILL: Micro Load Balancing for Low-latency Data Center Networks.. In *Proceedings of the ACM SIGCOMM 2017 Conference*. 225–238.

[26] grpc [n.d.]. gRPC. http://www.grpc.io/.

[27] haproxy [n.d.]. HAProxy. https://haproxy.org.

[28] Yutaro Hayakawa, Lars Eggert, Michio Honda, and Douglas Santry. 2017. Prism: a proxy architecture for datacenter networks.. In *Proceedings of the 2017 ACM Symposium on Cloud Computing (SOCC)*. 181–188.

[29] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John B. Carter, and Aditya Akella. 2015. Presto: Edge-based Load Balancing for Fast Datacenter Networks.. In *Proceedings of the ACM SIGCOMM 2015 Conference*. 465–478.

[30] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*.

[31] Guerney Hunt, Erich Nahum, and John Tracey. 1997. *Enabling Content-Based Load Distribution for Scalable Services.* Technical Report TR-97. IBM T.J. Watson Research Center.

[32] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing.. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*. 97–112.

[33] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast.. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*. 1–16.

[34] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web.. In *Proceedings of the 29th ACM Symposium on the Theory of Computing (STOC)*. 654–663.

[35] katran [n.d.]. Facebook's Katran. https://github.com/facebookincubator/katran.

[36] Naga Praveen Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing

Using Programmable Data Planes.. In *Proceedings of the Symposium on SDN Research (SOSR)*. 10.

[37] Marios Kogias, Stephen Mallon, and Edouard Bugnion. 2019. Lancet: A self-correcting Latency Measuring Tool.. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 881–896.

[38] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. R2P2: Making RPCs first-class datacenter citizens.. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 863–880.

[39] kubernetes [n.d.]. Kubernestes Container Orechestrator. https://kubernetes.io/.

[40] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan R. Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment.. In *Proceedings of the ACM SIGCOMM 2017 Conference*. 183–196.

[41] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kokonov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: a Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*.

[42] James Murphy McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. 2019. Thoughts on load distribution and the role of programmable switches. *Computer Communication Review* 49, 1 (2019), 18–23.

[43] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs.. In *Proceedings of the ACM SIGCOMM 2017 Conference*. 15–28.

[44] Michael Mitzenmacher. 2001. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.* 12, 10 (2001), 1094–1104.

[45] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. 2018. Homa: a receiver-driven low-latency transport protocol using network priorities.. In *Proceedings of the ACM SIGCOMM 2018 Conference*. 221–235.

[46] nginx-proxy [n.d.]. NGINX Reverse Proxy. https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/.

[47] Vladimir Andrei Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless Datacenter Load-balancing with Beamer.. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*. 125–139.

[48] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert G. Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: cloud scale load balancing.. In *Proceedings of the ACM SIGCOMM 2013 Conference*. 207–218.

[49] Benoit Pit-Claudel, Yoann Desmouceaux, Pierre Pfister, Mark Townsley, and Thomas H. Clausen. 2018. Stateless Load-Aware Load Balancing in P4.. In *Proceedings of the 26th IEEE International Conference on Network Protocols (ICNP)*. 418–423.

[50] Jon Postel. 1981. Transmission Control Protocol. *RFC* 793 (1981), 1–91.

[51] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network.. In *Proceedings of the ACM SIGCOMM 2015 Conference*. 183–197.

[52] Alex C. Snoeren, David G. Andersen, and Hari Balakrishnan. 2001. Fine-Grained Failover Using Connection Migration.. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*. 221–232.

[53] swarm [n.d.]. Docer Swarm. https://docs.docker.com/engine/swarm/.

[54] David Thaler and Christian E. Hopps. 2000. Multipath Issues in Unicast and Multicast Next-Hop Selection. *RFC* 2991 (2000), 1–9.

[55] thrift [n.d.]. Apache Thrift. https://thrift.apache.org/.

[56] url:bind [n.d.]. Berkeley Internet Name Domain (BIND). https://www.bind9.net.

[57] url:conntrack [n.d.]. Conntrack. https://manpages.debian.org/testing/conntrack/conntrack.8.en.html.

[58] url:dnsmasq [n.d.]. dnsmasq. http://www.thekelleys.org.uk/dnsmasq/doc.html.

[59] url:ebpf [n.d.]. extended Berkeley Packet Filter. https://www.iovisor.org/technology/ebpf.

[60] url:intel-atr [n.d.]. Intel Ethernet Flow Director - Application Targeting Routing. https://software.intel.com/content/www/us/en/develop/articles/setting-up-intel-ethernet-flow-director.html.

[61] url:kubernetes-doc [n.d.]. Kubernetes Service. https://cloud.google.com/kubernetes-engine/docs/concepts/service.

[62] url:netfilter [n.d.]. Netfilter. https://www.netfilter.org.

[63] url:netperf [n.d.]. NetPerf. https://fossies.org/linux/netperf/doc/netperf.pdf.

[64] url:nginx-tc [n.d.]. Enabling DSR on NGINX. https://www.nginx.com/blog/ip-transparency-direct-server-return-nginx-plus-transparent-proxy.

[65] url:quic-ietf [n.d.]. The QUIC Transport Protocol - IETF. https://tools.ietf.org/html/draft-ietf-quic-transport-27.

[66] url:rfs [n.d.]. Receive Flow Steering. https://lwn.net/Articles/382428.

[67] url:toeplitz [n.d.]. The Toeplitz Hash Algorithm. https://en.wikipedia.org/wiki/Toeplitz_Hash_Algorithm.

[68] url:xdp [n.d.]. eXpress Data Path. https://www.iovisor.org/technology/xdp.

[69] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching.. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*. 407–420.

[70] varnish [n.d.]. Varnish. https://varnish-cache.org/.

[71] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg.. In *Proceedings of the 2015 EuroSys Conference*. 18:1–18:17.

[72] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. 2014. WCMP: weighted cost multipathing for improved fairness in data centers.. In *Proceedings of the 2014 EuroSys Conference*. 5:1–5:14.