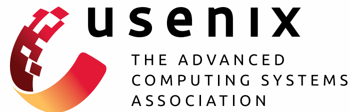

Rubik: Unlocking the Power of Locality and End-point Flexibility in Cloud Scale Load Balancing

Rohan Gandhi, Y. Charlie Hu and Cheng-kok Koh, *Purdue University;*
Hongqiang (Harry) Liu and Ming Zhang, *Microsoft Research*

https://www.usenix.org/conference/atc15/technical-sessions/presentation/gandhi

This paper is included in the Proceedings of the
2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Open access to the Proceedings of the
2015 USENIX Annual Technical Conference
(USENIX ATC '15) is sponsored by USENIX.

Rubik: Unlocking the Power of Locality and End-Point Flexibility in Cloud Scale Load Balancing

Rohan Gandhi, Y. Charlie Hu, Cheng-Kok Koh
Purdue University

Hongqiang (Harry) Liu, Ming Zhang
Microsoft Research

Abstract

Cloud scale load balancers, such as Ananta and Duet are critical components of the data center (DC) infrastructure, and are vital to the performance of the hosted online services. In this paper, using traffic traces from a production DC, we show that prior load balancer designs incur substantial overhead in the DC network bandwidth usage, due to the intrinsic nature of traffic redirection. Moreover, in Duet, traffic redirection results in extra bandwidth consumption in the core network and breaks the full-bisection bandwidth guarantees offered by the underlying networks such as Clos and FatTree.

We present RUBIK, a load balancer that significantly lowers the DC network bandwidth usage while providing all the performance and availability benefits of Duet. RUBIK achieves its goals by applying two principles in the scale-out load balancer design – exploiting locality and applying end-point flexibility in placing the servers. We show how to jointly exploit these two principles to maximally contain the traffic load balanced to be within individual ToRs while satisfying service-specific failure domain constraints. Our evaluation using a testbed prototype and DC-scale simulation using real traffic traces shows that compared to the prior art Duet, RUBIK can reduce the bandwidth usage by over 3x and the maximum link utilization of the DC network by 4x, while providing all the performance, scalability, and availability benefits.

1 Introduction

Load balancing is a foundational function of modern data center (DC) infrastructures that host online services. Typically, each service exposes one or more virtual IPs (VIPs) outside the service boundary, but internally runs on hundreds to thousands of servers, each with a unique direct IP (DIP). The load balancer (LB) stores the VIP-to-DIP mapping, receives the traffic destined to each VIP, and splits it across the DIPs assigned for that VIP. Thus, the LB touches every packet coming from the Internet, as well as a significant fraction of the intra-DC traffic. For a 40K-server DC, LB is expected to handle 44 Tbps of traffic at full network utilization [21].

Such enormous traffic volume significantly strains the data plane of the LB. The performance and reliability of the load balancer directly affects the performance (throughput and latency) as well as availability of the online services within the DC. Recently proposed scale-out LB designs such as Ananta [21] and Duet [14] provide low cost, high scalability and high availability by distributing the load balancing function among Multiplexers (Muxes), either implemented in commodity servers called software Muxes (SMuxes) or existing hardware switches, called hardware Muxes (HMuxes).

However, such LB designs incur high bandwidth usage of the DC network because of the intrinsic nature of traffic redirection. First, even if the traffic source and the DIPs that handle the traffic are under the same ToR, the traffic first has to be routed to the Muxes, which may be faraway and elongate the path traveled by the traffic. Second, in both Ananta and Duet, the Muxes select DIPs for a VIP by hashing the five-tuple of IP headers, and hence are oblivious to DIP locations. As a result, even if the Mux and some DIPs are located nearby the source, the traffic can be routed to faraway DIPs in the DC, again traversing longer paths. Lastly, these designs do not leverage the server location flexibility in placing the DIPs closer to the sources to shorten the path.

The second problem with the Duet LB design is that the traffic detouring through core links breaks the full-bisection bandwidth guarantees originally provided by full-provisioned networks such as Clos and FatTree.

Our evaluation of traffic paths in a production DC network shows that such traffic detour significantly inflates the bandwidth usage of the DC network. This high bandwidth usage not only requires the DC operator to provision high network bandwidth which is costly, but also makes the network prone to transient congestion which affects latency-sensitive services.

In this paper, we propose RUBIK, a new LB that sig-

nificantly reduces the high bandwidth usage by LB. Like Duet, RUBIK uses a hybrid LB design consisting of the HMuxes and SMuxes, and aims to maximize the VIP traffic handled by HMuxes to reduce the LB costs. While doing that, RUBIK reduces the bandwidth usage using two synergistic design principles. First, RUBIK exploits the locality, *i.e.,* it tries to load balance VIP traffic generated within individual ToRs across the DIPs residing in the same ToRs. This reduces the total traffic entering the core network. Second, RUBIK exploits end-point flexibility, *i.e.,* it tries to place the DIPs for a VIP in the same ToRs as the sources generating the VIP traffic.

To exploit locality, RUBIK uses a novel architecture that splits the VIP-to-DIP mapping for a VIP into multiple "local" and a single "residual" mappings stored in different HMuxes. The local mapping stored at a ToR handles the traffic generated in the ToR across the DIPs in the same ToR. The residual mapping assigned to an HMux handles the traffic not handled by local mappings and maximizes the total VIP traffic handled by HMuxes.

To exploit locality and end-point flexibility, RUBIK faces numerous challenges. First, there are limited resources – individual switches have limited memory (where VIP-to-DIP mappings are stored) and individual ToRs have limited servers (where DIPs can be assigned). Also, individual DIPs (servers) have limited capacities. Exploiting end-point flexibility is further compounded as there are dependencies across services. The dependencies arise because many large services are multi-tiered; when a subservice at tier i receives a request, it spawns multiple requests to the subservices at tier $(i+1)$. Because of such dependencies, traffic sources at a lower tier are not known until DIPs in the higher tier are placed. Furthermore, RUBIK needs to ensure that it assigns DIPs that satisfy SLAs.

We develop a practical two-step solution to address all of the above challenges. In the first step, we design an algorithm to jointly calculate the DIP placement and mappings to maximize the traffic contained in ToRs while satisfying various constraints using an LP solver. In the second step, we use a heuristic assignment to maximize the total traffic handled by HMuxes to reduce the costs.

Lastly, to adapt to the cloud dynamics such as changes in the VIP traffic, failures, *etc.*, RUBIK regularly updates its local, residual mappings and DIP placement while limiting the number of servers migrated.

We evaluate RUBIK using a prototype implementation and DC-scale simulation using real traffic traces. Our results show that compared to the prior art Duet, RUBIK can reduce the maximum link utilization (MLU) of the DC network by over 4x and the bandwidth usage by over 3x, while providing the same benefits as Duet.

In summary, this paper makes the following contributions. (1) Through careful analysis of the LB workload from one of our production DCs, we show the high DC network bandwidth usage by recently proposed LB design Duet and Ananta. (2) We present the design and implementation of RUBIK that overcomes these inefficiencies by exploiting traffic locality and end-point flexibility. To the best of our knowledge, this is the first LB design that exploits these principles. (3) Through testbed experiments and extensive simulations, we show that RUBIK reduces the DC network bandwidth usage by 3x and the MLU by over 4x while providing a high performance and highly available LB.

2 Background

In this section, we briefly explain the LB functionality, workloads, and the Duet LB.

2.1 Load balancer

VIP indirection: A DC hosts thousands of online services, *e.g.,* news, sports [21, 14]. Each service exposes one or more virtual IPs (VIPs) outside the service boundary to receive the traffic. Internally, each service runs on hundreds to thousands of servers. Each server in this set has a unique direct IP (DIP) address. The task of the LB is to forward the traffic destined to a VIP of a service to one of the DIPs for that service. Such indirection provided by VIPs provides location independence: each service is addressed with a few persistent VIPs, which simplifies the management of firewall rules and ACLs, while behind the scene individual servers can be maintained or migrated without affecting the dependent service.

VIP traffic: In the Azure DC, 18-59% (average 44%) of the total traffic is VIP traffic which requires load balancing [21]. This is because services within the same DC use VIPs to communicate with each other to use the benefits provided by the VIP indirection. As a result, all incoming Internet traffic to these services (close to 30% of the total VIP traffic in our DC) as well as a large amount of inter-service traffic (accounting for 70% of the total VIP traffic) go through the LB. For a DC with 40k servers, LB is expected to handle 44 Tbps of traffic at full network utilization [21]. Such indirection of large traffic volume requires a scalable, high performance (low latency, high capacity) and highly available LB.

2.2 Workload Characteristics

We make the following observations about the VIP traffic being load balanced in our production DC, by analyzing a 24-hour traffic trace, for 30K VIPs.

The traffic sources and DIPs for individual VIPs are scattered over many ToRs. Fig. 1 shows the number of ToRs where the traffic sources and DIPs for the top 10% VIPs which generate 90% of the total VIP traffic are located. We see that traffic sources are widely scattered – the number of ToRs generating traffic for each VIP varies between 0-44.5% of the total ToRs. Also, the number of

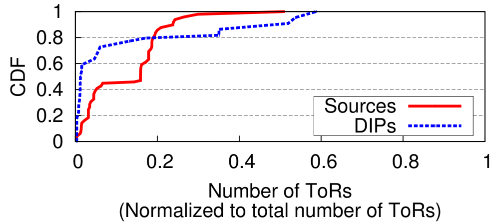

Figure 1: Distribution of the number of ToRs where the sources and DIPs are located.

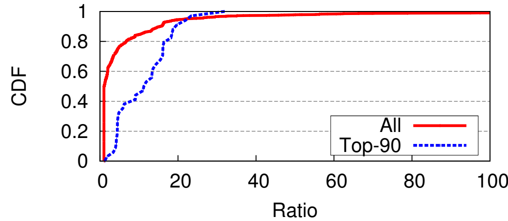

Figure 2: Ratio of 99th percentile to average traffic volume for each VIP across all sources.

ToRs where the DIPs for a VIP are located varies between 0-58% of the total ToRs in the DC.

The traffic volume of sources per VIP are highly skewed. We measure the traffic from all the ToRs for each VIP. Figure 2 shows the CDF of the ratio of the 99th percentile to the median per-ToR traffic volume for each VIP. We see that the source traffic volume for each VIP is highly skewed – the ratio varies between 1-35 (median 18) for the top VIPs generating 90% of the total traffic. The large skew happens for multiple reasons, including different numbers of servers, skew in the popularity of the objects that are served, and locality [17, 10].

VIP dependencies: Many large-scale web services are composed of multi-tier services, each with its own set of VIPs. When the top-level service receives a request, it spawns multiple requests to the services at the second tier, which in turn send requests to services at lower tiers. As a result, the VIP traffic exhibit hierarchical dependencies – the DIPs serving the VIPs at tier i become the traffic sources for the VIPs at tier $(i+1)$. We observe that 31.1% VIPs receive traffic from other VIPs. These VIPs employ 25.1% of the total DIPs and contribute to 27.6% of the total VIP traffic. The remaining 72.4% VIP traffic comes from the Internet, other DCs, and other servers in the same DC that are not assigned to any VIPs.

The dependency among the VIPs can be represented in a DAG. The depth of the DAG observed is similar to the depths reported by Facebook and Amazon [20].

2.3 Ananta LB

Ananta distributes the LB functionality among hundreds of commodity servers called software Muxes (SMuxes). Each SMux in Ananta stores VIP-to-DIP mapping for all the VIPs. When the VIP traffic hits one of the SMuxes, it selects the DIP based on the hash calculated over IP 5-tuple, and uses IP encapsulation to forward the VIP traffic to the selected DIP. Using SMuxes allows Ananta to be highly scalable, but is also costly, where supporting 15 Tbps VIP traffic for a 40K-server DC would require 4K SMuxes, and incurs high latency of 200 μsec to 1 msec for every packet handled by each SMux [14].

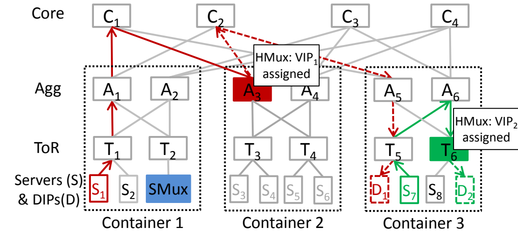

Figure 3: Duet architecture. Links marked with solid and dotted lines carry VIP and DIP traffic, respectively.

2.4 Duet LB

Duet [14] LB design consists of hardware switches and servers. Compared to Ananta, Duet lowers the LB cost by 12-24x and incurs a small latency of a few microseconds by using existing switches for load balancing. Duet runs hardware Mux (HMux) on every switch that stores the VIP-to-DIP mapping in the switch (ECMP) memory, splits the traffic for a VIP among its DIPs based on the hash value calculated over the 5-tuple in the IP header, and sends the packet to the selected DIP by encapsulating the packets using the tunneling table available in the switch.

However, switches have limited hardware resources, especially the routing and tunneling table space. The tunneling table size (typically 512 entries) limits the total number of DIPs (for multiple VIPs) that can be stored on a single HMux. Accordingly, Duet partitions the VIP-to-DIP mappings across HMuxes, where the mappings for a small set of VIPs are assigned to each HMux. This way of partitioning enables Duet to support a large number of DIPs. Second, the routing table size (typically 16K entries) per switch limits the total number of VIPs that can be supported in HMuxes. Therefore, Duet uses HMuxes to handle up to 16K *elephant VIPs*. The remaining mice VIP (that could not be assigned to any of the HMuxes) traffic is handled by deploying a small number of SMuxes, which have the same design as in Ananta. These SMuxes also load balance traffic otherwise handled by HMuxes during HMux failures which enables Duet to provide high availability.

3 Motivation

We next assess the impact of the VIP traffic characteristics (§2.2) on the DC network bandwidth usage under

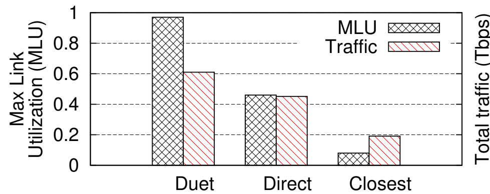

Figure 4: MLU and total traffic under various LB schemes. Total traffic is measured across all the DC network links.

Duet	Direct	Closest
5.47	3.94	1.78

Table 1: Path length for different LB designs.

the Duet LB. We simulate how Duet handles the VIP traffic using a 24-hour traffic trace from our production DC on a network topology that closely resembles our production DC. The topology, workload, and results are detailed in §10. Duet maximizes the total traffic handled by HMuxes, on average 97% in the 24-hour period.

High link utilization. Figure 4 shows the MLU and total traffic in the DC network[1]. While Duet is able to handle 97% of the total VIP traffic by leveraging HMuxes, it also inflates the MLU to 0.98 (or 98%). This high MLU can be explained by two design decisions of Duet.

First, Duet assigns a VIP only to a single HMux. But the traffic sources and DIPs for individual VIPs are spread in a large number of ToRs (Figure 1). The diverse location of traffic sources and DIPs per VIP suggests *no matter where the single Mux for a VIP is positioned in the network, it will be far away from most of the traffic sources and DIPs for that VIP*, and hence most VIP traffic will traverse through the network to reach the HMuxes and then the DIPs, which inflates the path length between the sources and DIPs.

Table 1 shows that the average number of hops between the sources and the DIPs across all individual VIPs is 5.47 in Duet. Notice that the traffic between two hosts that does not go through the LB would have a maximum of 4 hops (ToR-Agg, Agg-Core, Core-Agg, Agg-ToR). Thus the average path length of 5.47 in Duet indicates that most traffic goes through the core links and further experiences some detour in the DC network. Figure 3 shows an example where the VIP-1 traffic originated at S_1 has to travel 6 hops to reach DIP D_1 – 3 hops to reach the HMux at switch A_3, and 3 more hops to reach D_1.

To dissect the impact of the redirection, we measure the MLU and total traffic in the DC network in a hypothetical case where the HMuxes are located on a direct path between the sources and DIPs, labeled as "Direct". Figure 4 shows that in this case the MLU is reduced to 0.46 (from 0.98 in Duet), and the bandwidth used is lowered by 1.36x, compared to Duet. Also, the average path length in "Direct" is lowered to 3.94 (1.38x improvement). This means the redirection design in Duet inflates the MLU by 2.13x and bandwidth used by 1.36x.

[1] Absolute values for "total traffic" are omitted for confidentiality.

The second cause for the high link utilization is location-oblivious DIP selection in Duet. The HMux splits the VIP's traffic by hashing on the 5-tuples in the IP header, and chooses the DIP based on the hash. Thus, even if there is a DIP located under the same ToR as the HMux and has the capacity to handle all the local traffic for the VIP, the HMux will spread the local traffic among all DIPs, many of which can be far away in the DC.

To measure the impact of location-oblivious DIP selection, we measure the MLU and bandwidth used in a hypothetical case, where the traffic from the individual sources is routed to the closest DIP and assuming the HMuxes lie on the path. This mechanism is labeled as "Closest". Figure 4 shows that the MLU is reduced to just 0.08, and the bandwidth used reduces by 3.19x compared to Duet. Also, the average path length is lowered to just 1.78 hops.

Effective full bisection bandwidth reduced at core. Many DC networks have adopted topologies like FatTree and Clos [15] to achieve full-bisection bandwidth. Such networks guarantee that there is enough aggregate capacity between Core and Agg switches as between Agg and ToR switches, and hence the core links will never become a bottleneck for any traffic between the hosts.

However, traffic indirection can break this assumption, if the HMuxes reside in Agg or ToR switches. This happens to Duet, as Duet considers all the switches while assigning VIP-to-DIP mappings. This is illustrated in Figure 3. When VIP_1 is assigned to an Agg switch (A_3), the traffic from source S_1 travels the core links twice enroute to DIP D_1 – first to get to HMux A_3, and then to D_1. In contrast, direct host-to-host traffic only has to traverse core links at most once. As a result, the effective bandwidth in the core links is reduced – in Figure 3, the available bandwidth to container-2 (servers S_3-S_6) is reduced due to the LB traffic among other containers.

Our evaluation in §10.3 shows the traffic overhead in Duet, *i.e.,* the ratio of the additional traffic due to redirection to the total traffic without redirection is 44% in core links and 16% in containers. This means the remaining bisection bandwidth of the Agg-Core links is lower than the remaining bisection bandwidth in the ToR-Agg links. This breaks the full-bisection guarantee provided by the FatTree or Clos, which jeopardizes other applications that co-exist in the DC and assume full-bisection bandwidth is available (*e.g.,* [12, 23]).

4 RUBIK Overview

In the previous section, we saw that the traffic indirection in Duet incurs substantial overhead in the DC net-

work bandwidth usage. In this paper, we propose a new LB design, RUBIK, that significantly reduces the bandwidth usage in the DC network while providing low cost, high performance and high availability benefits.

RUBIK is based on two key ideas motivated by the observations in the last section. First, it exploits *locality*, *i.e.,* it tries to load balance traffic generated in individual ToRs across the DIPs present in the same ToRs. In this way, a substantial fraction of the load balanced traffic will not enter the links beyond ToRs which reduces the DC network bandwidth usage and MLU.

The second key idea of RUBIK is to exploit *DIP placement flexibility* to place DIPs closer to the sources. In RUBIK online services specify the number of DIPs for individual VIPs, and RUBIK decides the location of the servers to be assigned to individual VIPs. This idea is synergistic with the first idea, as it facilitates exploiting locality in load balancing within ToRs.

Realizing the two ideas is challenging, because (1) there are a limited number of servers in each ToR where DIPs can be assigned, (2) switches have limited memory for storing VIP-to-DIP mappings, (3) a VIP may have traffic sources in more ToRs than the total number of DIPs for that VIP. In such a case, a DIP cannot be assigned in every ToR that has traffic sources, (4) dependencies between the VIPs make it even harder, as the sources to some of the VIPs are not known until DIPs for other VIPs are placed.

RUBIK addresses the above capacity limitations (switch memory and DIPs in a ToR) using two complimentary ideas. First, RUBIK uses a new LB architecture that splits the VIP-to-DIP mapping for a VIP across multiple HMuxes to *enable* efficient use of switch memory while containing local traffic. Second, it employs a novel algorithm that *calculates* the most efficient use of switch memory for containing the most local traffic.

5 RUBIK Architecture

RUBIK uses a new LB design that splits the VIP-to-DIP mapping for each VIP into *multiple local and a single residual VIP-to-DIP mappings*. This idea is inspired by the observation that the traffic for individual VIPs is skewed (§2.2) – some ToRs generate more traffic than other ToRs for a given VIP. In RUBIK, we assign local mappings to the ToRs generating large fractions of the traffic and also assign enough DIPs to handle those traffic. The local mapping for a VIP load balances traffic for that VIP across the DIPs present under the same ToR (called local DIPs). We then assign a single residual mapping for that VIP to handle the traffic from all the remaining ToRs, where no local mapping is assigned.

Effectively, the VIP-to-DIP mapping for a VIP is split across the local and residual mappings such that a single DIP appears in only one mapping. Assigning a DIP only once makes the most efficient use of the limited tunneling table space of HMuxes so the total VIP traffic handled by the HMuxes can be maximized. The assignment module (§6) then calculates the actual assignment that maximizes the VIP traffic handled locally.

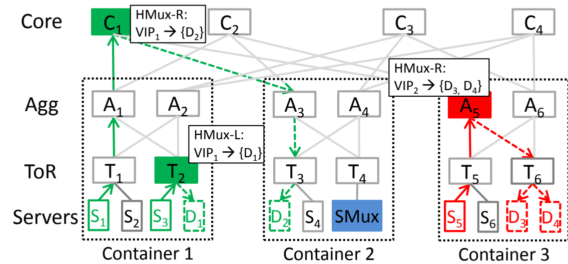

Figure 5: RUBIK Architecture. DIPs for VIP_1 are split in local (HMux-L) and residual mappings (HMux-R).

We now explain the RUBIK architecture in detail.

Local mapping. If some of the sources and DIPs for a VIP already reside in the same ToR, RUBIK exploits this locality by load balancing the source traffic across those local DIPs. To ensure that the traffic does not flow outside the ToR in detouring through the HMux, RUBIK stores a subset of the VIP-to-DIP mapping, *i.e.,* containing only the local DIPs, at the ToR itself (*e.g.,* HMux T_2 in Fig. 5). We denote such a mapping containing the subset of local DIPs as a *local mapping*.

Residual mapping. For an individual VIP, we assign a single residual mapping to handle the remaining traffic not handled by the local mappings (called residual traffic). We pool all the remaining DIPs for a VIP together in a single DIP-set, called the *residual mapping* for that VIP (*e.g.,* HMux on C_1 and A_5 in Fig. 5). The residual mapping for each VIP announces the VIP using BGP so that other routers (or switches) route the VIP traffic to the HMux where its residual mapping is assigned.

In principle, we can replicate the residual mapping at all the ToRs containing any remaining traffic sources. Such replication can reduce the number of hops between the sources and HMux, but it can also consume a significant amount of the limited tunneling table space. Therefore, we only assign the residual mapping for a VIP to a single HMux, and the optimal choice of HMux to store the residual mapping of a VIP depends on the location of the remaining traffic sources and residual DIPs.

VIP routing. The above DIP-set splitting design has one potential problem. If the HMuxes storing either the local mappings or the residual mapping of a VIP all announce the VIP via BGP to the network, some of the residual source traffic may be routed towards the HMuxes storing local mappings if they are closer than the HMux storing the residual mapping. This would significantly complicate the DIP placement, and DIP-set splitting and placement problem. We avoid this complication by making the

HMuxes storing local mappings *not* announce the VIP via BGP. In this way, only local source traffic within a ToR sees the local mapping and is split to the local DIPs.

SMuxes. Because of the limited switch memory, the numbers of VIPs and DIPs supported by HMuxes remain limited. Current HMuxes can support up to 16K VIPs [14], and our DC has 30K+ VIPs. Also, it remains challenging to provide high availability during HMux failures. We address both problems by deploying a small number of SMuxes as a backstop, to handle the VIP traffic that could not be handled using HMuxes. We also announce all the VIPs from all SMuxes. We use Longest prefix matching (LPM) to: (1) preferentially route the VIP traffic to the HMuxes for the VIPs assigned to both HMuxes and SMuxes, (2) route the traffic to the remaining VIPs not assigned to HMuxes to the SMuxes.

The use of SMuxes in this way also provides high availability during residual mapping failure. §7 gives details on how RUBIK recovers from a variety of failures.

Summary. The benefits of this architecture can only be realized by carefully calculating the DIP placement, and local and residual mappings for individual VIPs subject to a variety of constraints, which we describe next.

6 Joint VIP and DIP Assignment

RUBIK's objective is to maximize the traffic handled by the HMuxes, while maximizing the traffic handled locally within ToRs. The assignment algorithm determines for each VIP, (1) the location of its DIPs; (2) the number of DIPs in each ToR in the local VIP-to-DIP mapping; and (3) the number of DIPs in the residual mapping, and the HMux assigned to store the mapping.

RUBIK needs to calculate this assignment such that the capacity of all resources (switch tables, links, and servers per ToR) is not be exceeded. Also, RUBIK needs to ensure that it assigns DIPs in the failure domains (*i.e.,* ToRs) specified by the online services. The placement calculated at a given time may lose effectiveness over time as the VIP traffic changes, and VIPs and DIPs are added and removed. To adapt to such cloud dynamics, RUBIK reruns the placement algorithm from time to time. While calculating a new assignment, RUBIK has to ensure that the number of machines migrated from the old assignment is under the limit.

The assignment problem is a variant of the bin-packing problem (NP-hard [11]), where the resources are the bins, and the VIPs are the objects. It is further compounded because the VIP traffic exhibits hierarchical dependencies (§2.2).

To reduce the complexity, RUBIK decomposes the joint assignment problem into two independent modules, (1) *DIP and local mapping placement*, (2) *residual mapping placement*, as shown in Algorithm 1. The first module places the DIPs and local mappings for all the VIPs to

Algorithm 1: RUBIK Assignment Algorithm

```
1 Input: V, M, N_v, f_v, S, L, b_{t,v}, C_{t,v}
2 Output: x^D_{t,v}, x^M_{t,v}
3 topological_sort(V in DAG)
4 for l = 1, depth of DAG do
5 |   local_mapping_and_dip_placement(VIPs in
  |   DAG_level(l))
6 end
7 residual_mapping_placement()
```

Notation	Explanation
	Input
S, L, V	Sets of switches, links, and VIPs
M_t	# servers under t-th ToR
T_s	Table capacity of s-th switch
L_e	Link traffic capacity of link e
N_v, f_v	#DIPs and failure-domain for v-th VIP
$b_{t,v}$	Traffic sent to v-th VIP from t-th ToR
$C_{t,v}$	Traffic capacity of server in t-th ToR when assigned to v-th VIP
	Variables
$x^D_{t,v}$	Number of servers (DIPs) in t-th ToR assigned to v-th VIP
$x^M_{t,v}$	Number of table entries in t-th ToR assigned to v-th VIP

Table 2: Notations used in the algorithm.

maximize the total traffic load-balanced locally on individual ToRs. We calculate DIP and local mapping simultaneously, because the problem of DIP and local mapping placement are intertwined, as the traffic for a VIP is contained within a ToR only if the ToR has (1) enough DIPs to handle the traffic, and (2) enough memory to store the corresponding VIP-to-DIP mapping.

Since the VIP traffic exhibits hierarchical dependencies (§2.2), we create a DAG that captures the traffic flow and hence the dependency between the VIPs, and then perform a topological sort on the DAG to divide the VIPs into different levels. We then place the DIPs and local mappings for the VIPs level-by-level (lines 3:6 in Algo. 1). As we place DIPs for VIPs in one level, the sources in the next level become known.

The second module places the residual mappings of all the VIPs to maximize the total traffic handled by HMuxes. The residual mapping placement subproblem remains NP-hard. But, the residual VIP traffic is typically only a small portion of the total traffic and hence we can apply heuristics to solve it without significantly affecting the quality of the overall solution (line 7).

6.1 DIP and Local Mapping Placement

The first module places the DIPs and the local mappings of all the VIPs for which the sources are known such that the total VIP traffic load balanced within the

ToRs is maximized. We formulate the joint DIP and local mapping placement problem as ILP using notations shown in Table 2 as follows.

Input: The input includes (1) the network topology and resource information (capacity of switch tables, links, and servers in the ToRs), (2) for every VIP in current level, the number of DIPs and number of failure domain and traffic, and (3) max. number of DIPs to migrate (δ).

Output/Variables: The output includes the local VIP-to-DIP mappings on individual ToRs, and placement of all the DIPs (including residual DIPs), for all VIPs.

Let $x^D_{t,v}$ denote the number of machines in the t-th ToR assigned as the DIPs for the v-th VIP, and $x^M_{t,v}$ denote the number of machines out of these $x^D_{t,v}$ machines that are used in the local mapping for the VIP, *i.e.,* they will appear in the local VIP-to-DIP mapping of the t-th ToR.

Objective:

maximize Locality $L = \sum_{v \in V} \sum_{t \in T} y^M_{t,v} \cdot b_{t,v}$

where $y^D_{t,v}$ is set if there are any DIPs in the t-th ToR assigned to the v-th VIP, and $y^M_{t,v}$ is set if the t-th ToR switch (HMux) contains local VIP-to-DIP mapping for the v-th VIP. This way, $y^M_{t,v} \cdot b_{t,v}$ denotes if traffic for v-th VIP in t-th ToR is handled locally, and we maximize traffic handled locally across all VIPs and ToRs.

$$y^M_{t,v} = \begin{cases} 1 & x^M_{t,v} \geq 1 \\ 0 & \text{Otherwise} \end{cases} \qquad y^D_{t,v} = \begin{cases} 1 & x^D_{t,v} \geq 1 \\ 0 & \text{Otherwise} \end{cases}$$

Constraints:

(1,2) Switch table size and number of servers not exceeded on every ToR

$$\forall t \in T, \sum_{v \in V} x^M_{t,v} \leq T_t, \quad \sum_{v \in V} x^D_{t,v} \leq M_t$$

(3,4) Specified number of DIPs assigned for every VIP; failure domain constraints

$$\forall v \in V, \sum_{t \in T} x^D_{t,v} = N_v, \quad \sum_{t \in T} y^D_{t,v} \geq f_v$$

(5a, 5b) DIPs are not overloaded (no hot-spots)

$$\forall t \in T, \forall v \in V, y^M_{t,v} \cdot b_{t,v} \leq x^M_{t,v} \cdot C_{t,v}$$

$$\forall v \in V, \sum_{t \in T} (1 - y^M_{t,v}) \cdot b_{t,v} \leq \sum_{t \in T} (x^D_{t,v} - x^M_{t,v}) \cdot C_{t,v}$$

Constraint (5a) ensures the DIPs mapped in the local mapping are not overloaded. Constraint (5b) ensures the DIPs in the residual mapping are not overloaded.

(6) Limiting the number of DIP moves

$$\sum_{v \in V, t \in T} |x^M_{t,v} - x^{M,old}_{t,v}| \leq \delta$$

where $x^{M,old}_{t,v}$ denotes the number of DIPs in the ToR in the previous assignment, and δ is the threshold on the maximum number of DIPs to be moved. We convert constraint (6) into the linear form as:

$$\sum_{v \in V, t \in T} z_{t,v} \leq \delta$$

$$\forall t \in T, \forall v \in V, z_{t,v} \geq x^M_{t,v} - x^{M,old}_{t,v}, z_{t,v} \geq x^{M,old}_{t,v} - x^M_{t,v}$$

(7) ToRs have more DIPs than in local mappings

$$\forall v \in V, t \in T, x^D_{t,v} \geq x^M_{t,v}$$

(8a,8b) Writing $y^M_{t,v}$, $y^D_{t,v}$ in linear form

$$\forall t \in T, \forall v \in V, 0 \leq y^M_{t,v}, y^D_{t,v} \leq 1, y^M_{t,v} \leq x^M_{t,v}, y^D_{t,v} \leq x^D_{t,v}$$

6.2 Residual Mapping Placement

The second module places the residual mappings for the VIPs among the switches while maximizing the total VIP traffic load balanced by the residual mapping HMuxes (traffic not handled by local mappings), subject to switch memory and link capacity constraints.

This assignment problem is the same as that in Duet, and we solve it using the same heuristic algorithm as in Duet. Briefly, to assign the VIPs, we first sort the VIPs in decreasing traffic volume, and attempt to assign them one by one. We define the notion of maximum resource utilization (MRU). MRU represents the maximum utilization across all resources – switches and links. To assign a given VIP, we consider all switches as candidates. We calculate the MRU for each assignment, and pick the one that results in the smallest MRU, breaking ties at random. If the smallest MRU exceeds 100%, *i.e.,* no assignment can accommodate the traffic of the VIP, the algorithm terminates. The remaining VIPs are not assigned to any switch – their traffic will be handled by the SMuxes.

7 Failure Recovery

A key requirement of the LB design is to maintain high availability during failure: (1) the traffic to any VIP should not be dropped, (2) existing connections should not be broken. As in Duet, RUBIK relies on SMuxes to load balance the traffic during various failures. In addition to storing VIP-to-DIP mapping for all the VIPs, we use the ample memory on individual SMuxes to provide connection affinity by maintaining per-connection state.

Residual mapping HMux failure: Failure of the HMux storing the residual mapping of a VIP only affects the traffic going to that HMux; the traffic handled by other local and residual mappings is unaffected. The routing entries for the VIPs assigned to the failed HMux are removed from all other switches via BGP withdraw messages. After routing convergence, traffic to these VIPs is routed to the SMuxes, which announce all VIPs. Since each SMux stores the same residual DIPs and uses the same hash function as the residual mapping HMux to select a DIP, existing connections are not broken.

Local mapping failure: When a ToR switch fails, all the sources and DIPs for a VIP under it are also discon-

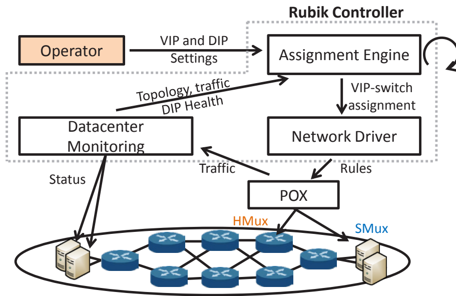

Figure 6: RUBIK implementation.

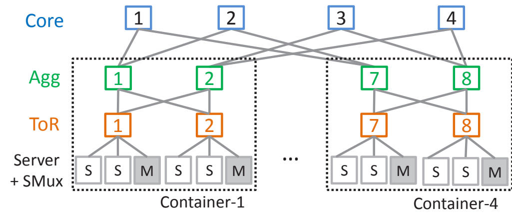

Figure 7: Our testbed. FatTree with 4 containers connected to 4 Core switches.

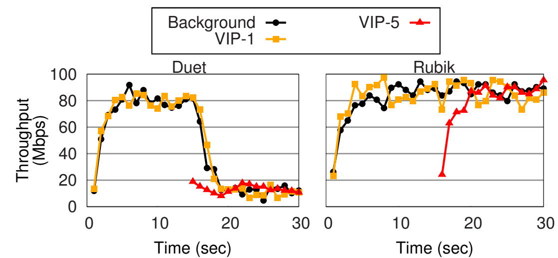

Figure 8: RUBIK reduces congestion.

nected. As a result, the traffic the local mapping was handling also disappears. Further, the rest of the traffic for that VIP continues to be routed to the residual mapping or other local mappings, and are not affected.

SMux failure: On an SMux failure, traffic going to that SMux is rerouted to the remaining SMuxes using ECMP. The connections are not broken as all the SMuxes use the same hash function.

DIP failure: Existing connections to the failed DIP would necessarily be terminated. For VIPs whose mapping are assigned to SMuxes, connection to the remaining DIPs are maintained as SMuxes use consistent hashing in DIP selection [21]. For VIPs assigned to HMuxes, the connections are maintained using smart hashing [2].

8 Implementation

We briefly describe the implementation of the three building blocks of RUBIK, (1) RUBIK controller, (2) network driver, (3) HMux and SMux, as shown in Figure 6.

RUBIK **controller:** The controller orchestrates all control activities in RUBIK. It consists of three key modules: (1) DC monitor, (2) Assignment engine, (3) Network driver. The DC monitor periodically captures the traffic and DIP health information from the DC network and sends it to the assignment engine. The assignment engine calculates the DIP placement, local and residual VIP-to-DIP mappings for all the VIPs, and pushes these new assignment to the network driver. We use CPLEX [7] to solve the LP (§6.1).

Network driver: This module is responsible for maintaining VIP and DIP traffic routing in the LB. Specifically, when the VIP-to-DIP assignment changes, the network driver announces or withdraws routes for the changed VIPs according to BGP.

HMux and SMux: We implement HMuxes and SMuxes using Open vSwitches that split the VIP traffic among its DIPs using ECMP based on the source addresses [5]. We implement smart hashing [2] using OpenFlow rules. The replies from the DIPs directly go to the sources using DSR [21].

Lastly, we use POX to push the rules and poll the traffic statistics. We developed a separate module to monitor the DIP health. The code for all the modules consists of 3.4K LOC in C++ and Python.

9 Testbed

Setup: We evaluate RUBIK prototype using Open vSwitches and Mininet. Our testbed (Fig. 7) consists of 20 switches (HMuxes) in 4 containers connected in a FatTree topology. Each ToR contains an SMux (marked "M") and 2 hosts that can be set as DIPs (marked "S").

Services: We evaluate the performance of RUBIK using two services that require load balancing: (1) HTTP web service, (2) Bulk data transfer service. The web service serves static web pages of size 1KB and generates a large number of short-lived TCP flows. The Bulk data transfer service receives a large amount of data using a small number of long-lived TCP flows. All the servers and clients for these services reside in the same DC.

Experiments: Our testbed evaluation shows: (1) RUBIK lowers congestion in the network; (2) RUBIK achieves high availability during a variety of failures – local mapping, residual mapping, and DIP failure.

9.1 Reduction in Congestion

First we show that RUBIK reduces congestion in the DC network by using local mappings. In this experiment, initially 4 VIPs (each with 1 source and 1 DIP) are assigned to 4 different HMuxes. Additionally, there is background traffic between 2 hosts. Figure 8 shows the per-second throughput measured across 2 flows. "VIP-1" denotes the throughput for one of the 4 VIPs added initially. "Background" denotes the throughput for the background flow (not going through the LB). Initially, there is no congestion in the network and as a result all flows experience high throughput. At time 15 sec, we add a new VIP (VIP-5) that has 2 DIPs and 2 sources sending equal volume of traffic, and assign it using Duet.

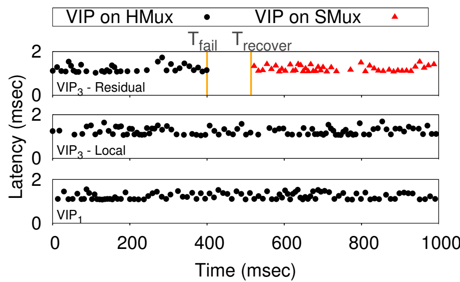

Figure 9: VIP availability when residual mapping fails.

However, assigning the new VIP causes congestion as the new flows compete with the old flows. As a result, the throughput for all the flows drop by almost 5-6x.

We repeat the same experiment with RUBIK. At time 15 sec, we assign the VIP-5 using RUBIK. RUBIK assigns local mappings to handle the VIP-5 traffic. As a result, adding VIP-5 does not cause congestion (no drop in throughput), as shown in Figure 8. This experiment shows that by exploiting locality, RUBIK reduces the congestion and improves the throughput by 5-6x.

9.2 Failure Mitigation

Next we show how RUBIK maintains high availability during various failures.

Residual mapping failure: Fig. 9 shows the availability of the VIP, measured using ping latency, when its residual mapping fails. In this experiment, we have 3 VIPs (VIP-1, 2, 3) assigned to the data-transfer service. VIP-1 and VIP-2 have one source and one DIP each in different ToRs, and their traffic is handled by residual mappings (no local mapping). VIP-3 has two sources and two DIPs. One source and one DIP are in the same ToR – the local mapping on that ToR handles their traffic. The remaining source and DIP are in two different ToRs, and their traffic is handled by the residual mapping.

At 400 msec, we fail the HMux storing the residual mapping for VIP-3. We make four observations: (1) On HMux failure, VIP-3 traffic handled by it is lost for 114 msec. (2) After 114 msec, VIP-3 is 100% available, *i.e.,* all of the pings are successful again. During this time, the routing converges, and the traffic that used to go to the HMux is rerouted to the SMuxes. (3) The traffic for VIP-3 handled by the local mapping (shown as VIP-3-Local) is not affected – no ping message is dropped. (4) Other VIPs (only VIP-1 is shown) are not affected – their ping messages are not dropped.

This shows that RUBIK provides high availability during residual mapping failure.

Local mapping failure: Figure 10 shows the impact of local mapping failure on the availability of the VIPs. We use the same setup as before, and fail the HMux where VIP-3's local mapping was assigned. We measure the ping message latency from 2 sources for VIP-3 (de-

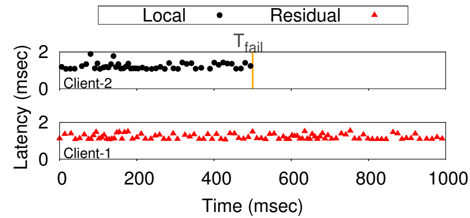

Figure 10: VIP availability during local mapping failure.

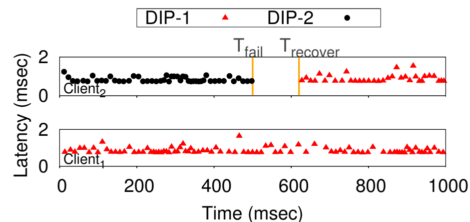

Figure 11: VIP availability during DIP failure.

noted as Client-1, 2). The traffic from Client-2 is handled locally, whereas Client-1 traffic is handled by the residual mapping. When local mapping fails (at 500 msec), all the sources and DIPs under it disappear. Therefore, ping messages for Client-2 are lost as Client-2 itself is down. Figure 10 shows that the traffic from Client-1, which is handled by the residual mapping, is not affected.

DIP failure: Lastly, we evaluate the impact of DIP failure on service availability. In this experiment, we use a single VIP with 2 sources (Client-1, 2) and 2 DIPs (DIP-1, 2), located in different ToRs. Therefore, both DIPs are assigned to the residual mapping. Initially, the traffic from Client-1 is served by DIP-1 and that of Client-2 is served by DIP-2. We fail DIP-2 at 500 msec.

Figure 11 shows the latency for the ping messages from Client-1 and Client-2. When DIP-2 fails, the ping messages for Client-2 are lost for about 120 msec. After 120 msec, Client-2 traffic is served by DIP-1. This is because when DIP-2 fails, the residual mapping is adjusted using smart-hashing, *i.e.,* the traffic going to the failed DIP is split across the remaining DIPs. As a result, the traffic going to DIP-2 is now served by DIP-1. It can also be seen that Client-1 traffic is not affected – there is no drop in the ping messages. This shows that a DIP failure does not affect the traffic going to other DIPs, and traffic going to the failed DIP is spread across remaining DIPs.

10 Simulation

In this section, we use large-scale simulations of RUBIK and Duet to show: (1) RUBIK handles a large percentage of traffic in HMuxes as in Duet but incurs significantly lower maximum link utilization (MLU); (2) RUBIK reduces the traffic in the core by 3.68x and in the container by 3.47x; (3) RUBIK contains 63% of VIP traffic within ToRs; (4) RUBIK does not create hotspots.

Network: Our simulated network closely resembles

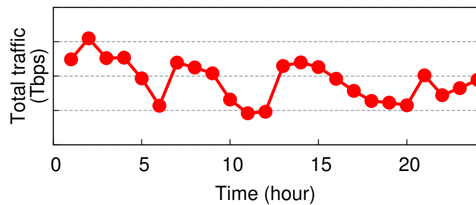

Figure 12: Total traffic variation over 24 hours.

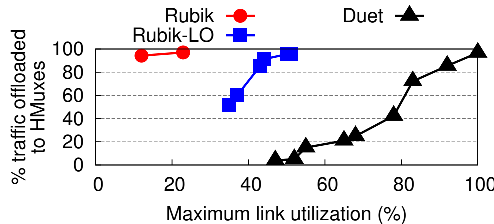

Figure 13: Traffic handled by HMuxes vs. MLU.

that of a production DC, with a FatTree topology connecting 500K VMs under 1600 ToRs in 40 containers. Each container has 40 ToRs and 4 Agg switches, and the 40 containers are connected with 40 Core switches. The link and switch memory capacity were set with values observed in the production DC.

Workload: We run the experiments using traffic trace collected from the production DC over a 24-hour duration. The trace consists of the number of bytes sent between all sources and all VIPs. Figure 12 shows the total traffic per hour fluctuates over the 24-hour period[2].

Comparison: We compare the performance of Duet, RUBIK-LO and RUBIK. Duet exploits neither locality nor DIP placement. RUBIK-LO is a version of RUBIK that only exploits locality without moving the DIPs; it assumes DIP placement is fixed and given, and only calculates the local and residual mappings. RUBIK exploits both locality and flexibility in moving the DIPs. RUBIK performs stage-by-stage VIP-to-DIP mapping assignment following the VIP dependency.

10.1 MLU Reduction

We first compare the trade-off between the MLU and fraction of the traffic handled by the HMuxes under the three schemes. Note that all three schemes try to maximize the total traffic handled by HMuxes. The traffic not handled by HMuxes is handled by SMuxes.

Figure 13 shows the fraction of traffic handled by HMuxes under the three schemes. The MLU shown is the total MLU which resulted from load balancing all VIP traffic, handled by HMuxes and by SMuxes. We see that Duet can handle 97% traffic using HMuxes, but incurs a high MLU of 98%. But when MLU is restricted to 47%, Duet can only handle 4% traffic using HMuxes.

In contrast, RUBIK-LO handles 97% VIP traffic using HMuxes at MLU of 51%. It handles 52% of VIP traffic using HMuxes at MLU of 35%. This improvement over Duet comes purely from exploiting locality.

Lastly, RUBIK significantly outperforms both Duet and RUBIK-LO. It handles 97% traffic with a low MLU of 22.9%, a 4.3x reduction from Duet. Also, at a MLU of 12%, RUBIK handles 94% traffic using HMuxes.

[2] Absolute values are omitted for confidentiality.

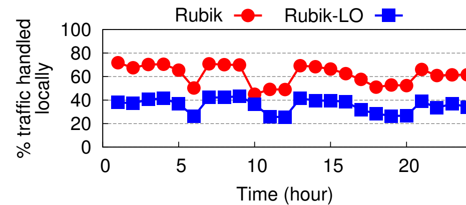

Figure 14: Traffic handled using local mappings.

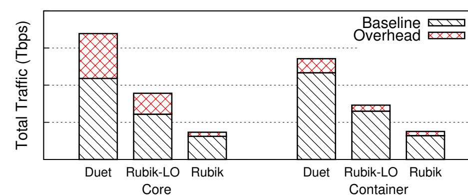

Figure 15: Total traffic in core and container.

10.2 Traffic Localized

RUBIK significantly reduces the MLU by containing significant amount of traffic within individual ToRs. Figure 14 shows the fraction of the total traffic contained within ToRs in RUBIK and RUBIK-LO over the 24-hour period, where these mechanisms calculate new assignment every hour. In RUBIK, we limit the machine moves to 1% based on the trade-off detailed in §10.5.

We see that RUBIK-LO localizes 25.5-43.4% (average 34.8%) of the total traffic within ToRs, and RUBIK localizes 46-71.8% (average 63%) of the total traffic within ToRs. Additionally, for the VIPs generating 90% of the total VIP traffic, we find that, the local mappings handle traffic from 37.8-48.6% (average 41.8%) sources, and 50.2-57.7% (average 53%) of the total DIPs are assigned to their local mappings.

10.3 Traffic Reduction

Figure 15 shows the total bandwidth usage across all the links caused by the VIP traffic under the three mechanisms. We separately show the total traffic on the core links (between Core and Agg switches) and containers links (between ToR and Agg switches). The total traffic shown is the average over 24 hours. Furthermore, we break down the total traffic into baseline and overhead due to redirection. The baseline traffic shows the amount of traffic generated if the HMuxes were on the direct path between source and DIPs, which would cause no redirection. The remaining traffic is the extra traffic due to the

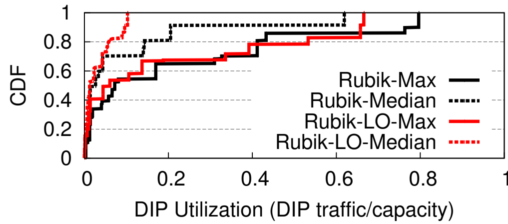

Figure 16: DIP utilization distribution across VIPs.

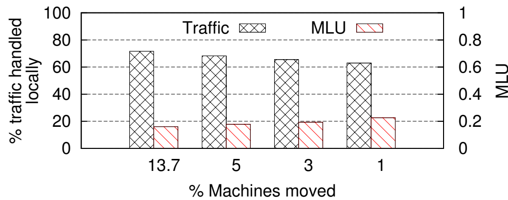

Figure 17: Impact of machine moves.

redirection to route traffic to and from HMuxes.

RUBIK and RUBIK-LO significantly reduce the total traffic in the core network and containers. Compared to Duet, on average RUBIK-LO reduces the total traffic by 1.94x and 1.88x, respectively. RUBIK reduces the total traffic by 3.68x and 3.47x, respectively.

Secondly, RUBIK-LO and RUBIK reduce the traffic overhead due to traffic redirection by 2.1x and 10.9x compared to Duet. It should be noted that both RUBIK and RUBIK-LO cannot eliminate the traffic overhead, because they cannot localize 100% of the VIP traffic. As a result, the traffic not localized is handled by the HMuxes storing residual mappings, which causes traffic detour.

10.4 DIP Load Balance

To exploit locality, RUBIK partitions the DIPs for a VIP into local and residual DIP-sets, which can potentially overload some of the DIPs (hotspots). We calculate the average and peak DIP utilization (DIP traffic/capacity) across all DIPs for every VIP. Figure 16 shows the CDF across all VIPs in RUBIK and RUBIK-LO. It shows that both schemes ensure that the peak utilization for all the DIPs is well under 80%, which is the constraint given to the assignment algorithm. Furthermore, for 80% VIPs, the peak utilization is under 40%. This shows RUBIK does not create hotspots.

10.5 Impact of Limiting Machine Moves

Lastly, we evaluate the impact of limiting machine moves in RUBIK's assignment LP formulation (§6.1) on the fraction of traffic localized and MLU. Figure 17 shows the two metrics as we reduce the percentage machine moves allowed. Without any restriction, RUBIK assignment results in moving 13.7% of the DIPs. When the percentage machine moves is 1%, the fraction of traffic localized decreases by 8.7% whereas the MLU increases by 6.6%, and the execution time to find the solution increases by 2.3x compared to unrestricted machine moves. This shows that most of the benefits of RUBIK are maintained after restricting the machine moves to just 1%. We therefore used this threshold in all the previous simulations and testbed experiments.

11 Related work

To our best knowledge, RUBIK is the first LB design that exploits locality and end-point flexibility. Below we review work related to DC LB design which has received much attention in recent years.

LB: Traditional hardware load balancers [4, 1] are expensive and typically only provide 1+1 availability. We have already discussed Duet [14] and Ananta [21] load balancers extensively. Other software-based load balancers [6, 8, 9, 3] have also been proposed, but they lack the scalability and availability of Ananta [21]. In contrast to these previous designs, RUBIK substantially reduces the DC network bandwidth usage due to traffic indirection while providing low cost, high performance benefits.

OpenFlow based LB: Several recent proposals focus on using OpenFlow switches for load balancing. In [24], the authors present a preliminary LB design using OpenFlow switches. They focus on minimizing the number of wildcard rules. In [18], the authors propose a hybrid hardware-software design and propose algorithms to calculate the weights for splitting the VIP traffic. Plug-n-Serve [16] is another preliminary design that uses OpenFlow switches to load balance web servers deployed in unstructured, enterprise networks. In contrast, RUBIK is designed for DC networks and efficiently load balances the traffic by exploiting locality and end-point flexibility.

SDN architecture and middleboxes: Researchers have leveraged the SDN designs in the context of middleboxes for policy enforcement and verification [22, 13], which is a different goal from RUBIK. Researchers have also proposed using OpenFlow switches for a variety of other purposes. *e.g.,* DIFANE [25] and vCRIB [19] use switches to cache rules and act as authoritative switches. Again their main focus is quite different from RUBIK.

12 Conclusion

RUBIK is a new load balancer design that drastically reduces the bandwidth usage while providing low cost, high performance and reliability benefits. RUBIK achieves this by exploiting two design principles: (1) locality: it load balances traffic generated in individual ToRs across DIPs present in the same ToRs, (2) end-point flexibility: it places the DIPs closer to the traffic sources. We evaluate RUBIK using a prototype implementation and extensive simulations using traces from our production DC. Our evaluation shows together these two principles reduce the bandwidth usage by the load balanced traffic by over 3x compared to prior art Duet.

References

[1] A10 networks ax series. `http://www.a10networks.com`.

[2] Broadcom smart hashing. `http://http://www.broadcom.com/collateral/wp/StrataXGS_SmartSwitch-WP200-R.pdf`.

[3] Embrane. `http://www.embrane.com`.

[4] F5 load balancer. `http://www.f5.com`.

[5] Fattree routing using openflow. `https://github.com/brandonheller/ripl`.

[6] Ha proxy load balancer. `http://haproxy.1wt.eu`.

[7] Ibm cplex lp solver. `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`.

[8] Loadbalancer.org virtual appliance. `http://www.load-balancer.org`.

[9] Netscalar vpx virtual appliance. `http://www.citrix.com`.

[10] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *EuroSys 2011*.

[11] C. Chekuri and S. Khanna. On multi-dimensional packing problems. In *SODA, 1999*.

[12] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *SIGCOMM 2013*.

[13] S. Fayazbakhsh, V. Sekar, M. Yu, and J. Mogul. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. *Proc. HotSDN*, 2013.

[14] R. Gandhi, H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *SIGCOMM 2014*.

[15] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *SIGCOMM 2009*.

[16] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-serve: Load-balancing web traffic using openflow. *ACM SIGCOMM Demo*, 2009.

[17] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. *SIGCOMM*, 2013.

[18] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Niagara: Scalable load balancing on commodity switches. In *Technical Report (TR-973-14), Princeton, 2014*.

[19] M. Moshref, M. Yu, A. Sharma, and R. Govindan. Scalable rule management for data centers. In *NSDI*, 2013.

[20] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, et al. The case for ramcloud. *Communications of the ACM, 2011*.

[21] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, et al. Ananta: Cloud scale load balancing. In *SIGCOMM*, 2013.

[22] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. In *SIGCOMM, 2013*.

[23] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI 2012*.

[24] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *Usenix HotICE*, 2011.

[25] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. In *SIGCOMM*, 2010.