Load Balancing with JET: Just Enough Tracking for Connection Consistency

Gal Mendelson
Stanford University

Shay Vargaftik
VMware Research

Dean H. Lorenz
IBM Research – Haifa

Kathy Barabash
IBM Research – Haifa

Isaac Keslassy
Technion

Ariel Orda
Technion

ABSTRACT

Hash-based stateful load-balancers employ connection tracking to avoid per-connection-consistency (PCC) violations that lead to broken connections. In this paper, we propose Just Enough Tracking (JET), a new algorithmic framework that significantly reduces the size of the connection tracking tables for hash-based stateful load-balancers without increasing PCC violations.

Under mild assumptions on how backend servers are added, JET adapts consistent hash techniques to identify which connections do not need to be tracked. We provide a model to identify these *safe* connections and a pluggable framework with appealing theoretical guarantees that supports a variety of consistent hash and connection-tracking modules.

We implement JET in two different environments and with four different consistent hash techniques. Using a series of evaluations, we demonstrate that JET requires connection-tracking tables that are an order of magnitude smaller than those required with full connection tracking while preserving PCC and balance properties. In addition, JET often increases the lookup rate due to improved caching.

CCS CONCEPTS

• **Networks** → **Network resources allocation**; **Middle boxes / network appliances**; **Packet scheduling**.

ACM Reference Format:
Gal Mendelson, Shay Vargaftik, Dean H. Lorenz, Kathy Barabash, Isaac Keslassy, and Ariel Orda. 2021. Load Balancing with JET: Just Enough Tracking for Connection Consistency. In *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21), December 7–10, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3485983.3494851

1 INTRODUCTION

Load balancing is a central building block in modern datacenters and cloud applications. A load balancer (LB) is required to dispatch millions of connections, or flows, to a multitude of servers, with servers being added and removed [14, 28]. Therefore, efficient and simple flow dispatching mechanisms are vital to the manageability, scalability, and performance of such systems.

When designing a load balancer, and specifically the mechanisms that are responsible for assigning flows to destinations, an essential requirement is to maintain per-connection-consistency (PCC). Namely, packets of the same flow must arrive at the same destination. Our interest and the focus of our work lies in hash-based load balancing that relies on connection tracking (CT) to maintain PCC. With this predominant approach [14, 18], often referred to as *hash-based* stateful load balancing, a flow's destination is determined by applying a hash function on the connection's identifier (*e.g.,* the TCP 5-tuple) of its first packet. The CT keeps for every flow its chosen destination, and a CT table lookup is performed on each packet to make sure PCC is not violated.

Despite its appealing attributes, the main limitation of CT is that large CT tables have a large memory footprint which often leads to slower lookups. Moreover, in systems with high processing rate requirements and memory limitation, tracking all connections may not be feasible, leading to PCC violation [7, 27].

Our goal is to address this limitation by significantly reducing the CT requirements of existing hash-based stateful load balancers and do so with a negligible computational overhead while preserving PCC. To achieve our goal, we first reexamine the fundamental question: which connections should a hash-based LB track to maintain PCC? Generally, due to the dynamic nature of changes to the backend servers, the answer is *all* connections, leading to full connection tracking (full CT) solutions. Taking a deeper look reveals that this property is not inherent. Ultimately, the specific LB decision mechanisms and backend server changes determine which connections should be tracked.

With mild assumptions on how backend servers are added, we are able to identify connections that do not need to be tracked to maintain their PCC. These are precisely the connections for which the destination choice made by the LB is not affected by a backend change. By not tracking these connections, we can significantly reduce the CT tables' size without sacrificing PCC.

In this work, we propose JET, an algorithmic framework to reduce the size of connection tracking tables by identifying connections that do not require tracking. JET can be integrated with existing hash-based stateful load balancing schemes, utilizing the existing decision processes to choose the destination for each new connection and identifying if it can forgo tracking.

Particularly, we show that for decision processes based on consistent hashing (CH) [19, 23, 31], most connections do not need tracking, and often the size of connection tracking tables can be significantly reduced. Moreover, we are able to identify such connections with minimal computational overhead. We show how JET

CoNEXT '21, December 7–10, 2021, Virtual Event, Germany

operates with several consistent hash techniques and derive strong theoretical guarantees on its expected performance.

We implement JET in two different environments (C++ and Python) and with multiple CH algorithms. Using a series of evaluations, including event-driven simulations and evaluations over real and synthetic traces, we demonstrate that JET requires CT tables that are an order of magnitude smaller than those required with full CT. In addition, JET often increases the lookup rate due to improved caching while maintaining the same balance properties.

To summarize, our contributions are: (1) A model to identify connections that do not require tracking; (2) The JET algorithmic framework that can be integrated with different connection dispatching methods; (3) Efficient implementations of JET for four different consistent hash techniques; (4) Theoretical guarantees, including the expected reduction in connections that require tracking; (5) Evaluation through event-driven simulation and over real and synthetic traces. The code for JET is available at [34].

1.1 Related Work

The decision-making techniques employed by some LB designs are related to JET. Particularly, we are interested in related mechanisms that preserve PCC. These are generally divided into two categories: stateful and stateless, which we overview next.

Stateful LB. Stateful LBs, *e.g.,* Ananta [28], Duet [15], Maglev [14] and Katran, [18], rely on full connection tracking to maintain PCC. As mentioned, large CT tables introduce a significant memory footprint which may also lead to a slow-down of the lookup rate. The software (SW)-based LBs cope with this problem by deploying more LB instances and implementing specific optimizations. Several LBs use programmable hardware (HW) to speed up the CT lookup process, e.g., Duet [15] uses forwarding and ECMP table-based virtual-to-physical mappings. SilkRoad [24] and SHELL [29] propose to use state-of-the-art programmable switches and tackle the memory limitation challenges of HW-based LBs. Lastly, Prism [11] is a recent hybrid LB design that combines full SW-level CT with partial programmable HW-level CT, copying flow states from SW to HW when needed.

Stateless LB. Stateless load balancers avoid using CT altogether. Instead, they rely on other network elements to maintain PCC. One approach is to encode the tracking information into the data plane (e.g., packet headers), e.g., Cheetah [7], QUIC-LB [12]. Another approach is to utilize the backend servers to track PCC, as they track their active connections in any case. The backend servers identify PCC violations and reroute ("daisy-chain") packets to the correct backend, e.g., Beamer [27], Faild [6].

The goal of our work is to reduce the CT table sizes of hash-based stateful LBs while maintaining PCC. A comparison between the stateful and the stateless approaches is out of the scope of our work. We refer the reader to [7] for an in-depth comparison and discussion.

Consistent Hashing. Several stateful and stateless LBs, e.g., [6, 14], use consistent hashing (CH) to make dispatching choices. This lowers the overhead of maintaining PCC. For stateless LBs, this reduces the number of connections that need rerouting; for stateful LBs, fewer PCC violations occur for untracked connections. There are many CH choices, e.g., HRW [31, 36], Ring [19, 20], Fast Robust Hashing [32], Jump [22], and recently, MaglevHash [14] and AnchorHash [23]. CH techniques are a key component in JET, and we later show how to efficiently adapt them to achieve our goal.

2 PRELIMINARIES

We focus on hash-based stateful load balancers that employ *connection tracking* to preserve *per-connection-consistency*; for each connection, the destination of its first packet is recorded and subsequently used for handling the rest of its packets.

Full CT is not always needed to maintain PCC. For example, in a static setting, where the backend servers do not change, the LB can apply a simple hash function on the connection identifier (*e.g.,* TCP's 5-tuple) to deterministically derive the destination. This maintains PCC and uniformly (at random) distributes the connections across the backend servers. However, in a dynamic setting, the hash-based destination of any connection may change as servers are being added or removed, violating PCC. We term an event in which a server is added or removed as a *backend change event*.

2.1 Connection Safety

An LB employs some decision rule (*e.g.,* a hash function on the connection identifier) to choose a destination for each new or untracked connection. For any connection, we refer to the destination assigned to its first packet by that rule as its *true destination*. To reason about the PCC implications of that rule, we examine how in a dynamic setting, that rule may change each connection's chosen destination with respect to the connection's *true destination*.

Observe that if the true destination of a connection points to a server that is removed, the connection breaks regardless of the LB state and decision rule. Otherwise, if a different server is removed, or a new server is added, the connection breaks only if it is untracked in the CT table and the LB's decision rule in the new state disagrees with the connection's true destination. This observation motivates us to make the following definitions. For a backend server change event, we divide the active connections into three categories:

- **Inevitably broken connections:** A connection is *inevitably broken* if the event is the removal of its true destination.
- **Safe connections:** A connection is *safe* if it is not *inevitably broken* and the LB's decision rule for that connection after the event agrees with the connection's true destination.
- **Unsafe connections:** A connection is *unsafe* if it is not *inevitably broken* and the LB's decision rule for that connection after the event disagrees with the connection's true destination.

We emphasize that safety is a state of the connection that depends on the LB's decision rule (that may change its decision over time depending on the backend state) and the connection's true destination. Safe connections do not require any connection tracking to maintain PCC. Inevitably broken connections break regardless of tracking and can be ignored. Only unsafe connections require tracking to maintain PCC, as otherwise, they might break after the backend change event.

We aim to reduce the number of tracked connections yet still maintain PCC by identifying and tracking only unsafe connections. As defined above, the set of unsafe connections depends on the

decision rule and on the backend change events. Tracking an unsafe connection does not have to start with its first packet; however, it must start *before* any change event that makes it unsafe.

Generally, we must consider *any* possible change event, which leads to full CT. However, as we describe below, it is often possible to anticipate backend changes that affect connection safety, thus efficiently identify the unsafe connections with respect to those changes.

2.2 Anticipating Backend Changes

We consider the two types of backend change events, namely, server removal and server addition.

Server Removal. Often, the removal of a server is planned. For example, removal may happen due to maintenance, down-scaling, and migration operations. However, unplanned removals may also occur due to unpredictable reasons such as permanent or temporary server failure or a network failure resulting in a connectivity loss. Obviously, due to the unplanned removals, we cannot always know in advance or even estimate which servers would be removed and when. Beyond the inevitably broken connection (which do not affect PCC), we must track all unsafe connections with respect to *any* server removal; that is, connections for which the decision changes due to a removal of a server that is not their destination.

Server Addition. Similarly to server removals, we must track all unsafe connections with respect to server additions. However, unlike server removals, server additions can be made predictable, allowing us to determine which connections are unsafe. We list below several ways in which this can be achieved.

- *Standby servers:* Servers may be pre-allocated and maintained in a standby mode. Additions of new servers to the backend are then made from this standby pool. The identifiers (e.g., IP addresses) of standby servers can be announced to the LBs and taken into consideration by their destination assigning mechanism. That is, any unsafe connections with respect to a standby server addition should be tracked.
- *Warm-up:* Servers may require a "warm-up" period before being added to the backend. Namely, new servers are first announced to the LBs, and only after a predefined warm-up period (*e.g.,* TCP timeout) the LBs may forward new connections to these servers. The connection tracking for unsafe connections due to each added server can start during the server's warm-up period.
- *Name allocation:* This case is similar to the "standby server" case, but instead of having standby servers we only have standby server identities. For example, if each added server obtains a new IP address from a controlled DNS service, we can configure the DNS to use a pool of addresses and announce them as standby server addresses in advance.
- *Transient failures:* Servers that are removed due to transient failures, such as reboots, temporary disconnects, migrations, or short maintenance, can be expected to have a short downtime period after which they rejoin the backend. While such a server is down, new connections that are unsafe with respect to the event of its recovery should be tracked.

2.3 The Horizon And Its Size

Leveraging the observations in Section 2.2, from this point on, we assume that the next possible server identifiers that may be immediately added to the backend are known. We call these servers the *Horizon set* and denote it by $\mathcal{H}$. Also, when clear from context, we slightly abuse notation and use *servers* instead of *server identifiers*. Note that $\mathcal{H}$ may change over time. New servers must first be added to $\mathcal{H}$ before they become eligible to join the backend. Moreover, a server must spend some minimal amout of time in $\mathcal{H}$, which we term as the *warmup period*, to make sure that the system has enough time to prepare for a possible addition of this server. Upon removal of a working server from the backend, it is added to $\mathcal{H}$. If the server is permanently removed, it is removed from $\mathcal{H}$.

2.3.1 Warmup Period. When adding a server from the horizon, to preserve PCC, the *unsafe* connections with respect to this addition must be already tracked. Thus, the warmup period should be sufficiently long such that at least one packet from each such *unsafe* connection will pass through the LB from the moment this server was added to $\mathcal{H}$ and until its possible addition to the backend. For example, a TCP timeout [13, 21] is sufficient.

2.3.2 The Horizon Size. The size of $\mathcal{H}$ is a hyperparameter in our design that also indirectly controls the warmup period. Intuitively, a small $\mathcal{H}$ results in a small connection tracking table but may limit the dynamicity of server additions, as only servers from $\mathcal{H}$ can be immediately added to the backend. A larger $\mathcal{H}$ means more flexibility, but potentially more unsafe connections and thus a larger connection tracking table. Thus, the size of $\mathcal{H}$ allows one to tradeoff memory with flexibility. We formalize this intuition in Section 4.

To conclude, a proper horizon should support the dynamic properties of the system, encapsulating knowledge about the expected change events and connection packet inter-arrival times. In the absence of such knowledge, the horizon may need to be large enough to accommodate the uncertainty. On the other hand, deriving such knowledge may lead to tiny connection tracking tables. For example, if consecutive server additions are slower than the maximal allowed inter-packet interval (e.g., TCP timeout), it may be sufficient to have a horizon of only a *single server*.

2.4 Connection Safety via Consistent Hashing

The decision rule by which an LB chooses the destination for each packet influences the number of unsafe connections. Often, a natural decision rule can result in most of the connections being unsafe, e.g., [16, 26, 35]. For example, in a system with N servers, the decision rule may assign a connection k to a server $s = hash(k) \bmod N$. However, this results in an expected fraction of $\approx 1 - 1/N$ unsafe connections for each backend change. To reduce connection tracking, we seek to minimize the number of unsafe connections and do so in a way that allows us to identify unsafe connections in a computationally light manner. We would like to obtain the following properties upon backend changes:

(1) No connections are unsafe due to server removal events.

(2) Only a small fraction of connections are unsafe due to server addition events and only for the purpose of ensuring connection balance.

Roughly speaking, property (1) means that we do not need to track *any* connections to handle server removal events (which we cannot anticipate). Property (2) implies that, on average, only a fraction of the connections must be tracked to maintain PCC upon adding servers from the horizon.

We utilize consistent hashing techniques to achieve these properties.[1] Accordingly, from this point on, we will assume that the destination choice function employed by the LBs is a consistent hash with the above properties. Moreover, as we describe next in Section 3, we can precisely identify the unsafe connections with several consistent hashing implementations in a computationally light manner.

3 JET

In this section, we introduce the JET framework and discuss its implementation details.

In a nutshell, JET leverages the fact that using a consistent hash and knowing which servers are going to be added in the near future is sufficient to efficiently identify the connections that are going to be mapped to a different destination by the consistent hash as a result of these server additions and *any* server removals. This allows JET to track only those (unsafe) connections.

JET utilizes two pluggable modules: (1) **CT** for connection tracking; (2) **CH** for computing connection destination using a *consistent hash*. $\mathbf{CT}[k]$ is the saved destination for a tracked connection k, where $k \in \mathcal{K}$ is the connection identifier (*e.g.,* TCP 5-tuple). $\mathbf{CT}[k]$ is NIL if k is untracked, evicted, or its destination is removed. $\mathbf{CH}(\mathcal{W}, k)$ is the calculated destination for a connection k over a working server set $\mathcal{W}$; that is, $\mathbf{CH}(\mathcal{W}, k) \in \mathcal{W}$.

As described in the previous section, we assume that servers can be added only from a known horizon set, $\mathcal{H}$. That is, each server addition event moves a server from $\mathcal{H}$ to $\mathcal{W}$. Likewise, each server removal event moves a server from $\mathcal{W}$ to $\mathcal{H}$.[2] Additionally, servers can be permanently removed by removing them from $\mathcal{H}$, and new servers can be introduced to the system by adding them to $\mathcal{H}$.

To maintain PCC, JET has to identify connections that are unsafe with respect to server additions from $\mathcal{H}$. At first glance, this appears to be computationally exhaustive since for each connection k, we need to check *all* possible sequences of server additions from $\mathcal{H}$ to $\mathcal{W}$ and see if there is a sequence in which k's destination changes. However, the consistent hashes we consider in this work, as we prove in Section 4, have an appealing property that it is sufficient to check only a single condition: whether $\mathbf{CH}(\mathcal{W}, k)$ differs from $\mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k)$, where $\mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k) \in \mathcal{W} \cup \mathcal{H}$ is the result of the consistent hash after adding all the servers from the horizon to the working set, in some *arbitrary order*. That is, to maintain PCC, we do not need to start tracking any connection k for which $\mathbf{CH}(\mathcal{W}, k) = \mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k)$.

Note that the decision of whether to start tracking a connection should be reexamined for each packet as $\mathcal{W}$ and $\mathcal{W} \cup \mathcal{H}$ may change during the connection's lifetime. Indeed, this may be the case for a long-lasting connection that experiences many backend changes during its lifetime. In particular, it may be the case that $\mathbf{CH}(\mathcal{W}, k) = \mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k)$ where both are not the true destination of k. But, this means that the condition for that connection failed after some older server addition event and thus it is already tracked.

Moreover, we further prove in Section 4 that the expected fraction of tracked connections is roughly $\frac{|\mathcal{H}|}{|\mathcal{W} \cup \mathcal{H}|}$. For example, if the size of $\mathcal{H}$ is no more than 10% of the size of $\mathcal{W}$, then the connection tracking space requirements of JET are expected to be 11× smaller than those of full **CT**.

Algorithm 1 JET

```
    CT                                                  ▷ Connection tracking
    CH                                                  ▷ Consistent hash
1:  function GetDestination(k)
2:      s ← CT[k]
3:      if not s then
4:          s ← CH(W, k)
5:          if s ≠ CH(W ∪ H, k) then                    ▷ Should track?
6:              CT[k] ← s
7:      return s
8:  function AddWorkingServer(s)
9:      H ← H \ {s}                                     ▷ s must be in H
10:     W ← W ∪ {s}
11: function RemoveWorkingServer(s)
12:     W ← W \ {s}
13:     H ← H ∪ {s}
14: function AddHorizonServer(s) :    H ← H ∪ {s}
15: function RemoveHorizonServer(s) :    H ← H \ {s}
```

3.1 The JET Framework

JET is given in Algorithm 1. The core function is GetDestination (Line 1) which takes as an input a unique connection identifier k and returns the destination server $s \in \mathcal{W}$ for that connection. First, we check whether the **CT** module tracks k. If so, the saved destination is returned and no further action is required. Otherwise, we turn to the **CH** module to compute the destination $s = \mathbf{CH}(\mathcal{W}, k)$.

We now check whether k is unsafe and should be tracked by the **CT** module. We do so by comparing s to $\mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k)$. k is unsafe if s differs from $\mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k)$ (Lines 5-6). If k is unsafe, it is tracked by the **CT** module (Line 6).

When adding a new server (AddWorkingServer), the server must be added from the horizon (Lines 9-10). When a working server is removed (RemoveWorkingServer),[3] we admit it immediately to the horizon (Lines 12-13).

For completeness, we introduce two additional functions to manage the horizon: AddHorizonServer, RemoveHorizonServer (Lines 14-15). These functions control the size of the horizon. If desired, a server s can be removed permanently by calling RemoveHorizonServer(s) after RemoveWorkingServer(s).

3.2 JET with HRW

Algorithm 2 provides a presudo-code of JET's GetDestination function using the HRW consistent hashing technique [31]. For a

[1] Properties (1) and (2) are implied by the *minimal disruption* and *balance* properties of consistent hashing. For example, see [23] for formal definitions.

[2] This is not mandatory but typical for transient failures and maintenance operations.

[3] Note that all connections to the removed server are inevitably broken. At this point, the connection tracking table can be cleaned from such connections (in an active or a lazy manner) to prevent broken **CT** lookups.

Algorithm 2 JET-HRW

```
1: function GetDestination(k)
2:     s ← CT[k]
3:     if not s then
4:         s ← arg max_{w∈W} hash(w, k)
5:         if hash(s, k) < max_{h∈H} hash(h, k) then
6:             CT[k] ← s
7:     return s
```

given connection with an identifier k that is not tracked, we compute its random weight $Hash(w, k)$ with each server $w \in \mathcal{W}$; the destination, $\mathbf{CH}(\mathcal{W}, k)$, is the server s with the highest weight (Line 4). To determine whether k should be tracked, we check whether s is different from $\mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k)$. However, we do not need to compute the random weight for every server in $\mathcal{W} \cup \mathcal{H}$ to do so. Rather, it is sufficient to check whether any $h \in \mathcal{H}$ has a higher random weight than s (Line 5).

Example. Figure 1 illustrates JET with HRW and $\mathcal{H}=\{h_1\}$. It exemplifies the three possible scenarios for an arriving packet: (Figure 1a) connection K_{12} is already tracked and thus the destination server $w_2 = \mathbf{CT}[K_{12}]$ is returned by the CT module (no further action is required); (Figure 1b) connection K_{32} is not tracked. We turn to the CH module to compute its destination. $w_3 = \mathbf{CH}(\mathcal{W}, K_{32})$ is returned since it has the highest random weight with K_{32}. K_{32} does not require tracking, since its random weight with the horizon h_1 is smaller than with w_3, *i.e.,* it is safe. This means that even if h_1 is added, we would still forward the packet to w_3; (Figure 1c) connection K_2 is not tracked. w_1 is returned by CH, since it has the highest random weight with K_2. K_2 requires tracking because its random weight with the horizon h_1 is larger than with w_1. This means that K_2 is unsafe due to h_1; that is, if h_1 is added, and without tracking this connection, we would forward K_2 to h_1 instead of w_3 and violate the per-connection-consistency.

Implementation notes. Usually, HRW is unfeasible for a large backend. However, for a small number of servers (e.g., 8), and a SIMD implementation (e.g., [4]) of the hash calculations, HRW can offer appealing balance properties while maintaining a high processing rate.

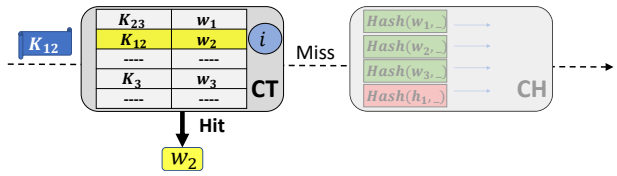
3.3 JET with Ring

Algorithm 3 provides a pseudo-code of JET's GetDestination function using Ring consistent hashing [19, 20].

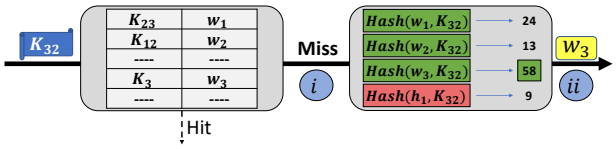
For a given connection with an identifier k that is not tracked, we first compute $\mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k)$ using the Ring technique; that is, we map k to a point on the ring using $hash(k)$,[4] then find the position p of the first entry on the ring going clockwise. We term p the *successor* of k on the ring (Line 4). We extend the ring entries to return both the destination server s and whether to track k (Line 5). We track k if needed and return s (Lines 6-8).

For given $\mathcal{W}$ and $\mathcal{H}$, the function PopulateRing populates the ring in two steps. The first step (Lines 10-11) is similar to the standard ring population; each working server w is mapped to a position on a ring ($\mathbf{Ring}_{\mathcal{W}}$) using a hash of its name. We initiate each

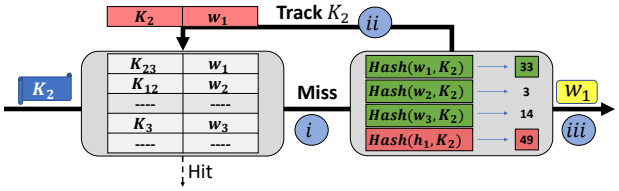
[4]Here, the hash maps onto the unit circle (i.e., $hash(x) \in [0, 1)$) and distance is measured along the circle (i.e., $(x_1 - x_2) \bmod 1 \in [0, 1)$).



(a) Connection K_{12} is already tracked; its destination is w_2.



(b) Connection K_{32} is not tracked. CH returns w_3 since it has the highest random weight with K_{32}. This connection is safe.



(c) Connection K_2 is not tracked. CH returns w_1 since it has the highest random weight with K_2. Additionally, K_2 requires tracking, since its random weight with the horizon h_1 is higher than with w_1; namely, it is unsafe due to h_1.

Figure 1: Illustrating JET with HRW. Showing the three possible scenarios for an arriving packet: (a) The connection is tracked; (b) The connection is not tracked but is safe; (c) The connection is not tracked currently but is unsafe and therefore requires tracking.

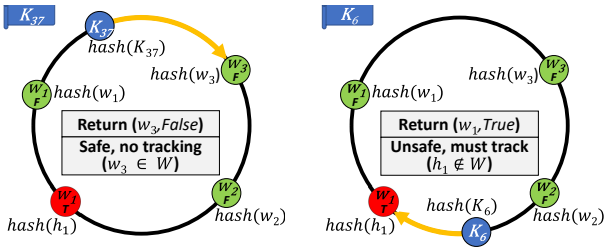
entry with (w, False), indicating no tracking is needed. In the second step (Lines 12-14), we map each horizon server h to a position on a second ring ($\mathbf{Ring}_{\mathcal{H}}$) using a hash of its name. We initiate the entry with (w, True) where w is the successor of h on the ring of the working servers ($\mathbf{Ring}_{\mathcal{W}}$). This indicates that keys mapped to h should be tracked and mapped to w rather than to h (i.e., keys for which $\mathbf{CH}(\mathcal{W}, k) = w$ and $\mathbf{CH}(\mathcal{W} \cup \mathcal{H}) = h$). Finally, we merge the two rings into a single one.

Example. Figure 2 illustrates JET with Ring and $\mathcal{H}=\{h_1\}$. It exemplifies the two possible scenarios for an arriving packet whose connection is currently not tracked: (Figure 2a) connection K_{37} is not in CT, so we turn to the CH ring module to compute its destination. We find the position of $hash(K_{37})$'s successor on the ring; namely $hash(w_3)$. The ring entry returns (w_3, False), indicating no tracking is needed since the connection is safe. (Figure 2b) connection K_6 is not in CT. We find the position of $hash(K_6)$'s successor on the ring; namely $hash(h_1)$. The ring entry returns (w_1, True), indicating the connection should be tracked since h_1 makes it unsafe. The destination is w_1 because $hash(w_1)$ is the successor of $hash(h_1)$, when considering only working servers.

Implementation notes. A standard implementation of Ring usually implements the **Ring**.*Successor* operation efficiently, using either a sorted dictionary or a search tree. Note that we add only a single

Algorithm 3 JET-RING

```
1:  function GetDestination(k)
2:      s ← CT[k]
3:      if not s then
4:          p ← Ring.Successor(hash(k))
5:          s, track ← Ring[p]
6:          if track then
7:              CT[k] ← s
8:      return s
9:  function PopulateRing(W, H)
10:     for w ∈ W do
11:         Ring_W[hash(w)] ← (w, False)
12:     for h ∈ H do
13:         p ← Ring_W.Successor(hash(h))
14:         Ring_H[hash(h)] ← (Ring[p].first, True)
15:     return Ring_W ∪ Ring_H
```



(a) K_{37} **is safe.** **(b)** K_6 **is not safe.**

Figure 2: Illustrating JET with Ring. Showing the two possible scenarios for a packet whose connection is not currently tracked: (a) CH returns $(w_3, False)$**, since** $hash(w_3)$ **is the successor of** $hash(K_{37})$ **on the ring. The connection is safe and thus no tracking is needed; (b) CH returns** $(w_1, True)$**, since** $hash(h_1)$ **is the successor of** $hash(K_6)$ **and** $hash(w_1)$ **is the successor of** $hash(h_1)$**.** K_6 **requires tracking, since it is unsafe due to** h_1**.**

Boolean flag per Ring entry to indicate if tracking is needed. As for backend changes, PopulateRing can be called after each backend addition/removal; alternatively, it can be modified to update only the successors/predecessors that are affected by the backend change.

3.4 JET with Table-based Consistent Hashing

Algorithm 4 provides a pseudo-code of JET using a table-based consistent hashing. In table-based consistent hashing, each connection is mapped to a table row, where each row is mapped to a server. The mapping between rows and servers is computed using a consistent hash. This method requires only single memory access for each lookup; it is typically faster than even CT lookup, as its table is smaller and has better caching properties. The downside of table-based lookups is some degradation in balance, as it is often the case that each backend is associated with a different number of table rows. The balance can be improved by holding several/lots copies of each backend, using a larger table with a bigger memory footprint, possibly causing an increase in cache misses.

Algorithm 4 JET- Table Based (HRW)

```
    CT                                          ▷ Connection tracking
    CH                                          ▷ Consistent hash table
    TR   ▷ Boolean table indicating CH(W, k) ≟ CH(W ∪ H, k)
1:  function GetDestination(k)
2:      s ← CT[k]
3:      if not s then
4:          r ← hash(k) mod size(CH)            ▷ Get table row
5:          s ← CH[r]
6:          if TR[r] then
7:              CT[k] ← s
8:      return s
9:  function AddWorkingServer(s)
10:     H ← H \ {s}                             ▷ s must be in H
11:     W ← W ∪ {s}
12:     for row r such that TR[r] is True do
13:         if hash(s, r) > hash(CH[r], r) then
14:             CH[r] ← s
15:             TR[r] ← hash(s, r) < max_{h∈H} hash(h, r)
16: function RemoveWorkingServer(s)
17:     W ← W \ {s}
18:     H ← H ∪ {s}
19:     for row r such that CH[r] = s do
20:         CH[r] ← arg max_{w∈W} hash(w, r)
21:         TR[r] ← True
22: function AddHorizonServer(s)
23:     H ← H ∪ {s}
24:     for row r such that TR[r] is False do
25:         TR[r] ← hash(s, r) > hash(CH[r], r)
26: function RemoveHorizonServer(s)
27:     H ← H \ {s}
28:     for row r such that TR[r] is True do
29:         TR[r] ← hash(CH[r], r) < max_{h∈H} hash(h, r)
```

In this algorithm, we use the HRW consistent hash function to precompute the row mappings. The main idea is to maintain two tables; table CH for computing $\mathbf{CH}(\mathcal{W}, k)$ and table TR for indicating whether k is safe or unsafe; namely whether $\mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k)$ differs from $\mathbf{CH}(\mathcal{W}, k)$.

The GetDestination function first checks if a connection k is in CT; if so, the tracked destination is returned. Otherwise, it uses $hash(k)$ to find the row r corresponding to k (Line 4), looks up the destination in the CH table (Line 5), and finds if k is unsafe according to the TR table (Line 6). If k is unsafe, then it is tracked in CT (Line 7).

The rest of Algorithm 4 depicts how to efficiently update the tables' content when changes are introduced to $\mathcal{W}$ or $\mathcal{H}$. We employ two key insights: (1) there is no need to update every row; (2) when updating a row, we do not need to recompute the random weight for each server.

For example, when adding a new working server (function AddWorkingServer), only rows that indicate tracking (i.e., $\mathbf{TR}[r]$ is *True*) may need to be updated. $\mathbf{CH}[r]$ needs to be updated only if the random weight of the added server is larger than the random weight of the current cached destination server. $\mathbf{TR}[r]$ may become

Algorithm 5 JET-AnchorHash

```
CT                          ▷ Connection tracking
hash_S(k)                   ▷ Hash mapping of k into a backend set S
A                           ▷ Anchor set, W ∪ H ⊂ A
W_s                         ▷ Worker set at the time of s's removal from W
1: function GetDestination(k)
2:     s ← CT[k]
3:     if not s then
4:         s ← hash_A(k)
5:         while s ∉ W do
6:             h ← s
7:             s ← hash_{W_s}(k)
8:         if h ∈ H then
9:             CT[k] ← s
10:    return s
```

False if no horizon server has a larger random weight than that of the added server.

Note that in comparison to a standard table-based consistent hash implementation, JET requires a memory overhead of only a single Boolean flag per row.

3.5 JET with AnchorHash

Algorithm 5 provides a pseudo-code of JET using AnchorHash consistent hashing. AnchorHash [23] is a consistent hash with appealing properties of small memory footprint, fast lookup and update times, and good balance. The main limitation of AnchorHash is that is requires some lightweight synchronization in a distributed setting among the LBs that must agree on the sequence of server addition/removal order.

AnchorHash naturally integrates with JET. Our GetDestination function for JET with AnchorHash is based on the GetResource function of AnchorHash.

The GetDestination function first checks if a connection k is in CT; if so, the tracked destination is returned. Otherwise, it uses $hash(k)$ to map the connection to a server s from the superset $\mathcal{A}$ of all backend servers (called the Anchor set). If s is a working backend, the loop ends. Otherwise (the crux of the function, Lines 5-7) it tries to map k onto the subset $\mathcal{W}_s \subset \mathcal{A}$, which captures $\mathcal{W}$ at the time s was removed.[5] The loop continues until a working server $s \in \mathcal{W}$ is found. Checking whether k is safe requires a single additional operation: the connection is unsafe if the penultimate backend examined by the loop is in $\mathcal{H}$ (Lines 8-9).

Implementation notes. The key to efficient implementation of JET with AnchorHash is the implementation of $hash_{\mathcal{W}_s}(k)$ (Line 7). [23] provides a linear space algorithm with proven bounds on the expected lookup time – about 1-2 hash calculations per lookup, in our evaluation scenario (see Section 5).

3.6 JET with MaglevHash?

MaglevHash [14] is an improved table-based consistent hashing that provides the same fast lookup rates but with improved balance in comparison to hash-based tables. The limitation of MaglevHash is that upon a backend change, it may remap connections that are not associated with the added/removed server. This phenomenon (termed "flips") has minimal impact on PCC when using full connection tracking. However, MaglevHash cannot be directly integrated with JET since these flips may add unsafe connections that should be identified and tracked by JET to avoid PCC violations. Specifically, for random backend removals, it is not clear how to efficiently identify the possible flips, the resulting unsafe connections and their total expected number. It is a challenging open question how to integrate JET with MaglevHash.

[5] In AnchorHash, when a server s is removed, the set $\mathcal{W}_s$ of servers that are working just after its removal is recorded (see [23] for details).

4 THEORETICAL GUARANTEES

We now provide a brief theoretical analysis of the JET framework. As before, $\mathcal{K}$, $\mathcal{W}$ and $\mathcal{H}$ denote the key, working and horizon sets, respectively. In the interest of space, the proofs of the claims in this section are deferred to Appendix A.

4.1 Balance

The first result is concerned with the connection balance achieved by JET compared to a hash-based full CT LB.

Proposition 4.1. *Assume JET and full* CT *use the same* CH *and have the same backend change events and packet arrivals. Then the two systems make identical packet dispatching decisions, except for broken connections. Specifically, they identically balance connections over the backend servers.*

4.2 Just Enough Tracking

Our next results provide the theoretical justification for the significant reduction in the size of the connection tracking tables JET offers. We begin by calculating the probability of tracking a new connection.

Theorem 4.2. *When a new connection arrives, the probability of it being tracked by JET is* $\frac{\alpha}{\alpha+1}$*, where* $\alpha = |\mathcal{H}|/|\mathcal{W}|$.

Knowing the tracking probability of a new connection allows us to provide guarantees on the expected number of tracked connections at any point in time.

Theorem 4.3. *Assume that there exists* $\gamma \in \mathbb{R}_+$ *such that at every point in time* $|\mathcal{H}| \leq \gamma|\mathcal{W}|$*. Let* $\mathcal{K}^* \subseteq \mathcal{K}$ *be the currently active connections. Then:*

(1) The expected number of tracked connections is upper-bounded by $|\mathcal{K}^*|\frac{\gamma}{1+\gamma}$.
(2) The probability that the number of tracked connections is larger than $|\mathcal{K}^*|\frac{\gamma}{1+\gamma}$ *decays exponentially.*

The implication of Theorem 4.3 is that to roughly get the same performance with JET as with full CT, one can use a CT table smaller by a factor of at least $\frac{1+\gamma}{\gamma}$.

Moreover, Theorems 4.2 and 4.3 imply that the number of tracked connections does not directly depend on the size of the backend; it only depends on the number of distinct connections and the ratio $|\mathcal{H}|/|\mathcal{W}|$. For example, given the same number of connections, a system with 50 servers and a horizon size of 5 requires, on average, the same CT table size as a system with 500 servers and a horizon size of 50 servers. Smaller $|\mathcal{H}|/|\mathcal{W}|$ ratios correspond to smaller CT tables.

4.3 Efficient Detection of Unsafe Connections

Finally, we prove our claim from Section 3 that for an horizon $\mathcal{H}$, the connections for which $\mathbf{CH}(\mathcal{W}, k) = \mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k)$ do not require tracking to maintain PCC.

Property 1. Let H be an arbitrary orderings of the horizon set $\mathcal{H}$ and let $\mathbf{CH}(\mathcal{W} \cup H, k)$ be the result of the consistent hash after adding H. Then, for any two arbitrary orderings H_1, H_2 of $\mathcal{H}$ and any key $k \in \mathcal{K}$: $\mathbf{CH}(\mathcal{W}, k) = \mathbf{CH}(\mathcal{W} \cup H_1, k)$ if and only if $\mathbf{CH}(\mathcal{W}, k) = \mathbf{CH}(\mathcal{W} \cup H_2, k)$.

THEOREM 4.4. *Assume that the* **CH** *module satisfies Property 1. Fix an arbitrary ordering H of $\mathcal{H}$. Then, connections that satisfy $\mathbf{CH}(\mathcal{W}, k) = \mathbf{CH}(\mathcal{W} \cup H, k)$ do not need tracking.*

Any **CH** that satisfies Property 1 can be efficiently plugged into JET. In Appendix A.5 we show that Property 1 holds for all consistent hashing techniques considered in Section 3.

5 EVALUATION

In this section, we evaluate JET using several scenarios, metrics and system parameters. JET's performance depends on the **CH** and **CT** modules and can be designed to adhere to different performance requirements.

For a **CT** module, two main points to consider when choosing a specific implementation are the lookup rate and the eviction policy. Our C++ simulations use a `robin_hood` map [5] to support fast lookup; however, since JET tracks only a fraction of all connections, it may be beneficial to consider implementations that provide faster negative answers. In an ideal eviction policy, inactive connections should be removed from the **CT**. However, one cannot rely on every connection to terminate properly for numerous reasons, including broken, stalled, and idled connections. The eviction policy attempts to limit the **CT** table size by heuristically evicting such connections; however, if these connections are still alive, it may cause PCC violations (especially if the table is not sufficiently large). In our evaluation, we employ the effective least-recently-used (LRU) policy in which the oldest entries in the table are removed.

The choice of **CH**, on average, does not affect the number of tracked connections. The main tradeoffs to consider for different **CH** implementations are memory footprint, lookup speed (rate), and the quality of connection balance. For example, HRW (Section 3.2) offers a good connection balance quality and requires a small memory footprint. However, for each key, it requires a hash calculation with each server. Therefore, for a large system, it has a slow lookup speed. Table-based consistent hashing (Section 3.4) requires only a single hash calculation for each key, resulting in a fast lookup speed. However, to achieve a good balance, it requires a large memory footprint. In particular, in several consistent hashing techniques, including Ring and table-based solutions, "virtual" copies for each server are introduced to achieve better balance. A typical choice for the number of virtual copies is 100-300. The number of copies increases the memory footprint and possibly slows down the lookup rate due to increased search complexity (*e.g.,* Ring), or increased cache misses. Some of these effects are evident in our evaluation.

5.1 Event Driven Simulation

To allow simulations on a larger scale, rather than simulating each packet arrival, we employ a Python-based event-driven simulation, inspired by [1, 7]. We consider four different events: (1) the beginning of a new connection; (2) the termination of an existing connection; (3) the removal of a server from the backend; (4) the addition of a server to the backend. Each simulation is governed by six parameters.

- *New connection rate*, *connection size distribution*, and *connection duration distribution*: these parameters determine the connection dynamics of the system. New connections are initiated following a Poisson arrival process parameterized by the *connection rate*. Unless stated otherwise, each simulation maintains a constant connection rate; namely, the Poisson parameter is constant. For each new connection, we draw its size (i.e., number of packets) and duration from the *size distribution* and *duration distribution*, respectively. We assume that flow packets in a time interval follow a binomial distribution, with a probability that reflects the proportion of the interval size to the remaining flow duration. For flows with a large number of packets, this approximates Poisson packet arrivals.
- *Backend update rate* and *down-time distribution*: these parameters control the dynamics of backend server changes. All simulations start with the same number of backend servers (468). Backend servers are removed following a Poisson removal process parameterized by the *update rate* (expected removals per minute). The downtime of each removed server is drawn from the *down-time distribution*.
- **CT** *table size*: this parameter determines how many connections can be tracked; once full, some tracked connections must be evicted to accommodate tracking of new connections.

All parameters and distributions are available in, and taken from [1], and represent real-world datacenter clusters [7, 24]. In particular, for the server down-time, connection size and connection duration distributions, we adopt the distributions used by [7, 24], which aim at capturing a large web service provider running over a Hadoop cluster. Each simulation is run for $1K$ seconds; with the connection duration distribution, this translates to about $5M$ connections for a connection rate of $100K$. For simplicity, we use an LRU eviction policy in the **CT**. Our focus is on the number of *broken unsafe connections* (*i.e.,* PCC violations). Recall that we can ignore inevitably broken connections, *i.e.,* those that were served by a removed server. The results shown in this subsection utilize the recently proposed AnchorHash [23, 33] as the **CH** module. We have repeated these experiments with Ring and table-based HRW and observed similar gains.

PCC violations vs. **CT** *table size.* In these experiments, we measure the number of PCC violations as a function of the **CT** table size. We repeat these experiments for different backend removal rates. For JET, we set the horizon to 10%, *i.e.,* 47 servers. The connection rate is fixed at $100K$.

As shown in Fig. 3, to achieve zero PCC violation, the **CT** table size must be larger than the number of active flows in the system. In our case, this translates to $150K$, *i.e.,* about 50% larger than the connection rate. This is expected as the connection rate represents only an average around which the number of flows fluctuates. Moreover,

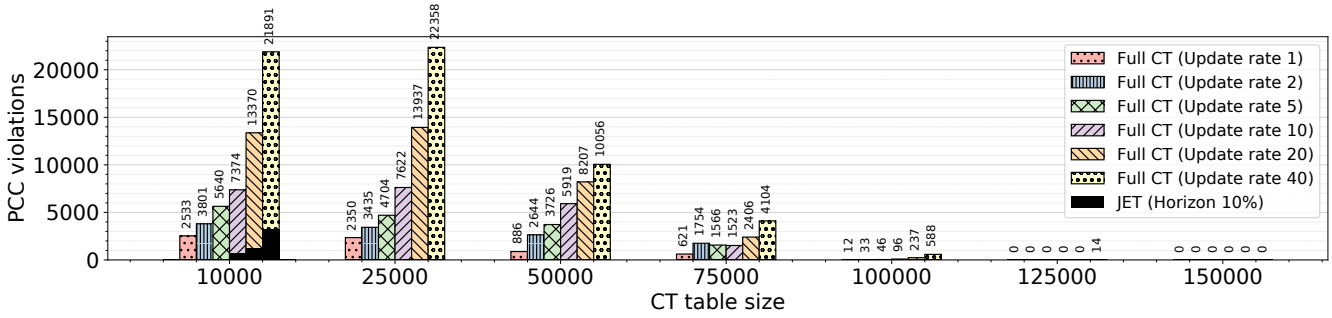
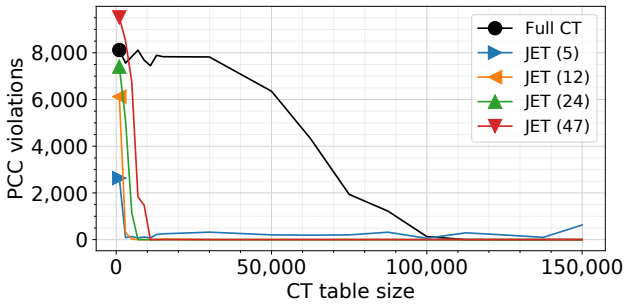
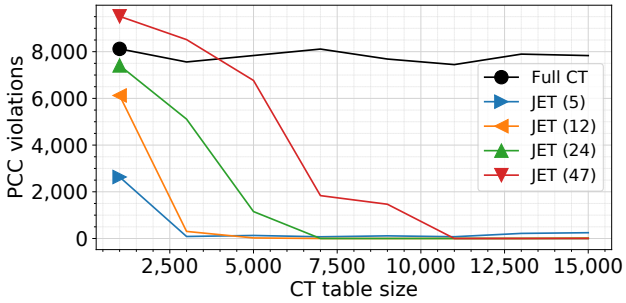


Figure 3: PCC violations vs. CT table size for different backend update rates. Each bar represents an experiment with $1K$ seconds and 468 servers. The results for JET with horizon size $\mathcal{H}$=47 (*i.e.*, 10%) are overlaid with black color.



(a) Sweeping over CT table sizes.



(b) Zooming in on small CT table sizes.

Figure 4: PCC violations vs. CT table size with different JET horizon sizes.

since we use a heuristic eviction policy (LRU), it is often the case that some memory slots are occupied by inactive flows.

When the CT table size is not large enough, some unsafe flows break. As expected, the number of PCC violations increases with the removal rate. As the CT table size grows, the number of PCC violations decreases. At small CT table sizes, the number of contending flows is so high that there is no perceivable difference in the number of PCC violations.

JET has zero PCC violations for almost all configurations (shown in black in Fig. 3). There are PCC violations only for a small CT table size of $10K$ (10% of the connection rate) and high removal rates (at least 10 removals per minute). Even in these violation instances,

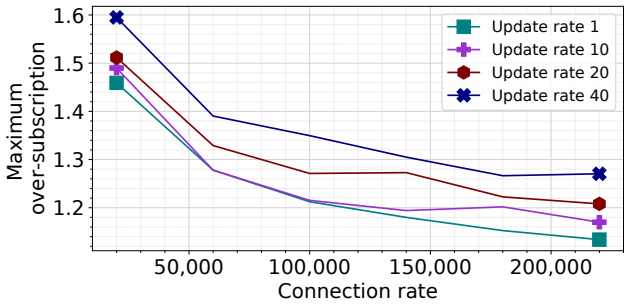


Figure 5: Oversubscription for different connection rates and server removal rates.

the number of violations by JET is much smaller (by an order of magnitude) compared to the corresponding full connection tracking experiments. These results indicate that, with limited memory, JET offers better PCC than full CT.

To better understand the behavior of JET, we repeat the above experiment for different horizon sizes, as shown in Fig. 4 for a fixed removal rate of 10. For most horizon sizes, we observe the same results in PCC violation compared to full CT. For a horizon size of 5, which is half of the removal rate, JET has some PCC violations, even when the CT table size is large (Fig. 4a). In this case, JET does not track enough connections to anticipate all the unsafe connections due to the fast rate of server additions. Namely, some added servers are placed in $\mathcal{H}$ only for a short time, during which some unsafe connections do not receive new packets and thus are not tracked. Figure 4b zooms in on the smaller CT table sizes. It is evident that the smaller the horizon, the less CT size is needed to achieve zero PCC violations. This is expected, as the number of tracked connections by JET is proportional to the ratio of the horizon size to the backend size. Furthermore, JET outperforms full CT for every horizon setting except for a horizon of 5, which is too low for the backend update rate of 10. This implies there is no need to fine-tune the horizon size. It is sufficient to make sure it is not too small.

Load balance. Our measure of load is the number of active connections that are currently mapped to each server. In particular, our

Table 1: Evaluation over trace from IMC University Data Center 1 (UNI1), 2010.

14.7M Packets 334K flows	n=50					n=500				
	Table-based HRW		AnchorHash		MaglevHash	Table-based HRW		AnchorHash		MaglevHash
	Full CT	JET	Full CT	JET	Full CT	Full CT	JET	Full CT	JET	Full CT
Maximum oversubscription	1.088 ±0.011	1.088 ±0.011	1.028 ±0.005	1.028 ±0.005	1.028 ±0.006	1.182 ±0.021	1.182 ±0.021	1.124 ±0.014	1.124 ±0.014	1.119 ±0.010
Tracked connections	334, 399 ±0	30, 300 ±541.182	334, 399 ±0	30, 442 ±121.682	334, 399 ±0	334, 399 ±0	30, 275 ±234.571	334, 399 ±0	30, 366 ±135.559	334, 399 ±0
Rate pkt/sec [millions]	54.040 ±0.498	61.811 ±2.504	45.900 ±0.245	33.097 ±2.183	54.383 ±0.828	52.770 ±0.126	60.153 ±3.288	45.849 ±0.320	34.282 ±0.971	53.599 ±1.885

Table 2: Evaluation over trace from CAIDA Equinix New-York (NY18), 2018.

34.1M Packets 1.6M flows	n=50					n=500				
	Table-based HRW		AnchorHash		MaglevHash	Table-based HRW		AnchorHash		MaglevHash
	Full CT	JET	Full CT	JET	Full CT	Full CT	JET	Full CT	JET	Full CT
Maximum oversubscription	1.085 ±0.016	1.085 ±0.016	1.014 ±0.003	1.014 ±0.003	1.016 ±0.004	1.139 ±0.017	1.139 ±0.017	1.052 ±0.004	1.052 ±0.004	1.054 ±0.005
Tracked connections	1, 602, 007 ±0	144, 940 ±2889.821	1, 602, 007 ±0	145, 541 ±296.824	1, 602, 007 ±0	1, 602, 007 ±0	145, 378 ±895.286	1, 602, 007 ±0	145, 543 ±230.205	1, 602, 007 ±0
Rate pkt/sec [millions]	24.410 ±1.998	49.493 ±2.641	22.772 ±0.072	30.998 ±0.187	25.564 ±0.446	22.883 ±2.573	45.567 ±4.113	22.702 ±0.134	30.856 ±0.187	23.446 ±2.839

metric for balance is maximum oversubscription, defined as the number of connections at the most loaded server divided by the average number of flows (*i.e.,* the total number of active flows divided by the number of active servers). A ratio of 1.0 represents a perfect balance in the number of connections.[6] As shown in Fig. 5, the maximum oversubscription is less than 1.6 and the balance improves with a higher connection rate.[7] The imbalance increases with the update rate. However, the effect of server additions is more prominent than that of server removals. In both cases, the number of active servers immediately changes. Upon server removal, its connections are broken. Thus the number of active connections drops immediately as well. In contrast, upon server addition, it takes time for the added server to shoulder some of the load.

As stated by Proposition 4.1, the maximal oversubscription is the same for JET and full CT. Specifically, in Fig.5, there is a single line per update rate (as JET and full CT use the same seeds).

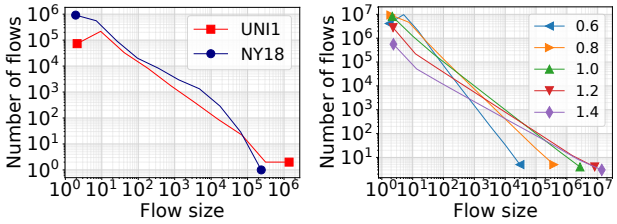
5.2 Evaluation over real traces

In this section, we evaluate the behavior of JET over real traces. We use two traces: (1) a trace from IMC University Data Center 1 (UNI1), 2010 [2, 8]; and (2) a trace from CAIDA Equinix New-York (NY18), 2018 [3]. The UNI1 trace has about $334K$ connection flows and $14.7M$ packets. The newer NY18 trace has $34.1M$ packets and $1.6M$ flows.

We run both traces with 50 and with 500 backend servers and with two hash functions: a table-based HRW (with 300 copies per server) and AnchorHash. We also compare against a full CT with MaglevHash to position the potential gains of JET in comparison to a full CT LB that utilizes a recent state-of-the-art CH technique that is used by Google's Maglev [14]. In these runs, we eliminate all PCC violations by allowing the CT table to grow as needed (*i.e.,* no flows are evicted from CT). We evaluate three metrics: *maximum oversubscription*, the number of *tracked connections*, and the $rate$. All these simulations are implemented in C++, and executed over a single core on a PC with an Intel Core i7-7700 CPU @3.60GHz (256KB L1 cache, 1MB L2 cache, and 8MB L3 cache) and 32GB DDR3 2133MHz RAM. We execute each simulation ten times and report the mean and standard deviation.

[6]Note that a ratio of 1.0 does not mean perfect load balancing, as each connection may have a different number of packets.

[7]This is in line with the theoretical imbalance analysis for $25K$ balls in 468 bins [30].



(a) Real traces. **(b) Synthetic (Zipf) traces.**

Figure 6: Histogram of flow sizes in a log-log scale: (a) UNI1 is considerably more skewed than NY18; there are less flows and the heavy hitters are larger. (b) Displaying the effect of the skew parameter of the Zipf distribution on the resulting traces.

UNI1. The results are shown in Table 1.

- *Maximum oversubscription:* Two trends are evident. First, the maximum oversubscription for $n = 50$ backend servers is better than for $n = 500$. This is expected since we randomly distribute the same number of connections among a larger set, leading to a higher deviation from the mean.
  Second, the balance for AnchorHash and MaglevHash is better than for the table-based HRW. AnchorHash and MaglevHash distribute very closely to random, so their imbalance worsens when the number of servers increases or when the number of connections decreases. Table-based HRW suffers from the same imbalance as AnchorHash and MaglevHash but has an additional imbalance factor due to the mapping between table entries and servers. A random mapping may allocate more rows to some servers; the balance improves for larger tables, *i.e.,* more copies per server. Note that for most runs, the balance of AnchorHash

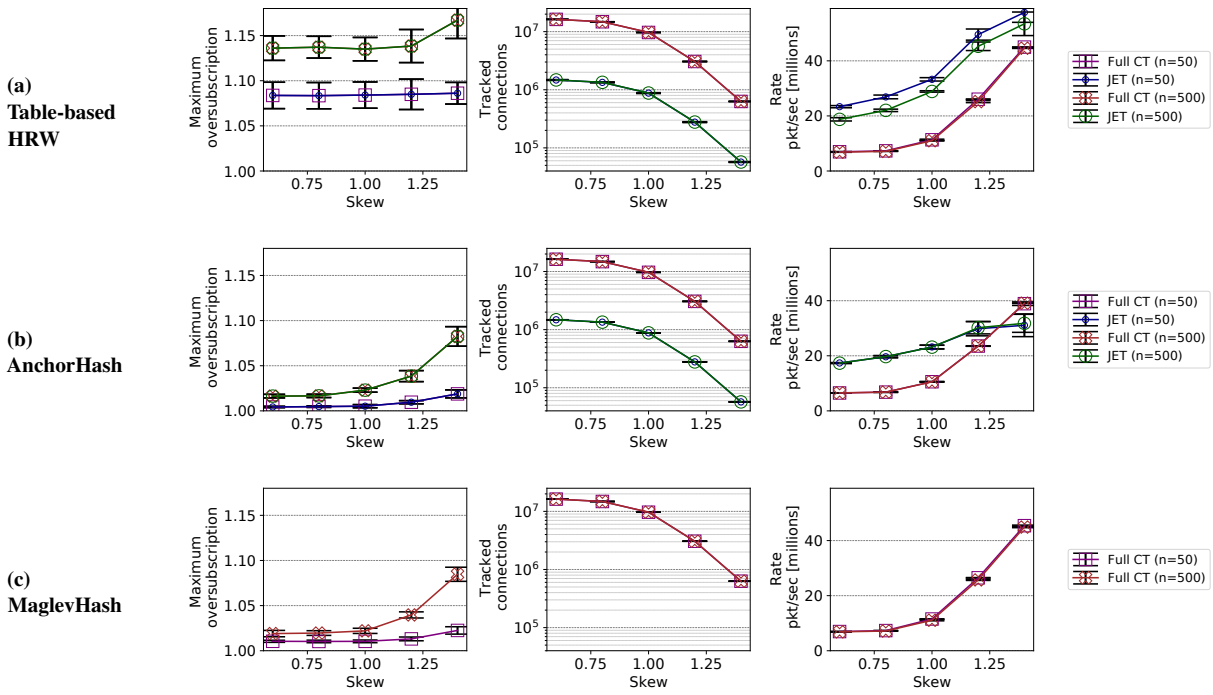


Figure 7: JET vs. Full CT — maximal oversubscription, number of tracked connections, and rate for different hash implementations.

is slightly better than that of MaglevHash. We refer the reader to [23] for an in-depth comparison of these two techniques.

- *Tracked connections:* The number of tracked connections for full CT, including the variant with MaglevHash, is the number of flows in the trace. For JET, this number is determined by the ratio between the size of the backend and the horizon. In these experiments, guided by the results of the event-driven simulations, we set the size of the horizon to be 10% of the backend, namely 5 and 50 for $n = 50$ and $n = 500$, respectively. As a result, the number of tracked connections for JET is approximately 10% of the full CT. We also find little sensitivity of this parameter with respect to the hash type.
- *Rate:* For full CT, the rate is mainly determined by the size of the CT table. Larger table sizes result in more cache misses and slower processing. In comparison, JET has a significantly smaller CT table, but it makes consistent hash computations for most packets. Consistent hash computations are typically faster than table lookups (unless the specific lookup entry resides in a low-level cache). The UNI1 trace (see Figure 6a) has heavy hitters, which results in a high rate of L1/L2 cached CT lookups. With table-based HRW, both CT lookups and consistent hash calculations require a single table lookup; JET is consistently faster since its table is smaller. On the other hand, table lookups are faster than AnchorHash's computations with such high skew. Thus JET with AnchorHash is slower than full CT. Note that the rate of MaglevHash rate is almost identical to that of table-based HRW as both require a single table lookup for each consistent hash calculation.

NY18. The results are shown in Table 2. The trends are similar to those in Table 1. Nevertheless, the are several differences. NY18 is longer (*i.e.,* 34.1M packets vs. 14.7M packers) and considerably less skewed than UNI1 (see Figure 6a); it contains more flows with fewer packets. In terms of the number of tracked connections, it maintains the 1:10 ratio between JET and full CT. Note that the absolute number has increased since NY18 is both less skewed and longer.

For full CT, the rate of MaglevHash is almost identical to that of table-based HRW and very similar to that of AnchorHash. The large CT tables for this trace result in increased cache misses of full CT, meaning CT lookups become slower than consistent hash calculations. For JET, the CT table size is considerably smaller, increasing the chance for L1-L2 cache hits for the unsafe connections. Thus, the rate of JET is now faster than full CT for both AnchorHash and table-based HRW (by ≈ 30%-100%). The best rate is achieved by JET with table-based HRW. Maximal oversubscription is similar for AnchorHash and MaglevHash and both offer a better balance than table-based HRW.

5.3 Synthetic Traces

In this section, we repeat the real traces evaluation, but using synthetic Zipf traces [9, 10] with a skew varying from 0.6 (*e.g.,* internet traffic) and up to 1.4 (highly skewed). Each trace is 100M packets long. We use the same C++ implementation and the same computational environments. As before, we run all traces with 50 and with 500 backend servers; with two hash functions (table-based HRW and AnchorHash); compare against full CT with MaglevHash; and do not bound the CT table size. We capture the same three metrics:

maximum oversubscription, number of *tracked connections*, and $rate$. We execute each simulation ten times and report the mean and standard deviation. The results are shown in Fig. 7.

The trends for *maximal oversubscription* (left column) are similar to those of the real traces. As expected, the connection balance is identical for JET and full CT, whereas JET with AnchorHash and MaglevHash offer better balance than with table-based HRW (also, the standard deviation is smaller for AnchorHash and MaglevHash). Evidently, the imbalance increases with the skew and with the backend size. Also, for a low skew, the balance by AnchorHash is slightly better than that of MaglevHash, as expected [23].

The trends for the number of *tracked connections* (middle column) are also similar to those of the real traces. The number of tracked connections has low sensitivity to both the number of backend servers and to the specific choice of consistent hash. As expected, JET tracks less connections than full CT and full CT with MaglevHash (about 10%). As the skew grows, the number of distinct flows drops, but the ratio remains roughly the same.

As before, the *rate* (right column) is higher for more skewed traces. For full CT, it is evident that the rate is not affected by the backend size and is similar for all implementations. The rate increase as the skew grows is attributed to the higher rate of CT lookup hits. With JET, the rate is almost always higher than full CT and full CT with MaglevHash, but the difference drops for higher skews; full CT and full CT with MaglevHash are even faster than JET with AnchorHash for a skew of 1.4. For JET with table-based HRW, the rate is better for 50 servers than for 500; this is due to the smaller table-based HRW table, which allows more L1-L2 cache hits on table-based HRW lookups. To summarize, in our evaluation, and for a wide range of parameters, JET consistently offers a higher processing rate than full CT and full CT with MaglevHash due to the smaller CT table sizes that result in better cache locality. We believe that such a better cache locality may help hash-based software LBs in reaching a higher LB decision processing rates given the same hardware resources.

6 DISCUSSION

6.1 Simultaneous Server Additions And Removals

For some systems, it may be the case that simultaneous backend change events take place. For example, several servers may fail concurrently or it may be of interest to add several servers at the same time to increase the cluster's capacity.

Observe that JET naturally supports any number of simultaneous server removals due the CH's appealing minimal disruption property. JET maintains PCC through simultaneous server additions, provided that all added servers are admitted from the horizon.

6.2 LB Pool Changes

In a system with multiple hash-based stateful LBs, and without a state synchronization mechanism, PCC may be violated in the case of a change in the pool of LBs [14]. Specifically, a connection k will break only if $\mathbf{CH}(\mathcal{W}, k)$ is different from k's true destination and it is redirected to an LB for which $\mathbf{CT}[k]$ does not exist. This is also the case for JET. Note that if synchronization is employed, JET's smaller CT size means that a smaller state needs to be synchronized.

6.3 Load Awareness

Hash-based load balancing distributes load by choosing a random server for each request. In some systems, it may not yield satisfactory balance properties; therefore, a load-aware technique is required [25]. For example, we may want to forward a new connection to the least loaded server or employ power-of-choice techniques [26] to make sure no server is oversubscribed above a predefined ratio.

In this work, we do not provide support for such techniques. It is of interest for future work to explore whether JET can be efficiently extended to support load-awareness. For example, a naive integration of the power-of-2-choices technique into JET is still expected to save up to 50% of CT table sizes. Namely, for a new connection, the CH result can serve as one of the two choices. The connection is registered in the CT if it is unsafe *or the choice disagrees with the hash result*. We believe that further investigation can indicate better ways for such integration.

7 CONCLUSIONS

We introduced JET, a pluggable algorithmic framework for reducing the connection tracking requirements of hash-based stateful LBs. We derived appealing theoretical guarantees and demonstrated the effectiveness of JET via a series of evaluations. JET requires an order of magnitude smaller tracking tables to preserve PCC compared to full CT, often leading to a higher processing rate due to better caching properties. Finally, we showed how to efficiently implement JET with several consistent hashes, including HRW [31], table-based HRW, Ring [19], and AnchorHash [23].

ACKNOWLEDGMENTS

We would like to thank Ben Pfaff, Soudeh Ghorbani (the paper shepherd), and the anonymous reviewers for their most valuable comments and suggestions. This work was partly supported by the Fulbright Postdoctoral Scholar Program, the Hasso Plattner Institute Research School, the Israel Science Foundation (grant No. 1119/19), the Israeli Consortium for Network Programming (Neptune), the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel Cyber Bureau.

REFERENCES

[1] Cheetah authors. Github - cheetah source code, 2020. https://github.com/cheetahlb/simulations.

[2] Data Set for IMC 2010 Data Center Measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html, 2010.

[3] The CAIDA equinix-newyork packet trace, 20181220-130000, 2018.

[4] J. Alakuijala, B. Cox, and J. Wassenberg. Fast Keyed Hash/Pseudo-Random Function Using Simd Multiply and Permute. *arXiv preprint arXiv:1612.06257*, 2016.

[5] M. Ankerl (martinus) et al. Fast & Memory Efficient Hashtable Based on Robin Hood Hashing for C++11/14/17/20. https://github.com/martinus/robin-hood-hashing, 2021.

[6] J. T. Araujo, L. Saino, L. Buytenhek, and R. Landa. Balancing on the Edge: Transport Affinity without Network State. In *Usenix NSDI*, 2018.

[7] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. Maguire Jr, P. Papadimitratos, and M. Chiesa. A High-Speed Load-Balancer Design With Guaranteed per-Connection-Consistency. In *Usenix NSDI*, pages 667–683, 2020.

[8] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[9] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. On the Implications of Zipf's Law for Web Caching. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1998.

[10] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *IEEE Infocom*, volume 1, pages 126–134, 1999.

[11] R. Cohen, M. Kadosh, A. Lo, and Q. Sayah. LB Scalability: Achieving the Right Balance Between Being Stateful and Stateless. *IEEE/ACM Transactions on Networking*, 2021.

[12] M. Duke and N. Banks. QUIC-LB: Generating Routable QUIC Connection IDs. Internet-Draft draft-ietf-quic-load-balancers-06, Internet Engineering Task Force, Feb. 2021. Work in Progress.

[13] L. Eggert and F. Gont. Tcp User Timeout Option. Technical report, RFC 5482, March, 2009.

[14] D. E. Eisenbud, C. Yi, C. Contavalli, et al. Maglev: A Fast and Reliable Software Network Load Balancer. In *Usenix NSDI*, 2016.

[15] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud Scale Load Balancing With Hardware and Software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2014.

[16] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based load balancing for fast datacenter networks. *ACM SIGCOMM Computer Communication Review*, 45(4):465–478, 2015.

[17] W. Hoeffding. Probability Inequalities for Sums of Bounded Random Variables. In *The Collected Works of Wassily Hoeffding*, pages 409–426. Springer, 1994.

[18] C. Hopps. Katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran, 2021.

[19] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, 1997.

[20] D. Karger, A. Sherman, A. Berkheimer, et al. Web Caching With Consistent Hashing. *Comp. Netw.*, 1999.

[21] A. Kesselman and Y. Mansour. Optimizing Tcp Retransmission Timeout. In *International Conference on Networking*, pages 133–140. Springer, 2005.

[22] J. Lamping and E. Veach. A Fast, Minimal Memory, Consistent Hash Algorithm. *arXiv preprint arXiv:1406.2294*, 2014.

[23] G. Mendelson, S. Vargaftik, K. Barabash, D. H. Lorenz, I. Keslassy, and A. Orda. AnchorHash: A Scalable Consistent Hash. *IEEE/ACM Transactions on Networking*, 29(2):517–528, 2021.

[24] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching Asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[25] V. Mirrokni, M. Thorup, and M. Zadimoghaddam. Consistent hashing with bounded loads. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 587–604. SIAM, 2018.

[26] M. Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[27] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu. Stateless Datacenter Load-balancing with Beamer. In *Usenix NSDI*, 2018.

[28] 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. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.

[29] B. Pit-Claudel, Y. Desmouceaux, P. Pfister, M. Townsley, and T. Clausen. Stateless Load-Aware Load Balancing in P4. In *IEEE ICNP*, pages 418–423, 2018.

[30] M. Raab and A. Steger. "Balls into bins"—A simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998.

[31] D. G. Thaler and C. V. Ravishankar. Using Name-Based Mappings to Increase Hit Rates. *IEEE/ACM Trans. Netw.*, 1998.

[32] M. Uruena, D. Larrabeiti, and P. Serrano. Fast Robust Hashing. In *IEEE Globecom*, 2006.

[33] S. Vargaftik and D. H. Lorenz. Implementation of AnchorHash - A Scalable Consistent Hash. https://github.com/anchorhash/cpp-anchorhash, 2021.

[34] S. Vargaftik and D. H. Lorenz. Implementation of Load Balancing with JET: Just Enough Tracking for Connection Consistency. https://github.com/anchorhash/jetlb, 2022.

[35] W. Wang and G. Casale. Evaluating Weighted Round Robin Load Balancing for Cloud Web Services. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 393–400. IEEE, 2014.

[36] W. Wang and C. V. Ravishankar. Hash-Based Virtual Hierarchies for Scalable Location Service in Mobile Ad-Hoc Networks. *Mobile Networks and Applications*, 2009.

A PROOFS

A.1 Proof of Proposition 4.1

PROOF. A new connection k would be directed to the same destination $\mathbf{CH}(\mathcal{W}, k)$ in both systems. If k is not new and not broken, then all of its subsequent packets follow the first. Thus, the destination for all packets of non-broken connections is the same in both systems. It follows that both systems have the same oversubscription ratio. □

A.2 Proof of Theorem 4.2

PROOF. The destination returned by GETDESTINATION(k) for a new connection is $\mathbf{CH}(\mathcal{W}, k)$ (Algorithm 1, Line 5). The connection is tracked if

$$\mathbf{CH}(\mathcal{W}, k) \neq \mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k). \tag{1}$$

Since **CH** implements a consistent hash, by its *minimum disruption* property,[8]

$$\mathbf{CH}(\mathcal{W}, k) \neq \mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k) \rightarrow \mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k) \in \mathcal{H}. \tag{2}$$

In other words, if the destination changes due to server additions, the new destination must be an added server. By **CH**'s *balance* property, $\mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k)$ divides connections with equal probability over the set $\mathcal{W} \cup \mathcal{H}$. Therefore,

$$\mathbb{P}(\mathbf{CH}(\mathcal{W} \cup \mathcal{H}, k) \in \mathcal{H}) = |\mathcal{H}|/|\mathcal{H} \cup \mathcal{W}|. \tag{3}$$

The results follows by combining Eqs. (1) to (3). □

A.3 Proof of Theorem 4.3

PROOF.

(1) By linearity of expectation, the expected number of tracked connections is upper-bounded by the sum of the probabilities that each of $k \in \mathcal{K}^*$ is tracked. By Theorem 4.2, each such probability is upper-bounded by $\frac{\gamma}{\gamma+1}$. This concludes the proof.

(2) As argued above, the number of tracked connections is stochastically dominated by the sum of $|\mathcal{K}^*|$ $Ber(\frac{\gamma}{\gamma+1})$ random variables, namely a $Bin(|\mathcal{K}^*|, \frac{\gamma}{\gamma+1})$ random variable. By Hoeffding's inequality [17], the probability that the latter is larger than its expected value of $\frac{\gamma}{\gamma+1}|\mathcal{K}^*|$ decays exponentially. In particular,

$$\mathbb{P}(X - \frac{\gamma}{\gamma+1}|\mathcal{K}^*| \geq t) \leq e^{-\frac{2t^2}{|\mathcal{K}^*|}},$$

where X is the number of tracked connections. This concludes the proof.

□

A.4 Proof of Theorem 4.4

PROOF. We show that if $\mathbf{CH}(\mathcal{W}, k) = \mathbf{CH}(\mathcal{W} \cup H, k)$, then the destination of k does not change by the addition of *any* subset of $\mathcal{H}$ that is added to $\mathcal{W}$ in *any* order. This implies that k does not require tracking. Assume, by way of contradiction, that $\mathbf{CH}(\mathcal{W}, k) = \mathbf{CH}(\mathcal{W} \cup H, k)$, yet there exists an arbitrary prefix of servers $\bar{H}^{[p]}$, from an arbitrary ordering $\bar{H}$ of $\mathcal{H}$, for which $\mathbf{CH}(\mathcal{W}, k) \neq \mathbf{CH}(\mathcal{W} \cup \bar{H}^{[p]}, k)$.

Recall that by the minimum disruption property of **CH**, if the destination changes due to a server addition, the new destination must be the added server. Therefore, if $\mathbf{CH}(\mathcal{W}, k) \neq \mathbf{CH}(\mathcal{W} \cup \bar{H}^{[p]}) \notin \mathcal{W}$ then also $\mathbf{CH}(\mathcal{W} \cup \bar{H}, k) \notin \mathcal{W}$, implying $\mathbf{CH}(\mathcal{W}, k) \neq \mathbf{CH}(\mathcal{W} \cup \bar{H}, k)$. Finally, by property 1 we must have $\mathbf{CH}(\mathcal{W}, k) \neq \mathbf{CH}(\mathcal{W} \cup H, k)$. This is a contradiction and concludes the proof. □

A.5 Application of Property 1 to different CH techniques

HRW. For any connection k, its random weight is computed independently with each server in $\mathcal{W} \cup \mathcal{H}$. In particular, it does not depend on the order by which $\mathcal{H}$ is examined.

Table-based HRW. The mapping from a connection to a table row is fixed (it is determined by a hash function on the connection identifier that is independent of both $\mathcal{W}$ and $\mathcal{H}$). The destination for each row is computed by HRW, so as explained above, does not depend on the ordering of $\mathcal{H}$.

Ring. The mapping from a server to the ring is computed by a hash that is not affected by the state of the ring. Thus the final result of mapping all servers in $\mathcal{W} \cup \mathcal{H}$ is the same, regardless of order.

AnchorHash. AnchorHash maps keys to buckets, which are always added in the same order (see [23] for more details). AnchorHash uses indirection to decouple server identities from buckets, allowing servers to be added in any order without affecting the bucket addition order. Thus AnchorHash trivially satisfies Property 1.

[8] Minimum disruption is also called *monotonicity* [19].