VOL. 5, NO. 1
JANUARY 2016

TECHNICAL JOURNAL

Editors: Rita Tavilla and Curt Kolovson

TABLE OF CONTENTS

INTRODUCTION

Welcome to Volume 5, Number 1 of the VMware Technical Journal. This issue presents six articles that were co-written by VMware authors that were published elsewhere, and appear here with permission of the authors and the associations that originally published them. We thank them all for their consideration.

In this issue, we feature three system papers, one on performance, and two theory papers.

Systems

- *The Design and Implementation of Open vSwitch*, by Pfaff et al (appeared in USENIX NSDI 2015) describes an open source virtual switch for multiple hypervisor platforms. Open vSwitch was designed specifically for networking in virtual environments, resulting in a major departure from traditional software switching design approaches.
- *Yesquel: Scalable SQL Storage for Web Applications*, by Aguilera, Leners, and Walfish (appeared in ACM SOSP 2015) describes a novel approach to designing a database system that combines the scalability of NoSQL systems with the features of a SQL relational database.
- *Taming Uncertainty In Distributed Systems with Help from the Network*, by Leners et al (appeared in ACM Eurosys 2015) describes a method of discovering and enforcing network partitions in software defined networks in order to minimize uncertainty.

Performance

- *Performance Analysis of Database Virtualization with the TPC-VMS Benchmark*, by Deehr et al (appeared in TPCTC 2014) describes the experience and results of performing a TPC-VMS benchmark consisting of the TPC-E workload running on three OLTP database systems consolidated on a single server.

Theory

- *Byzantine Agreement with Optimal Early Stopping, Optimal Resilience and Polynomial Complexity*, by Abraham and Dolev (appeared in ACM STOC 2015) describes a protocol that solves Byzantine agreement with optimal early stopping and optimal resilience using polynomial message size and computation.
- *Distributed Resource Discovery in Sub-Logarithmic Time*, by Haeupler and Malkhi (appeared in ACM PODC 2015) presents a new distributed algorithm for the resource discovery problem that features significantly improved running time while maintaining an optimal message complexity.

We hope that you enjoy this issue of the VMware Technical Journal. As always, I welcome your comments.

Sincerely,

Curt Kolovson
Co-Editor, VMware Technical Journal

SYSTEMS

– *Design and Implementation of Open vSwitch*

– *Yesquel: Scalable SQL Storage for Web Applications*

– *Taming Uncertainty in Distributed Systems with Help from the Network*

The Design and Implementation of Open vSwitch

Ben Pfaff[*], *Justin Pettit*[*], *Teemu Koponen*[*], *Ethan J. Jackson*[*],
Andy Zhou[*], *Jarno Rajahalme*[*], *Jesse Gross*[*], *Alex Wang*[*],
Jonathan Stringer[*], *Pravin Shelar*[*], *Keith Amidon*[†], *Martín Casado*[*]
[*]VMware [†]Awake Networks

Abstract

We describe the design and implementation of Open vSwitch, a multi-layer, open source virtual switch for all major hypervisor platforms. Open vSwitch was designed de novo for networking in virtual environments, resulting in major design departures from traditional software switching architectures. We detail the advanced flow classification and caching techniques that Open vSwitch uses to optimize its operations and conserve hypervisor resources. We evaluate Open vSwitch performance, drawing from our deployment experiences over the past seven years of using and improving Open vSwitch.

1 Introduction

Virtualization has changed the way we do computing over the past 15 years; for instance, many datacenters are entirely virtualized to provide quick provisioning, spillover to the cloud, and improved availability during periods of disaster recovery. While virtualization is still to reach all types of workloads, the number of virtual machines has already exceeded the number of servers and further virtualization shows no signs of stopping [1].

The rise of server virtualization has brought with it a fundamental shift in datacenter networking. A new network access layer has emerged in which most network ports are virtual, not physical [5] – and therefore, the first hop switch for workloads increasingly often resides within the hypervisor. In the early days, these hypervisor "vSwitches" were primarily concerned with providing basic network connectivity. In effect, they simply mimicked their ToR cousins by extending physical L2 networks to resident virtual machines. As virtualized workloads proliferated, limits of this approach became evident: reconfiguring and preparing a physical network for new workloads slows their provisioning, and coupling workloads with physical L2 segments severely limits their mobility and scalability to that of the underlying network.

These pressures resulted in the emergence of network virtualization [19]. In network virtualization, virtual switches become the primary provider of network services for VMs, leaving physical datacenter networks with transportation of IP tunneled packets between hypervisors. This approach allows the virtual networks to be decoupled from their underlying physical networks, and by leveraging the flexibility of general purpose processors, virtual switches can provide VMs, their tenants, and administrators with logical network abstractions, services and tools identical to dedicated physical networks.

Network virtualization demands a capable virtual switch – forwarding functionality must be wired on a per virtual port basis to match logical network abstractions configured by administrators. Implementation of these abstractions, across hypervisors, also greatly benefits from fine-grained centralized coordination. This approach starkly contrasts with early virtual switches for which a static, mostly hard-coded forwarding pipelines had been completely sufficient to provide virtual machines with L2 connectivity to physical networks.

It was this context: the increasing complexity of virtual networking, emergence of network virtualization, and limitations of existing virtual switches, that allowed Open vSwitch to quickly gain popularity. Today, on Linux, its original platform, Open vSwitch works with most hypervisors and container systems, including Xen, KVM, and Docker. Open vSwitch also works "out of the box" on the FreeBSD and NetBSD operating systems and ports to the VMware ESXi and Microsoft Hyper-V hypervisors are underway.

In this paper, we describe the design and implementation of Open vSwitch [26, 29]. The key elements of its design, revolve around the performance required by the production environments in which Open vSwitch is commonly deployed, and the programmability demanded by network virtualization. Unlike traditional network appliances, whether software or hardware, which achieve high performance through specialization, Open vSwitch, by

contrast, is designed for flexibility and general-purpose usage. It must achieve high performance without the luxury of specialization, adapting to differences in platforms supported, all while sharing resources with the hypervisor and its workloads. Therefore, this paper foremost concerns this tension – how Open vSwitch obtains high performance without sacrificing generality.

The remainder of the paper is organized as follows. Section 2 provides further background about virtualized environments while Section 3 describes the basic design of Open vSwitch. Afterward, Sections 4, 5, and 6 describe how the Open vSwitch design optimizes for the requirements of virtualized environments through flow caching, how caching has wide-reaching implications for the entire design, including its packet classifier, and how Open vSwitch manages its flow caches. Section 7 then evaluates the performance of Open vSwitch through classification and caching micro-benchmarks but also provides a view of Open vSwitch performance in a multi-tenant datacenter. Before concluding, we discuss ongoing, future and related work in Section 8.

2 Design Constraints and Rationale

The operating environment of a virtual switch is drastically different from the environment of a traditional network appliance. Below we briefly discuss constraints and challenges stemming from these differences, both to reveal the rationale behind the design choices of Open vSwitch and highlight what makes it unique.

Resource sharing. The performance goals of traditional network appliances favor designs that use dedicated hardware resources to achieve line rate performance in *worst-case* conditions. With a virtual switch on the other hand, resource conservation is critical. Whether or not the switch can keep up with worst-case line rate is secondary to maximizing resources available for the primary function of a hypervisor: running user workloads. That is, compared to physical environments, networking in virtualized environments optimizes for the *common case* over the worst-case. This is not to say worst-case situations are not important because they do arise in practice. Port scans, peer-to-peer rendezvous servers, and network monitoring all generate unusual traffic patterns but must be supported gracefully. This principle led us, *e.g.,* toward heavy use of flow caching and other forms of caching, which in common cases (with high hit rates) reduce CPU usage and increase forwarding rates.

Placement. The placement of virtual switches at the edge of the network is a source of both simplifications and complications. Arguably, topological location as a leaf, as well as sharing fate with the hypervisor and VMs remove many standard networking problems. The placement complicates scaling, however. It's not uncommon for a single virtual switch to have thousands of virtual switches as its peers in a mesh of point-to-point IP tunnels between hypervisors. Virtual switches receive forwarding state updates as VMs boot, migrate, and shut down and while virtual switches have relatively few (by networking standards) physical network ports directly attached, changes in remote hypervisors may affect local state. Especially in larger deployments of thousands (or more) of hypervisors, the forwarding state may be in constant flux. The prime example of a design influenced by this principle discussed in this paper is the Open vSwitch classification algorithm, which is designed for $O(1)$ updates.

SDN, use cases, and ecosystem. Open vSwitch has three additional unique requirements that eventually caused its design to differ from the other virtual switches:

First, Open vSwitch has been an *OpenFlow switch* since its inception. It is deliberately not tied to a single-purpose, tightly vertically integrated network control stack, but instead is re-programmable through OpenFlow [27]. This constrasts with a *feature datapath* model of other virtual switches [24, 39]: similar to forwarding ASICs, their packet processing pipelines are fixed. Only configuration of prearranged features is possible. (The Hyper-V virtual switch [24] can be extended by adding binary modules, but ordinarily each module only adds another single-purpose feature to the datapath.)

The flexibility of OpenFlow was essential in the early days of SDN but it quickly became evident that advanced use cases, such as network virtualization, result in long packet processing pipelines, and thus higher classification load than traditionally seen in virtual switches. To prevent Open vSwitch from consuming more hypervisor resources than competitive virtual switches, it was forced to implement flow caching.

Third, unlike any other major virtual switch, Open vSwitch is open source and multi-platform. In contrast to closed source virtual switches which all operate in a single environment, Open vSwitch's environment is usually selected by a user who chooses an operating system distribution and hypervisor. This has forced the Open vSwitch design to be quite modular and portable.

3 Design

3.1 Overview

In Open vSwitch, two major components direct packet forwarding. The first, and larger, component is `ovs-vswitchd`, a userspace daemon that is essentially the same from one operating system and operating environment to another. The other major component, a

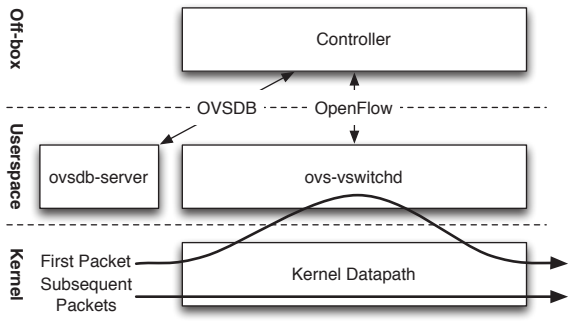


Figure 1: The components and interfaces of Open vSwitch. The first packet of a flow results in a miss, and the kernel module directs the packet to the userspace component, which caches the forwarding decision for subsequent packets into the kernel.

datapath kernel module, is usually written specially for the host operating system for performance.

Figure 1 depicts how the two main OVS components work together to forward packets. The datapath module in the kernel receives the packets first, from a physical NIC or a VM's virtual NIC. Either `ovs-vswitchd` has instructed the datapath how to handle packets of this type, or it has not. In the former case, the datapath module simply follows the instructions, called *actions*, given by `ovs-vswitchd`, which list physical ports or tunnels on which to transmit the packet. Actions may also specify packet modifications, packet sampling, or instructions to drop the packet. In the other case, where the datapath has not been told what to do with the packet, it delivers it to `ovs-vswitchd`. In userspace, `ovs-vswitchd` determines how the packet should be handled, then it passes the packet back to the datapath with the desired handling. Usually, `ovs-vswitchd` also tells the datapath to cache the actions, for handling similar future packets.

In Open vSwitch, flow caching has greatly evolved over time; the initial datapath was a *microflow cache*, essentially caching per transport connection forwarding decisions. In later versions, the datapath has two layers of caching: a microflow cache and a secondary layer, called a *megaflow cache*, which caches forwarding decisions for traffic aggregates beyond individual connections. We will return to the topic of caching in more detail in Section 4.

Open vSwitch is commonly used as an SDN switch, and the main way to control forwarding is OpenFlow [27]. Through a simple binary protocol, OpenFlow allows a controller to add, remove, update, monitor, and obtain statistics on flow tables and their flows, as well as to divert selected packets to the controller and to inject packets from the controller into the switch. In Open vSwitch, `ovs-vswitchd` receives OpenFlow flow tables from an SDN controller, matches any packets received from the datapath module against these OpenFlow tables, gathers the actions applied, and finally caches the result in the kernel datapath. This allows the datapath module to remain unaware of the particulars of the OpenFlow wire protocol, further simplifying it. From the OpenFlow controller's point of view, the caching and separation into user and kernel components are invisible implementation details: in the controller's view, each packet visits a series of OpenFlow flow tables and the switch finds the highest-priority flow whose conditions are satisfied by the packet, and executes its OpenFlow actions.

The flow programming model of Open vSwitch largely determines the use cases it can support and to this end, Open vSwitch has many extensions to standard OpenFlow to accommodate network virtualization. We will discuss these extensions shortly, but before that, we turn our focus on the performance critical aspects of this design: packet classification and the kernel-userspace interface.

3.2 Packet Classification

Algorithmic packet classification is expensive on general purpose processors, and packet classification in the context of OpenFlow is especially costly because of the generality of the form of the match, which may test any combination of Ethernet addresses, IPv4 and IPv6 addresses, TCP and UDP ports, and many other fields, including packet metadata such as the switch ingress port.

Open vSwitch uses a *tuple space search* classifier [34] for all of its packet classification, both kernel and userspace. To understand how tuple space search works, assume that all the flows in an Open vSwitch flow table matched on the same fields in the same way, *e.g.,* all flows match the source and destination Ethernet address but no other fields. A tuple search classifier implements such a flow table as a single hash table. If the controller then adds new flows with a different form of match, the classifier creates a second hash table that hashes on the fields matched in those flows. (The *tuple* of a hash table in a tuple space search classifier is, properly, the set of fields that form that hash table's key, but we often refer to the hash table itself as the tuple, as a kind of useful shorthand.) With two hash tables, a search must look in both hash tables. If there are no matches, the flow table doesn't contain a match; if there is a match in one hash table, that flow is the result; if there is a match in both, then the result is the flow with the higher priority. As the controller continues to add more flows with new forms of match, the classifier similarly expands to include a hash table for each unique match, and a search of the classifier must look in every hash table.

While the lookup complexity of tuple space search is far from the state of the art [8, 18, 38], it performs well with the flow tables we see in practice and has three attractive properties over decision tree classification algorithms. First, it supports efficient constant-time updates (an up-

date translates to a single hash table operation), which makes it suitable for use with virtualized environments where a centralized controller may add and remove flows often, sometimes multiple times per second per hypervisor, in response to changes in the whole datacenter. Second, tuple space search generalizes to an arbitrary number of packet header fields, without any algorithmic change. Finally, tuple space search uses memory linear in the number of flows.

The relative cost of a packet classification is further amplified by the large number of flow tables that sophisticated SDN controllers use. For example, flow tables installed by the VMware network virtualization controller [19] use a minimum of about 15 table lookups per packet in its packet processing pipeline. Long pipelines are driven by two factors: reducing stages through cross-producting would often significantly increase the flow table sizes and developer preference to modularize the pipeline design. Thus, even more important than the performance of a single classifier lookup, it is to reduce the number of flow table lookups a single packet requires, on average.

3.3 OpenFlow as a Programming Model

Initially, Open vSwitch focused on a reactive flow programming model in which a controller responding to traffic installs microflows which match every supported OpenFlow field. This approach is easy to support for software switches and controllers alike, and early research suggested it was sufficient [3]. However, reactive programming of microflows soon proved impractical for use outside of small deployments and Open vSwitch had to adapt to proactive flow programming to limit its performance costs.

In OpenFlow 1.0, a microflow has about 275 bits of information, so that a flow table for every microflow would have 2^{275} or more entries. Thus, proactive population of flow tables requires support for wildcard matching to cover the header space of all possible packets. With a single table this results in a "cross-product problem": to vary the treatment of packets according to n_1 values of field A and n_2 values of field B, one must install $n_1 \times n_2$ flows in the general case, even if the actions to be taken based on A and B are independent. Open vSwitch soon introduced an extension action called *resubmit* that allows packets to consult multiple flow tables (or the same table multiple times), aggregating the resulting actions. This solves the cross-product problem, since one table can contain n_1 flows that consult A and another table n_2 flows that consult B. The resubmit action also enables a form of programming based on multiway branching based on the value of one or more fields. Later, OpenFlow vendors focusing on hardware sought a way to make better use of the multiple tables consulted in series by forwarding ASICs, and OpenFlow 1.1 introduced multi-table support. Open vSwitch adopted the new model but retained its support for the resubmit action for backward compatibility and because the new model did not allow for recursion but only forward progress through a fixed table pipeline.

At this point, a controller could implement programs in Open vSwitch flow tables that could make decisions based on packet headers using arbitrary chains of logic, but they had no access to temporary storage. To solve that problem, Open vSwitch extended OpenFlow in another way, by adding meta-data fields called "registers" that flow tables could match, plus additional actions to modify and copy them around. With this, for instance, flows could decide a physical destination early in the pipeline, then run the packet through packet processing steps identical regardless of the chosen destination, until sending the packet, possibly using destination-specific instructions. As another example, VMware's NVP network virtualization controller [19] uses registers to keep track of a packet's progress through a logical L2 and L3 topology implemented as "logical datapaths" that it overlays on the physical OpenFlow pipeline.

OpenFlow is specialized for flow-based control of a switch. It cannot create or destroy OpenFlow switches, add or remove ports, configure QoS queues, associate OpenFlow controller and switches, enable or disable STP (Spanning Tree Protocol), etc. In Open vSwitch, this functionality is controlled through a separate component, the *configuration database*. To access the configuration database, an SDN controller may connect to `ovsdb-server` over the OVSDB protocol [28], as shown in Figure 1. In general, in Open vSwitch, OpenFlow controls potentially fast-changing and ephemeral data such as the flow table, whereas the configuration database contains more durable state.

4 Flow Cache Design

This section describes the design of flow caching in Open vSwitch and how it evolved to its current state.

4.1 Microflow Caching

In 2007, when the development of the code that would become Open vSwitch started on Linux, only in-kernel packet forwarding could realistically achieve good performance, so the initial implementation put all OpenFlow processing into a kernel module. The module received a packet from a NIC or VM, classified through the OpenFlow table (with standard OpenFlow matches and actions), modified it as necessary, and finally sent it to another port. This approach soon became impractical because of the relative difficulty of developing in the kernel and distribut-

ing and updating kernel modules. It also became clear that an in-kernel OpenFlow implementation would not be acceptable as a contribution to upstream Linux, which is an important requirement for mainstream acceptance for software with kernel components.

Our solution was to reimplement the kernel module as a *microflow cache* in which a single cache entry exact matches with all the packet header fields supported by OpenFlow. This allowed radical simplification, by implementing the kernel module as a simple hash table rather than as a complicated, generic packet classifier, supporting arbitrary fields and masking. In this design, cache entries are extremely fine-grained and match *at most* packets of a single transport connection: even for a single transport connection, a change in network path and hence in IP TTL field would result in a miss, and would divert a packet to userspace, which consulted the actual OpenFlow flow table to decide how to forward it. This implies that the critical performance dimension is flow setup time, the time that it takes for the kernel to report a microflow "miss" to userspace and for userspace to reply.

Over multiple Open vSwitch versions, we adopted several techniques to reduce flow setup time with the microflow cache. Batching flow setups that arrive together improved flow setup performance about 24%, for example, by reducing the average number of system calls required to set up a given microflow. Eventually, we also distributed flow setup load over multiple userspace threads to benefit from multiple CPU cores. Drawing inspiration from CuckooSwitch [42], we adopted optimistic concurrent cuckoo hashing [6] and RCU [23] techniques to implement nonblocking multiple-reader, single-writer flow tables.

After general optimizations of this kind customer feedback drew us to focus on performance in latency-sensitive applications, and that required us to reconsider our simple caching design.

4.2 Megaflow Caching

While the microflow cache works well with most traffic patterns, it suffers serious performance degradation when faced with large numbers of short lived connections. In this case, many packets miss the cache, and must not only cross the kernel-userspace boundary, but also execute a long series of expensive packet classifications. While batching and multithreading can somewhat alleviate this stress, they are not sufficient to fully support this workload.

We replaced the microflow cache with a *megaflow cache*. The megaflow cache is a single flow lookup table that supports generic matching, *i.e.,* it supports caching forwarding decisions for larger aggregates of traffic than connections. While it more closely resembles a generic OpenFlow table than the microflow cache does, due to its support for arbitrary packet field matching, it is still strictly simpler and lighter in runtime for two primary reasons. First, it does not have priorities, which speeds up packet classification: the in-kernel tuple space search implementation can terminate as soon as it finds any match, instead of continuing to look for a higher-priority match until all the mask-specific hash tables are inspected. (To avoid ambiguity, userspace installs only disjoint megaflows, those whose matches do not overlap.) Second, there is only one megaflow classifier, instead of a pipeline of them, so userspace installs megaflow entries that collapse together the behavior of all relevant OpenFlow tables.

The cost of a megaflow lookup is close to the general-purpose packet classifier, even though it lacks support for flow priorities. Searching the megaflow classifier requires searching each of its hash tables until a match is found; and as discussed in Section 3.2, each unique kind of match in a flow table yields a hash table in the classifier. Assuming that each hash table is equally likely to contain a match, matching packets require searching $(n+1)/2$ tables on average, and non-matching packets require searching all n. Therefore, for $n > 1$, which is usually the case, a classifier-based megaflow search requires more hash table lookups than a microflow cache. Megaflows by themselves thus yield a trade-off: one must bet that the per-microflow benefit of avoiding an extra trip to userspace outweighs the per-packet cost of the extra hash lookups in form of megaflow lookup.

Open vSwitch addresses the costs of megaflows by retaining the microflow cache as a first-level cache, consulted before the megaflow cache. This cache is a hash table that maps from a microflow to its matching megaflow. Thus, after the first packet in a microflow passes through the kernel megaflow table, requiring a search of the kernel classifier, this exact-match cache allows subsequent packets in the same microflow to get quickly directed to the appropriate megaflow. This reduces the cost of megaflows from per-packet to per-microflow. The exact-match cache is a true cache in that its activity is not visible to userspace, other than through its effects on performance.

A megaflow flow table represents an active subset of the cross-product of all the userspace OpenFlow flow tables. To avoid the cost of proactive crossproduct computation and to populate the megaflow cache only with entries relevant for current forwarded traffic, the Open vSwitch userspace daemon computes the cache entries incrementally and reactively. As Open vSwitch processes a packet through userspace flow tables, classifying the packet at every table, it tracks the packet field bits that were consulted as part of the classification algorithm. The generated megaflow must match any field (or part of a field) whose value was used as part of the decision. For

example, if the classifier looks at the IP destination field in any OpenFlow table as part of its pipeline, then the megaflow cache entry's condition must match on the destination IP as well. This means that incoming packets drive the cache population, and as the aggregates of the traffic evolve, new entries are populated and old entries removed.

The foregoing discussion glosses over some details. The basic algorithm, while correct, produces match conditions that are more specific than necessary, which translates to suboptimal cache hit rates. Section 5, below, describes how Open vSwitch modifies tuple space search to yield better megaflows for caching. Afterward, Section 6 addresses cache invalidation.

5 Caching-aware Packet Classification

We now turn our focus on the refinements and improvements we made to the basic tuple search algorithm (summarized in Section 3.2) to improve its suitability for flow caching.

5.1 Problem

As Open vSwitch userspace processes a packet through its OpenFlow tables, it tracks the packet field bits that were consulted as part of the forwarding decision. This bitwise tracking of packet header fields is very effective in constructing the megaflow entries with simple OpenFlow flow tables.

For example, if the OpenFlow table only looks at Ethernet addresses (as would a flow table based on L2 MAC learning), then the megaflows it generates will also look only at Ethernet addresses. For example, port scans (which do not vary Ethernet addresses) will not cause packets to go to userspace as their L3 and L4 header fields will be wildcarded resulting in near-ideal megaflow cache hit rates. On the other hand, if even one flow entry in the table matches on the TCP destination port, tuple space search will consider the TCP destination port of every packet. Then every megaflow will also match on the TCP destination port, and port scan performance again drops.

We do not know of an efficient online algorithm to generate optimal, least specific megaflows, so in development we have focused our attention on generating increasingly good approximations. Failing to match a field that must be included can cause incorrect packet forwarding, which makes such errors unacceptable, so our approximations are biased toward matching on more fields than necessary. The following sections describe improvements of this type that we have integrated into Open vSwitch.

```
function PRIORITYSORTEDTUPLESEARCH(H)
    B ← NULL      /* Best flow match so far. */
    for tuple T in descending order of T.pri_max do
        if B ≠ NULL and B.pri ≥ T.pri_max then
            return B
        if T contains a flow F matching H then
            if B = NULL or F.pri > B.pri then
                B ← F
    return B
```

Figure 2: Tuple space search for target packet headers H, with priority sorting.

5.2 Tuple Priority Sorting

Lookup in a tuple space search classifier ordinarily requires searching every tuple. Even if a search of an early tuple finds a match, the search must still look in the other tuples because one of them might contain a matching flow with a higher priority.

We improved on this by tracking, in each tuple T, the maximum priority $T.pri\_max$ of any flow entry in T. We modified the lookup code to search tuples from greatest to least maximum priority, so that a search that finds a matching flow F with priority $F.pri$ can terminate as soon as it arrives at a tuple whose maximum priority is $F.pri$ or less, since at that point no better match can be found. Figure 2 shows the algorithm in detail.

As an example, we examined the OpenFlow table installed by a production deployment of VMware's NVP controller [19]. This table contained 29 tuples. Of those 29 tuples, 26 contained flows of a single priority, which makes intuitive sense because flows matching a single tuple tend to share a purpose and therefore a priority. When searching in descending priority order, one can always terminate immediately following a successful match in such a tuple. Considering the other tuples, two contained flows with two unique priorities that were higher than those in any subsequent tuple, so any match in either of these tuples terminated the search. The final tuple contained flows with five unique priorities ranging from 32767 to 36866; in the worst case, if the lowest priority flows matched in this tuple, then the remaining tuples with $T.pri\_max > 32767$ (up to 20 tuples based on this tuple's location in the sorted list), must also be searched.

5.3 Staged Lookup

Tuple space search searches each tuple with a hash table lookup. In our algorithm to construct the megaflow matching condition, this hash table lookup means that the megaflow must match all the bits of fields included in the tuple, even if the tuple search fails, because every one of those fields and their bits may have affected the

lookup result so far. When the tuple matches on a field that varies often from flow to flow, *e.g.,* the TCP source port, the generated megaflow is not much more useful than installing a microflow would be because it will only match a single TCP stream.

This points to an opportunity for improvement. If one could search a tuple on a subset of its fields, and determine with this search that the tuple could not possibly match, then the generated megaflow would only need to match on the subset of fields, rather than all the fields in the tuple.

The tuple implementation as a hash table over all its fields made such an optimization difficult. One cannot search a hash table on a subset of its key. We considered other data structures. A trie would allow a search on any prefix of fields, but it would also increase the number of memory accesses required by a successful search from $O(1)$ to $O(n)$ in the length of the tuple fields. Individual per-field hash tables had the same drawback. We did not consider data structures larger than $O(n)$ in the number of flows in a tuple, because OpenFlow tables can have hundreds of thousands of flows.

The solution we implemented statically divides fields into four groups, in decreasing order of traffic granularity: metadata (*e.g.,* the switch ingress port), L2, L3, and L4. We changed each tuple from a single hash table to an array of four hash tables, called *stages*: one over metadata fields only, one over metadata and L2 fields, one over metadata, L2, and L3 fields, and one over all fields. (The latter is the same as the single hash table in the previous implementation.) A lookup in a tuple searches each of its stages in order. If any search turns up no match, then the overall search of the tuple also fails, and only the fields included in the stage last searched must be added to the megaflow match.

This optimization technique would apply to any subsets of the supported fields, not just the layer-based subsets we used. We divided fields by protocol layer because, as a rule of thumb, in TCP/IP, inner layer headers tend to be more diverse than outer layer headers. At L4, for example, the TCP source and destination ports change on a per-connection basis, but in the metadata layer only a relatively small and static number of ingress ports exist.

Each stage in a tuple includes all of the fields in earlier stages. We chose this arrangement, although the technique does not require it, because then hashes could be computed incrementally from one stage to the next, and profiling had shown hash computation to be a significant cost (with or without staging).

With four stages, one might expect the time to search a tuple to quadruple. Our measurements show that, in fact, classification speed actually improves slightly in practice because, when a search terminates at any early stage, the classifier does not have to compute the full hash of all the fields covered by the tuple.

This optimization fixes a performance problem observed in production deployments. The NVP controller uses Open vSwitch to implement multiple isolated logical datapaths (further interconnected to form logical networks). Each logical datapath is independently configured. Suppose that some logical datapaths are configured with ACLs that allow or deny traffic based on L4 (*e.g.,* TCP or UDP) port numbers. Megaflows for traffic on these logical datapaths must match on the L4 port to enforce the ACLs. Megaflows for traffic on other logical datapaths need not and, for performance, should not match on L4 port. Before this optimization, however, all generated megaflows matched on L4 port because a classifier search had to pass through a tuple that matched on L4 port. The optimization allows megaflows for traffic on logical datapaths without L4 ACLs to avoid matching on L4 port, because the first three (or fewer) stages are enough to determine that there is no match.

5.4 Prefix Tracking

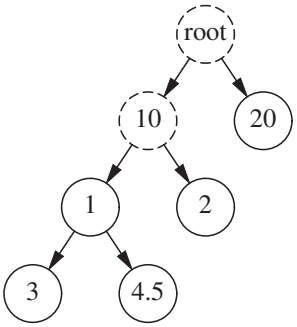
Flows in OpenFlow often match IPv4 and IPv6 subnets to implement routing. When all the flows that match on such a field use the same subnet size, *e.g.,* all match /16 subnets, this works out fine for constructing megaflows. If, on the other hand, different flows match different subnet sizes, like any standard IP routing table does, the constructed megaflows match the longest subnet prefix, *e.g.,* any host route (/32) forces all the megaflows to match full addresses. Suppose, for example, Open vSwitch is constructing a megaflow for a packet addressed to 10.5.6.7. If flows match subnet 10/8 and host 10.1.2.3/32, one could safely install a megaflow for 10.5/16 (because 10.5/16 is completely inside 10/8 and does not include 10.1.2.3), but without additional optimization Open vSwitch installs 10.5.6.7/32. (Our examples use only octet prefixes, *e.g.,* /8, /16, /24, /32, for clarity, but the implementation and the pseudocode shown later work in terms of bit prefixes.)

We implemented optimization of prefixes for IPv4 and IPv6 fields using a trie structure. If a flow table matches over an IP address, the classifier executes an LPM lookup for any such field *before* the tuple space search, both to determine the maximum megaflow prefix length required, as well as to determine which tuples can be skipped entirely without affecting correctness.[1] As an example, suppose an OpenFlow table contained flows that matched on some IPv4 field, as shown:

[1] This is a slight simplification for improved clarity; the actual implementation reverts to prefix tracking if staged lookups have concluded to include an IP field to the match.

20 /8
10.1 /16
10.2 /16
10.1.3 /24
10.1.4.5/32

These flows correspond to the following trie, in which a solid circle represents one of the address matches listed above and a dashed circle indicates a node that is present only for its children:



To determine the bits to match, Open vSwitch traverses the trie from the root down through nodes with labels matching the corresponding bits in the packet's IP address. If traversal reaches a leaf node, then the megaflow need not match the remainder of the address bits, *e.g.,* in our example 10.1.3.5 would be installed as 10.1.3/24 and 20.0.5.1 as 20/8. If, on the other hand, traversal stops due to the bits in the address not matching any of the corresponding labels in the tree, the megaflow must be constructed to match up to and including the bits that could not be found, *e.g.,* 10.3.5.1 must be installed as 10.3/16 and 30.10.5.2 as 30/8.

The trie search result also allows Open vSwitch to skip searching some tuples. Consider the address 10.1.6.1. A search of the above trie for this address terminates at the node labeled 1, failing to find a node to follow for the address's third octet. This means that no flow in the flow table with an IP address match longer than 16 bits matches the packet, so the classifier lookup can skip searching tuples for the flows listed above with /24 and /32 prefixes.

Figure 3 gives detailed pseudocode for the prefix matching algorithm. Each node is assumed to have members *bits*, the bits in the particular node (at least one bit, except that the root node may be empty); *left* and *right*, the node's children (or NULL); and *n_rules*, the number of rules in the node (zero if the node is present only for its children, otherwise nonzero). It returns the number of bits that must be matched, allowing megaflows to be improved, and a bit-array in which 0-bits designate matching lengths for tuples that Open vSwitch may skip searching, as described above.

While this algorithm optimizes longest-prefix match lookups, it improves megaflows even when no flow explicitly matches against an IP prefix. To implement a longest prefix match in OpenFlow, the flows with longer prefix must have higher priorities, which will allow the tuple priority sorting optimization in Section 5.2 to skip prefix matching tables after the longest match is found, but this alone causes megaflows to unwildcard address bits according to the longest prefix in the table. The main practical benefit of this algorithm, then, is to prevent policies (such as a high priority ACL) that are applied to a specific host from forcing all megaflows to match on a full IP address. This algorithm allows the megaflow entries only to match with the high order bits sufficient to differentiate the traffic from the host with ACLs.

```
function TRIESEARCH(value, root)
    node ← root, prev ← NULL
    plens ← bit-array of len(value) 0-bits
    i ← 0
    while node ≠ NULL do
        c ← 0
        while c < len(node.bits) do
            if value[i] ≠ node.bits[c] then
                return (i+1, plens)
            c ← c+1, i ← i+1
        if node.n_rules > 0 then
            plens[i−1] ← 1
        if i ≥ len(value) then
            return (i, plens)
        prev ← node
        if value[i] = 0 then
            node ← node.left
        else
            node ← node.right
    if prev ≠ NULL and prev has at least one child then
        i ← i+1
    return (i, plens)
```

Figure 3: Prefix tracking pseudocode. The function searches for *value* (*e.g.,* an IP address) in the trie rooted at node *root*. It returns the number of bits at the beginning of *value* that must be examined to render its matching node unique, and a bit-array of possible matching lengths. In the pseudocode, $x[i]$ is bit i in x and $\text{len}(x)$ the number of bits in x.

We also eventually adopted prefix tracking for L4 transport port numbers. Similar to IP ACLs, this prevents high-priority ACLs that match specific transport ports (*e.g.,* to block SMTP) from forcing all megaflows to match the entire transport port fields, which would again reduce the megaflow cache to a microflow cache [32].

5.5 Classifier Partitioning

The number of tuple space searches can be further reduced by skipping tuples that cannot possibly match. OpenFlow

supports setting and matching metadata fields during a packet's trip through the classifier. Open vSwitch partitions the classifier based on a particular metadata field. If the current value in that field does not match any value in a particular tuple, the tuple is skipped altogether.

While Open vSwitch does not have a fixed pipeline like traditional switches, NVP often configures each lookup in the classifier as a stage in a pipeline. These stages match on a fixed number of fields, similar to a tuple. By storing a numeric indicator of the pipeline stage into a specialized metadata field, NVP provides a hint to the classifier to efficiently only look at pertinent tuples.

6 Cache Invalidation

The flip side of caching is the complexity of managing the cache. In Open vSwitch, the cache may require updating for a number of reasons. Most obviously, the controller can change the OpenFlow flow table. OpenFlow also specifies changes that the switch should take on its own in reaction to various events, *e.g.,* OpenFlow "group" behavior can depend on whether carrier is detected on a network interface. Reconfiguration that turns features on or off, adds or removes ports, etc., can affect packet handling. Protocols for connectivity detection, such as CFM [10] or BFD [14], or for loop detection and avoidance, *e.g.,* (Rapid) Spanning Tree Protocol, can influence behavior. Finally, some OpenFlow actions and Open vSwitch extensions change behavior based on network state, *e.g.,* based on MAC learning.

Ideally, Open vSwitch could precisely identify the megaflows that need to change in response to some event. For some kinds of events, this is straightforward. For example, when the Open vSwitch implementation of MAC learning detects that a MAC address has moved from one port to another, the datapath flows that used that MAC are the ones that need an update. But the generality of the OpenFlow model makes precise identification difficult in other cases. One example is adding a new flow to an OpenFlow table. Any megaflow that matched a flow in that OpenFlow table whose priority is less than the new flow's priority should potentially now exhibit different behavior, but we do not know how to efficiently (in time and space) identify precisely those flows.[2] The problem is worsened further by long sequences of OpenFlow flow table lookups. We concluded that precision is not practical in the general case.

Therefore, early versions of Open vSwitch divided changes that could require the behavior of datapath flows to change into two groups. For the first group, the changes whose effects were too broad to precisely identify the needed changes, Open vSwitch had to examine every datapath flow for possible changes. Each flow had to be passed through the OpenFlow flow table in the same way as it was originally constructed, then the generated actions compared against the ones currently installed in the datapath. This can be time-consuming if there are many datapath flows, but we have not observed this to be a problem in practice, perhaps because there are only large numbers of datapath flows when the system actually has a high network load, making it reasonable to use more CPU on networking. The real problem was that, because Open vSwitch was single-threaded, the time spent re-examining all of the datapath flows blocked setting up new flows for arriving packets that did not match any existing datapath flow. This added high latency to flow setup for those packets, greatly increased the overall variability of flow setup latency, and limited the overall flow setup rate. Through version 2.0, therefore, Open vSwitch limited the maximum number of cached flows installed in the datapath to about 1,000, increased to 2,500 following some optimizations, to minimize these problems.

The second group consisted of changes whose effects on datapath flows could be narrowed down, such as MAC learning table changes. Early versions of Open vSwitch implemented these in an optimized way using a technique called *tags*. Each property that, if changed, could require megaflow updates was given one of these tags. Also, each megaflow was associated with the tags for all of the properties on which its actions depended, *e.g.,* if the actions output the packet to port x because the packet's destination MAC was learned to be on that port, then the megaflow is associated with the tag for that learned fact. Later, if that MAC learned port changed, Open vSwitch added the tag to a set of tags that accumulated changes. In batches, Open vSwitch scanned the megaflow table for megaflows that had at least one of the changed tags, and checked whether their actions needed an update.

Over time, as controllers grew more sophisticated and flow tables more complicated, and as Open vSwitch added more actions whose behavior changed based on network state, each datapath flow became marked with more and more tags. We had implemented tags as Bloom filters [2], which meant that each additional tag caused more "false positives" for revalidation, so now most or all flows required examination whenever any state changed. By Open vSwitch version 2.0, the effectiveness of tags had declined so much that to simplify the code Open vSwitch abandoned them altogether in favor of always revalidating the entire datapath flow table.

Since tags had been one of the ways we sought to minimize flow setup latency, we now looked for other ways. In Open vSwitch 2.0, toward that purpose, we divided userspace into multiple threads. We broke flow setup into separate threads so that it did not have to wait behind

[2] Header space analysis [16] provides the algebra to identify the flows but the feasibility of efficient, online analysis (such as in [15]) in this context remains an open question.

revalidation. Datapath flow eviction, however, remained part of the single main thread and could not keep up with multiple threads setting up flows. Under heavy flow setup load, though, the rate at which eviction can occur is critical, because userspace must be able to delete flows from the datapath as quickly as it can install new flows, or the datapath cache will quickly fill up. Therefore, in Open vSwitch 2.1 we introduced multiple dedicated threads for cache revalidation, which allowed us to scale up the revalidation performance to match the flow setup performance and to greatly increase the kernel cache maximum size, to about 200,000 entries. The actual maximum is dynamically adjusted to ensure that total revalidation time stays under 1 second, to bound the amount of time that a stale entry can stay in the cache.

Open vSwitch userspace obtains datapath cache statistics by periodically (about once per second) polling the kernel module for every flow's packet and byte counters. The core use of datapath flow statistics is to determine which datapath flows are useful and should remain installed in the kernel and which ones are not processing a significant number of packets and should be evicted. Short of the table's maximum size, flows remain in the datapath until they have been idle for a configurable amount of time, which now defaults to 10 s. (Above the maximum size, Open vSwitch drops this idle time to force the table to shrink.) The threads that periodically poll the kernel for per flow statistics also use those statistics to implement OpenFlow's per-flow packet and byte count statistics and flow idle timeout features. This means that OpenFlow statistics are themselves only periodically updated.

The above describes how userspace invalidates the datapath's megaflow cache. Maintenance of the first-level microflow cache (discussed in Section 4) is much simpler. A microflow cache entry is only a hint to the first hash table to search in the general tuple space search. Therefore, a stale microflow cache entry is detected and corrected the first time a packet matches it. The microflow cache has a fixed maximum size, with new microflows replacing old ones, so there is no need to periodically flush old entries. We use a pseudo-random replacement policy, for simplicity, and have found it to be effective in practice.

7 Evaluation

The following sections examine Open vSwitch performance in production and in microbenchmarks.

7.1 Performance in Production

We examined 24 hours of Open vSwitch performance data from the hypervisors in a large, commercial multi-tenant data center operated by Rackspace. Our data set contains statistics polled every 10 minutes from over 1,000 hypervisors running Open vSwitch to serve mixed tenant workloads in network virtualization setting.

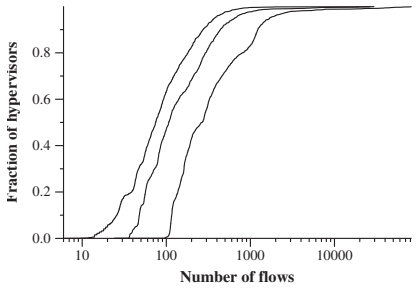


Figure 4: Min/mean/max megaflow flow counts observed.

Cache sizes. The number of active megaflows gives us an indication about practical megaflow cache sizes Open vSwitch handles. In Figure 4, we show the CDF for minimum, mean and maximum counts during the observation period. The plots show that small megaflow caches are sufficient in practice: 50% of the hypervisors had mean flow counts of 107 or less. The 99th percentile of the maximum flows was still just 7,033 flows. For the hypervisors in this environment, Open vSwitch userspace can maintain a sufficiently large kernel cache. (With the latest Open vSwitch mainstream version, the kernel flow limit is set to 200,000 entries.)

Cache hit rates. Figure 5 shows the effectiveness of caching. The solid line plots the overall cache hit rate across each of the 10-minute measurement intervals across the entire population of hypervisors. The overall cache hit rate was 97.7%. The dotted line includes just the 25% of the measurement periods in which the fewest packets were forwarded, in which the caching was less effective than overall, achieving a 74.7% hit rate. Intuitively, caching is less effective (and unimportant) when there is little to cache. Open vSwitch caching is most effective when it is most useful: when there is a great deal of traffic to cache. The dashed line, which includes just the 25% of the measurement periods in which the most packets were forwarded, demonstrates this: during these periods, the hit rate rises slightly above the overall average to 98.0%.

The vast majority of the hypervisors in this data center do not experience high volume traffic from their workloads. Figure 6 depicts this: 99% of the hypervisors see fewer than 79,000 packets/s to hit their caches (and fewer than 1500 flow setups/s to enter userspace due to misses).

CPU usage. Our statistics gathering process cannot separate Open vSwitch kernel load from the rest of the kernel load, so we focus on Open vSwitch userspace. As we

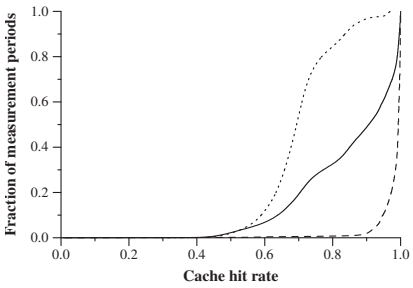


Figure 5: Hit rates during all (solid), busiest (dashed), and slowest (dotted) periods.

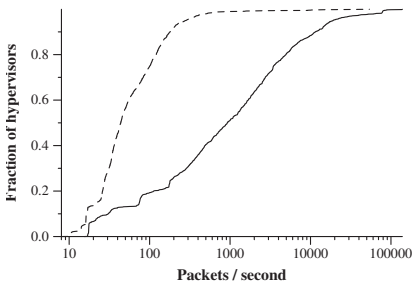


Figure 6: Cache hit (solid) and miss (dashed) packet counts.

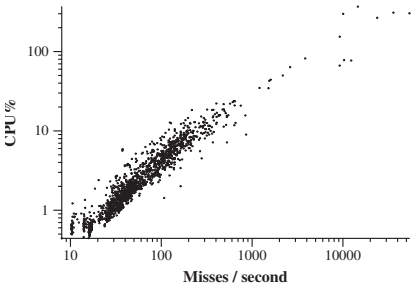


Figure 7: Userspace daemon CPU load as a function of misses/s entering userspace.

will show in Section 7.2, the megaflow CPU usage itself is in line with Linux bridging and less of a concern. In Open vSwitch, the userspace load is largely due to the misses in kernel and Figure 7 depicts this. (Userspace CPU load can exceed 100% due to multithreading.) We observe that 80% of the hypervisors averaged 5% CPU or less on `ovs-vswitchd`, which has been our traditional goal. Over 50% of hypervisors used 2% CPU or less.

Outliers. The upper right corner of Figure 7 depicts a number of hypervisors using large amounts of CPU to process many misses in userspace. We individually examined the six most extreme cases, where Open vSwitch averaged over 100% CPU over the 24 hour period. We found that all of these hypervisors exhibited a previously unknown bug in the implementation of prefix tracking, such that flows that match on an ICMP type or code caused all TCP flows to match on the entire TCP source or destination port, respectively. We believe we have fixed this bug in Open vSwitch 2.3, but the data center was not upgraded in time to verify in production.

7.2 Caching Microbenchmarks

We ran microbenchmarks with a simple flow table designed to compactly demonstrate the benefits of the caching-aware packet classification algorithm. We used the following OpenFlow flows, from highest to lowest priority. We omit the actions because they are not significant for the discussion:

```
arp                                                    (1)
ip   ip_dst=11.1.1.1/16                                (2)
tcp  ip_dst=9.1.1.1      tcp_src=10 tcp_dst=10         (3)
ip   ip_dst=9.1.1.1/24                                 (4)
```

With this table, with no caching-aware packet classification, any TCP packet will always generate a megaflow that matches on TCP source and destination ports, because flow #3 matches on those fields. With priority sorting (Section 5.2), packets that match flow #2 can omit matching on TCP ports, because flow #3 is never considered. With staged lookup (Section 5.3), IP packets not destined to 9.1.1.1 never need to match on TCP ports, because flow #3 is identified as non-matching after considering only the IP destination address. Finally, address prefix tracking (Section 5.4) allows megaflows to ignore some of the bits in IP destination addresses even though flow #3 matches on the entire address.

Optimizations	ktps	Flows	Masks	CPU%
Megaflows disabled	37	1,051,884	1	45/ 40
No optimizations	56	905,758	3	37/ 40
Priority sorting only	57	794,124	4	39/ 45
Prefix tracking only	95	13	10	0/ 15
Staged lookup only	115	14	13	0/ 15
All optimizations	117	15	14	0/ 20

Table 1: Performance testing results for classifier optimizations. Each row reports the measured number of Netperf `TCP_CRR` transactions per second, in thousands, along with the number of kernel flows, kernel masks, and user and kernel CPU usage.

Microflows	Optimizations	ktps	Tuples/pkt	CPU%
Enabled	Enabled	120	1.68	0/ 20
Disabled	Enabled	92	3.21	0/ 18
Enabled	Disabled	56	1.29	38/ 40
Disabled	Disabled	56	2.45	40/ 42

Table 2: Effects of microflow cache. Each row reports the measured number of Netperf `TCP_CRR` transactions per second, in thousands, along with the average number of tuples searched by each packet and user and kernel CPU usage.

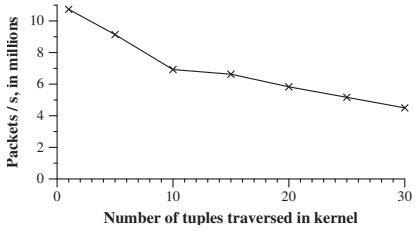


Figure 8: Forwarding rate in terms of the average number of megaflow tuples searched, with the microflow cache disabled.

Cache layer performance. We measured first the baseline performance of each Open vSwitch cache layer. In all following tests, Open vSwitch ran on a Linux server with two 8-core, 2.0 GHz Xeon processors and two Intel 10-Gb NICs. To generate many connections, we used Netperf's `TCP_CRR` test [25], which repeatedly establishes a TCP connection, sends and receives one byte of traffic, and disconnects. The results are reported in transactions per second (tps). Netperf only makes one connection attempt at a time, so we ran 400 Netperf sessions in parallel and reported the sum.

To measure the performance of packet processing in Open vSwitch userspace, we configured `ovs-vswitchd` to disable megaflow caching, by setting up only microflow entries in the datapath. As shown in Table 1, this yielded 37 ktps in the `TCP_CRR` test, with over one million kernel flow entries, and used about 1 core of CPU time.

To quantify the throughput of the megaflow cache by itself, we re-enabled megaflow caching, then disabled the kernel's microflow cache. Table 2 shows that disabling the microflow cache reduces `TCP_CRR` performance from 120 to 92 ktps when classifier optimizations are enabled. (When classifier optimizations are disabled, disabling the microflow cache has little effect because it is overshadowed by the increased number of trips to userspace.)

Figure 8 plots packet forwarding performance for long-lived flows as a function of the average number of tuples searched, with the kernel microflow cache disabled. In the same scenarios, with the microflow cache enabled, we measured packet forwarding performance of long-lived flows to be approximately 10.6 Mpps, independent of the number of tuples in the kernel classifier. Even searching only 5 tuples on average, the microflow cache improves performance by 1.5 Mpps, clearly demonstrating its value. To put these numbers in perspective in terms of raw hash lookup performance, we benchmarked our tuple space classifier in isolation: with a randomly generated table of half a million flow entries, the implementation is able to do roughly 6.8M hash lookups/s, on a single core – which translates to 680,000 classifications per second with 10 tuples.

Classifier optimization benefit. We measured the benefit of our classifier optimizations. Table 1 shows the improvement from individual optimizations and all of the optimizations together. Each optimization reduces the number of kernel flows needed to run the test. Each kernel flow corresponds to one trip between the kernel and userspace, so each reduction in flows also reduces userspace CPU time used. As can be seen from the table, as the number of kernel flows (Flows) declines, the number of tuples in the kernel flow table (Masks) increases, increasing the cost of kernel classification, but the measured reduction in kernel CPU time and increase in `TCP_CRR` shows that this is more than offset by the microflow cache and by fewer trips to userspace. The `TCP_CRR` test is highly sensitive to latency, demonstrating that latency decreases as well.

Comparison to in-kernel switch. We compared Open vSwitch to the Linux bridge, an Ethernet switch implemented entirely inside the Linux kernel. In the simplest configuration, the two switches achieved identical throughput (18.8 Gbps) and similar `TCP_CRR` connection rates (696 ktps for Open vSwitch, 688 for the Linux bridge), although Open vSwitch used more CPU (161% vs. 48%). However, when we added one flow to Open vSwitch to drop STP BPDU packets and a similar `iptables` rule to the Linux bridge, Open vSwitch performance and CPU usage remained constant whereas the Linux bridge connection rate dropped to 512 ktps and its CPU usage increased over 26-fold to 1,279%. This is because the built-in kernel functions have per-packet overhead, whereas Open vSwitch's overhead is generally fixed per-megaflow. We expect enabling other features, such as routing and a firewall, would similarly add CPU load.

8 Ongoing, Future, and Related Work

We now briefly discuss our current and planned efforts to improve Open vSwitch, and briefly cover related work.

8.1 Stateful Packet Processing

OpenFlow does not accommodate stateful packet operations, and thus, per-connection or per-packet forwarding state requires the controller to become involved. For this purpose, Open vSwitch allows running on-hypervisor "local controllers" in addition to a remote, primary controller. Because a local controller is an arbitrary program, it can maintain any amount of state across the packets that Open vSwitch sends it.

NVP includes, for example, a local controller that implements a stateful L3 daemon responsible for sending and processing ARPs. The L3 daemon populates an L3 ARP cache into a dedicated OpenFlow table (not managed by the primary controller) for quick forwarding of common case (packets with a known IP to MAC binding). The L3 daemon only receives packets resulting in an ARP cache miss and emits any necessary ARP requests to remote L3 daemons based on the packets received from Open vSwitch. While the connectivity between the local controller and Open vSwitch is local, the performance overhead is significant: a received packet traverses first from kernel to userspace daemon from which it traverses across a local socket (again via kernel) to a separate process.

For performance critical stateful packet operations, Open vSwitch relies on kernel networking facilities. For instance, a solid IP tunneling implementation requires (stateful) IP reassembly support. In a similar manner, transport connection tracking is a first practical requirement after basic L2/L3 networking; even most basic firewall security policies call for stateful filtering. OpenFlow is flexible enough to implement *static* ACLs but not stateful ones. For this, there's an ongoing effort to provide a new OpenFlow action that invokes a kernel module that provides metadata which the subsequent OpenFlow tables may use the connection state (new, established, related) in their forwarding decision. This "connection tracking" is the same technique used in many dedicated firewall appliances. Transitioning between kernel networking stack and kernel datapath module incurs overhead but avoids the duplication of functionality, critical in upstreaming kernel changes.

8.2 Userspace Networking

Improving the virtual switch performance through userspace networking is a timely topic due to NFV [9, 22]. In this model, packets are passed directly from the NIC to VM with minimal intervention by the hypervisor userspace/kernel, typically through shared memory between NIC, virtual switch, and VMs. To this end, there is an ongoing effort to add both DPDK [11] and netmap [30] support to Open vSwitch. Early tests indicate the Open vSwitch caching architecture in this context is similarly beneficial to kernel flow cache.

An alternative to DPDK that some in the Linux community are investigating is to reduce the overhead of going through the kernel. In particular, the SKB structure that stores packets in the Linux kernel is several cache lines large, contrary to the compact representation in DPDK and netmap. We expect the Linux community will make significant improvements in this regard.

8.3 Hardware Offloading

Over time, NICs have added hardware offloads for commonly needed functions that use excessive host CPU time. Some of these features, such as TCP checksum and segmentation offload, have proven very effective over time. Open vSwitch takes advantage of these offloads, and most others, which are just as relevant to virtualized environments. Specialized hardware offloads for virtualized environments have proven more elusive, though.

Offloading virtual switching entirely to hardware is a recurring theme (see, *e.g.,* [12]). This yields high performance, but at the cost of flexibility: a simple fixed function hardware switch effectively replaces the software virtual switch with no ability for the hypervisor to extend its functionality. The offload approach we currently find most promising is to enable NICs to accelerate kernel flow classification. The Flow Director feature on some Intel NICs has already been shown to be useful for classifying packets to separate queues [36]. Enhancing this feature simply to report the matching rule, instead of selecting the queue, would make it useful as such for megaflow classification. Even if the TCAM size were limited, or if the TCAM did not support all the fields that the datapath uses, it could speed up software classification by reducing the number of hash table searches – without limiting the flexibility since the actions would still take place in the host CPU.

8.4 Related Work

Flow caching. The benefits of flow caching generally have been argued by many in the community [4, 13, 17, 31, 41]. Lee et al. [21] describes how to augment the limited capacity of a hardware switch's flow table using a software flow cache, but does not mention problems with flows of different forms or priorities. CacheFlow [13], like Open vSwitch, caches a set of OpenFlow flows in a fast path, but CacheFlow requires the fast path to directly implement all the OpenFlow actions and requires building a full flow dependency graph in advance.

Packet classification. Classification is a well-studied problem [37]. Many classification algorithms only work with static sets of flows, or have expensive incremental update procedures, making them unsuitable for dynamic OpenFlow flow tables [7, 8, 33, 38, 40]. Some classifiers require memory that is quadratic or exponential in the number of flows [8, 20, 35]. Other classifiers work only with 2 to 5 fields [35], whereas OpenFlow 1.0 has 12 fields and later versions have more. (The effective number of fields is much higher with classifiers that must treat each bit of a bitwise matchable field as an individual field.)

9 Conclusion

We described the design and implementation of Open vSwitch, an open source, multi-platform OpenFlow virtual switch. Open vSwitch has simple origins but its performance has been gradually optimized to match the requirements of multi-tenant datacenter workloads, which has necessitated a more complex design. Given its operating environment, we anticipate no change of course but expect its design only to become more distinct from traditional network appliances over time.

References

[1] T. J. Bittman, G. J. Weiss, M. A. Margevicius, and P. Dawson. Magic Quadrant for x86 Server Virtualization Infrastructure. Gartner, June 2013.

[2] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

[3] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *Proc. of SIGCOMM*, 2007.

[4] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking Packet Forwarding Hardware. In *Proc. of HotNets*, 2008.

[5] Crehan Research Inc. and VMware Estimate, Mar. 2013.

[6] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, pages 385–398, 2013.

[7] A. Feldman and S. Muthukrishnan. Tradeoffs for Packet classification. In *Proc. of INFOCOM*, volume 3, pages 1193–1202 vol.3, Mar 2000.

[8] P. Gupta and N. McKeown. Packet Classification Using Hierarchical Intelligent Cuttings. In *Hot Interconnects VII*, pages 34–41, 1999.

[9] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. of NSDI*, Apr. 2014.

[10] IEEE Standard 802.1ag-2007: Virtual Bridged Local Area Networks, Amendment 5: Connectivity Fault Management, 2007.

[11] Intel. *Intel Data Plane Development Kit (Intel DPDK): Programmer's Guide*, October 2013.

[12] Intel LAN Access Division. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. `http://www.intel.com/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf`, January 2011.

[13] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Infinite CacheFlow in Software-Defined Networks. In *Proc. of HotSDN*, 2014.

[14] D. Katz and D. Ward. Bidirectional Forwarding Detection (BFD). RFC 5880 (Proposed Standard), June 2010.

[15] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *Proc. of NSDI*, 2013.

[16] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. of NSDI*, 2012.

[17] C. Kim, M. Caesar, A. Gerber, and J. Rexford. Revisiting Route Caching: The World Should Be Flat. In *Proc. of PAM*, 2009.

[18] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster. SAX-PAC (Scalable And eXpressive PAcket Classification). In *Proc. of SIGCOMM*, 2014.

[19] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *Proc. of NSDI*, Seattle, WA, Apr. 2014.

[20] T. Lakshman and D. Stiliadis. High-speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. *SIGCOMM CCR*, 28(4):203–214, 1998.

[21] B.-S. Lee, R. Kanagavelu, and K. M. M. Aung. An Efficient Flow Cache Algorithm with Improved Fairness in Software-Defined Data Center Networks. In *Proc. of Cloudnet*, pages 18–24, 2013.

[22] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proc. of NSDI*, Apr. 2014.

[23] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *AUUG Conference Proceedings*, page 175. AUUG, Inc., 2001.

[24] Microsoft. Hyper-V Virtual Switch Overview. `http://technet.microsoft.com/en-us/library/hh831823.aspx`, September 2013.

[25] The Netperf homepage. `http://www.netperf.org/`, January 2014.

[26] Open vSwitch – An Open Virtual Switch. `http://www.openvswitch.org`, September 2014.

[27] OpenFlow. `http://www.opennetworking.org/sdn-resources/onf-specifications/openflow`, January 2014.

[28] B. Pfaff and B. Davie. The Open vSwitch Database Management Protocol. RFC 7047 (Informational), Dec. 2013.

[29] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. In *Proc. of HotNets*, Oct. 2009.

[30] L. Rizzo. netmap: A novel framework for fast packet I/O. In *Proc. of USENIX Annual Technical Conference*, pages 101–112, 2012.

[31] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang. Leveraging Zipf's Law for Traffic Offloading. *SIGCOMM CCR*, 42(1), Jan. 2012.

[32] N. Shelly, E. Jackson, T. Koponen, N. McKeown, and J. Rajahalme. Flow Caching for High Entropy Packet Fields. In *Proc. of HotSDN*, 2014.

[33] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. In *Proc. of SIGCOMM*, 2003.

[34] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification Using Tuple Space Search. In *Proc. of SIGCOMM*, 1999.

[35] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and Scalable Layer Four Switching. In *Proc. of SIGCOMM*, 1998.

[36] V. Tanyingyong, M. Hidell, and P. Sjodin. Using Hardware Classification to Improve PC-based OpenFlow Switching. In *Proc. of High Performance Switching and Routing (HPSR)*, pages 215–221. IEEE, 2011.

[37] D. E. Taylor. Survey and Taxonomy of Packet Classification Techniques. *ACM Computing Surveys (CSUR)*, 37(3):238–275, 2005.

[38] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing Packet Classification for Memory and Throughput. In *Proc. of SIGCOMM*, Aug. 2010.

[39] VMware. vSphere Distributed Switch. `http://www.vmware.com/products/vsphere/features/distributed-switch`, September 2014.

[40] T. Y. C. Woo. A Modular Approach to Packet Classification: Algorithms and Results. In *Proc. of INFOCOM*, volume 3, pages 1213–1222 vol.3, Mar 2000.

[41] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-based Networking with DIFANE. In *Proc. of SIGCOMM*, 2010.

[42] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance Ethernet forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 97–108, New York, NY, USA, 2013. ACM.

Yesquel: scalable SQL storage for Web applications

Marcos K. Aguilera
Joshua B. Leners
UT Austin
Ramakrishna Kotla
Michael Walfish
NYU

ABSTRACT

Based on a brief history of the storage systems for Web applications, we motivate the need for a new storage system. We then describe the architecture of such a system, called Yesquel. Yesquel supports the SQL query language and offers performance similar to NOSQL storage systems.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed databases*; H.2.4 [**Database Management**]: Systems—*Distributed databases, relational databases, transaction processing*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms

Design, Performance, Reliability

Keywords

Distributed storage systems, SQL, NOSQL, transactions, database systems, scalability, Web applications

1. INTRODUCTION

Web applications (web mail, web stores, social networks, etc) keep massive amounts of data (account settings, user preferences, passwords, emails, shopping carts, wall posts, etc). The design of such an application often revolves around the underlying storage system.

This paper briefly describes a new storage system for Web applications, called Yesquel. Yesquel combines several advantages of prior systems: it supports the SQL query language to facilitate the design of applications, and it offers performance similar to NOSQL key-value storage systems.

To achieve these goals, Yesquel leverages techniques for scalability and fault tolerance previously used in NOSQL systems, and uses them to obtain a SQL system. In addition, Yesquel incorporates a unique architecture that provides an embedded query processor to each of its clients, that implements distributed transactions at a low level, and that builds a distributed index structure on top of such transactions. This architecture is accompanied by new and efficient mechanisms to execute transactions and to store database tables and indexes.

This paper focuses on Yesquel's motivation and architecture. A subsequent paper will describe Yesquel's transactions and storage engine.

ICDCN '15, January 04 - 07 2015, Goa, India

http://dx.doi.org/10.1145/2684464.2684504.

2. HISTORICAL PERSPECTIVE AND MOTIVATION

The storage systems used in Web applications have evolved dramatically over the past 25 years, and the history brings interesting insights. We can roughly divide these storage systems into four generations.

- *First generation: file systems.* In the early 1990s, Web pages were static. Web servers received HTTP requests for a file, read the file from their local file system, and returned it to the user. The storage system for Web applications was simply the file system holding such files.

- *Second generation: file and* SQL *database systems.* In the mid to late 1990s, the Web saw the emergence of *dynamic content*—content that depends on who the user is or what the user has done (e.g., the items in a shopping cart). To generate a page, the Web server invoked a program written in languages such as Perl, Python, PHP, Java, etc. The program stored data needed to generate the page in a central SQL database system. The use of SQL was convenient, because SQL has many useful features (joins, secondary indexes, transactions, aggregations, many data types, etc). The storage system, thus, was a combination of the file system for static content and the database system for dynamic content.

- *Third generation: highly scalable systems.* In the 2000s, large Web sites emerged, such as Amazon, Hotmail, Google, Yahoo, and others. These sites had a rapid growth in the number of users; soon the database system became a performance and scalability bottleneck. Scaling the database system was hard or expensive, so developers decided to replace the SQL database system with their own home-grown storage systems, such as the Google File System [9], BigTable [4], Dynamo [7],

and others. This was the beginning of a movement that later became known as NOSQL.

- *Fourth generation: cloud storage systems.* In the 2010s, many Web applications moved to the cloud, where many vendors share the same computing and storage infrastructure. Storage systems were designed not just to scale, but also to provide isolation, so that applications do not interfere with one another. Examples of such storage systems include Amazon S3, SimpleDB, Azure Blobs, and Azure Tables.

A highlight in this history is the emergence of the NOSQL movement, ten years ago, which sought to replace the SQL database system with cheaper custom-built alternatives. These alternatives were much simpler than a SQL database system, and thus were easier to scale, but they offered more restricted functionality: few NOSQL systems offer transactions, secondary indexes, joins, or aggregations, and certainly no NOSQL systems offer all the features found in SQL. Thus, the move from the SQL database system to NOSQL systems came at the cost of functionality.

Today, there are several dozens of NOSQL systems, each offering a different set of features, and each with its own application interface. Web application developers face a difficult choice of what storage system to use, because it is unclear a priori what features the application might need. Even if developers can identify an appropriate system, the application may evolve and later require functionality from the storage system that was not originally deemed useful. Because each storage system provides its own interface, once an application is developed to use one storage system, it becomes hard to replace it with another storage system should the need arise—a problem known as vendor lock-in.

We start with the observation that NOSQL is not a feature but rather the absence of a feature. What is the NOSQL movement really trying to achieve? The answer is a distributed storage system that is scalable; fault tolerant; simple and sane to design, implement, and maintain; and nimble and cheap to run.

We took these characteristics as our goal in developing Yesquel, but we also wanted to provide full support for the SQL query language. Besides having a rich set of features, SQL is a well-known query language (it is taught in database courses), and it is an industry standard, so applications developed for SQL avoid the problem of vendor lock-in.

3. ARCHITECTURE

A SQL database system has two main components: a query processor and a storage engine. The query processor parses and executes SQL queries, while the storage engine stores tables and indexes.

The architecture of Yesquel is depicted in Figure 1. Each client has its own embedded query processor (box ① in the figure), which is a user library that links against the client application. As a result, as the number of clients increases, the number of query processors also increases proportionately. This property allows Yesquel to scale the query processing capacity of the system. The query processors all share a common storage engine.

For this idea to work, the storage engine must be designed to handle a large number of query processors, which at a given time may try to access the same tables and indexes.

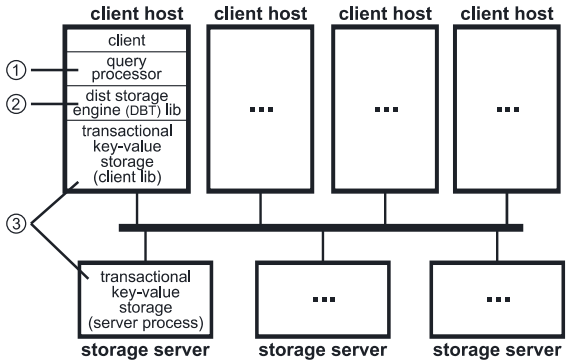


Figure 1: Architecture of Yesquel.

Handling concurrent access to such data is a key challenge addressed by Yesquel.

In Yesquel, the storage engine is implemented as a *distributed balanced tree (DBT)* (box ②). A DBT is a balanced tree data structure whose nodes are distributed over a set of storage servers. The reason for distribution is to scale the performance of the DBT; in Yesquel, this is done by increasing the number of storage servers.

The Yesquel DBT library is implemented on top of a transactional key-value storage system—a simple transactional system that stores key-value pairs on the storage servers (boxes ③). That is, the nodes of the Yesquel DBT are stored as key-value pairs in the key-value storage system. This design separates the implementation of the DBT from the implementation of the distributed transactions. The transactions in the key-value storage system provide *snapshot isolation* [3], a property often used in commercial database systems. Because the DBT is implemented above the transactions, the DBT benefits from the full power of transactions; for example, the Yesquel DBT uses transactions to atomically move data across DBT nodes to balance load.

Note that transactions are provided at the lowest layer—the key-value storage system, which is the layer that actually stores the data bits on storage servers. As a result, the transactional protocol enjoys greater efficiency. Specifically, Yesquel uses multi-version concurrency control—the most sophisticated form of concurrency control—which requires managing multiple versions of each data item, and this can be done most efficiently and effectively at the layer that stores the actual data.

4. RELATED WORK

F1 [13] is a distributed storage system designed for Google's AdWords; it provides support for SQL with a non-relational hierarchical model. F1 has a different architecture from Yesquel: F1 is layered on top of the Spanner system, which in turn is layered onto a Bigtable-based implementation. Spanner [5] provides distributed transactions, while Bigtable provides the DBT functionality. Thus, the DBT is implemented *below* transactions. In contrast, in Yesquel distributed transactions are provided at the lowest level (the key-value storage system), and the DBT is implemented *above* the transactions. As explained, this choice allows the DBT to leverage transactions. Moreover, Yesquel provides an embedded query processor to clients. Yesquel also uses different protocols for transactions, which unlike F1 do not

require special hardware clocks. However, F1 provides a stronger transaction isolation property (strict serializability) than Yesquel, and F1 works in a geo-distributed deployment, whereas Yesquel runs within a single data center.

MOSQL [15, 16] is a distributed storage engine for MYSQL. MOSQL has a different architecture from Yesquel: its transactional layer runs on top of (1) a distributed storage layer without transactions, and (2) a certifier service implemented as a replicated state machine. Atop the transaction layer, MOSQL has a B+tree. In contrast, Yesquel provides distributed transactions at the lowest level, Yesquel uses a more sophisticated DBT, and Yesquel does not use a logically centralized certifier. As a result, we believe Yesquel is more efficient and scalable than MOSQL. However, MOSQL provides a stronger transaction isolation (serializability) than Yesquel.

Traditionally, there are two broad architectures for a distributed SQL database system: shared-nothing and shared-disk [11]. Each has advantages, and there is a long-running debate about the two approaches, with commercial vendors supporting one or another, and sometimes both.

In *shared-nothing* systems (Clustrix, Greenplum, H-Store/VoltDB [10], IBM DB2 DPF, Microsoft SQL Server, MySQL Cluster, Netezza, Tandem NonStop, Teradata, etc.), each database server stores part of the data; the system decomposes queries, executes the sub-queries at the appropriate servers, and combines the results for the client. The benefits of this architecture can be substantial; for example, subquery processing happens close to the data, thereby avoiding network communication. However, performance crucially depends on a good partition of the data, and such a good partition may not exist (if the query set is dynamic). Or a partition may be expensive to identify [14]: the classical approach is to rely on a (well-paid) database administrator to partition manually. Software can help (as in H-Store/VoltDB [10, 12]) but not in all cases.

In *shared-disk* systems (IBM DB2 pureScale, Oracle RAC, ScaleDB, Sybase ASE Cluster Edition, etc.), every database server can access every data item; the disks are shared over the network. The advantage is easier management as there is no need to carefully partition the database. However, in this architecture, servers must carefully coordinate access to storage; the traditional solutions use distributed locks, distributed leases, and cache coherence protocols, which bring complexity and cost.

A recent movement called NEWSQL advocates new architectures for database systems. NEWSQL systems include H-Store/VoltDB and Hekaton. H-Store/VoltDB is an in-memory shared-nothing system, where each server runs a single thread to avoid synchronization overheads. Similar to other shared-nothing systems, performance critically depends on a good partition of the data. Hekaton is a centralized in-memory database system that features a lock-free index structure; it performs well, but scalability is limited to a single machine.

Finally, there has been much work on Big Data systems that process massive data sets at many hosts (e.g., [1, 2, 6]). Some of these systems support SQL [17, 8]. However, they are intended for data analytics applications and are not directly applicable to Web applications.

5. SUMMARY

We described the architecture of Yesquel, a storage system designed for Web applications, whose workload consists of relatively small queries that must execute quickly. Yesquel provides full support for SQL and offers performance similar to NOSQL key-value storage systems. Basically, Yesquel maps SQL queries to operations on a DBT, which in turn are mapped to operations on a transactional key-value storage system. In other words, Yesquel internally leverages a NOSQL storage system to provide its scalability.

6. REFERENCES

[1] `http://hbase.apache.org`.

[2] `http://hadoop.apache.org`.

[3] H. Berenson et al. A critique of ANSI SQL isolation levels. In *International Conference on Management of Data*, pages 1–10, May 1995.

[4] F. Chang et al. Bigtable: A distributed storage system for structured data. In *Symposium on Operating Systems Design and Implementation*, pages 205–218, Nov. 2006.

[5] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *Symposium on Operating Systems Design and Implementation*, pages 251–264, Oct. 2012.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation*, pages 137–150, Dec. 2004.

[7] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles*, pages 205–220, Oct. 2007.

[8] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB Endowment*, 2(2):1402–1413, Aug. 2009.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, pages 29–43, Oct. 2003.

[10] R. Kallman et al. H-Store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, Aug. 2008.

[11] K. D. Levin and H. L. Morgan. Optimizing distributed data bases: a framework for research. In *National computer conference*, pages 473–478, May 1975.

[12] A. Pavlo, C. Curino, and S. B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *International Conference on Management of Data*, pages 61–72, May 2012.

[13] J. Shute et al. F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, Aug. 2013.

[14] M. Stonebraker. The case for shared nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, Mar. 1986.

[15] A. Tomic. *MoSQL, A Relational Database Using NoSQL Technology.* PhD thesis, Faculty of Informatics, University of Lugano, 2011.

[16] A. Tomic, D. Sciascia, and F. Pedone. MoSQL: An

elastic storage engine for MySQL. In *Symposium On Applied Computing*, pages 455–462, Mar. 2013.

[17] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *International Conference on Management of Data*, pages 13–24, June 2013.

Taming uncertainty in distributed systems with help from the network

Joshua B. Leners[*‡] Trinabh Gupta[*‡] Marcos K. Aguilera[†] Michael Walfish[‡]

[*]The University of Texas at Austin [†]VMware Research Group [‡]NYU

Abstract

Network and process failures cause complexity in distributed applications. When a remote process does not respond, the application cannot tell if the process or network have failed, or if they are just slow. Without this information, applications can lose availability or correctness. To address this problem, we propose Albatross, a service that quickly reports to applications the current status of a remote process—whether it is working and reachable, or not. Albatross is targeted at data centers equipped with software defined networks (SDNs), allowing it to discover and *enforce* network partitions: Albatross borrows the old observation that it can be better to cause a problem than to live with uncertainty, and applies this idea to networks. When enforcing partitions, Albatross avoids disruption by disconnecting only individual processes (not entire hosts), and by allowing them to reconnect if the application chooses. We show that, under Albatross, distributed applications can bypass the complexity caused by network failures and that they become more available.

1 Introduction

In a distributed application, if a process stops responding, the rest of the application cannot be sure what is going on. Has the process crashed? Is there a network failure? Maybe there are no failures, and the issue is that the process is slow or the network is congested?

Unfortunately, if the application guesses incorrectly, it ends up with problems. These include split-brain scenarios (an incorrect guess that there was a problem can cause multiple instantiations of a process), consistency violations (if clients access the wrong server), and loss of availability (if the process incorrectly guesses that there are no problems).

This *uncertainty* about whether a failure exists is a perennial source of complexity in distributed systems (§2.2). Indeed, many applications devote substantial code to the problem (e.g., they implement fault-tolerant protocols that are impervious to uncertainty [12, 39, 48]). Other applications avoid the problem by leveraging *membership services* that track working processes. However, these services have various limitations. They take tens of seconds[1] or longer to react to failure [13, 36] (and faster reaction times would cause collateral damage [13, §2.8]), they sometimes halt working machines to eliminate uncertainty [26, 51],[2] or they cannot handle network problems [50].

In this paper, we propose a new membership service, called *Albatross*. We target data center networks and assume the presence of Software Defined Networks (SDNs). Albatross is a new design point. To our knowledge, it is the first membership service that achieves the following combination: (1) it addresses common network failures (as discussed below, Albatross cannot address *all* network failures: doing so would violate known impossibility results), (2) it is quick (it answers in less than a second), which improves availability, and (3) it avoids interfering with working processes and machines, which also improves availability. Albatross is a complete system and, as such, handles process failures too, but we will not focus on this aspect, since it is well-studied [9, 15, 26, 34, 50, 51, 67].

Albatross is based on two high-level insights. First, the ability to monitor and configure network elements makes it possible to provide (1)–(3), for reasons that will be explained in the coming pages. Second, SDNs provide a standard interface to the required network functionality.

1.1 Components and requirements

Albatross consists of a *host module* (installed on the hosts of applications that use the service) and a few replicated servers, called *managers*. The host module communicates with the managers, and exposes an interface that a process can query to learn the failure status of remote processes. Using SDN functionality, the managers receive notifications about the state of the network, determine which processes are reachable, and enforce their determinations by installing *drop rules* on switches. Albatross adopts several requirements:

(1) Provide guarantees that are well-defined and useful to applications. Ideally, Albatross would be an oracle that answers any query with perfect information, immediately. But this ideal is impossible: Albatross may be ignorant of a process's true status, and Albatross cannot provide information to processes that it cannot reach. Thus, Albatross relaxes the strength and scope of its guarantees. First, rather than promise perfect information, Albatross provides *definitive* reports, which guarantee the failure status of a remote process. To provide this guarantee, Albatross sometimes *inter-*

EuroSys '15, April 21–24, 2015, Bordeaux, France.
ACM 978-1-4503-3238-5/15/04.
http://dx.doi.org/10.1145/2741948.2741976

[1]Tens of seconds can be an expensive outage: some Web properties make tens or even hundreds of thousands of dollars per minute.

[2]This technique is called STONITH (for Shoot The Other Node In The Head).

feres with processes (as noted in the next requirement), which amounts to applying an old technique (STONITH [26, 50, 51]) to a new context (SDN-enabled networks). Second, Albatross provides *asymmetric* guarantees: it categorizes processes as *excluded* or *non-excluded* and promises definitive answers only to non-excluded processes. Third, Albatross allows reports to be delayed in favor of being definitive, but it strives to be quick (sub-second detection time). To our knowledge, this combination is new, and we find that it is strong enough to be useful to applications.

(2) Limit interference. Ideally, Albatross would only report events, not influence them. On the other hand, a convenient way to provide definitive responses is to interfere [26, 50, 51]. To balance these concerns, Albatross does several things. First, it inspects the state of the network rather than relying on coarse inferences from timeouts. Second, it manipulates that state at fine grain; its technique here is to embed a name space for applications in source MAC addresses, which enables switches to block the traffic of individual processes. The result is to avoid unwarranted interference with processes that use Albatross, and to eliminate interference with processes that do not use Albatross. Finally, Albatross allows disconnected processes to later reconnect.

(3) Use few resources. Switches have limited space for drop rules [35]. To work within this constraint, Albatross names processes according to their starting time and enclosing application; this naming scheme allows drop rules to be aggregated when failures affect many processes. Furthermore, Albatross must eventually garbage collect unused drop rules and other resources. To this end, Albatross introduces a reference counting scheme that handles distributed references and tolerates faults.

(4) Tolerate failures within Albatross itself. Albatross is itself a distributed system and, as such, is subject to the very failures that it wishes to detect. To be useful, Albatross must function under reasonable and common failures. (As an analogy, a fire alarm must function under usual types of fires.) Albatross responds in various ways. First, Albatross is replicated using Paxos [46]. Second, Paxos requires that a majority of the servers are responsive and mutually connected, which Albatross achieves by carefully placing servers (§5.3). Third, Albatross has a principled design, to avoid architecture flaws that lead to spurious failures.

Albatross cannot survive and report every conceivable network and process problem: doing so would mean tolerating arbitrary network partitions, which is impossible [29]. Albatross's limitations are, first, that it cannot survive failures that disconnect or kill a majority of its managers (as explained above). Second, to obtain service from Albatross, a host must be able to reach a majority of Albatross managers. The upshot is that Albatross does not work under catastrophic events (e.g., failure of power and backup); however, such failures take the data center offline anyway. Albatross does tolerate failures that affect only a restricted part of the network—which are the common case, according to a study of the failure events in the data centers of a large production service (§2.1).

1.2 Performance, results, and contributions

In evaluating Albatross (§7), we find that it has low cost: it requires little state in the network (fewer than 5 rules per switch to enforce disconnection), and uses little CPU and memory. Yet, it detects network failures an order of magnitude more quickly than the ZooKeeper membership service [36] (we will refer to this membership service as simply "ZooKeeper"). This gain owes to the design of Albatross: if ZooKeeper were to lower its timeouts to gain the same speed, its servers would be overwhelmed (§7.2).

Furthermore, we demonstrate that Albatross's guarantees are useful to applications. We show that integrating RAMCloud [59] with Albatross prevents clients from communicating with servers that have been declared failed; this eliminates a consistency bug in RAMCloud. We also show that Albatross can be used in place of existing membership services in distributed algorithms, despite its different guarantees.

The concrete contributions of this paper are as follows:

- Albatross, a service that provides a new combination: it reports network (and process) failures definitively, quickly, and with little interference (§3).
- A formalization of Albatross's guarantees, so that applications can reason about reports (§4).
- The design and implementation of Albatross; the design carefully composes novel techniques (a process naming scheme, a way of dropping processes' traffic at fine grain, a readmission protocol, algorithms for garbage collection) with existing ones (SDNs, Paxos [46], Falcon spies [50]) to produce a coherent system (§5–§6).
- An experimental evaluation (§7) of the implementation.

There is one more contribution of this paper, and it is conceptual. Whereas the purpose of SDNs was originally simplifying network management, this paper identifies a different use of SDNs: enhancing classical distributed systems. This connection had not been observed before, and we think that it may be more widely applicable.

2 Further motivation

2.1 What network partitions happen in data centers?

We analyzed a year-long trace of the failures that occurred in several data centers of a company with a strong Internet presence, using similar methods to Gill et al. [30]. Events in the failure trace were generated by in-device monitoring and were collected in a central repository using monitoring protocols (such as SNMP) or manual intervention. Events are tagged with metadata, including what type of device failed, and whether the failure was masked by redundancy; we used these tags to determine which events created partitions.

We found that larger data centers (more than a thousand net-

work elements) had about 12 partitions per month, of which about half disconnected an entire rack and half disconnected a single host. The partitions in the larger data centers never disconnected more than a single rack (owing to path redundancy), but we found that smaller data centers (fewer than 600 network elements) experienced multi-rack partitions. With this information in mind, we focus attention on partitions in a single data center that affect a subset of the network.

2.2 The whys, whats, and hows of membership services

One way to detect failures is by using end-to-end timeouts [9, 15, 34, 67]. However, timeouts are troublesome. First, a poor choice can compromise availability: if the timeout is too long, the system is forced to wait, and if the timeout is too short, the system wastes time responding to non-failures. Worse, the variability of response times means that there may be no good choice of timeout [74]. A second problem with timeouts is their inherent *uncertainty*: a timeout does not imply a failure.

To address this uncertainty directly, distributed systems typically use one or more of the following three techniques. The first technique uses a majority of processes to obtain agreement among correct processes [11, 16, 39, 46, 47, 58]. This technique is useful in building high-performance systems [12, 48], but it imposes particular structures on application developers. For example, applications must be built using the replicated state machine approach [45, 64] or using group communication primitives [11, 16].

A second technique is to use a mechanism like leases [13, 32, 36] or watchdogs [26] to ensure that suspected processes kill themselves in effect or fact. This technique assumes that the system has bounded timeliness. Also, short timeouts (as needed for fast detection) consume many resources (§7.2). Furthermore, with leases in particular, short timeouts can trigger complicated corner cases (§7.1).

The third technique is to kill processes that are suspected of having failed [50, 51]. A problem here is that killing tends to be coarse-grained (e.g., shutting down a machine). Furthermore, this technique can founder under network partitions, as the command to kill may not get through.

The aforementioned techniques bring complexity. Many services have been built that place these techniques behind a layer of abstraction that hides the complexity, allowing developers of distributed applications to interact with a clean abstraction. We refer to these services as *membership services*.[3] Membership services provide *definitive reports* about which processes are working, in the sense that processes can safely treat such reports as ground truth.

An alternative to end-to-end timeouts is to use information local to a system's components to detect failures [49, 50]. This approach allows a membership service, in the common case, to react to failures as they happen without waiting for an *end-to-end* timeout to expire. However, prior attempts at building membership services this way are incomplete: they do not handle network problems [50], or they give ambiguous reports [49]. (Section 9 elaborates.)

2.3 How does Albatross fit?

Albatross uses local information as hints but with a new mechanism for making reports definitive: before reporting a suspected process as not working, Albatross modifies the network to prevent the suspected process from sending messages to working processes. Albatross uses SDNs for this purpose. For its own fault-tolerance, Albatross uses majority-based techniques internally, thus following the tradition of implementing the complex technique once (§2.2).

As a concrete example, consider an application that uses primary-backup replication [4];[4] we use primary-backup as a running example for its conceptual simplicity, though membership services enable other fault-tolerance techniques (recovering from snapshots, raising an alarm, switching to a safe state, etc). In this application, the primary receives a request, replicates that request at the backup, and only then executes the request and responds to the requesting node. This setup provides fault-tolerance safely, via the invariant that replication happens before responding to the requestor.

However, for availability, the application needs a way to make progress if the primary or backup fails. Here is where the membership service enters. A standard choice would be ZooKeeper [36], which uses leases, and is used by production data center applications (e.g., [55]). The idea is that each replica acquires a lease at ZooKeeper named by its role ("primary" or "backup") and then monitors the other's lease. The backup learns that the primary is no longer working when the "primary" lease expires, and the backup can safely take over by acquiring the "primary" lease.

What if, as an alternative, the primary-backup application uses Albatross? Then, when Albatross suspects a partition (e.g., because a switch reports a link as down), it can install rules to stop the primary from using the network and then report the problem to the backup. The backup can then take over immediately—without having to wait for a lease to expire—because it knows that Albatross is preventing the primary from using the network.

3 Overview of Albatross

Albatross is a service that a process of a distributed application can query to learn about the status of a remote process. The status can be "disconnected" or "connected"; roughly, "disconnected" means crashed or partitioned, and "connected" means alive and reachable. If Albatross reports a process as "disconnected", it is safe to assume that process cannot affect the world. For the rest of this paper, a process refers to an

[3] This name is inspired by group membership services [11] but expanded in scope to include services such as ZooKeeper [36] and Chubby [13].

[4] More generally, membership services enable algorithms for consensus and atomic broadcast that can tolerate f failures using only $f+1$ processes [14] (versus $2f+1$).

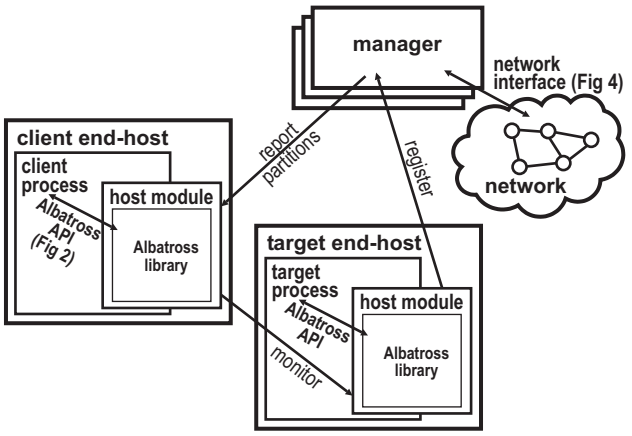


Figure 1—High-level view of Albatross. The host module provides the API through which applications use Albatross; the host module helps detect crashes of local processes. The manager is replicated at dedicated servers and coordinates Albatross's response to network and host failures. The manager interacts with the network through an abstract interface, and notifies clients about partitioned processes.

Function	Description
becomeAlbatrossProcess(appid)	register target process of app
handle = init((IP, proto, port), cb)	monitor a target, given by (IP, proto, port). callback cb is invoked when the target fails or is partitioned
query(handle)	return the state of target
startTimer(handle, timeout)	start timeout on target
stopTimer(handle)	cancel timeout on target
ackDisconnect()	acknowledge disconnection

Figure 2—The Albatross API.

operating system process.

Environmental assumptions. We target data centers with a single administrative domain. We presume the ability to configure network switches (e.g., via SDNs). We assume that minor modifications to end-host software are acceptable.

Components. Figure 1 depicts the components of Albatross. We survey them briefly below. (Section 5 gives details).

The *manager* detects, enforces, and reports network failures. Detecting and enforcing happens via a *network interface* that abstracts SDN-like features. Reporting happens by calling back client processes that have registered for notifications. The manager is a single logical entity that is replicated over several servers, using state machine replication. A *host module* detects and reports local process failures (to remote host modules); like the manager, this component uses callbacks for reporting. Much of the logic for detecting local process failures is borrowed [49, 50]. The host module also implements the Albatross API, described immediately below.

Albatross API. Figure 2 shows the Albatross API; Figure 3 gives an example use of the API and explains what causes communication among the components in Figure 1. A monitoring process is known as a *client*; a monitored process is called a *target*. To request monitoring, a client calls init(); this call generates a message to the target's host. Albatross returns notifications about the target via a client-supplied callback function or in response to query().

Action	Resulting communication
1. target calls becomesAlbatrossProcess()	target host module sends "register" RPC to the manager
2. client calls init(...), gets handle	client host module sends "monitor" RPC to the target's host module
3. client calls query(handle), gets "connected"	none
4. target crashes	target host module or manager sends RPC to the client's host module; host module invokes client callback (if any)
5. client calls query(handle), gets "disconnected"	none
6. target recovers, calls ackDisconnect()	target host module sends "deregister" RPC to the manager (not shown)
7. client calls init(...), gets new handle	client host module sends "monitor" RPC to the target's host module
8. client calls query(handle), gets "connected"	none

Figure 3—Example sequence of actions using the Albatross API and the resulting communication by Albatross.

The API serves three other purposes. First, processes register as targets with Albatross, by invoking becomeAlbatrossProcess(appid). This call may generate a message to the manager (the manager tracks applications). The specified appid should uniquely identify the application and should be used by all processes of the application.

Second, clients use the API to set an end-to-end timeout that serves as a *backstop* when Albatross cannot otherwise detect a problem. Specifically, Albatross expects a client process to call startTimer() when it is waiting for a message from a target process and to call stopTimer() when it receives the expected message. If the timer fires, Albatross disconnects the target and reports "disconnected".

Third, disconnected targets can reconnect, by calling ackDisconnect() (possibly after rolling back state), at which point monitoring clients must call init() again.

Informal contract. Albatross covers all host failures and common network failures. Its reports are definitive but asymmetric: it excludes some processes and promises definitive reports (about whether a process is excluded) only to non-excluded processes. Intuitively, the non-excluded processes are the ones that a majority of Albatross manager replicas can reach. These guarantees are formalized in Section 4.

In addition, Albatross provides fast (sub-second) detection time, which it achieves through its overall architecture: visibility into the network (which provides timely information), callbacks (which enable low latency without the overhead of

frequent polling), etc. Of course, one way to provide speed is to indiscriminately disconnect processes at any suspicion of a problem, but Albatross also limits interference, using the techniques summarized in Section 1.1.(2).

Rationale. Reporting *all* network failures is impossible [27]. Similarly, providing definitive, symmetric reports seems infeasible: how can a system give a report to a node that it cannot reach? Of course, just because a contract is feasible does not mean that it is useful to an application (Albatross could promise to return the string "elephant" always, which is feasible to implement but useless). Fortunately, Albatross's guarantees are useful to applications (§7.1), though there are some small corner cases, covered in the next section.

4 Albatross's contract

This section precisely describes Albatross's guarantees. We will define a set of *excluded* processes, and the guarantees will be asymmetric: processes outside the excluded set receive assurances that processes inside the set do not. The high-level concepts of exclusion and asymmetric guarantees have appeared before [10, 11, 16] but not, to our knowledge, in our specific context, namely failure reports [14].

Albatross's guarantees refer to a notion of *time*, which is a *logical* time at the Albatross manager; we are not assuming that entities in Albatross have synchronized clocks. We say that a *process p cannot reach process q at time t* if a message sent by p at time t would fail to be delivered to q (because, for example, q crashes before the packet arrives or there are no routes to q, or the routes to q disappear as the packet is traveling, etc.). Observe that this definition of "reachable" collapses a message's future and fate into a label associated with the sending time (t). We say that *processes p and q are partitioned at time t* (or p is partitioned from q at time t) if either p cannot reach q or q cannot reach p at time t.

The guarantees of Albatross are relative to a monotonically increasing set E of excluded processes. Intuitively, these are the processes that Albatross disconnects from the rest of the system (and the outside world). We denote by E_t the membership of E at time t. Albatross ensures the following:

- *(Exclusion Monotonicity) Processes are excluded permanently.* More precisely, if $t \leq t'$ then $E_t \subseteq E_{t'}$.
- *(Isolation) Non-excluded processes do not receive messages from excluded processes.* More precisely, if $p \in E_t$, $q \notin E_t$, and p sends a message to q at time t, then q never receives that message. In particular, if q receives a message from p, that message must have been sent before time t.

Exclusions are permanent, but in practice an application may wish to reconnect the process. This is allowed and modeled by having the process assume a new id.

The next property states that a process is indeed excluded if something bad happens to it:

- *(Exclusion Completeness) If a process has a problem for sufficiently long, then it is eventually excluded.* If q has crashed, or q is permanently partitioned from a process that is never excluded, then $q \in E_t$ for some t.

The above property does not guarantee immediate exclusion when the problem occurs, because the system may take some time to detect the problem; in practice, it is desirable that this delay be as small as possible. Also, exclusion is not guaranteed if q is partitioned temporarily, because the partition can heal before Albatross notices it. Similarly, exclusion is not guaranteed if q is partitioned from a process r that later gets excluded, because the exclusion of r may happen before Albatross notices the partition between q and r.

The final property states that queries by a non-excluded process return "disconnected" or "connected" according to whether the remote process is excluded.

- *(Correspondence) If a process is excluded then eventually a query about it by a non-excluded process always returns "disconnected". Moreover, a query about a process by a non-excluded process returns "disconnected" only if the process is excluded.* More precisely, if $q \in E_t$ then there is a time t_q such that, for all $t' > t_q$, a query about q by $p \notin E_{t'}$ returns "disconnected". If $p \notin E_t$ and a query about q by p returns "disconnected" at time t, then $q \in E_t$.

All properties above are conditional; Albatross provides them if the application follows the expectations in Section 3 (processes register, set backstop timeouts, etc.), and if a majority of Albatross managers remains alive and mutually connected (per the fault-tolerance discussions in Sections 1.1 and 5.3).

Consequences of the guarantees, and an example

- Albatross may return incorrect answers to queries done *by* excluded processes. This asymmetry is acceptable: to the *non-excluded* part of the system—which includes the outside world—these processes are as good as dead.
- Messages sent by an excluded process *before* Albatross reports a partition may still be received by a non-excluded process *after* Albatross's report. However, all of the non-excluded processes know that these messages causally precede Albatross's report because of the Isolation property, and can act accordingly (e.g., by dropping stale messages).
- Excluded processes may continue to interact with, and affect, *each other*. Thus, prior to reconnecting, excluded processes must rollback their state to some checkpoint that causally precedes [45] Albatross's "disconnected" report. By rolling back their state, excluded processes accept their effective deaths, and can be safely reintegrated using standard catch-up techniques (e.g., replay).
- It is possible for a crashed process to be temporarily reported as "connected"; the Completeness and Correspondence properties together imply that if a process has crashed or been partitioned, Albatross *eventually* reports it as "disconnected".

We now revisit the primary-backup example from Section 2.3, focusing on corner cases. First, consider the case that the backup receives a request from the primary *after* it hears

that the primary is "disconnected". The backup can safely discard this message because it knows that *the primary could not have responded to the requesting node (causally) before it was excluded* (and if it responds to the requesting node causally after exclusion, then the requesting node is also excluded, by Isolation). The italicized phrase holds because during the period when the primary was *not* excluded, it would have correctly observed the backup as "connected" (by Correspondence), and thus waited for an acknowledgment from the backup before responding to the requesting node.

Second, suppose a backup "takes over" for a non-excluded primary (a potential split-brain scenario). A correct backup will take over only if it hears that the primary is "disconnected"; since the primary is not in fact excluded, then the backup must be (by Correspondence). Thus, the "take-over" by the backup is something akin to a delusion (experienced by the backup and perhaps other excluded hosts).

What about reconnection? An excluded replica may eventually learn that it is excluded, for example, by querying its own state or receiving a "you are disconnected" message from the other replica. Then, the replica must determine a checkpoint from before it was excluded and rollback to it before reconnecting (via ackDisconnect()) and then replaying. For an excluded backup, a suitable checkpoint would be the one prior to the last request received from the primary.

5 Detailed design

This section describes Albatross's design, bottom up; we begin with the scheme by which processes are named and end with the core logic that enforces partitions and rehabilitates processes. Section 6 describes notable implementation details.

5.1 Names and identifiers

Under Albatross, each target process receives a *process id (pid)* when it registers (§3) with the host module. This pid uniquely identifies the process in terms of its host, application, and birth period. A pid contains the following fields:

- A *host id* (for example, an IP address);
- An *application id* (appid), which is programmer-supplied and unique to applications within the given network (§3);
- A *local id*, which differentiates multiple processes of the same application on the same host; and
- An *epoch number*, which identifies the epoch in which the process registered.

Epochs are determined by the manager; an epoch corresponds to a view of the network's topology and partitions.

Pids are carried in packets. Albatross uses the fields of a pid to create partitions (by filtering traffic). The choice of field depends on the desired granularity of a partition. For example, Albatross uses epochs when it needs to create a partition affecting an entire rack of end-hosts. We describe the interface for enforcing partitions next.

Primitive	Description
CUT-APP(switch, appid, port)	drop incoming traffic of application appid entering port of a switch
CUT-EPOCH(switch, epoch, port)	drop incoming traffic of epoch entering port of a switch
BLOCK(switch, pid)	drop all incoming traffic of process pid at a switch
SUBSCRIBE(destination)	request topology information and failure events to be sent to a destination

Figure 4—Network interface used by Albatross.

5.2 Network interface

As noted earlier, Albatross relies on SDN-like functionality from the network (though SDNs per se are not required to implement Albatross, as discussed in Section 8). Here, we describe the functionality in terms of an abstract interface, depicted in Figure 4. (Section 6.2 describes an implementation of this interface, using OpenFlow and NOX [33].)

CUT-APP and CUT-EPOCH tell a switch to block incoming traffic that (a) enters the given port and (b) matches the given appid or epoch (§5.1). BLOCK tells a switch to block traffic belonging to a given process id on all ports. Albatross also requires the ability to undo CUT-APP, CUT-EPOCH, and BLOCK (not shown in Figure 4). SUBSCRIBE tells switches where to send information about network topology and failure events. Events of interest are *link failure*, indicating that a link is deemed down; and *end-host failure*, indicating that a host connected to a port is deemed down. The intent is that these events are sent to the manager, described next.

5.3 Manager

Albatross's manager coordinates the response to most failures.

Network failures. At a high level, the manager tracks the network topology; when the topology experiences a partition, the manager chooses a main partition, asks switches at the edge of the main partition to block the traffic of Albatross applications coming from outside the partition, and then calls back clients to notify them about which processes have been disconnected. This procedure does not affect applications that do not use Albatross; it also does not affect applications that use Albatross but are launched after the failure is resolved. (One can think of this approach, loosely, as virtualizing partitions, in that different applications see different views of the network topology.)

In more detail, the manager runs the logic in Figure 5. The manager maintains a model of the current network topology. To that end, the manager, on starting up, requests notifications about topology changes, using SUBSCRIBE (our implementation assumes that the manager also begins with a correctly configured base topology; a less lazy implementation could build the topology as switches join). When the manager re-

```
at startup call SUBSCRIBE(self)

function handle_failure_event(link):
  remove link from topology
  if topology has a new partition:
    enforced := false
    pick candidate excluded set P
    while not enforced:
      enforced := enforce_partition(P)
    report_partition(P)

function enforce_partition(P):
  for each switch.port connecting to P:
    try:
      if switch.port connects to an end-host:
        for each appid in activeAppid running at end-host:
          call CUT-APP(switch, appid, switch.port)
      else: // switch.port connects to another switch
        for each epoch in activeEpochs:
          call CUT-EPOCH(switch, epoch, switch.port)
        currentEpoch := get_inactive_epoch()
        activeEpochs.insert(currentEpoch)
    except call failure:
      add switch to P
      return false
  return true

function report_partition(P):
  broadcast list of hosts in P and activeEpochs
```

Figure 5—Logic for detecting, enforcing, and reporting partitions.

ceives an *end-host failure* or *link failure* event, it updates its model. If the model has a partition, the manager chooses an *excluded set P* of switches and hosts. *P* is chosen to be all switches and hosts outside the largest strongly connected component that is reachable by a majority of manager replicas. Ties are broken arbitrarily.

Before Albatross reports a problem to clients, the manager enforces the partition: it invokes CUT-APP or CUT-EPOCH for every switch port bordering *P*. The choice of primitive carries a trade-off. On the one hand, if the port bordering *P* connects to an end-host, then the manager uses CUT-APP: each end-host has a small set of applications, so enforcing a partition requires few per-application rules. On the other hand, if the port bordering *P* connects to another switch, then using CUT-APP would require a rule for each application whose traffic is carried by the switch—potentially hundreds of rules. Instead, the manager handles this case by excluding at coarser granularity: it uses CUT-EPOCH, which tells that switch, and only that switch, to exclude all applications of active epochs. While CUT-EPOCH compromises on surgical disconnection, we note, first, that applications that begin in a new epoch are not affected, since the use of CUT-EPOCH induces a change of epoch. Second, CUT-EPOCH is invoked only when a switch fails or is partitioned; the common case, end-host failures, is handled with CUT-APP.

If the call to CUT-APP or CUT-EPOCH fails, the manager adds the switch to *P* and continues.[5] If the manager cannot use the network interface to install rules at *any* switch, then Albatross may be unable to report some failures, but this case is rare and means that the whole network is likely unusable (§1.1).

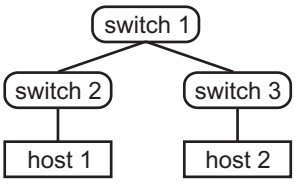
```
function handle_backstop_timeout(client, pid):
 for switch in topology such that switch is connected to host of pid:
  try: call BLOCK(switch, pid)
  except call failure:
   for link in switch:
    handle_failure_event(link)
 reply_to_client(client)
```

Figure 6—Logic to handle client backstop timeout on target *pid*.

When the procedure finishes, the manager broadcasts the list of hosts in *P* and the affected epochs. This information is received by the Albatross host modules, which mark the processes in *P* as down. The broadcast packets might be dropped; in that case, Albatross can still detect failures using the client's backstop timeout (see below)—albeit more slowly.

Host and process failures. Albatross treats a *host failure* as a one-host network failure, using the mechanism described immediately above. *Process* crashes, however, are handled differently; they separate into two cases. The first case is that a backstop timeout (§3) fires; this event causes the monitoring process to request help from the manager, which disconnects the target process, using BLOCK. Figure 6 shows the detailed logic. The second case is that a module running on the remote host is aware of a process crash; this case does not involve the manager at all and is covered in Section 6.3.

Example. Consider the example network below:



host 1 appids: 1, 2
host 2 appids: 2, 3
active epochs in network: 1, 3
appids of other hosts at switch 2: 4–47
appids of other hosts at switch 3: 13–63

If switch 3 reports an *end-host failure* event to the manager, then the manager will install at switch 3 a CUT-APP rule for appids 2 and 3. If switch 1 reports a *link-failure* event for its link to switch 3, or the manager suspects that switch 3 has failed (e.g., because switch 3 failed to install a CUT-APP rule), then the manager will install at switch 1 a CUT-EPOCH rule for epochs 1 and 3 and choose an inactive epoch as the current epoch. The manager use CUT-EPOCH instead of CUT-APP because CUT-EPOCH requires two invocations, whereas

[5] This failure can be detected via timeouts. These timeouts differ from *end-to-end* timeouts (§2.2) because, first, they are monitoring a constrained component, and, second, SDN management traffic can be prioritized, which would make message latencies predictable and thus avoid spurious timeouts.

CUT-APP would require fifty-three (the fifty-one ids at switch 3 plus appids 2 and 3); this choice is important because, as we will explain in Section 6.2, each invocation consumes scarce resources at the switch. This example ignores how failures might affect the manager; we describe the manager's fault tolerance next.

Fault tolerance. Recall that the manager is replicated for fault-tolerance (§3), following the state machine replication approach [45, 64], which is a majority-based technique for fault-tolerance (see Section 2.2). If manager replicas are placed at diverse parts of the network (such as different racks), then under common network partitions (described in Section 2.1), the majority of servers remains with the majority of network elements.[6]

Revisiting the guarantees. The formal guarantees (§4) reference a set E. This set is never materialized explicitly. Instead, recall that the manager maintains a set P, which is a set of (a) blocked processes, together with (b) a set of blocked applications, switches, and hosts (keyed by epoch). Because appids and epoch ids imply a set of processes, P implicitly represents the membership of E.

Albatross provides Completeness through the backstop timeout; if all else fails, a client will eventually request that the manager block the target (Figure 6). Albatross guarantees Isolation by configuring network switches to drop the traffic of excluded processes (Figure 5). Albatross guarantees Monotonicity because it never unblocks processes; it does recycle identifiers, however, as described in the next section. The first part of Correspondence is provided, also, by the backstop timeout; if a report from the manager is dropped, the client will eventually timeout and request blockage of the target (forcing the manager to retry its message). The second half of Correspondence is provided by the sequencing of events; the manager reports partitions only after enforcing them.

Albatross provides speed by reacting to failure events as opposed to end-to-end timeouts, in the common case. It limits interference by inspecting network state, by using surgical rules, and by allowing reconnection (described next).

5.4 Reconnecting processes and recycling identifiers

We now describe how Albatross reconnects processes and recycles epochs and pids. Although epochs change infrequently, they are important to recycle since, in our implementation (§6.1), there are only a handful of them, network-wide. Pids are scarce because the local id field—which identifies a process within a given application on a given host—is small.

Reconnecting processes. When a process tries to reconnect by calling ackDisconnect(), its host module gives it a new pid (§3,§4). Although Albatross's contract allows the host module to return any unallocated pid, in the interest of progress, the host module first checks that the new pid is not being blocked by any switch. To this end, before allocating a new pid, the host module asks the manager (a) what is the current epoch, and (b) which of the host's applications have been excluded (via CUT-APP). If no applications have been excluded, the host module returns a new pid with the current epoch and appid. If applications have been excluded, the host module locally blocks the processes belonging to those applications using a packet filter, asks the manager to undo the blanket exclusion (meaning, undo the CUT-APP calls at the edge switch), and only then returns the new pid. The order of these steps is important to upholding the Isolation property (§4); if the CUT-APP rules are undone before the excluded processes are blocked by their host module locally, then the excluded processes could affect non-excluded processes.

Garbage collecting epochs. Recall that when the manager enforces a partition of more than one end-host, it must activate a previously inactive epoch number. To allow the manager to track which epochs are active, host modules inform the manager which epochs they are using (by attaching a list to their normal messages to the manager). When the manager sees that an active epoch is not used by any host module, it undoes the CUT-EPOCH for the epoch, and marks it inactive.

Garbage collecting pids. A host module must be careful about when it reuses the pid of a process that has exited or has acknowledged a disconnection. Suppose, for example, that a host module were to give to a new process the pid of a process that had recently terminated; later, a third process could time out on the original terminated process, and have the manager enforce a partition using that pid, which would disrupt the new process. To avoid this and similar scenarios, Albatross includes the following counting scheme.

Each pid has a counter that is physically stored at the host module that allocated that pid (the local host); the counter tracks references to that pid held by other hosts. The local host module increments (or decrements) the counter when it hears that a remote process has started (or stopped) monitoring the associated process. A pid can be reused when these conditions all hold: (1) the pid of the process has reference count zero, (2) the local process has crashed or acknowledged the disconnection, and (3) the manager is not blocking the process's pid (with BLOCK).

The challenge in keeping the counter accurate is that there can be failures, both of clients referencing the pid and the host module storing the counter. To handle both cases, the local host module tracks, in a persistent write-ahead log, which clients have references; periodically, the local host module queries remote host modules to confirm that clients referencing its allocated pids are still running.

6 Selected implementation details

6.1 Packet marking

Figure 7 depicts the format of a pid. It consists of a 4-byte host identifier (currently, the host's IP address), together with 16 per-process bits. The per-process bits are the process's

[6] Even if the manager-majority partition holds a minority of processes, the minority can keep operating, under "$f+1$" algorithms (see footnote 4).

host id (IP addr) (32 bits)	epoch (3 bits)	appid (10 bits)	local id (3 bits)

Figure 7—Format of a Albatross process id (pid). Pids are six bytes; a process's pid appears in the source MAC address field of packets originated by the process. The number of epoch bits is small, but epochs are recycled (§5.4). The local id disambiguates multiple processes of the same application on the same host.

epoch number, the appid, and the local id.

Under Albatross, a process's pid appears in the source MAC address field of the packets that it originates. If a packet is sent by a process that is not using Albatross (including packets of ICMP, ARP, etc.), the bottom 16 bits are set to 0. Only the source MAC fields are used this way; the destination MAC field uses the usual MACs, obtained from ARP. This scheme assumes a scalable layer-two network in the data center (e.g., SEATTLE [41]).

The scheme has three features. First, it is easy to identify the traffic of applications that use Albatross—by observing a non-zero value in the bottom 16 bits. Second, blocking the traffic of a process at a switch requires a single rule (to match the source MAC); likewise, bit fields within the source MAC can be used to block the traffic of an entire application or epoch with one rule. Third, once the rule is installed, it need not be updated based on how and where the process sends data. By contrast, a scheme that blocked based on source TCP or UDP ports would require one rule per port used by the process, and updates in response to port changes. We discuss how this scheme affects existing Layer 2 protocols in Section 8.

6.2 Network interface implementation

Our implementation of Albatross assumes a network with OpenFlow switches and a NOX controller [33]. Given this environment, one can implement the network interface (Figure 4, §5.2) as follows. The CUT-APP(switch, appid, port) and CUT-EPOCH(switch, epoch, port) primitives direct the NOX controller to install an OpenFlow drop rule that matches on the appid or epoch bits of the pid; similarly, the BLOCK(switch, pid) primitive results in the installation of an OpenFlow drop rule that matches the entire pid.

SUBSCRIBE(destination) is implemented by augmenting the NOX controller to forward topology changes and failure events to the destination (which is the Albatross manager). Additionally, the destination needs to receive the link and end-host failure events (§5.2). *Link failure* events correspond to port- or link-down status events, and OpenFlow switches (by nature) notify the controller of such events. The controller simply forwards these notifications to the destination.

The more difficult case is *end-host failure* events. These are not directly supported by OpenFlow, so our implementation must synthesize them. Our solution leverages *SDN rule timeouts*, as follows. Each host module sends a special heartbeat packet to its switch every $T_{heartbeat}$ time units. On the first heartbeat, the switch sends an *unknown packet* event to the SDN controller. The SDN controller then configures the switch to (a) drop these heartbeat packets, and (b) send a timeout notification if the rule is not used for $T_{net\text{-}check}$ time units. If the controller receives such a notification, it sends an end-host failure event to the destination (the manager). Our implementation sets $T_{heartbeat}$ to 10 ms and $T_{net\text{-}check}$ to 1 s (the smallest OpenFlow timeout), which provide reasonably fast detection while tolerating dropped or delayed heartbeats.

6.3 Detecting process crashes

As noted in Section 5.3, process crash detection involves an additional module. This module mostly reuses prior work; we cover it for completeness. The core logic is a modified Falcon spy [50]. A Falcon spy uses local information (e.g., an OS's process table) to detect process crashes and report them to clients monitoring the crashed (target) processes. In particular, the Falcon spy inspects a process's internal state by invoking an application-specific function over IPC; this mechanism detects problems that may be hidden externally, such as deadlock. The modification from Falcon is that, instead of terminating unresponsive processes, the spy drops their traffic locally with a packet filter (using iptables). This modification reduces the impact of a false suspicion.

6.4 Miscellaneous implementation details

- Each host module caches the status of monitored target processes: when a client's host module receives a notification from a target's host module or from the manager, the client's module invokes the relevant callback function (Figure 2) and stores the "disconnected" for future queries.
- Albatross's manager is separate from the SDN controller. The manager's solution to replication makes use of a library [54]. (§8 discusses SDN controller fault-tolerance.)
- A final detail is interprocess communication (IPC). Albatross must enforce Isolation even when processes are on the same host. Thus, Albatross requires that all IPC be sent through the host's top-of-rack switch. If this requirement is burdensome (e.g., if processes use IPC extensively), two local processes can share the same Albatross pid, the tradeoff being that Albatross treats such processes as a unit.

7 Evaluation of Albatross

Using empirics and experiments, our evaluation reprises the arguments in Sections 2.2 and 2.3. First, we explain the benefits of a membership service that provides definitive reports, showing in the process that Albatross's contract is sufficient to derive these benefits (§7.1). Second, we investigate how well Albatross does the job of providing this contract (§7.2).

All experiments run on a prototype network (with 13 switches connected in a complete ternary tree) implemented using QEMU/KVM [62] virtual machines (version 1.0.1) and CPqD OpenFlow 1.2 software switches [60]. The hypervisor is a 64-core Dell PowerEdge R815 with AMD Opteron Processors and 128 GB of memory, running Linux (kernel ver-

sion 3.7.10-gentoo-r1). The network controller is NOX [33], modified to work with OpenFlow 1.2 [56].

7.1 What are the benefits of Albatross's contract?

On the one hand, the fact that membership services simplify the design of the distributed applications that use them has long been established: the fail-stop model (which assumes that all processes can detect all crashes correctly) is known to enable "easier" algorithms than the crash model. As just one example, Chain Replication [69] (a form of primary-backup) is simpler than Viewstamped Replication [57], Paxos-based replication [46], and Raft [58].

On the other hand, Albatross's contract (§4), with its asymmetric guarantees, is not precisely the fail-stop model. Thus, this section investigates whether Albatross's contract is sufficient to provide the same qualitative benefits. We do this by illustrating what can go wrong without a membership service; demonstrating that Albatross's guarantees are sufficient to simplify distributed algorithms; and describing the subtle relationships among Albatross, ZooKeeper (as an alternate membership service), and majority-based agreement.

Without a membership service, what can go wrong? We use RAMCloud [59] as a short case study. RAMCloud is a storage system that keeps data in memory at a set of *master servers*. These servers also process client requests to read and write data. For durability, a master server writes copies of data on the disks of multiple *backup servers*. A *coordinator* manages the configuration of the servers (which servers are masters for what data, etc.). To avoid losing writes or reading stale data, RAMCloud must guarantee that exactly one master server is responsible for a piece of data. One way to do this would be to use a membership service, but RAMCloud instead[7] uses several mechanisms internally: short timeouts, self-killing, propagation of crash information, and coordination among backups. These mechanisms must be orchestrated carefully to handle corner cases.

We first determine if RAMCloud ever returns stale (incorrect) data. We inject network failures at times carefully chosen to trigger the following corner case: a master is transiently disconnected from the coordinator, causing the coordinator to initiate the master's recovery. We find that RAMCloud can indeed return incorrect data; this bug was observed empirically and confirmed by the RAMCloud developers. Specifically, RAMCloud detects failures using a short timeout of hundreds of milliseconds; if the coordinator times out on a master, the coordinator starts data recovery, which is very fast. Because the timeout is short and recovery is fast, the entire process may complete before the old master realizes that it was replaced, resulting in two masters: a split-brain scenario. Intuitively, the issue is that RAMCloud does not make its suspicion of failure definitive (e.g., by waiting for the old master to shut down) before acting on that suspicion.

[7] RAMCloud uses ZooKeeper but only for coordinator failures [59, §3.10].

Algorithm	**Aab (§7.1)**	**Zab [39]**
Description length	half page	~3 pages
# phases	2	3
# roundtrips on recovery	2	3
# message types	3	9
# timestamps/counters	1	2
At most one leader?	yes	no
failures (f) tolerated relative to total (n)	$f<n$	$f<n/2$

Figure 8—Comparison of atomic broadcast with and without definitive reports. Aab uses Albatross (and would be similar if it used any other membership service), and Zab [39] uses majority-based agreement; both algorithms are described in the text.

We replaced RAMCloud's failure detector with Albatross (65 lines of C++) and found that RAMCloud then worked correctly: when the master is reported as "disconnected", it is excluded and cannot serve clients, by Correspondence and Isolation (§4). This benefit is not unique to Albatross; other membership services would eliminate this error too.

How do Albatross's guarantees simplify algorithm design? As another case study, we examine *atomic broadcast*: it is a building block of many distributed systems, and it has solutions with and without definitive reports [21]. We specifically compare (a) *Zab* [39], a protocol that uses majority-based agreement (as opposed to definitive reports), and (b) *Aab*, a protocol that uses Albatross.

Zab: atomic broadcast without definitive reports. Zab [39] takes a standard approach, which we briefly summarize here. A leader orders messages. Because partitions can result in multiple leaders (one leader becomes disconnected, another leader is elected, and the original reconnects), the protocol relies on a majority (quorum) of processes to approve leader actions. As a result, if two leaders try to act, only one succeeds in getting approval from a majority.

Aab: atomic broadcast under Albatross. Under Albatross, processes can select a unique leader by picking the smallest process id among processes that Albatross considers to be "connected". This scheme works because, if there could be two non-excluded leaders at the same time, let p be the one with higher id; then p considers the other leader as "disconnected", otherwise it would not have picked itself as leader. Thus, by Correspondence (§4), the other leader is excluded—contradiction. Thus, we have essentially unique leaders. We say "essentially" because there could be many self-styled leaders; however, all but one will be excluded.

Given (essentially) unique leaders, we can implement atomic broadcast using a sequencer-based algorithm [21], adapted to use Albatross. The algorithm proceeds in periods; each period has a unique leader (chosen as described above). In each period, a process that wants to broadcast a message sends it to the leader and waits for an acknowledgment; if the leader changes, the process resends to the new leader (in a new period). The leader handles each period in two phases, recovery and order. In the recovery phase, the leader completes the broadcast of pending messages from prior periods (if any).

where injected?	what failure is injected?	what does the failure model?
network	link failure	network partition
network	switch failure	network partition
network	misconfiguration that causes a partition	operator error
network	host floods UDP traffic	sudden traffic spike
network	dropped OpenFlow messages	problems in the SDN
network	spurious failure event	link flapping
end-host	process crash (segfault)	problem in the application
end-host	host crash (kernel panic)	machine crash or reboot
Albatross	crash of host module	bug in host module
Albatross	crash of leader in manager	bug in manager

Figure 9—Panel of synthetic failures. We inject failures in the network, at the end hosts, and into Albatross itself.

In the order phase, the leader serves as a sequencer: it gets a new message to broadcast, assigns it a sequence number, and sends it to processes for delivery. Processes then deliver the messages in sequence number order.

Comparison. Figure 8 compares the two algorithms. Aab has a smaller description, fewer phases, fewer round-trips, fewer message types, and fewer counters for ordering messages. Moreover, it tolerates the failure of all but one process; Zab, by contrast, tolerates the failure of fewer than half of the processes. (Equivalently, to tolerate f failures, the Albatross-based Aab requires $f + 1$ processes, whereas Zab requires $2f+1$ processes.) The fundamental source of these differences is that Zab is built on majority-based agreement, which brings complexity, as noted earlier (§2.2).

Albatross vs. ZooKeeper vs. consensus vs. atomic broadcast. The preceding comparison immediately raises a question. Namely, *Albatross also uses majority-based techniques internally*—in fact, the consensus-based algorithm for replicating the manager (§3, §5.3) has the same qualitative complexity as Zab. So why is this fact omitted in the Aab-vs-Zab comparison? Because under Albatross, the complexity is localized to the manager, and handled once; the clients of Albatross are not exposed to the complexity, and the additional resource cost is amortized over all clients of Albatross.

But can't the same kind of amortization work for Zab? Yes and no. On the one hand, ZooKeeper's lease server abstraction is built on Zab (Zab stands for "ZooKeeper atomic broadcast"; Zab is used to order commands to a replicated lease server state machine), and the intent is that many different applications can be clients of ZooKeeper's lease server. On the other hand, ZooKeeper cannot achieve the same performance under the same number of clients as Albatross. The reason is that short leases require frequent polling, which can overwhelm a server with many clients; this is demonstrated in Section 7.2.

Can ZooKeeper be modified to use Albatross? Yes. ZooKeeper is built on an atomic broadcast interface, which is implemented by Zab (as noted above). We could replace Zab with Aab. However, the resulting system would inherit the disadvantages of leases (§2.2, §2.3).

failure type	action taken by Albatross
link failure	the network interface reports link failures (§5.2); the mgr. detects a partition, enforces an excluded set, and reports it to clients (Figure 5)
switch failure	ditto
network misconfiguration	detected by client timeout (§3) and enforced by the manager's backstop logic (Figure 6)
network flooding	no failure is detected
dropped OpenFlow messages	detected by an OpenFlow timeout in the SDN controller; the controller treats this as a switch failure and reports it to the manager as multiple link failure events (Figure 5)
spurious failure event	handled as a link failure (see above)
host crash	detected with an end-host failure event (§5.2), and enforced by the manager (Figure 5)
process crash	Falcon spy [50] detects and reports failure (§6.3)
crash of host module	detected by backstop timeout (§3) and enforced by the manager (§5.3)
crash of manager replica + partition	replication library (§6.4) elects a new leader (§5.3), then the manager handles the failure as above

Figure 10—Albatross's reaction to the failure panel (Figure 9). Albatross detects all failures save network flooding (a non-failure), and its enforcement actions affect only applications that use it.

Can Albatross be built atop ZooKeeper? Yes. Albatross could replicate its manager using ZooKeeper's lease servers (or Zab directly). This represents an alternative instantiation of Albatross; it is essentially equivalent to the one covered in the rest of this paper.

7.2 Is Albatross a good membership service?

We now experimentally investigate the qualities of Albatross: (a) how it responds to failures, (b) how its timeliness compares with two baseline mechanisms, (c) how well it limits interference, and (d) what system resources it uses. The experiments use a panel of synthetic failures, depicted in Figure 9. These failures model problems in the network, at end-hosts, and in Albatross itself; link and switch failures are derived from our failure analysis (§2.1). While deploying Albatross on physical hardware and measuring its response to failures in the wild would be better than a synthetic evaluation, this is beyond our scope, as we currently seek a more basic understanding of how Albatross performs. Thus, this evaluation should be read as suggestive rather than conclusive.

How does Albatross respond to failures? We run an experiment where a *client* process monitors a remote *target* process; we inject a failure of some chosen type, affecting the target process, and we record Albatross's response. We repeat the experiment 25 times for each failure type.

We find that Albatross reacts the same way in the 25 repetitions for a given failure type; the reactions for each failure type are in Figure 10.

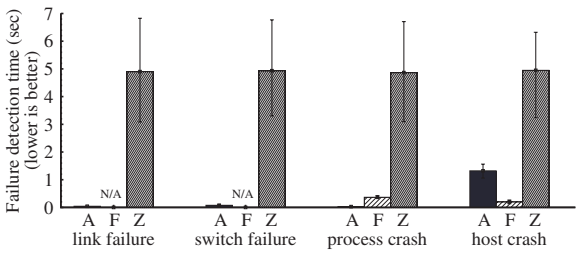


Figure 11—Detection time and coverage of Albatross (**A**), compared to Falcon [50] (**F**) and ZooKeeper [36] (**Z**). Error bars are minimum and maximum observed detection times. Two of Falcon's bars are labeled N/A because it does not detect link or switch failures. Albatross detects failures quickly by using information at end-hosts and in the network. ZooKeeper's detection time reflects its timeout (4s); a shorter one causes overload (see text).

How does Albatross compare with baseline mechanisms? We take as baselines (1) ZooKeeper [36], and (2) Falcon [50], both of which are described in Section 2.[8] For each failure that Albatross detects (without using backstop timeouts), we repeat the aforementioned experiments 100 times and measure the *detection time* from when the failure occurs to when it is reported to the application. We experiment with Albatross and with ZooKeeper. For Falcon, we report published results [50] (because two of Falcon's spies are incompatible with the testbed used for Albatross).

Figure 11 shows the results. Network problems are detected by Albatross quickly, usually in less than a second. The specifics of these numbers depend on the implementation of our testbed's switches, which are software; deploying Albatross on real hardware may have different performance characteristics, though we expect the order of magnitude will be similar.

Process failures are detected by Falcon and Albatross quickly; Albatross is faster than Falcon here (even though Albatross uses a Falcon spy to detect these failures) because Falcon's published results include a delay for confirming that a process has left the process table whereas Albatross needs only to install an iptables rule (§6.3).

On host failures (e.g., kernel panics), Albatross takes 1s longer than Falcon; the difference is that Albatross detects host failures using OpenFlow rule timeouts (§6.2), which have a minimum duration of 1s. Unlike Albatross, Falcon cannot detect switch or link failures [50].

ZooKeeper's detection speed reflects its timeout, which we configure to be 4 seconds, as suggested in its tutorial [1]. This choice is not arbitrary: if one lowers ZooKeeper's timeout to match Albatross's detection speed, ZooKeeper would be overloaded by keep-alives. To establish this, we experiment with ZooKeeper. We find, first, that ZooKeeper can monitor 1500 targets, each using a 4s timeout on their leases. But when we reduce the timeout to 500 ms, ZooKeeper drops the connections of about 70 targets, even though the network is not saturated. We believe this effect is similar to Burrows's observations [13]: timeouts shorter than 12s overwhelmed Chubby's servers in Google's clusters (which monitors many more targets). In contrast, we find that Albatross's manager can monitor over 1500 targets. Essentially, ZooKeeper polls clients with ping messages whereas Albatross watches for the *causes* of dropped pings (crashes, partitions, etc.), and can thus react quickly.

[8] We do not use Pigeon [49] as a baseline because it lacks definitive reports for network failures and because it targets Layer 3 networks.

max additional rules installed at a switch	
rules installed	1 rule
CPU *usage per component (§5)*	
host module	1.8 %
manager	0.03 %
bandwidth used	
at each end-host	61.0 kBps
at manager	6.9 kBps

Figure 12—Summary of Albatross's costs under link failure. Albatross uses few resources. Scalability is discussed in the text.

How well does Albatross limit interference? We evaluate whether some common network behaviors might cause Albatross to disconnect processes without cause. We inject two non-failures into our testbed: (a) heavy traffic (modeling congestion) and (b) a spurious link failure event (modeling link flapping), for a link whose removal splits the network. We observe that Albatross does not disconnect processes under heavy traffic. Albatross does not detect a problem because the duration of the spike in traffic is less than the client process's end-to-end timeout. Albatross does disconnect under the spurious failure. While this behavior is not ideal, it is not disastrous because, first, a known down link may be better than persistent link flapping; second, Albatross does not interfere if there are alternate paths or the link is not used by Albatross processes; and third, applications can reconnect (§5.4). Reconnection takes about one second in our experiments.

What are Albatross's costs? We measure Albatross's resource cost for detecting and enforcing a partition for a single application. Figure 12 shows the results. As expected, Albatross installs one rule per application id before reporting the target process as "disconnected".

We must also consider what happens when there are more applications and hosts. In general, the number of rules grows with the number of disconnected processes; for example, a switch with 40 disconnected end-hosts, each with 20 distinct applications, would have 800 rules. On the one hand, numbers like these are acceptable: the HP ProCurve J9451A switch, for example, has capacity of 1500 OpenFlow rules [35]. On the other hand, the linear in-network costs could become undesirable. In that case, Albatross could reduce the number of rules that it uses; on links that connect to end-hosts, it could block at the granularity of epochs instead of appids (§5.3), at the cost of possibly blocking additional processes.

Albatross uses few resources at the manager replicas in terms of CPU and network bandwidth. Albatross's cost at

end-hosts is higher, as the host module generates heartbeat packets (§6.2). However, the effect is local: these packets are dropped by a host's switch before entering the network.

Albatross is implemented with 4044 lines of C++ code.

8 Discussion and future work

Does Albatross have all the features of existing membership services? While Albatross implements the basics of a membership service, there are other features of existing membership services that Albatross does not implement, including metadata storage [13, 36], message ordering [11, 16], and access control [36]. However, Albatross can be used within existing services to help reduce both failure detection time (§7.2) and implementation complexity (§7.1).

Does Albatross require SDNs? While the current implementation of Albatross uses OpenFlow, Albatross requires relatively few things from the network: the ability to receive failure events and to block traffic based on packet fields. These requirements are made explicit by the network interface (§5.2), and Albatross can work in any network where this interface can be implemented.

Is the SDN controller a single point of failure? This issue is mostly orthogonal to Albatross. Albatross currently uses NOX, which is centralized and thus a single point of failure. However, Albatross could instead use recent fault-tolerant controllers (see Section 9).

Must Albatross repurpose the source MAC field? Albatross's embedding of process identifiers in packets' source MAC field (§6.1) is not fundamental. Albatross could use other space in packets: MPLS labels, a shim layer for Albatross, bits in an RPC header, etc. The only requirement is that switches can filter packets based on these fields.

How does Albatross's MAC rewriting scheme affect existing Layer 2 protocols? Under existing Layer 2 protocols, such as IEEE 802.1d [37], switches will use the source MAC addresses of incoming packets to learn the mapping between MAC addresses and output ports, for future forwarding decisions. Since Albatross's MAC-rewriting scheme creates source addresses that will never be used as destination addresses (§6.1), a Layer 2 protocol deployed alongside Albatross should be modified to never learn from these packets. Fortunately, Albatross works in the context of SDNs, so many Layer 2 protocol changes would require only software changes at the SDN controller.

Can Albatross work across data centers? One challenge is that wide-area delays will worsen detection time when the monitoring and target processes are far apart. Another challenge is finding reasonable guarantees for Albatross to provide when the data centers are partitioned. Addressing these challenges is future work.

Can Albatross work with virtual machine migration? Albatross assumes that processes and end-hosts remain stationary, which conflicts with virtual machine migration [17]. This issue is surmountable, if the manager and migration mechanism collaborate to migrate filter rules. This, too, is future work.

Does Albatross consider network policy? Albatross models only the *physical* network topology (§5.3). Yet policies (e.g., ACLs) can constrain communication. The Albatross manager might thus be unable to detect unreachability: it might think a path exists, when in reality it is prohibited. This problem would be handled by the client's backstop timeout (§3, §5.3). Using policy information in Albatross is future work.

Can cooperating applications have inconsistent views of the network? As mentioned in Section 5.3, one can think of Albatross as virtualizing partitions. A natural question is whether the discrepancies in the views of applications create problems when applications communicate. The answer is no. Albatross guarantees that, if a process is partitioned away, it is partitioned for all applications.

What are the security implications of Albatross? Processes can block any Albatross-enabled process by starting and never canceling an end-to-end timeout (Figure 2). Adding access control to Albatross's API is future work.

Does Albatross contradict the end-to-end argument? No, because Albatross's guarantees are (a) about the state of the network and (b) require help from the network, two cases that fall outside the end-to-end argument's jurisdiction.

9 Related work

Albatross leverages software-defined networking [33, 61] (as described earlier). Albatross also builds on the distributed systems literature, borrows from the networking literature, and relates to work at their intersection.

Distributed systems. Earlier (§2.2), we walked through distributed systems solutions that relate to Albatross generally. Here, we delve into closely related approaches.

Albatross can be seen as a type of failure detector, a service that indicates the operational status of remote processes. This service has a well-developed theory (e.g., [14]), including some extensions for network partitions [3, 5], and a well-developed practice [9, 15, 34, 67], based on timeouts (§2.2). Unlike Albatross, this work does not leverage information and mechanisms in the network.

Falcon [50] and Pigeon [49] are failure detectors that, like Albatross, rely on local (or inside) information. Albatross has some debts to Falcon, most notably the spy module to detect process crashes (§6.3). A minor difference is that Albatross better limits interference (Falcon kills end-hosts, whereas Albatross merely disconnects processes). The major innovations over Falcon are: Albatross's handling of network failures (Falcon handles only process failures; it hangs if there are partitions), its precisely articulated contract, its design (process naming, fine-grained dropping using SDNs, etc.), and using SDNs to enhance classical distributed systems.

Like Albatross, Pigeon [49] is designed to handle network failures; unlike Albatross, Pigeon does not leverage SDNs. Another major difference is that Pigeon does not report failures definitively: under network problems, its interface pro-

vides only hints (§2.2). We built a membership service atop Pigeon, in the form of a lease server [49, §5.2]. Under host failures, inside information allows quick lease breakage; under *network* failures, however, this membership service waits for the lease to expire, as in ZooKeeper. One might be able to build a better membership service atop Pigeon; we plan to investigate this in future work. But even if such an improvement is possible, it would represent a different design point from our work here. Among other things, we expect the response times of this membership service to be longer (since it would achieve definitiveness via agreement as opposed to killing).

Various systems introduce abstractions that can be used in place of a failure detector. For example, cluster management services (ZooKeeper, Chubby, etc.) can be used to assist primary-backup replication, as explained in Section 2.3. Another example is FUSE [25], which reports the failures of process groups in overlay networks. Unlike Albatross, FUSE has symmetric guarantees for failure notification. However, the semantics of these reports cannot be used to implement primary-backup (since FUSE may report failure even if the primary and the backup are both up). One body of work that deserves special mention is *group communication services* [11, 16] (GCS); these maintain a *view* of processes and provide multicast within a view. GCS have an exclusion concept similar to Albatross, but the mechanisms are different: a GCS uses timeouts and distributed algorithms (§2.2) while Albatross observes and modifies the network.

Another related system is Fault-Tolerant CORBA (FT-CORBA) [2]. FT-CORBA has a hierarchical monitoring scheme, which is reminiscent of how Albatross implements end-host failure events (§6.2); FT-CORBA also includes object-specific monitoring functions, which are similar to the spy that Albatross borrows from Falcon (§6.3). However, Albatross and FT-CORBA have different goals and mechanisms: Albatross provides a membership service by leveraging SDN whereas FT-CORBA replicates objects.

Networking. The general area of resilience in networking has received much research attention. This work is largely orthogonal to Albatross, as Albatross concerns how to reliably report partitions to applications. However, some of this work has a similar ethos to Albatross. For instance, NetPilot [72] seeks to turn partial failures into total failures (called *failure mitigation*). Nevertheless, the two systems take different approaches, in the service of complementary goals: NetPilot restarts switches and ports, to mitigate network failures, whereas Albatross blocks traffic to make its reports accurate, when such failures do occur.

A large body of work (far more than we can cover in depth, but see [7, 8, 18, 19, 22, 31, 42, 43, 52, 53, 66, 71, 75, 76]) is concerned with extracting intelligence about the network, and reporting it to operators or applications. Although some of Albatross's elements are reminiscent of some of this work, generally the goals are different. For instance, some work [65, 70] monitors routing state to track topology (as does the Albatross manager), but their goal is different: analysis and diagnosis for researchers and operators. In Network Exception Handlers (NEH) [40], the network notifies hosts of state changes, but the purpose is end-host participation in traffic engineering. Like Albatross, Packet Obituaries (POs) [6] report network failure information to end-hosts, but POs do so at a different level of abstraction (which packets were dropped versus which processes are reachable).

Distributed systems and networking. Research that combines distributed systems and networking tends to apply distributed systems techniques to make better networks (whereas Albatross works the other way around).

Consistent networking aims to keep the network in a valid state at all times, under configuration changes. For instance, consensus routing [38] uses Paxos [46] to apply updates to BGP routers to avoid black holes and loops. More recent work has examined primitives for consistent updates to OpenFlow networks, to preserve routing state [63] and bandwidth guarantees [28]. These goals are different from Albatross's.

Distributed SDN. Some researchers have used techniques from distributed systems to replicate and distribute the SDN controller [20, 23, 24, 44, 68, 73]. This work would complement Albatross, as noted in Section 8.

10 Conclusion

> For all averred, I had killed the bird
>
> —"The Rime of the Ancient Mariner", Samuel Taylor Coleridge

Albatross leverages software defined networks to give distributed applications reports about the reachability of their component processes. These reports are timely and definitive, and cover network problems—a combination that we believe is new in membership services. Furthermore, Albatross as built handles failures at end-hosts, making it a complete solution for failure detection in a data center environment.

Albatross also brings up a potential direction for future work. A number of projects have applied distributed systems techniques to obtain better networks. However, we think that Albatross is the first to apply modern networking techniques to refine the guarantees of distributed systems. Perhaps there are other opportunities along these lines.

Acknowledgments

Youngjin Kwon helped implement an early prototype of Albatross. Many people helped to improve this work, including Sebastian Angel, Mahesh Balakrishnan, Manos Kapritsos, Jeff Mogul, John Ousterhout, Rama Ramasubramanian, Ryan Stutsman, Yang Wang, Emmett Witchel, and Edmund L. Wong. The presentation of this paper was greatly improved by the careful comments of Dejan Kostić, Jinyang Li, Scott Shenker, Riad Wahby, and the anonymous reviewers of EuroSys, OSDI, and SIGCOMM. The research was supported in part by NSF grants CNS-1055057, CNS-1040083, and CCF-1048269.

References

[1] `http://zookeeper.apache.org/doc/current/zookeeperStarted.html`.

[2] Faut tolerant CORBA. OMG Specification formal/2010-05-07, Object Management Group, 2010.

[3] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999.

[4] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *International Conference on Software Engineering (ICSE)*, pages 562–570, 1976.

[5] L. Arantes, P. Sens, G. Thomas, D. Conan, and L. Lim. Partition participant detector with dynamic paths in mobile networks. In *IEEE International Symposium on Network Computing and Applications (NCA)*, pages 224–228, July 2010.

[6] K. Argyraki, P. Maniatis, D. Cheriton, and S. Shenker. Providing packet obituaries. In *ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2004.

[7] H. Ballani and P. Francis. Fault management using the CONMan abstraction. In *INFOCOM*, Apr. 2009.

[8] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 335–348, Apr. 2009.

[9] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *International Conference on Dependable Systems and Networks (DSN)*, pages 354–363, June 2002.

[10] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):272–314, Aug. 1991.

[11] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 123–138, Nov. 1987.

[12] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 141–154, Apr. 2011.

[13] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350, Dec. 2006.

[14] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[15] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580, May 2002.

[16] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, Dec. 2001.

[17] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, May 2005.

[18] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski. A knowledge plane for the Internet. In *ACM SIGCOMM*, pages 3–10, Aug. 2003.

[19] E. Cooke, R. Mortier, A. Donnelly, P. Barham, and R. Isaacs. Reclaiming network-wide visibility using ubiquitous end system monitors. In *USENIX Annual Technical Conference*, June 2006.

[20] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *ACM SIGCOMM*, pages 254–265, Aug. 2011.

[21] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, Dec. 2004.

[22] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot. Troubleshooting network unreachabilities using end-to-end probes and routing data. In *ACM Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2007.

[23] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed SDN controller. In *SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 7–12, Aug. 2013.

[24] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy. ETTM: A scalable fault tolerant network manager. In *Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2011.

[25] J. Dunagan, N. J. A. Harvey, M. B. Jones, D. Kostić, M. Theimer, and A. Wollman. FUSE: Lightweight guaranteed distributed failure notification. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 151–166, Dec. 2004.

[26] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112, Feb. 2003.

[27] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[28] S. Ghorbani and M. Caesar. Walk the line: Consistent network updates with bandwidth guarantees. In *SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 67–72, Aug. 2012.

[29] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):48–51, June 2002.

[30] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *ACM SIGCOMM*, pages 350–361, Aug. 2011.

[31] S. Goldberg, D. Xiao, E. Tromer, B. Barak, and J. Rexford. Path-quality monitoring in the presence of adversaries. In *SIGMETRICS*, pages 193–204, June 2008.

[32] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–210, Dec. 1989.

[33] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an operating system for networks. *ACM Computer Communications Review (CCR)*, 38(3):105–110, July 2008.

[34] N. Hayashibara, X. Défago, R. Yared, and T. Katayama. The ϕ accrual failure detector. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 66–78, Oct. 2004.

[35] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 43–48, Aug. 2013.

[36] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX Annual Technical Conference*, pages 145–158, June 2010.

[37] Standard for local an metropolitan area networks: media access control (MAC) bridges. IEEE Standard 802.1d, Institute of Electrical and Electronics Engineers, 2004.

[38] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. Consensus routing: The Internet as a distributed system. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 351–364, Apr. 2008.

[39] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *International Conference on Dependable Systems and Networks*

(DSN), pages 245–256, June 2011.

[40] T. Karagiannis, R. Mortier, and A. Rowstron. Network exception handlers: Host-network control in enterprise networks. In *ACM SIGCOMM*, pages 123–134, Aug. 2008.

[41] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: a scalable Ethernet architecture for large enterprises. In *ACM SIGCOMM*, pages 3–14, Aug. 2008.

[42] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP fault localization via risk modeling. In *Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[43] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and localization of network black holes. In *INFOCOM*, pages 2180–2188, May 2007.

[44] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 351–364, Oct. 2010.

[45] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978.

[46] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[47] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 312–313, Aug. 2009.

[48] E. K. Lee and C. Thekkath. Petal: Distributed virtual disks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 84–92, Dec. 1996.

[49] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Improving availability in distributed systems with failure informers. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 427–442, Apr. 2013.

[50] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the FALCON spy network. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 279–294, Oct. 2011.

[51] Linux-HA, High-Availability software for Linux. `http://www.linux-ha.org`.

[52] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 367–380, Nov. 2006.

[53] H. V. Madhyastha, E. Katz-Bassett, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane Nano: path prediction for peer-to-peer applications. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 137–152, Apr. 2009.

[54] D. Mazières. Paxos made practical. `http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`, as of Sept. 2011.

[55] T. Muraus. Service registry behind the scenes why we built it. `http://www.rackspace.com/blog/service-registry-behind-the-scenes-why-we-built-it`, Nov. 2012.

[56] NOX Zaku with OpenFlow 1.2 support. `http://github.com/CPqD/nox12oflib`.

[57] B. M. Oki and B. Liskov. Viewstamped replication: A general primary copy. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 8–17, Aug. 1988.

[58] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–309, June 2014.

[59] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–41, Oct. 2011.

[60] OpenFlow 1.2 Software Switch. `http://github.com/CPqD/of12softswitch`.

[61] Openflow. `http://www.openflow.org/`.

[62] Kernel based virtual machine. `http://www.linux-kvm.org/`.

[63] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *ACM SIGCOMM*, pages 323–334, Aug. 2012.

[64] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[65] A. Shaikh and A. Greenberg. OSPF monitoring: Architecture, design, and deployment experience. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 57–70, Mar. 2004.

[66] A. Shieh, E. G. Sirer, and F. B. Schneider. NetQuery: A knowledge plane for reasoning about network properties. In *ACM SIGCOMM*, pages 278–289, Aug. 2011.

[67] K. So and E. G. Sirer. Latency and bandwidth-minimizing failure detectors. In *European Conference on Computer Systems (EuroSys)*, pages 89–99, Mar. 2007.

[68] A. Tootoonchian and Y. Ganjali. HyperFlow: A distributed control plane for OpenFlow. In *Internet Network Management Workshop / Workshop on Research on Enterprise Networking (INM/WREN)*, Apr. 2010.

[69] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–104, Dec. 2004.

[70] D. Watson, F. Jahanian, and C. Labovitz. Experiences with monitoring OSPF on a regional service provider network. In *International Conf. on Distributed Computing Systems (ICDCS)*, pages 204–213, May 2003.

[71] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2003.

[72] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. NetPilot: Automating datacenter network failure mitigation. In *ACM SIGCOMM*, pages 419–430, Aug. 2012.

[73] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *ACM SIGCOMM*, pages 351–362, Aug. 2010.

[74] L. Zhang. Why TCP timers don't work well. In *ACM SIGCOMM*, pages 397–405, Aug. 1986.

[75] M. Zhang, C. Zhang, V. Pai, L. Peterson, and R. Wang. PlanetSeer: Internet path failure monitoring and characterization in wide-area services. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 167–182, Dec. 2004.

[76] Y. Zhao, Y. Chen, and D. Bindel. Towards unbiased end-to-end network diagnosis. In *ACM SIGCOMM*, pages 219–230, Sept. 2006.

PERFORMANCE

– Performance Analysis of Database Virtualization with TCP-VMS Benchmark

Performance analysis of database virtualization with the TPC-VMS benchmark

Eric Deehr[1], Wen-Qi Fang[1], H. Reza Taheri[2,] and Hai-Fang Yun[1]

[1]Hewlett-Packard, Inc.; [2]VMware, Inc.
{eric.deehr, anton.fang hai-fang.yun}@hp.com, rtaheri@vmware.com

Abstract. TPC-VMS is a benchmark designed to measure the performance of virtualized databases using existing, time-tested TPC workloads. In this paper, we will present our experience in using the TPC-E workload under the TPC-VMS rules to measure the performance of 3 OLTP databases consolidated onto a single server. We will describe the tuning steps employed to more than double the performance and reach 98.6% of the performance of a non-virtualized server – if we aggregate the throughputs of the 3 VMs for quantifying the tuning process. The paper will detail lessons learned in optimizing performance by tuning the application, the database manager, the guest operating system, the hypervisor, and the hardware on both AMD and Intel processors.
Since TPC-E results have been disclosed with non-virtualized databases on both platforms, we can analyze the performance overheads of virtualization for database workloads. With a native-virtual performance gap of just a few percentage points, we will show that virtualized servers make excellent platforms for the most demanding database workloads.

Keywords: Database performance; virtualization; SQL Server; workload consolidation; performance tuning; cloud computing

1 Introduction

Server virtualization is a cornerstone of cloud computing and has fundamentally changed the way computing resources are provisioned, accessed and managed within a data center. To understand this new environment, customers need a tool to measure the effectiveness of a virtualization platform in consolidating multiple applications onto one server. According to IDC [5], in 2014, 32% of the new server shipments are deployed as virtualized servers, running not only the light weighted applications, but also CPU-intensive and I/O-intensive workloads, such as OLTP applications. As a result, there exists a strong demand for a benchmark that can compare the performance of virtualized servers running database workloads. TPC-VMS is the first benchmark that addresses this need by measuring database performance in a virtualized environment.

With all the benefit of virtualization technology, comes a price – the overhead of server virtualization. Vendors and customers need to have effective ways of measuring

adfa, p. 1, 2011.

this overhead, as well as characterizing the applications running in virtualized environments. Although TPC rules prohibit comparing TPC-VMS results with results of other TPC benchmarks on native servers for competitive or commercial purposes, we are able to make that comparison in an academic paper to quantify the overhead database workloads experience when the deployment platform is a virtualized server.

HP, in partnership with VMware, published the first two TPC-VMS results. In this paper, we will share our experience in running and tuning the TPC-E workload under the TPC-VMS rules to measure the performance of 3 OLTP databases consolidated to a single server for both Intel and AMD platforms. We will analyze the performance overhead of the virtualized database using native TPC-E result as the reference point, and will provide insights into optimizing large databases in a virtualized environment.

The paper will start with a short introduction to the TPC-VMS benchmark, present published results, describe the tuning process, share the lessons learned, and conclude with characterizing the overheard of server virtualization using the benchmark results.

1.1 TPC-VMS benchmark

In responding to the need for a standardized benchmark that measures the performance of virtualized databases, TPC introduced TPC Virtual Measurement Single System Specification, TPC-VMS [13], in December 2012. TPC-VMS models *server consolidation*, which is the simplest, oldest use of virtualization: a single, virtualized server consolidating 3 workloads that could have been deployed on 3 smaller, or older, physical servers.

TPC-VMS provided a methodology for test sponsors to use four existing TPC benchmarks, TPC-C [9], TPC-E [10], TPC-H [11] or TPC-DS [12], to measure and report database performance in a virtualized environment. The three virtualized workloads, running on the same physical server, have to be one of the four existing TPC benchmarks and they have to be identical (with the same attributes). The TPC-VMS score is the minimum score of the three TPC benchmarks running in the virtualized environment, specifically, VMStpmC, VMStpsE, VMSQphDS@ScaleFactor or VMSQphH@ScaleFactor. Each of those four TPC-VMS results is a standalone TPC result and can't be compared to each other, or to its native version of the TPC result for commercial or competitive purposes.

1.1.1 Other virtualization benchmarks.

Besides TPC-VMS, there are two other existing virtualization benchmarks in wide usage – VMmark 2.x and SPECvirt_sc2013.

VMmark 2.x [14], developed by VMware, is a multi-host data center virtualization benchmark designed to measure the performance of virtualization platforms. It mimics the behavior of complex consolidation environments by incorporating not only the traditional application-level workloads, but also the platform-level operations, such as guest VM deployment, dynamic virtual machine relocation (vMotion) and dynamic datastore relocation (storage vMotion). The application workloads include LoadGen, Olio, and DS2. With a tile-based scheme, it measures the scalability of a virtualization platform by adding more tiles. The benchmark also provides an infrastructure for measuring power usage of virtualized data centers.

SPECvirt_sc2013 [8], developed by SPEC, is a single-host virtualization benchmark designed to measure the performance of a consolidated server running multiple workloads. It measures the end-to-end performance of all system components including the hardware, virtualization platform, the virtualized guest operating system and the application software. It leverages several existing SPEC benchmarks, such as SPECweb2005, SPECjAppServer2004, SPECmail2008 and SPEC INT2006. It also uses a tile-base scheme and allows benchmarkers to measure power usage of the virtualized servers.

Both VMmark 2.x and SPECvirt_sc2013 workloads are relatively light, and as a result, the guest VMs are relatively small, and usually take a fraction of a physical core. Most of real-life database workloads, however, are CPU-intensive and require much bigger guest VMs to handle the loads. TPC-VMS fills the need for measuring the performance of databases running on virtualized servers.

Another benchmark under development by the TPC is the TPC-V benchmark [3], which will address several features that existing benchmarks do not:

- The VMs in some benchmarks, such as VMmark 2.x and SPECvirt_sc2013, have the same load regardless of the power of the server. At the other end of the spectrum, TPC-VMS will always have exactly 3 VMs; so more powerful servers will have VMs that handle heavier loads. TPC-V emulates a more realistic scenario where not only do more powerful server have more VMs, but also each VM handles a heavier load than less powerful servers. Both the number of VMs and the load handled by each VM grow as servers become more powerful.
- TPC-V VMs have a variety of load levels and two different workload types – OLTP and DSS – to emulate the non-uniform nature of databases virtualized in the cloud.
- The load to each VM varies greatly during a TPC-V execution, emulating the load elasticity that is typical in cloud environments.
- Unlike previous TPC benchmarks, TPC-V will be released with a publicly available, complete end-to-end benchmarking kit.

TPC-V is not yet available. So the only representative option for benchmarking virtualized databases is still TPV-VMS.

1.2 Characteristics of the TPC-E workload

Of the 4 possible workloads to use under TPC-VMS rules, we used the TPC-E workload. TPC Benchmark™ E is composed of a set of transactional operations designed to exercise system functionalities in a manner representative of complex OLTP database application environments. These transactional operations have been given a life-like context, portraying the activity of a brokerage firm, to help users relate intuitively to the components of the benchmark. The brokerage firm must manage customer accounts, execute customer trade orders, and be responsible for the interactions of customers with financial markets. **Fig. 1** illustrates the transaction flow of the business model portrayed in the benchmark:

The customers generate transactions related to trades, account inquiries, and market research. The brokerage firm in turns interacts with financial markets to execute orders on behalf of the customers and updates relevant account information. The number of

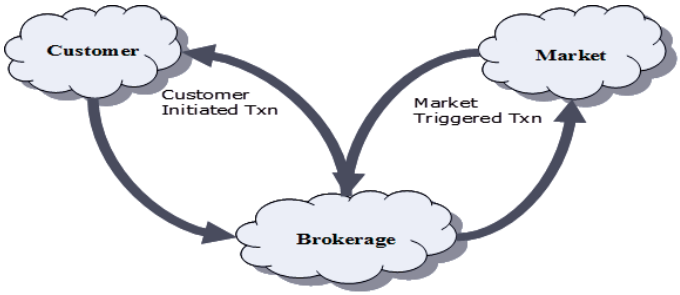


Fig. 1. TPC-E Business Model Transaction Flow

customers defined for the brokerage firm can be varied to represent the workloads of different size businesses.

The benchmark is composed of a set of transactions that are executed against three sets of database tables that represent market data, customer data, and broker data. A fourth set of tables contains generic dimension data such as zip codes.

The benchmark has been reduced to simplified form of the application environment. To measure the performance of the OLTP system, a simple Driver generates Transactions and their inputs, submits them to the System Under Test (SUT), and measures the rate of completed Transactions being returned. This number of transactions is considered the performance metric for the benchmark.

2 Published Results

In this section we will present and analyze the two published results for TPC-VMS. Since the authors were responsible for both disclosures, we can share the details of the tuning process, explain why settings were changed, and quantify the impact of those changes.

2.1 First disclosure

The first TPC-VMS result is on an HP ProLiant DL385p Gen8 server running the vSphere 5.5 hypervisor, Microsoft Windows Server 2012 guest operating system, and Microsoft SQL Server 2012 DBMS. In this section, we will discuss the benchmark configuration and the tuning process.

2.1.1 Configuration

The DL385p Gen8 VSUT[1] test environment is shown in **Fig. 2**.There are three major components to the test environment:

[1] VSUT is a term coined by the TPC-VMS benchmark, and is formally defined in the TPC-VMS specification [13]. In our case, it includes the Consolidated Database Server plus the client systems required by the TPC-E specification.

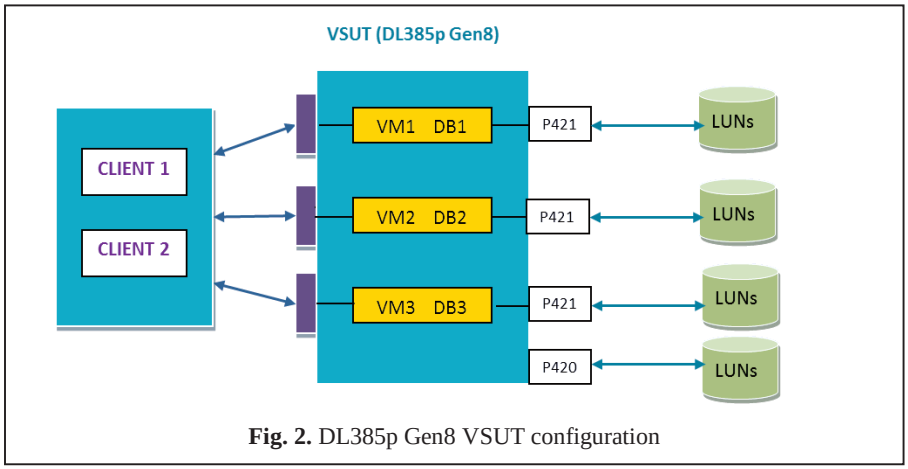


Fig. 2. DL385p Gen8 VSUT configuration

- the clients, on the left
- the *Consolidated Database Server*, which is the hardware and software that implements the virtualization environment which consolidates the TPC-E database server functionality, in the center
- the storage subsystem, on the right

The two client servers are HP ProLiant DL360 G7 using 2 x Hex-Core 2.93 GHz Intel Xeon X5670 Processors with 16 GB PC3-10600 Memory and are connected to the host through a 1 Gb network.

The Consolidated Database Server is an HP ProLiant DL385p Gen8 using 2 x AMD Opteron 6386SE 16-core 2.8 GHz processors with 256GB of memory. There are a total of 32 physical processor cores in 2 sockets. However, note that there are 4 distinct physical NUMA nodes present in the hardware.

Three P421 SmartArray controllers are used; two SAS ports from each controller are directly connected to a D2700 enclosure which hosts LUNs for a database. One P420 SmartArray controller is connected to the internal storage array which hosts LUNs for the database logs. Three 1 Gb NIC ports from client 2 are directly connected to the tier B database server. The fourth NIC port is connected to lab net and is used for remote access and management. Three HP StorageWorks D2700 disk Enclosures are used, one per VM, each attaches to 8 x 400 GB 6G SATA MLC SFF SSDs for a total of 24 drives.

2.1.2 Comparing native and virtualized performance

In the remainder of this paper, we will show results with TPC-E running on a single VM, on 4 VMs, or TPC-VMS running on 3 VMs in a virtualized server, and compare them to native TPC-E results on the same server. It should be emphasized that we used the comparison to native results only as a yardstick to measure our progress. Comparing TPC-VMS results to native TPC-E results are not only against TPC fair use rules, but as we will show in section 4, such comparisons are also misleading if intended as a way to directly contrast the performance of a virtualized server against a native server.

2.1.3 Tuning process

An audited TPC-E result of 1,416.37 tpsE[2] on this exact server had been published just prior to the start of this project. We started the TPC-VMS tuning project by reproducing the native TPC-E result on the setup to establish a baseline. After we got to the satisfactory performance level, we installed vSphere 5.5, and moved to tuning in the virtualized environment. Since DL385p Gen8 has four NUMA nodes, running 3 guest VMs on a 4 NUMA node machine introduces imbalance. We decided to approach the tuning in three phases: first, have one guest VM running one instance of TPC-E; second, have 4 guest VMs each running one instance of TPC-E; last, have 3 guest VMs each running one instance of TPC-E, which is the configuration for TPC-VMS.

The goal for phase one tuning is to find tunes that reduce the overhead of hypervisor and get as close to native performance as we can with one big guest VM. The end result was 85% of native performance. **Table 1** and **Fig. 3** chart the progress of the tuning exercise. Some notes are in order:

- Virtual Raw Device Mapping (RDM) provides the benefits of direct access to physical devices in addition to most of the advantages of a virtual disk on VMFS storage [15]. The difference is akin to the difference between file system I/O and raw I/O at the operating system level.
- Spreading the LUNs over multiple vHBAs allows the guest OS to distribute the I/O interrupts over multiple CPUs. It also trades off some interrupt coalescing, which is good for CPU usage efficiency, for better latency, which is important for TPC-E.
- The TPC-E workload on SQL Server is a heavy user of timer queries. Normally, reading the RDTSC results in an expensive VMexit, but vSphere 5.5 allows certain operating systems such as Windows Server 2012 to access the time source directly

Table 1. Tuning steps for 1 VM

Throughput	Tuning
587	Baseline
618	Switch the virtual disks from VMFS to virtual RDM
683	Instead of all 4 (virtual) LUNs of each VM on a single virtual HBA in the guest, use 4 virtual HBAs with a single LUN per vHBA
1090	Improve RDTSC access
1095	32 vCPUs, 4 NUMA nodes X 8 vCPUs, bind vCPUs in 4 nodes, bind vmkernel threads
1139	32 vCPUs, 4 NUMA nodes X 8 vCPUs, bind vCPUs in 2 x 16 groups, bind vmkernel threads
1198	30 vCPUs, 4 NUMA nodes with 8/8/8/6 vCPUs, bind vCPUs in 2 x 15 groups, bind vmkernel threads

[2] As of 6/13/2014. Complete details available at http://www.tpc.org/4064

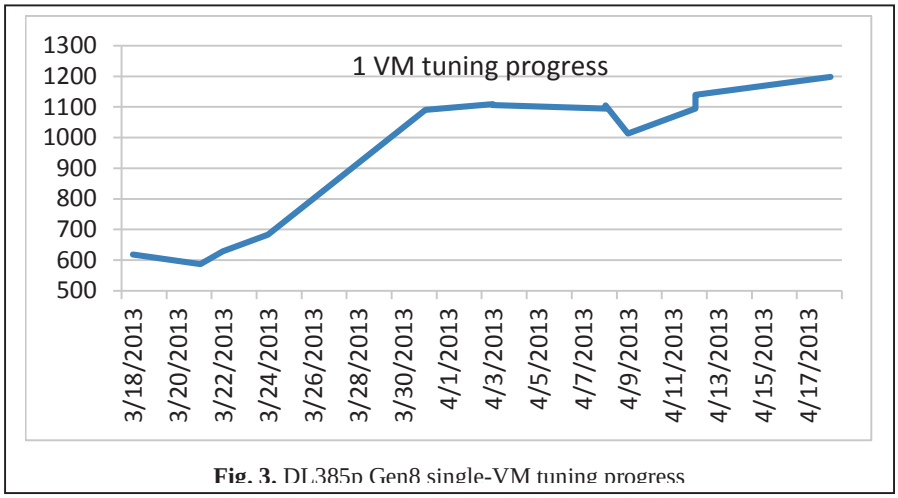


Fig. 3. DL385p Gen8 single-VM tuning progress

through the RDTSC instruction, avoiding the VMexit. This is a vSphere 5.5 feature, and we saw a 60% boost when we switched to the code base that had this feature.

- Setting aside CPUs for the auxiliary threads of the hypervisor kernel (*vmkernel* [16]) and binding virtual CPUs and vmkernel threads to specific cores is a common tuning practice, and it allowed us to go from 1.86X the original results to 2.04X.
- VM sizing and vCPU binding choices are discussed in sections 3.1and 3.2.

Phase one tuning brought the virtual performance up to 85% of native. Rather than further tuning of the 1-VM case, we continued with tuning of a 4-VM configuration.

The goal for phase two tuning is to see what kind of overhead we get when the number of guest VMs align with the number of NUMA nodes on the server. In this case, we assign each guest to a NUMA node to reduce the remote memory access overhead.

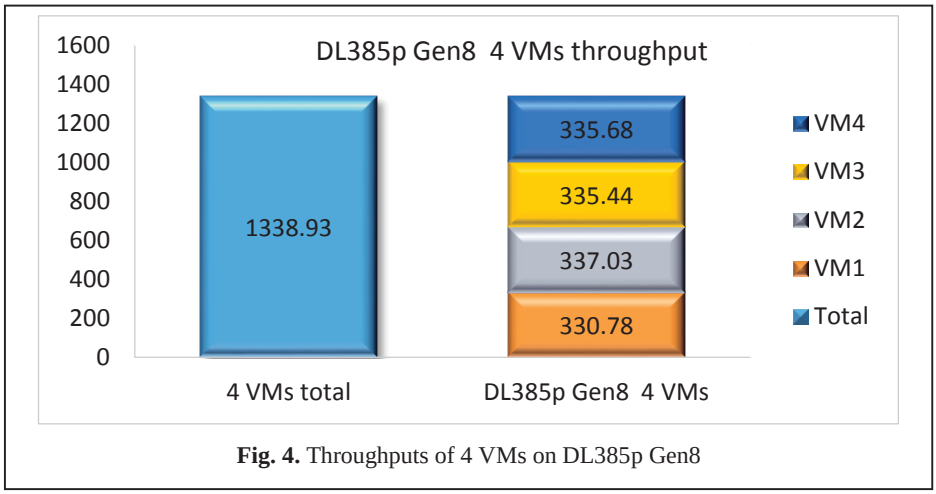


Fig. 4. Throughputs of 4 VMs on DL385p Gen8

We used 4 DB instances, each running in one of the 4 guest VMs. Each DB has one dedicated P421 SmartArray Controller, each VM has 8 vCPUs and is bound to a separate NUMA node. The sum of the throughputs of the 4 VMs, shown in **Fig. 4** reached 94.5% of native performance. This is not surprising: 8-way systems have less SMP overhead than 32-way systems, whether native or virtual. Also, each of the 4 VMs enjoyed 100% local memory.

What is also worth noting is that with little tuning, the 4-VM aggregate throughput was 12% higher than the 1-VM throughput, confirming the benefits of NUMA locality and better SMP scaling of smaller VMs.

The TPC-VMS benchmark was designed to run with 3 VMs expressly to pose a scheduling challenge to the hypervisor. Our first runs in this configuration proved the difficulty of achieving good performance with 3 VMs on a server with 4 NUMA nodes.

The throughput values in **Table 2** are the sums of the throughputs of the 3 VMs. This aggregate throughput is not a TPC-VMS metric, but is a good way of charting our progress in comparison with 1-VM and 4-VM configurations. **Fig. 5** shows the improvement in throughput as we applied the optimizations detailed in **Table 2**, and **Fig. 6** shows the throughputs of the 3 VMs for the published result.

The goal of phase three is to get the best performance for publishing. Measured by the unofficial metric of the overall throughput of the 3 VMs, we achieved close to 100% of the native result. Section 4 will explore the reasons behind this good performance.

In phase one of the tuning, the challenge is to reduce the hypervisor overhead, while in phase three, the challenge is to balance the Consolidated Database Sever resources to achieve the best result. Since the TPC-VMS score is the lowest throughput of the three TPC-E instances, the goal is to maximize the lowest throughput, or to get the three throughput values as close to each other as possible.

The hypervisor has full control over how to expose the NUMA properties of the hardware to the guest OS. It presents a number of *virtual NUMA nodes* to the guest OS.

Table 2. Tuning steps for 3 VMs on the DL385p Gen8

Throughput	Tuning
840	Five 2-vCPU virtual NUMA clients per VM
1150	For each VM, distribute and bind the 5 virtual NUMA clients and its vmkernel threads over 3 physical NUMA nodes
1224	For each VM, bind 4 2-vCPU virtual NUMA clients to an 8-core physical NUMA node; the 4th physical NUMA node runs the remaining 2 vCPUs of each VM and its vmkernel threads
1262	Fine tune binding policy
1355	Increase database size from 200,000 Customer to 220,000
1396	Use the new vSphere 5.5 feature: virtual hardware version 10
1395.97	Published result of 457.77 VMStpsE

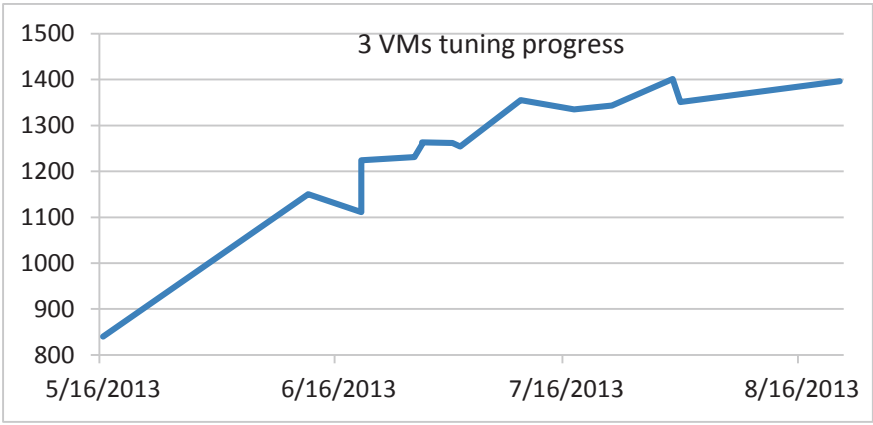


Fig. 5. DL385p Gen8 3-VM tuning progress

A *NUMA client* is a group of vCPUs that are scheduled on a physical NUMA node as a single entity. Typically, each virtual NUMA node is a NUMA client, unless a virtual NUMA node is larger than a physical NUMA node and maps to multiple virtual NUMA clients. We used five 2-vCPU virtual NUMA clients for each VM. This provides good granularity to distribute the workload from each VM evenly amongst two NUMA nodes. Upgrading vSphere 5.5 hardware version to 10 gave us a 4.6% performance improvement.

The phase three tuning culminated in the published result of 457.77 VMStpsE. The throughputs of the 3 VMs were 457.55 tpsE, 468.11 tpsE, and 470.31 tpsE. So the official reported TPC-VMS result was 457.77 VMStpsE[3]. We can see that the lowest throughput VM was at 97.3% of the highest throughput VM, showcasing a successful division of resources among the 3 VMs, which as [7] points out, is not trivial.

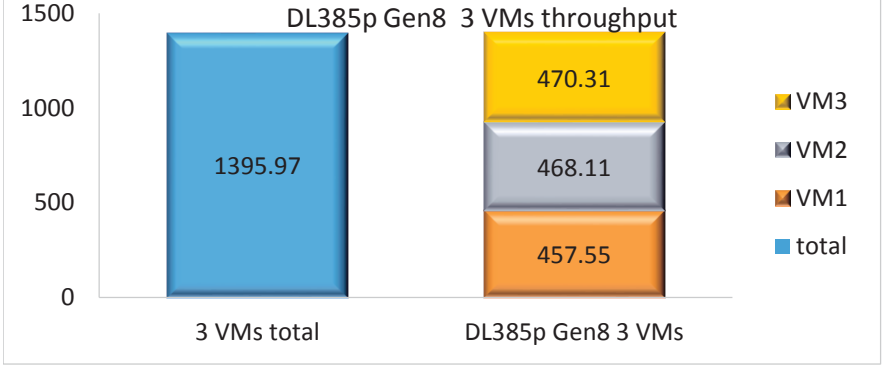


Fig. 6. Throughput of 3 VMs on DL385

[3] As of 6/13/2014. Complete details available at http://www.tpc.org/5201

2.2 Second disclosure

The second TPC-VMS result is on an HP ProLiant DL380p Gen8 server again running vSphere 5.5 and Microsoft Windows Server 2012, but the newer Microsoft SQL Server 2014 as the DBMS. We will cover the benchmark configuration and the tuning process.

2.2.1 Configuration

The client hardware is identical to the one described in Section 2.1.1. The Consolidated Database Sever server, **Fig. 7** is an HP ProLiant DL380p Gen8 using 2 x Intel Xeon E5-2697 v2 processors with 256GB of memory. Turbo boost and HyperThreading were enabled; so we have 24 cores and 48 HyperThread (logical) processors in 2 physical NUMA nodes. Six P421 SmartArray controllers are used, each attaches to one D2700 enclosure containing 4 X 800GB 6G MLC SSDs for a total of 24 drives.

2.2.2 Tuning process

Although no native TPC-E results have been published on this exact server, there are two published results that also used servers with 2 Intel Xeon E5-2697 v2 processors, 512 GB of memory, Microsoft SQL Server (one used the 2012 edition, one 2014), and Microsoft Windows Server 2012. Based on published throughputs of 2,472.58 tpsE[4] and 2,590 tpsE[5] on these systems, we used an even 2,500 tpsE for baseline native performance for our server in order to gauge the progress of the tuning project.

Having already completed a full tuning cycle for the first disclosure, we did not need to experiment with the 1-VM or 4-VM configurations, and started the tests with 3 VMs. Some notes regarding the tuning steps in **Table 3**:

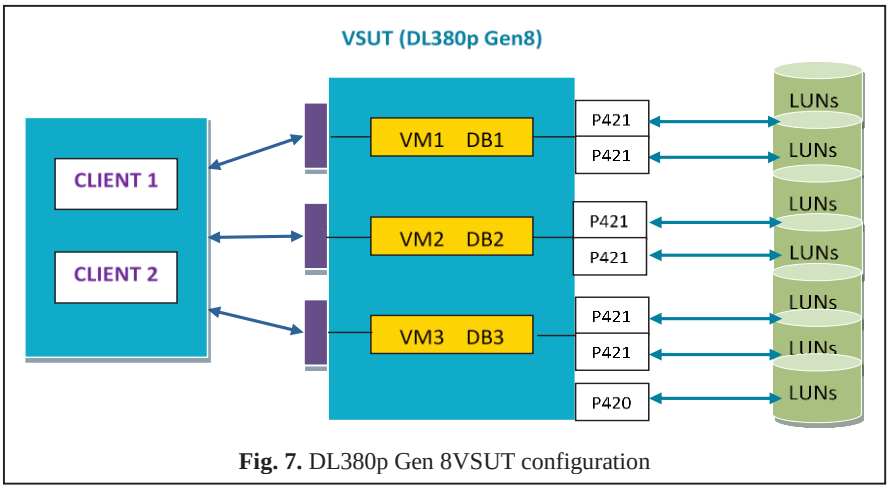


Fig. 7. DL380p Gen 8VSUT configuration

[4] As of 6/13/2014. Complete details available at http://www.tpc.org/4065
[5] As of 6/13/2014. Complete details available at http://www.tpc.org/4066

Table 3. Tuning steps for 3VMs on the DL380p Gen8

Throughput	Tuning
1851	Baseline: 16 vCPUs for each VM, all 3 VMs split across 2 NUMA nodes, no placement of vmkernel threads
1951	15 vCPUs for each VM
1974	Bind the vmkernel threads associated with VMs
2022	Also bind the vmkernel threads associated with physical devices
2089	VM1 split across NUMA nodes; VMs 2 and 3 bound to 1 NUMA node
2139	Drop the Soft-NUMA setting in SQL Server
2191	monitor.virtual_mmu=software
2179	Final configuration with fewer, denser drivers, 375K Customers, and SQL Server 2014
2179.11	Published result of 718.12 VMStpsE

- Much of the tuning exercise was focused on choosing the right number of vCPUs (15 or 16), and the placement of the vCPUs of 3 VMs and the vmkernel threads on the 2 NUMA nodes
- The aggregate throughput of 2022 is with each of the 3 VMs split across both NUMA nodes. By placing VM2 and VM3 only on one NUMA node each, their performance increases by 4-5%, without impacting VM1, giving us the 2089 aggregate performance. More on VM sizing and vCPU binding choices in sections 3.1and 3.2.
- SQL Server allows CPUs in hardware NUMA nodes to be grouped into smaller *Soft-NUMA* nodes for finer control, e.g. for load balancing of networking traffic. The Soft-NUMA node setting enhanced performance by enabling finer granularity in the first TPC-VMS result. However in the second result with the DL380p Gen8 system,

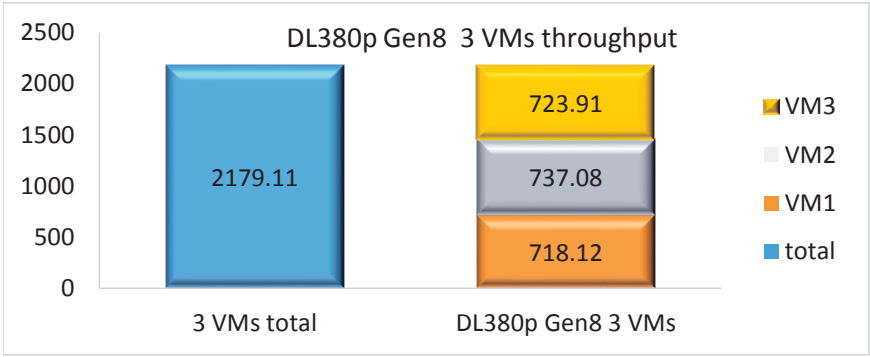


Fig. 8. Throughputs of 3 VMs on DL380p Gen8

it hurt performance. When the Soft-NUMA setting is dropped, and we instead expose the server's 2 Hard-NUMA nodes to the DBMS, throughput improves by 2.4%

- Section 3.3 discusses the reasoning behind setting monitor.virtual_mmu=software.

Fig. 8 shows the throughputs of the 3 VMs at 718.12 tpsE, 737.08 tpsE, and 723.91 tpsE. So the official reported TPC-VMS result was 718.12 VMStpsE[6]. We can see that the lowest throughput VM was at 97.4% of the highest throughput VM, matching the first publication even though having only 2 NUMA nodes made this a more challenging project.

The virtual-native ratio is lower than that of the first disclosure mainly due to the difficulty of evenly fitting 3 VMs on two NUMA nodes, compared to the 4 NUMA nodes of the first disclosure. It is worth noting that VM1, which was split between the two NUMA nodes, had a throughput of only 2.6% lower than VM2, which was entirely contained within one NUMA node. So we did not see a large benefit from NUMA locality, which as Section 4 demonstrates, has a profound impact on performance. One reason is that we have to favor VM1, which determines the reported metric, even at the cost of a *larger* negative impact on the other two VMs. In our configuration, 45 of the 48 physical CPUs run the 45 vCPUs of the 3 VMs. It would be intuitive to assign the vmkernel auxiliary threads of each VM to one of the 3 remaining physical CPUs. This indeed gives us the highest aggregate throughput, but not the highest VM1 throughput. So we distribute some of VM1's auxiliary vmkernel threads to the pCPUs that handle the vmkernel threads of VM2 and VM3.Although this has enough of a negative impact on VM2 and VM3 to cause a small (0.4%) drop in the aggregate performance, VM1 performance improves by 2% to the final reported value.

3 Lessons learned

3.1 VM size matters

Since TPC-E tests typically utilize the compute resources and the memory of the SUT to nearly 100%, we configured each of the 3 VMs with nearly 1/3 of the memory, after allowing for a small virtualization tax. Since memory can be allocated in arbitrary units, dividing the host memory into 3 chunks for the 3 VMs was trivial, as was the hypervisor automatically placing the memory pages on the same physical NUMA nodes as the VM's vCPUs. But choosing the right number of vCPUs posed a challenge. For the AMD-based platform with a total of 32 cores, the best configuration proved to be 3 10-vCPU VMs, and 2 cores left for the vmkernel threads. With the 4 NUMA nodes (see section 3.2), both the vCPU count and vCPU placement was rather easy.

The Intel-based platform with 24 cores/48 HyperThreads and 2 NUMA localities presented more of a challenge. An intuitive choice was a configuration with 16 vCPUs per VM, and allowing the hypervisor's scheduler to provision CPU time between the vCPUs and the vmkernel threads. But in practice, we found that allocating 15 vCPUs

[6] As of 6/13/2014. Complete details available at http://www.tpc.org/5202

per VM and leaving 3 logical processors for the vmkernel threads gave us the best performance even though it left a few percent idle unused. This may appear to be a negligible amount of idle, but in fact it is significant in a well-tuned TPC-E configuration.

The 16-vCPU configuration faced a lot more idle than the 15-vCPU case due to the high latency of storage I/O caused by the vmkernel threads competing with vCPUs for CPU resources. One could extrapolate the throughput to a slightly higher value than what we achieved with 15 vCPUs, but we were not able to push more load through the system and utilize the remaining CPU power due to high storage latencies.

3.2 vCPU placement and binding plays an important role

The choice of 3 VMs for TPC-VMS certainly succeeded in producing a hard workload for the hypervisor scheduler! Utilizing all system resources in a way that a) resulted in full CPU utilization, and b) uniform throughput for the 3 VMs proved to be very difficult. It was somewhat easier with 4 NUMA nodes because we could dedicate 3 of the NUMA nodes to the 3 VMs, one node per VM, and use the 8 cores in the 4th node for 2 more virtual CPUs for each VM as well as the vmkernel threads. But mapping vCPUs and vmkernel threads to physical CPUs posed a bigger challenge with 2 NUMA nodes. Keep in mind that the TPC-VMS metric is the lowest of the 3 VMs' throughputs. So maximizing the overall performance – informally measured as the sum of the throughputs of the 3 VMs – does not benefit us if it comes at the cost of lowering the throughput of the lowest VM.

The optimal configuration was splitting VM1 across the two NUMA nodes, and binding VMs 2 and 3 to a NUMA node each. With 15 vCPUs per VM, this left 3 HyperThreads to be dedicated to vmkernel threads. The overall percentage of time in vmkernel, a portion of which is the time in the vmkernel threads, was measured to be around 8%. So dedicating 2 of the 32 cores on the DL385p Gen8 or the 3 HyperThreads on the 48-HyperThread DL380p Gen8 to the vmkernel threads proved to be a good fit.

3.3 Reduce TLB miss processing overhead

A complete treatment of the two-dimensional TLB walk is outside the scope of this paper. Consult [1, 2, 4, and 6] for a more thorough treatment of Intel's EPT and AMD's NPT. Briefly, modern microprocessors have hardware assist features for virtualizing memory, and handling the address translation from the guest Virtual Address to the guest Physical Address and ultimately to the host Machine Address. Intel's EPT and AMD's NPT make managing virtualized virtual memory much easier, but the trade-off is a near doubling of the TLB miss processing costs for certain workloads. The vast majority of workloads run faster with EPT and NPT even though TLB misses are more expensive. But occasionally, the higher TLB miss processing costs may be a heavy overhead for an application. TPC-E on SQL Server on Windows appears to be one such workload [7]. Hardware counter collections showed an overhead of as much as 9% in two-dimensional TLB miss processing. By switching to the older software MMU option [2], we were able to raise the performance of the DL380p Gen8 server by 2.7%.

4 Comparing the performance of virtual and native platforms

The TPC fair use rules disallow comparisons between different TPC benchmarks, including those of the same benchmark with different scale factors. And, although TPC-VMS is based on existing benchmarks, this rule still holds, and TPC-VMS results can't be compared to their native counterparts. However, since TPC-VMS can be run on the same system/software that may have a full native result also published, there is a strong tendency to simply add the performance metrics of the three VMs together and directly compare to this native result. It may be assumed that the fair use rules are for marketing or administrative purposes and that this is an excellent way to determine the overhead of the hypervisor used. There are several technical reasons to avoid this comparison.

Firstly, the native system has a database three times the size of the virtualized test. Comparing databases of different sizes is not allowed by the benchmark specification, and the effects on IO and memory due to database size are non-linear in fashion. Second, the memory is divided into three, allowing the SQL Server buffer pool to use more local accesses for each guest VM, making it more efficient. And most importantly, each instance of SQL server is affinitized to only one third of the total processors, incurring less contention and scaling issues than the full native system which must scale across all logical processors.

From this, one can see that the summation of the TPC-VMS throughputs of the 3 VMs is not a proper direct comparison to a native result. One method to make a comparison to the non-virtualized system would be to use three native instances of Microsoft SQL Server, running against three databases of the same size as the virtualized TPC-VMS case. A proper comparison of this type was completed using the HP DL385p Gen8 server. See **Fig. 9**.

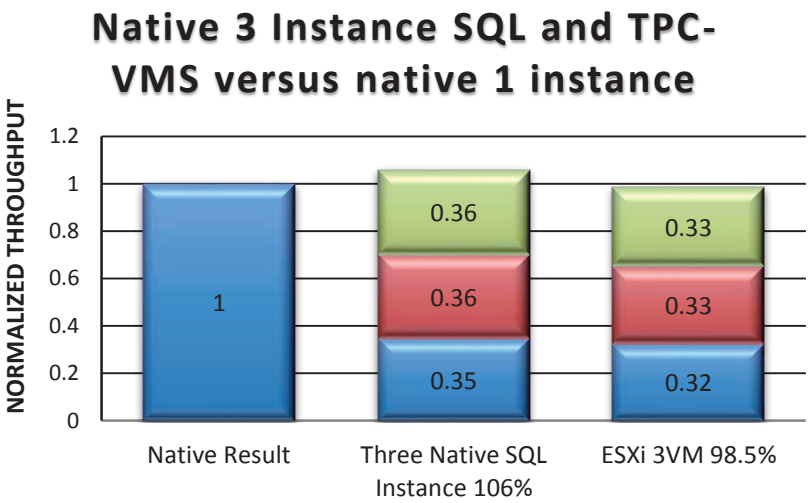


Fig. 9. Three databases on a native server and a virtualized server

Table 4. Native and Virtualized Comparison Points

Comparison	Performance ratio
TPC-VMS versus published TPC-E	98.6%
3-instance native TPC-E versus 1-instance native TPC-E	105.9%
Estimated TPC-VMS versus 3-instance native TPC-E	93.1%

Table 4 summarizes the findings when running three instances of SQL server. In the case of the HP DL385p Gen8 server, one only sees a difference of 1.5% when comparing the published native TPC-E and virtualized TPC-VMS results. However, due to the aforementioned technical reasons, three instances of SQL Server on a native system yields nearly 6% more performance than the full native system. Thus, it can be concluded that there is an overhead of at least 7% simply due to virtualization. The main take-away here is not an exact calculation of virtualization overhead. Rather, it is that an oversimplified comparison, especially using different benchmarking rules, can be misleading.

5 Conclusions and Future Work

TPC-VMS is a virtualized database benchmark using the existing TPC workloads. HP, working with VMware, published the first two TPC-VMS results with TPC-E. In this paper, we shared the configuration and tunings of the two setups, the analysis of the native and virtualized results, and virtualized database tuning tips. As the results show, with the very low overhead of the virtualization, virtualized servers make excellent platforms for the most demanding database workloads. Nonetheless, virtualization introduces an extra layer of software that needs to be tuned for optimal performance.

There are a number of areas for future work, including running 3 and 4 database instances on one VM, and comparing the aggregate performance to multiple instances on a native system as well as to multiple VMs, each with one database instance. This will quantify more accurately the overhead of the virtualization layer, as well as the additional costs of running multiple VMs on one server. Note that although we collected data on some of these configurations, these were stepping stones on the way to the final TPC-VMS configurations, and not well-tuned.

It would be instructive to collect hardware event counts on the two server to explain why one server seems to record an aggregate throughput closer to a native system. How much of a role does the number of NUMA nodes – and more generally, TPC-VMS's unusual VM count of 3 – play in the performance difference with native systems? Are there performance differences between the virtualization-assist features of the processors?

Achieving the good performance reported here required fine tuning of the hypervisor settings. Although these optimizations are well-known to the performance community, we still had to manually apply them. A direction for future work is incorporating the

optimizations into the hypervisor scheduler to detect DBMS workloads and automatically optimize for them.

Finally, this tuning methodology can be applied to other database workloads, other DBMS products, and other hypervisors.

6 Acknowledgements

We would like to thank the TPC for making the TPC-VMS benchmark available, and the following individuals for their contribution to the benchmark results: Bao Chun-Jie, Seongbeom Kim, Paul Cao, Juan Garcia-Rovetta, Bruce Herndon, and Chethan Kumar.

7 References

1. Ravi Bhargava, Ben Serebrin, Francesco Spanini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2008.
2. Nikhil Bhatia, Performance Evaluation of Intel EPT Hardware Assist, http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf .
3. Andrew Bond, Doug Johnson, Greg Kopczynski, and H. Reza Taheri: Architecture and Performance characteristics of a PostgreSQL implementation of the TPC-E and TPC-V workloads. In: Raghunath Nambiar, Meikel Poess (Eds.): Selected Topics in Performance Evaluation and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Springer 2013, LNCS Volume 8391, ISBN 978-3-319-04935-9.
4. Jeffrey Buell, et al., Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications, VMware Technical Journal, Summer 2013.
5. IDC: "Worldwide Virtual Machine 2013–2017 Forecast: Virtualization Buildout Continues Strong", http://www.idc.com/getdoc.jsp?containerId=242762.
6. Intel 64 and IA-32 Architectures Developer's Manual.
7. Wayne D. Smith and Shiny Sebastian, Virtualization Performance Insights from TPC-VMS, http://www.tpc.org/tpcvms/tpc-vms-2013-1.0.pdf
8. SPECvirt_sc2013 benchmark info, SPEC Virtualization Committee: http://www.spec.org/virt_sc2013/
9. TPC: Detailed TPC-C description: http://www.tpc.org/tpcc/detail.asp
10. TPC: Detailed TPC-E Description: http://www.tpc.org/tpce/spec/TPCEDetailed.doc
11. TPC: TPC Benchmark H Specification: http://www.tpc.org/tpch/spec/tpch2.14.4.pdf
12. TPC: TPC Benchmark DS Specification: http://www.tpc.org/tpcds/spec/tpcds_1.1.0.pdf
13. TPC: TPC-VMS benchmark: http://www.tpc.org/tpcvms/default.asp
14. VMware, Inc., VMmark 2.x, http://www.vmware.com/products/vmmark/overview.html
15. VMware, Inc., Performance Characteristics of VMFS and RDM, http://www.vmware.com/files/pdf/vmfs_rdm_perf.pdf
16. VMware, Inc., The Architecture of VMware ESXi, http://www.vmware.com/files/pdf/ESXi_architecture.pdf.

THEORY

– *Byzantine Agreement with Optimal Early Stopping, Optimal Resilience and Polynomial Complexity*

– *Distributed Resource Discovery in Sub-Logarithmic Time*

Byzantine Agreement with Optimal Early Stopping, Optimal Resilience and Polynomial Complexity

Ittai Abraham*
VMware Research
Palo Alto, CA, USA
iabraham@vmware.com

Danny Dolev†
Hebrew University of Jerusalem
Jerusalem, Israel
dolev@cs.huji.ac.il

ABSTRACT

We provide the first protocol that solves Byzantine agreement with optimal early stopping ($\min\{f+2, t+1\}$ rounds) and optimal resilience ($n > 3t$) using polynomial message size and computation.

All previous approaches obtained sub-optimal results and used resolve rules that looked only at the immediate children in the EIG (*Exponential Information Gathering*) tree. At the heart of our solution are new resolve rules that look at multiple layers of the EIG tree.

1. INTRODUCTION

In 1980 Pease, Shostak and Lamport [PSL80, LSP82] introduced the problem of Byzantine agreement, a fundamental problem in fault-tolerant distributed computing. In this problem n processes each have some initial value and the goal is to have all correct processes decide on some common value. The network is reliable and synchronous. If all correct processes start with the same initial value then this must be the common decision value, and otherwise the value should either be an initial value of one of the correct processes or some pre-defined default value.[1] This should be done in spite of at most t corrupt processes that can behave arbitrarily (called Byzantine processes). Byzantine agreement abstracts one of the core difficulties in distributed computing and secure multi-party computation — that of coordinating a joint decision. Pease et al. [PSL80] prove that Byzantine agreement cannot be solved for $n \leq 3t$. Therefore we say that a protocol that solves Byzantine agreement for $n > 3t$ has *optimal resilience*. Fisher and Lynch [FL82] prove that any protocol that solves Byzantine agreement must have an execution that runs for $t+1$ rounds. Dolev et al. [DRS90] prove that any protocol must have executions that run for $\min\{f+2, t+1\}$ rounds, where f is the actual number of corrupt processes. Therefore we say that a protocol that solves Byzantine agreement with $\min\{f+2, t+1\}$ rounds has *optimal early stopping*.

The protocol of [PSL80] has optimal resilience and optimal worst case $t+1$ rounds. However the message complexity of their protocol is exponential. Following this result, many have studied the question of obtaining a protocol with optimal resilience and optimal worst case rounds that uses only polynomial-sized messages (and computation).

Dolev and Strong [DS82] obtained the first polynomial protocol with optimal resilience. The problem of obtaining a protocol with optimal resilience, optimal worst case rounds and polynomial-sized messages turned out to be surprisingly challenging. Building on a long sequence of results, Berman and Garay [BG93] presented a protocol with optimal worst case rounds and polynomial-sized messages for $n > 4t$. In an exceptional tour de force, Garay and Moses [GM93, GM98], presented a protocol for binary-valued Byzantine agreement obtaining optimal resilience, polynomial-sized messages and $\min\{f+5, t+1\}$ rounds. We refer the reader to [GM98] for a detailed and full account of the related work. Recently Kowalski and Mostéfaoui [KM13] improved the message complexity to $\tilde{O}(n^3)$ but their solution does not provide early stopping and requires exponential computation.

Worst case running of $t+1$ rounds is the best possible if the protocol is to be resilient to an adversary that controls t processes. However, in executions where the adversary controls only $f < t$ processes, the optimal worst case can be improved to $f+2$ rounds. Berman et al. [BGP92] were the first to obtain optimal resilience and optimal early stopping (i.e. $\min\{f+2, t+1\}$ rounds) using exponential size messages. Early stopping is an extremely desirable property in real world replication systems. In fact, agreement in a small number of rounds when $f = 0$ is a core advantage of several practical state machine replication protocols (for example [CL99] and [KAD+07] focus on optimizing early stopping in the fault free case).

Somewhat surprisingly, after more than 30 years of research on Byzantine Agreement, the problem of obtaining the best of all worlds is still open. There is no protocol with

*Part of the work was done at Microsoft Research Silicon Valley.

†Part of the work was done while visiting Microsoft Research Silicon Valley. Danny Dolev is Incumbent of the Berthold Badler Chair in Computer Science. This research project was supported in part by The Israeli Centers of Research Excellence (I-CORE) program, (Center No. 4/11), and by grant 3/9778 of the Israeli Ministry of Science and Technology.

[1]Other versions of the problem may not restrict to a value on one of the correct processes, if not all initial values are the same, or require agreement on a leader's initial value, which can be reduced to the version we defined.

STOC'15, June 14–17, 2015, Portland, Oregon, USA.

http://dx.doi.org/10.1145/2746539.2746581.

optimal resilience, optimal early stopping and polynomial-sized message. The conference version of [GM98] claimed to have solved this problem but the journal version only proves a $\min\{f+5, t+1\}$ round protocol, then says it is *possible* to obtain a $\min\{f+3, t+1\}$ round protocol and finally the authors say they *believe* it should be possible to obtain a $\min\{f+2, t+1\}$ round protocol. We could not see how to directly extend the approach of [GM98] to obtain optimal early stopping. The main contribution of this paper is solving this long standing open question and providing the optimal $\min\{f+2, t+1\}$ rounds with optimal resilience and polynomial complexity. Moreover, our result applies directly for arbitrary initial values and not only to binary initial values, as some of the previous results.

Our Byzantine agreement protocol obtains a stronger notion of *multi-valued validity*. If $v \neq \perp$ is the decision value then at least $t+1$ correct processes started with value v. The multi-valued validity property is crucial in our solution for early stopping with monitors. This property is also more suitable in proving that Byzantine agreement implements an ideal world centralized decider that uses the majority value. We note that several previous solutions (in particular [GM98]) are inherently binary and their extension to multi-valued agreement does not have the stronger multi-valued validity property.

THEOREM 1. *Given n processes, there exists a protocol that solves Byzantine agreement. The protocol is resilient to any Byzantine adversary of size $t < n/3$. For any such adversary, the total number of bits sent by any correct process is polynomial in n and the number of rounds is $\min\{f+2, t+1\}$ where f is the actual size of the adversary.*

Overview of our solution. At a high level we follow the framework set by Berman and Garay [BG93]. In this framework, if at a given round all processes seem to behave correctly then the protocol stops quickly thereafter. So if the adversary wants to cause the protocol to continue for many rounds it must have at least one corrupt process behave in a faulty manner in each round. However, behaving in a faulty manner will expose the process and in a few rounds the misbehaving process will become publicly exposed as corrupt.

This puts the adversary between a rock and a hard place: if too few corrupt processes are publicly exposed then the protocol reaches agreement quickly, if too many corrupt process are exposed then a "monitor" framework (also called "cloture votes") that runs in the background causes the protocol to reach agreement in a few rounds. So the only path the adversary can take in order to generate a long execution is to publicly expose exactly one corrupt process each round. In the $t < n/4$ case, this type of adversary behavior keeps the communication polynomial.

For $t < n/3$ a central challenge is that a corrupt process can cause communication to grow in round i but will be publicly exposed only in round $i+2$. Naively, such a corrupt process may also cause communication to grow both in round i and $i+1$ and this may cause exponential communication blowup. Garay and Moses [GM98] overcome this challenge by providing a protocol such that, if there are at most two new corrupt processes in round i and no new corrupt process in round $i+1$ then even though they are publicly exposed in round $i+2$ they cannot increase communication in round $i+1$ (also known as preventing "cross corruption").

At the core of the binary-valued protocol of Garay and Moses is the property that one value can only be decided on even rounds and the other only on odd rounds. This property seems to raise several unsolved challenges for obtaining optimal early stopping. We could not see how to overcome these challenges and obtain optimal early stopping using this property. Our approach allows values to be fixed in a way that is indifferent to the parity of the round number (and is not restricted to binary values).

Two key properties of our protocol that makes it quite different from all previous protocols. First, the value of a node is determined by the values of its children and grandchildren in the EIG tree ([BNDDS92]). Second, if agreement is reached on a node then the value of all its children is changed to be the value of the node. This second property is crucial because otherwise even though a node is fixed there could be disagreement about the value of its child. Since the value of the parent of the fixed node depends on its children and grandchildren, the disagreement on the grandchild may cause disagreement on the parent and this disagreement could propagate to the root.

The decision to change the value of the children when their parent is fixed is non-trivial. Consider the following scenario with a node σ, child σp and grandchild σpq: some correct reach agreement that the value of σpq is d, then some correct reach agreement that the value of σp is $d' \neq d$ and hence the value of σpq is changed (colored) to d'. So it may happen that some correct decide the value of σ based on σpq being fixed on d and some other correct decide the value of σ based on σpq being colored to d'. Making sure that agreement is reached in all such scenarios requires us to have a relatively complex set of complementary agreement rules.

To bound the size of the tree by a polynomial size we prove that the adversary is still between a rock and a hard place: roughly speaking there are three cases. If just one new process is publicly exposed in a given round then the tree grows mildly (remains polynomial). If three or more new processes are exposed in the same round then this increases the size of the tree but can happen at most a constant number of times before a monitor process will cause the protocol to stop quickly.

The remaining case is when exactly two new processes are exposed, then a sequence of (possibly zero) rounds where just one new process is exposed in each round, followed by a round where no new process is exposed. This is a generalized version of the "cross corruption" case of [GM98] where the adversary does not face increased risk of being caught by the monitor process. We prove that in these cases the tree essentially grows mildly (remains polynomial).

In order to deal with this generalized "cross corruption" we introduce a special resolve rule (SPECIAL-BOT RULE) tailored to this scenario. In particular, in some cases we fix the value of a node σ to $\perp$ (a special default value) if we detect enough support. This solves the generalized "cross corruption" problem but adds significant complications. Recall that when we fix a value to a node then we also fix (color) the children of this node with the same value.

Suppose a process fixes a node σ to $\perp$. The risk is that some correct processes may have used a child σp with value d but some other correct process will see $\perp$ for σp (because when σ is fixed to $\perp$ we color all its children to $\perp$). Roughly speaking, we overcome this difficulty by having two resolve rule thresholds. The base is the $n-t$ thresh-

old (RESOLVE RULE, IT-TO-RT RULE) and the other is with a $n-t-1$ threshold (RELAXED RULE). In essence this $n-t-1$ rule is resilient to disagreement on one child node (that may occur due to coloring). We then make sure that the SPECIAL-BOT RULE can indeed change only one child value. This delicate interplay between the resolve rules is at the core of our new approach.

The adversary. Given $n > 3t$ and $\phi \leq t$, as in [GM98], we will consider a (t, ϕ)-*adversary* - an adversary that can control up to ϕ corrupt processes that behave arbitrarily and at most $t - \phi$ corrupt processes that are always silent (send some default value $\perp$ to all processes every round). The (t, ϕ)-adversary will be useful to model executions in which all correct processes have detected beforehand some common set of at least $t - \phi$ corrupt processes and hence ignore them throughout the protocol. Note that the standard t-adversary is just a (t, t)-adversary.

2. THE EIG STRUCTURE AND RULES

In this section we define the EIG structure and rules.

Let N be the set of processes, $n = |N|$ and assume that $n > 3t$. Let D be a set of possible decision values. We assume some decision $\perp \in D$ is the designated default decision.

Let Σ_r be the set of all sequences of length r of elements of N without repetition. Let $\Sigma_0 = \epsilon$, the empty sequence. Let $\Sigma = \bigcup_{0 \leq j \leq t+1} \Sigma_j$. An *Exponential Information Gathering* tree (EIG in short) is a tree whose nodes are elements in Σ and whose edges connect each node to the node representing its longest proper prefix. Thus, node ϵ has n children, and a node from Σ_k has exactly $n - k$ children.

We will typically use the Greek letter σ to denote a sequence (possibly empty) of labels corresponding to a node in an EIG tree. We use the notation σq to denote the node in the EIG tree that corresponds to the child of node σ that corresponds to the sequence σ concatenated with $q \in N$. We denote by $\bar{\epsilon}$ the root node of the tree that corresponds to the empty sequence. Given two sequences $\sigma, \sigma' \in \Sigma$, let $\sigma' \prec \sigma$ denote that σ' is a proper prefix of σ and $\sigma' \sqsubseteq \sigma$ denote that σ' is a prefix of σ (potentially $\sigma' = \sigma$).

In the EIG consensus protocol each process maintains a dynamic tree data structure $\mathcal{IT}$. This data structure maps a set of nodes in σ to values in D. Intuitively, this tree contains all the information the process has heard so far. Each process z also maintains two global dynamic sets $\mathcal{F}, \mathcal{FA}$. The set $\mathcal{F}$ contains processes that z detected as faulty, and $\mathcal{FA}$ contains processes that z knows are detected by all correct processes. The protocol for updating $\mathcal{F}, \mathcal{FA}$ is straightforward:

- In each round the processes exchange their $\mathcal{F}$ lists and update their $\mathcal{F}$ and $\mathcal{FA}$ sets once a faulty process appears in $t + 1$ or $2t + 1$ lists, respectively.
- When a process is detected as faulty every correct process masks its future messages to $\perp$.

The basic EIG protocol will be invoked repeatedly, and several copies of the EIG protocol may be running concurrently. The accumulated set of faulty processes will be used across all copies (the rest of the variables and data structures are local to each EIG invocation). Therefore, we assume that when the protocol is invoked the following property holds:

PROPERTY 1. *When the protocol is invoked, no correct process appears in the faulty sets of any other correct process. Moreover, $\mathcal{FA}_p \subseteq \mathcal{F}_p$ and $\mathcal{FA}_p \subseteq \mathcal{F}_q$ for any two correct processes p and q,*

Each invocation of the EIG protocol is tagged with a parameter ϕ, known to all processes. An EIG protocol with parameter ϕ, will run for at most $\phi+1$ rounds. At the beginning of the agreement protocol the faulty sets are empty at all correct processes and the EIG protocol with parameter $\phi = t$ is executed. Each additional invocation of the EIG protocol is with a smaller value of ϕ. In the non-trivial case, when the EIG protocol with parameter ϕ is invoked then $|\bigcup_i \mathcal{FA}_i| \geq t - \phi$. There will be one exception to this assumption, and it is handled in Lemma 1. Thus, other than in that specific case, it is assumed that we have a (t, ϕ)-adversary during the execution of the EIG protocol with parameter ϕ.

The basic EIG protocol for a correct process z with initial value $d_z \in D$ is very simple:

1. **Init:** Set $\mathcal{IT}(\bar{\epsilon}) := d_z$, so $\mathcal{IT}(\bar{\epsilon})$ is set to be the initial value.
2. **Send:** in each round r, $1 \leq r \leq \phi + 1$, for every $\sigma \in \mathcal{IT} \cap \Sigma_{r-1}$, such that $z \notin \sigma$, send the message $\langle \sigma, z, \mathcal{IT}(\sigma) \rangle$ to every process.
3. **Receive set:** in each round r, let $\mathcal{S}_r := \{\sigma x \in \Sigma_r\}$.
4. **Receive rule:** in each round r, for all $\sigma x \in \mathcal{S}_r$ set

$$\mathcal{IT}(\sigma x) := \begin{cases} \perp & \text{if } x \in \mathcal{F} \\ d & \text{if } x \notin \mathcal{F} \text{ sent } \langle \sigma, x, d \rangle \text{ and } d \in D; \\ \mathcal{IT}(\sigma) & \text{otherwise.} \end{cases}$$

Note: assigning of $\mathcal{IT}(\sigma x) := \mathcal{IT}(\sigma)$ when $x \notin \mathcal{F}$ is crucial for the case where x is correct and has halted in the previous round. Thus, if a process is silent but is not detected (possibly because it has halted due to early stopping) z assigns it the value it heard in the previous round.

We use a second dynamic EIG tree data structure $\mathcal{RT}$. Intuitively, if a process puts a value in a node of this tree then, essentially, all correct processes will put the same value in the same node in at most 2 more rounds. Processes use several rules to close branches of the $\mathcal{IT}$ tree whose value in $\mathcal{RT}$ is already determined by all. We present later the rules for closing branches of the $\mathcal{IT}$ tree. To handle this, we modify lines 2 and 3 as described below (and keep lines 1 and 4 as above).

2. **Send:** in each round r, $1 \leq r \leq \phi + 1$, for every $\sigma \in \mathcal{IT} \cap \Sigma_{r-1}$, such that $z \notin \sigma$, and the branch σ is not closed send the message $\langle \sigma, z, \mathcal{IT}(\sigma) \rangle$ to every process.
3. **Receive set:** in each round r, let $\mathcal{S}_r = \{\sigma x \in \Sigma_r \mid$ branch σx is not closed$\}$.

Informally, $\mathcal{IT}_z(\sigma p) = d$ (where $\mathcal{IT}_z$ denotes the $\mathcal{IT}$ tree at process z) indicates that process z received a message from process p that said that his value for σ was d. $\mathcal{RT}_z(\sigma p) = d$ indicates, essentially, that process z knows that every correct process x will agree and have $d \in \mathcal{RT}_x(\sigma p)$ in at most two more rounds.

Observe that we record in the EIG tree only information from sequences of nodes that do not contain repetition, therefore, not every message a process receives will be recorded.

At the end of each round, we apply the rules below to determine whether to assign values to nodes in $\mathcal{RT}$, assigning that value in $\mathcal{RT}$ is called *resolving* the node.

2.1 The Resolve Rules

A key feature of our algorithm is that whenever we put a value into $\mathcal{RT}(\sigma)$ we also color (assign) all the descendants of σ in $\mathcal{RT}$ with the same value. Observe that this means we may color a node σw in $\mathcal{RT}$ to d even if w is correct and sent $d' \neq d$ to all other correct processes.

Rules for IT-to-RT resolve: The following definitions and rules cause a node to be resolved based on information in $\mathcal{IT}$.

1. If $\mathcal{IT}(\sigma w) = d$ **then** we say: (1) w is a voter of (σ, w, d); (2) w is confirmed on (σ, w, d); (3) For all $v \in N \setminus \{\sigma\}$, w is a supporter of v on (σ, w, d).

   Note: the reason that we count w as a voter, as confirmed and as a supporter for all its echoers is that due to the EIG structure w does not appear in the subtree of σw.

2. If $\mathcal{IT}(\sigma wv) = d$, **then** we say that v is a supporter of v for (σ, w, d).

   Note: again we need v to be a supporter of itself because of the EIG structure.

3. If $\mathcal{IT}(\sigma wvu) = d$ **then** we say that u is a supporter of v for (σ, w, d).

4. If there is a set $|U| = n - t$, such that for each $u' \in U$, u' is a supporter of v on (σ, w, d) **then** we say that v is confirmed on (σ, w, d).

   Note: if σ contains no correct and w is correct, then any correct child v (of σw) will indeed have $n - t$ supporters for σw and hence will be confirmed. Note that one supporter is w, the other is v and the remaining are all the $n - t - 2$ correct children of σwv. Also note that w is confirmed, so all $n - t$ correct will be confirmed on (σ, w, d).

5. If $u \neq w$ has a set $|V| = n - t$, such that for each $v' \in V$, u is a supporter of v' on (σ, w, d) and v' is confirmed on (σ, w, d) **then** u is a voter of (σ, w, d).

   Note: this is somewhat similar to the notion of a Voter in grade-cast ([FM97, FM88]). But there is a crucial difference: all the $n - t$ echoers need to be *confirmed*. Also note that w is a voter for itself.

6. IT-TO-RT RULE: If w has a set $|U| = n - t$, such that for each $u' \in U$, u' is a voter of (σ, w, d) **then** if $\sigma w \notin \mathcal{RT}$, then put $\mathcal{RT}(\sigma w) := d$ and color descendants of σw with d as well.

   Note: this is somewhat similar to the notion of a grade 2 in grade-cast. A crucial difference is that the $n - t$ voters needed are defined with respect to *supported* echoers. This is a non-trivial change that breaks the standard grade-cast properties. Also note that we not only put a value in σw but also color all the descendants.

7. ROUND $\phi + 1$ RULE: if $\mathcal{IT}(\sigma w) = d$ and $\sigma \in \Sigma_t$ **then** if $\sigma w \notin \mathcal{RT}$, then put $\mathcal{RT}(\sigma w) := d$.

   Note: this is a standard rule to deal with the last round.

Rules for $\mathcal{RT}$ tree resolve: The following definitions and rules cause a node to be resolved based only on information in $\mathcal{RT}$ (these rules do not look at $\mathcal{IT}$).

1. If there is a set $|U| = t + 1$, such that for each $u' \in U$, $\mathcal{RT}(\sigma wvu') = d$ **then** we say v is $\mathcal{RT}$-confirmed on (σ, w, d).

   Note: if any correct sees a node as confirmed then it has $n - t$ that echo its value. At least $t + 1$ of them are correct and they all cause all correct to see the node as $\mathcal{RT}$-confirmed. Of course a node may become $\mathcal{RT}$-confirmed even if it was never confirmed by any correct. Observe that if $\mathcal{RT}(\sigma wu) = d$ then, by coloring, u is $\mathcal{RT}$-confirmed on (σ, w, d).

2. If $u \neq w$ has a set $|V| = n - t$, such that each $v' \in V$ is $\mathcal{RT}$-confirmed on (σ, w, d) and for each $v' \in V \setminus \{u\}$, $\mathcal{RT}(\sigma wv'u) = d$ and if $u \in V$ then also $\mathcal{RT}(\sigma wu) = d$, **then** u is $\mathcal{RT}$-voter of (σ, w, d).

   Note: if any correct process sees a node as a voter then it has $n-t$ echoers that are confirmed. So each of these $n - t$ echoers will be $\mathcal{RT}$-confirmed. So all correct processes will see this node as $\mathcal{RT}$-voter. Of course a node can become $\mathcal{RT}$-voter even if it was never a voter at any correct process.

3. RESOLVE RULE: If w has a set $|U| = t+1$, such that for each $u' \in U$, u' is a $\mathcal{RT}$-voter of (σ, w, d) **then** if $\sigma w \notin \mathcal{RT}$, then put $\mathcal{RT}(\sigma w) := d$, and color descendants of σw with d as well. The rule applies also for node $\sigma w = \bar{\epsilon}$.

   Note: if any correct process does IT-TO-RT RULE then this rule tries to guarantee that all correct processes will also put this node in $\mathcal{RT}$. The problem is that SPECIAL-BOT RULE (see below) may be applied to one of the echoers and this may cause some of the $\mathcal{RT}$-voters to lose their required support. The following rule fixes this situation. It reduces the threshold to $n - t - 1$ but requires that all children nodes are fixed.

4. RELAXED RULE: If all the children of σw are in $\mathcal{RT}$ (*i.e.,*$\forall \sigma wv \in \Sigma$: $\sigma wv \in \mathcal{RT}$) and exists a set $|V| = n - t - 1$, such that for each $v' \in V$, $\mathcal{RT}(\sigma wv') = d$, **then** if $\sigma w \notin \mathcal{RT}$, then put $\mathcal{RT}(\sigma w) := d$, and color descendants of σw with d as well. The rule applies only for nodes $|\sigma w| \geq 1$.

   Note: as mentioned above, the RELAXED RULE requires a threshold of $n - t - 1$ so that it can take into account the possibility of one value changing to $\perp$ due to the following rule:

5. SPECIAL-BOT RULE: If there is a set $|V| = t + 2 - |\sigma wu|$ such that for all $v \in V$, $\mathcal{RT}(\sigma wuv) = \perp$ and for all $u' \neq u$ such that $\sigma wu' \in \Sigma$, $\sigma wu' \in \mathcal{RT}$ **then** if $\sigma wu \notin \mathcal{RT}$, then put $\mathcal{RT}(\sigma wu) := \perp$, and color descendants of σwu with $\perp$ as well. The rule applies only for $|\sigma wu| \geq 2$.

   Note: This rule can be applied to at most one child.

6. SPECIAL-ROOT-BOT RULE: If exists a set $|U| = t+1$ such that for each $u \in U$, $\mathcal{RT}(u) = \perp$ then if $\bar{\epsilon} \notin \mathcal{RT}$, then put $\mathcal{RT}(\bar{\epsilon}) := \perp$, and color descendants of $\bar{\epsilon}$ with $\perp$ as well.

   Note: this rule is important in order to stop quickly if $t+1$ correct processes start with the value $\perp$.

To prevent the data structures from expanding too much processes close branches of the tree, and from that point on they do not send messages related to the closed branches. We use the notation $\{\sigma \in \mathcal{RT}[r]\}$ to denote an indicator variable that equals true if $\mathcal{RT}(\sigma)$ was assigned some value by the end of round r, and false otherwise.

Branch Closing and Early Resolve rules: There are three rules to close a branch in $\mathcal{IT}$ two of them also trigger an early resolve. By the end of round r, $r \leq \phi$,

1. DECAY RULE: if $\exists \sigma' \sqsubseteq \sigma$ such that $\sigma' \in \mathcal{RT}[r-1]$, then close the branch $\sigma \in \mathcal{IT}$.

   Note: this is the simple case: if a process already fixed the value of σ' in $\mathcal{RT}$ in round $r-1$ then it stops in the end of round r, since by the end of round $r+1$ all correct processes will put σ' in $\mathcal{RT}$ (and will interpret this process's silence in the right way during round $r+1$). There is no need to continue. Coloring will fix all the values of this subtree.

2. EARLY_IT-TO-RT RULE: if $\sigma \in \Sigma_{\mathsf{r}-1}$ and exists $U \subseteq N$, $U \cap \{u' \mid u' \in \sigma\} = \emptyset$, $|U| = n - r$, such that for every $u, v \in U \setminus \mathcal{F}$, $\mathcal{IT}(\sigma u) = \mathcal{IT}(\sigma v)$, then if $\sigma \notin \mathcal{RT}$, then put $\mathcal{RT}(\sigma) := \mathcal{IT}(\sigma)$ and close the branch $\sigma \in \mathcal{IT}$.

   Note: this is a case where the process can forecast that all correct processes will put σ in $\mathcal{RT}$ in the next round (because the process sees that all children nodes agree). So the process can fix σ in this round and stop now, because all correct processes will fix σ in $\mathcal{RT}$ next round (and will interpret this process's silence in the right way).

3. STRONG_IT-TO-RT RULE: if $\sigma \in \Sigma_{\mathsf{r}-2}$ and exists $U \subseteq N$, $U \cap \{u' \mid u' \in \sigma\} = \emptyset$, $|U| = n - r + 1$ such that for every $u, v \in U \setminus \mathcal{F}$, where $v \neq u$, $\mathcal{IT}(\sigma uv) = \mathcal{IT}(\sigma vu)$ then, if $\sigma \notin \mathcal{RT}$, then put $\mathcal{RT}(\sigma) := \mathcal{IT}(\sigma)$ and close the branch $\sigma \in \mathcal{IT}$.

   Note: in this case all the correct children of σ except for at most one will be fixed in the next round to the same value, so the RELAXED RULE will be applied to σ in the next round. So we can fix σ in this round and stop now.

In each round all the above rules are applied repeatedly until none holds any more.

The rules above imply that there are two ways to give a value to a node in $\mathcal{RT}$. One is assigning it a value using the various rules, and the other is coloring it as a result of assigning a value to one of its predecessors. We will use the term *color* for the second one and the term *put* for the first one.

Rules for fault detection and masking: The following definitions and rules are used to detect faulty processes, put them into $\mathcal{F}$ and hence mask them (all messages from $\mathcal{F}$ are masked to $\perp$). The last rule also defines an additional masking. The process first updates its $\mathcal{F}$ and $\mathcal{FA}$ sets using the sets received from the other processes during the current round. A process is added to $\mathcal{F}$ or $\mathcal{FA}$ once it appears in $t+1$ or $2t+1$ sets, respectively. Next the process applies the following fault detection rules. The fault detection is executed before applying any of the resolve rules above. When a new process is added to $\mathcal{F}$, the new masking is applied and the fault detection is repeated until no new process can be added. Only then the resolve rules above are applied.

At process z by the end of round r:

1. **Not Voter**: If $\exists \sigma w \in \Sigma_{\mathsf{r}-1}$ and $w \neq z$ and $\nexists \sigma' \sqsubseteq \sigma w$ such that $\sigma' \in \mathcal{RT}$ and it is not the case that there exists a set $|U| = n - t - 1$ such that for each $u' \in U$, $\mathcal{IT}(\sigma w u') = \mathcal{IT}(\sigma w)$ **then** add w to $\mathcal{F}$.

   Note: this is the standard detection rule after one round - if anything looks suspicions then detect.

2. **Not IT-to-RT**: If $\exists \sigma w \in \Sigma_{\mathsf{r}-2}$ for which w does not have a set $|U| = n - t$, such that for each $u' \in U$, u' is a voter of (σ, w, d), and $\nexists \sigma' \sqsubseteq \sigma$ such that $\sigma' \in \mathcal{RT}$ **then** add w to $\mathcal{F}$.

   Note: this is the standard detection rule after two rounds - if anything looks suspicions then detect.

3. If u, $u \neq w$, has a set $|V| = n - t$, such that for each $v' \in V$, u is a supporter of v' on (σ, w, d) **then** we say that u is an *unconfirmed voter* of (σ, w, d).

   Note: the notion of an *unconfirmed voter* is exactly that of a voter in the standard grade-cast protocol.

4. If w has a set $|U| = t+1$, such that for each $u' \in U$, u' is an unconfirmed voter of (σ, w, d) **then** we say that σw is *leaning towards* d.

   Note: the notion of *leaning towards* is exactly that of getting grade ≥ 1 in the standard grade-cast protocol.

5. **Not Masking**: If $\sigma w \in \Sigma_{\mathsf{r}-3}$ is leaning towards d and there exists u, $|V| = t+1$, and $d' \neq d$ such that for each $v' \in V$, $\mathcal{IT}(\sigma w u v') = d'$ and there exists $|\sigma''| > |\sigma|$ such that $\mathcal{IT}(\sigma'' w u) \neq \perp$ then
   (a) $\mathcal{IT}(\sigma'' w u) = \perp$;
   (b) if by the end of the round $\nexists \sigma' \sqsubseteq \sigma'' w$ such that $\sigma' \in \mathcal{RT}$ **then** add u to $\mathcal{F}$.

   Note: If σw is leaning towards d then u must have heard at least $t+1$ say d on σw. If $t+1$ say u said d' then u must have said d' to some correct. So u must have received d' from σw but in the next round u hears $t+1$ say σw said d. So u must conclude that w is faulty and u must mask him from the next round. If u did not mask some $\sigma'' w u$ then the **Not Masking** rule will detect u as faulty and mask all such $\sigma'' w u$ for you and also mark you as faulty. The reason we wait until the end of the round to add that node to $\mathcal{F}$ is that it might be a node of a correct process that stopped in the previous round and hence did not send any messages in the current round, and therefore did not send masking. In such a case we mask its virtual sending, but do not add it to $\mathcal{F}$.

Finalized Output: By the end of each round (after applying all the resolve rules), the process checks whether there

is a frontier in $\mathcal{RT}$. A *frontier* (also called a cut) is said to exist if for all $\sigma \in \Sigma_{\square+1}$ there exists some sub-sequence $\sigma' \sqsubseteq \sigma$ such that $\sigma' \in \mathcal{RT}$.

1. **Early Output rule**: By the end of a round, if $\bar{\epsilon} \in \mathcal{RT}$, **output** $\mathcal{RT}(\bar{\epsilon})$.
2. **Final Output rule**: Otherwise, if there is a frontier, **output** $\perp$.

Observe that the existence of a frontier can be tested from the current $\mathcal{IT}$ in $O(|\mathcal{IT}|)$ time.

Stopping rule: If all branches of IT are closed, stop the protocol.

3. THE CONSENSUS PROTOCOL ANALYSIS

The EIG protocol implicitly presented in the previous section is a consensus protocol $\mathcal{D}_\square$, where ϕ, $1 \leq \phi \leq t$ is a parameter. Protocol $\mathcal{D}_\square$ runs for at most $\phi+1$ rounds and solves Byzantine agreement against a (t,ϕ)-adversary. Denote by G the set of correct processes, $|G| \geq n-t$, where $n = |N|$, and by S, $S = \bigcap_{\mathsf{q}\in G} \mathcal{F}_\mathsf{q}$, the set of processes that are masked to $\perp$ by all correct processes. Let $s := |S|$.

Our solution invokes several copies of the EIG protocol. For each invoked protocol, $\mathcal{D}_\square$, there are two cases: either $s \geq t-\phi$, or we are guaranteed that the input of all correct processes that start the protocol is the same (in particular, it may be that some correct processes have halted and do not start the protocol). The following lemma deals with this latter case.

LEMMA 1. *[Validity and Fast Termination] For any (t,t)-adversary, and $n \geq 3t+1$,*

1. *if every correct process that starts the protocol holds the same input value d then d is the output value of all correct processes that start the protocol, by the end of round 2, and all of them complete the protocol by the end of round 3.*
2. *if all correct processes start the protocol and $t+1$ correct processes start with $\perp$ then all correct processes output $\perp$ by the end of round 3 and stop the protocol by the end of round 4.*
3. *For $p,q \in G$, no p will add q to $\mathcal{F}_\mathsf{p}$ in either of the above cases.*

The only case in which not all correct processes invoke a $\mathcal{D}_\square$ protocol is when some of the background running monitors are being invoked by some of the correct processes, while others may have already stopped. This special case is guaranteed to be when the inputs of all participating correct processes is $\perp$, and consensus can be still be achieved. Lemma 1 implies the following:

COROLLARY 1. *For any (t,t)-adversary, and $n \geq 3t+1$, if every correct process that invokes the protocol start with input $\perp$, then $\perp$ is the output value at each participating correct process by the end of round 2, and each participating correct process completes the protocol by the end of round 3. Moreover, for $p,q \in G$, no p will add q to $\mathcal{F}_\mathsf{p}$.*

The gossip exchange among correct processes about identified faults ensures the following:

LEMMA 2. *For a (t,ϕ)-adversary and protocol $\mathcal{D}_\square$, $n \geq 3t+1$, assuming Property 1, for any k, $1 \leq k \leq \phi+1$, by the end of round k, for every two correct processes p,q, $\mathcal{FA}_\mathsf{p} \subseteq \mathcal{F}_\mathsf{q}$ and $\mathcal{FA}_\mathsf{p}[k-1] \subseteq \mathcal{FA}_\mathsf{p}[k]$.*

A node may initially assign a value using one of the "put" rules and later it may color it to a different value. In the arguments below we sometimes need to refer to the value that was put to a node rather than the value it might be colored to. Once a node has a value it is not assigned a value using any put rule any more. Thus, the value assigned using a put rule is an initial value that may be assigned to a node before it is colored, or that node may never have a value put to it. To focus on these put operations, we will add, for proof purposes, that whenever a node p uses a put rule for some σ, except ROUND $\phi+1$ RULE, it also puts σ in $\mathcal{PT}_\mathsf{p}$ (The "Put-Tree") and as a result at that moment, $\mathcal{PT}_\mathsf{p}(\sigma) = \mathcal{RT}_\mathsf{p}(\sigma)$. We do not color nodes in $\mathcal{PT}_\mathsf{p}$, thus for σ that is colored, but was not assigned a value prior to that, $\mathcal{PT}_\mathsf{p}(\sigma)$ is undefined. We exclude ROUND $\phi+1$ RULE from $\mathcal{PT}$ on purpose.

The following is the core statement of the technical properties of the protocol. The only way we found to prove all these is via an induction argument that proves all properties together. The theorem contains four items.

The detection part proves that correct processes are never suspected as faulty. The challenge is that the various rules instruct processes when to stop sending messages, and that might cause other correct processes to be suspected as faulty.

The validity part proves that if a correct process sends a value, it will reach the $\mathcal{RT}$ of every other correct process within two rounds. It also proves that if a correct process decides not to send a value (thus, closed a branch), the appropriate node will be in $\mathcal{RT}$ of every correct process. The third claim in the validity part is that if a process appears in $\mathcal{FA}$, then it appears in $\mathcal{RT}$ of every correct process within two rounds.

The safety part intends to prove consistency in the $\mathcal{RT}$. The challenge is that coloring may cause the trees of correct processes to defer. Therefore the careful statements looks at $\mathcal{PT}$, and which rule was used in order to assign the value to it. The $\perp$ value is a default value, therefore there is a special consideration of whether the value the process puts is $\perp$ or not. The end result is that if a node appears in $\mathcal{PT}$ of two correct processes, it carries the same value.

The liveness part shows that if a node appears in $\mathcal{RT}$ of a correct process, it will appear in $\mathcal{RT}$ of any other correct process within two rounds.

THEOREM 2. *For a (t,ϕ)-adversary and protocol $\mathcal{D}_\square$, $n \geq 3t+1$, assuming Property 1 and that all correct processes participate in the protocol, then for any $1 \leq k \leq \phi+1$:*

1. **No False Detection:** *For $p,q \in G$, no q will add p to $\mathcal{F}_\mathsf{q}$ in round k.*
2. **Validity:**
   (a) *For $\sigma \in \Sigma_{\mathsf{k}-3}$ if $p \in G$, sends $\langle \sigma, p, d_\mathsf{p} \rangle$, then at the end of round k, at every correct process x, either $\mathcal{RT}_\mathsf{x}(\sigma p) = d_\mathsf{p}$ or $\exists \sigma' \prec \sigma$ such that $\sigma' \in \mathcal{RT}_\mathsf{x}$. For $k = \phi+1$, the property holds also for any $\sigma \in \Sigma_{\mathsf{k}-2}$ and for any $\sigma \in \Sigma_{\mathsf{k}-1}$.*
   (b) *If $z \in \mathcal{FA}$ in the beginning of round $k-2$, then by the end of round k, at every correct process, either $\mathcal{RT}(\sigma z) = \perp$ or $\exists \sigma' \prec \sigma$ such that $\sigma' \in \mathcal{RT}$. For*

$k = \phi + 1$, *the property holds for* $z \in \mathcal{FA}$ *in the beginning of rounds* $k-1$ *or* k.

(c) *For* $\sigma \in \Sigma_{k-1}$, *if* $p \in G$, *does not send* $\langle \sigma, p, d \rangle$ *for any* $d \in D$, *then at the end of round* k, *at every correct process* x, $\exists \sigma' \prec \sigma$ *such that* $\sigma' \in \mathcal{RT}_x$.

3. **Safety:** *For* $p, q \in G$, $x \in N$, $|\sigma x| \leq \phi$,$\sigma x \in \mathcal{PT}_p[k]$, *then*

(a) *if* p *applies* RESOLVE RULE *to put* $\mathcal{PT}_p(\sigma x) = d$, $d \neq \perp$, *and* v *is one of the* $\mathcal{RT}$*-confirmed nodes on* (σ, x, d) *in* $\mathcal{RT}_p$ *used in applying this rule in* $\mathcal{RT}_p$, *and in addition* $\mathcal{PT}_q(\sigma x v) = \perp$, *then* q *applied* SPECIAL-BOT RULE *to put* $\sigma x v$;

(b) *if* $|\sigma x| \geq 1$ *and* $\mathcal{PT}_p(\sigma x) = d$, $d \neq \perp$, *then, by the end of round* k, $|V_q| \leq t$, *where* $V_q = \{u \mid \mathcal{PT}_q(\sigma x u) = \perp\}$;

(c) *if* $|\sigma x| \geq 1$ *and* $\mathcal{PT}_p(\sigma x) = \perp$ *and it wasn't put using* SPECIAL-BOT RULE, *then, by the end of round* k, $|V_q| \leq t$, *where* $V_q = \{u \mid \mathcal{PT}_q(\sigma x u) \neq \perp\}$;

(d) *if* $\sigma x \in \mathcal{PT}_q[k]$, *then* $\mathcal{PT}_p(\sigma x) = \mathcal{PT}_q(\sigma x)$.

4. **Liveness:** *For* $p, q \in G$, *if* $\sigma \in \mathcal{RT}_p[k-2]$ *then* $\sigma \in \mathcal{RT}_q[k]$. *For* $k = \phi + 1$, *if* $\sigma \in \mathcal{RT}_p$ *then* $\sigma \in \mathcal{RT}_q$.

The following Theorem summarizes the properties needed from our protocol.

THEOREM 3. *For a* (t, ϕ)*-adversary and protocol* $\mathcal{D}_\square$ *and* $n \geq 3t+1$ *and assuming that all correct processes participate in the protocol:*

1. *Every correct process outputs the same value.*
2. *If the input values of all correct processes are the same, this is the output value. Every correct process outputs it by round* 2 *and stops by round* 3.
3. *If* $t+1$ *of the correct processes hold an input value of* $\perp$, *then all correct processes output* $\perp$ *by the end of round* 3 *and stop by the end of round* 4.
4. *If the actual number of faults is* $f_\square < \phi$, *then all correct processes complete the protocol by the end of round* $f_\square + 2$.
5. *If the actual number of faults is* $f_\square = 0$, *and all correct processes start with the same initial value, then all correct processes complete the protocol by the end of round* 1.
6. *If the actual number of faults is* $f_\square = 1$, *and all correct processes start with the same initial value, then all correct processes complete the protocol by the end of round* 2.
7. *If a correct process outputs in round* k, *it stops by the end of round* $k+1$.
8. *If a correct process stops in the end of round* k, *all correct processes output by round* $k+1$ *and stop by round* $k+2$.

4. MONITORS

We follow the approach of [BG93, GM93, GM98] with some modifications for guaranteeing early stopping.

In round $r = 1$ we run $\mathcal{D}_t$ using the initial values. For each integer k, in round $1 < r = 1 + 4k < t - 1$ we invoke protocol $\mathcal{D}_{t-1-4k}$ whose initial values is either $\perp$ (meaning everything is OK) or BAD (meaning that too many corrupt processes were detected). We call this sequence of protocols the *basic monitor sequence*. We will actually run 4 such sequences.

4.1 The Basic Monitor Protocol

Each process z stores two variables: $v \in D$, the current value, and *early*, a boolean value. Initially v equals the initial input of process z and $early := false$. Later, $early = true$ will be an indicator that the next decision protocol must decide $\perp$ (because there is not enough support for BAD). Each process remembers the last value of $early_q$ it received from every other process q, even if q did not send one recently.

Throughout this section we use the notation: $\bar{r} \equiv r \pmod 4$.

Algorithm 1: The Basic Monitor protocol (at process z)

```
1:  if r̄ = 1:
2:      if r < t − 1 then invoke protocol D_{t+1−r} with
        initial value v_z;
3:  if r̄ = 2:
4:      at the end of the round:
5:          if |FA| ≥ r + 3 then set v_z := BAD
6:              otherwise set v_z := ⊥;
7:  if r̄ = 3:
8:      send v_z to all;
9:      at the end of the round:
10:         if |{q | v_q = BAD}| ≤ t then set early_z := true
11:             otherwise set early_z := false;
12: if r̄ = 0:
13:     send early_z to all;
14:     at the end of the round:
15:         if |{q | early_q = true}| ≥ t+1 then set v_z := ⊥;
16:         if every previously invoked protocol produced
        an output then set v_z := ⊥.
```

The monitor protocol runs in the background until the process halts. The monitor protocol invokes a new $\mathcal{D}_\square$ protocol every 4 rounds. In each round, the monitor's lines of code are executed before running all the other protocols, and its end of round lines of code are executed before ending the current round in all currently running protocols. This is important, since it needs to detect, for example, whether all currently running protocols produced outputs for determining its variable for the next round. At the end of each round the monitor protocol applies the monitor_halting and monitor_decision rules below to determine whether to halt all the running protocols at once, or only to commit to the final decision value.

When a process is instructed to apply a monitor_decision it applies the following definition. If it is instructed to halt (monitor_halting), then if it did not previously apply the monitor_decision, it applies monitor_decision first and then halts all currently running protocols that were invoked by the monitor at once.

DEFINITION 1 (MONITOR_DECISION). *A process that did not previously decide,* **decides** BAD, *if any previously invoked protocol outputs* BAD. *Otherwise, it decides on the output of* $\mathcal{D}_t$.

When a process is instructed to decide without halting, it may need to continue running all protocols for few more rounds to help others to decide. We define "halt by $r + x$" to mean continue to run all active protocols until the end of round $\min\{r + x, t + 1\}$, unless an halt is issued earlier.

4.2 Monitor Halting and Decision Conditions

Given that different processes may end various invocations of the protocols in different rounds we need a rule to make sure that all running protocols end by the end of round $f+2$. The challenge in stopping all protocols by the end of $f+2$ is the fact that individual protocols may end at round $f+2$ and we do not have a room to exchange extra messages among the processes. This also implies that we need to have a halting rule at every round of the monitor protocol, since $f+2$ may occur at any round.

Each halting rule implies how other rules need to be enforced in later rounds, since any process may be the first to apply a monitor_halting at a given round and we need to ensure that for every extension of the protocols, until everyone decides, all will reach the same decision despite the fact that those that have halted are not participating any more. The conditions take into account processes that may have halted. A process considers another one as halted if it doesn't receive any message from it in any of the concurrently running set of invoked protocols, monitors and the gossiping of $\mathcal{F}$.

To achieve that we add the following set of rules.

Monitor Halting Rules:

H_{BAD}. Apply monitor_halting if any monitor stops with output BAD. Otherwise if any monitor outputs BAD, apply monitor_decision now and monitor_halting by $r+2$.

H_1. Case $\bar{r}=1$:

(a) If all previously invoked protocols stopped, apply monitor_halting.

(b) Otherwise, if only the latest invoked protocol did not stop and $|\{q \mid early_{\mathsf{q}} = true \text{ or } q \text{ halted}\}| \geq n-t$, then apply monitor_halting.

(c) Otherwise, if only the latest invoked protocol did not stop and $|\{q \mid early_{\mathsf{q}} = true \text{ or } q \text{ halted}\}| \geq t+1$, then apply monitor_decision now and monitor_halting by $r+2$.

H_2. Case $\bar{r}=2$:

(a) If all previously invoked protocols stopped, apply monitor_halting.

(b) Otherwise, if only the latest invoked protocol did not stop and $|\{q \mid early_{\mathsf{q}} = true \text{ or } q \text{ halted}\}| \geq n-t$ was true in the previous round, then apply monitor_halting.

(c) Otherwise, if only the latest invoked protocol did not stop and $|\{q \mid early_{\mathsf{q}} = true \text{ or } q \text{ halted}\}| \geq t+1$ was true in the previous round, then apply monitor_decision and now and monitor_halting by $r+1$.

H_3. Case $\bar{r}=3$: If all previously invoked protocols stopped, apply monitor_halting.

H_4. Case $\bar{r}=0$: If all previously invoked protocols stopped and $|\{q \mid early_{\mathsf{q}} = true \text{ or } q \text{ halted}\}| \geq n-t$ then apply monitor_halting.

LEMMA 3. *If $n > 3t$ and there are f, $f \leq t$, corrupt processes then all correct processes apply monitor_halting by the end of round* $\min(t+1, f+2)$.

LEMMA 4. *If the first process applies monitor_halting in round r on d then every correct process applies monitor_decision by round* $\min\{r+4, f+2, t+1\}$*, applies monitor_halting by round* $\min\{r+5, f+2, t+1\}$*, and obtains the same decision value, d.*

Lemma 3 and 4 complete the correctness part of Theorem 1. To simplify the polynomial considerations we look at a pipeline of monitors.

4.3 Monitors Pipeline

The basic monitor protocol runs a sequence of monitors and tests the number of faults' threshold every 4 rounds (Line 5). This allows the adversary to expose more faults in the following round, and be able to further expand the tree before the threshold is noticed the next time the processes execute Line 5. To circumvent this we will run a pipeline of 3 additional sequences of monitors on top of the basic one appearing above. Doing this we obtain that in every round r one of the 4 monitor sequences will be testing the threshold on the number of faults

Monitor sequence i, for $1 \leq i \leq 4$ begins in round i and invokes protocols every 4 rounds, in every round r, $1 < r = i + 4k < t-1$, it invokes protocol $\mathcal{D}_{\mathsf{t}-\mathsf{i}-4\mathsf{k}}$. Monitor sequence 1 is the basic monitor sequence defined in the previous subsection. Each monitor sequence independently runs the basic monitor protocol (Figure 1) every 4 rounds. In the monitor protocol, the test $\bar{r}=j$, which stands for $\bar{r} \equiv r \pmod 4$ in the basic monitor sequence, is replaced with $\bar{r_{\mathsf{i}}} = j$, which stands for $\bar{r_{\mathsf{i}}} \equiv r+1-i \pmod 4 = j$ (naturally only for $r+1-i > 0$). Each of the four monitor sequences decides and halts separately, as in the previous section above.

Notice that protocol $\mathcal{D}_{\mathsf{t}}$ is invoked only by the basic sequence (Sequence 1). For each of the three other monitor sequences, the decision rule is: decide BAD, if any invoked protocol (in this sequence) outputs BAD, and $\perp$ otherwise. Observe that Lemma 3 and 4 hold for each individual sequence.

We now state the global decision and global halting rules:

DEFINITION 2 (GLOBAL HALTING). *If any monitor sequence halts with* BAD*, or all 4 monitor sequences halt, the process halts.*

DEFINITION 3. *The global_decision is the output of $\mathcal{D}_{\mathsf{t}}$, unless any monitor sequence returns* BAD*, in which case the decision is* BAD.

The following are immediate consequences of Lemma 3 and 4 and the above definitions.

COROLLARY 2. *If $n > 3t$ and there are f, $f \leq t$, corrupt processes then all correct processes halt by the end of round* $\min(t+1, f+2)$.

COROLLARY 3. *If the first correct process halts in round r on d then every correct process applies global_decision by round* $\min\{r+4, f+2, t+1\}$*, halts by round* $\min\{r+5, f+2, t+1\}$*, and obtains the same decision value.*

5. BOUNDING THE SIZE OF THE TREE

Following the approach of [GM98], we make the following definitions:

DEFINITION 4. *A node $\sigma z \in \Sigma$ is* fully corrupt *if there does not exist $p \in G$ and $\sigma' \sqsupseteq \sigma z$ such that $\sigma' \in \mathcal{RT}_{\mathsf{p}}[|\sigma z|+2]$.*

DEFINITION 5. *We say that* a process z becomes fully corrupt at i *if exists a node $\sigma z \in \Sigma$ that is fully corrupt, $|\sigma z| = i$*

and for every previous node $|\sigma' z| < i$, node $\sigma' z$ is not fully corrupt.

The following is immediate from the definitions above.

CLAIM 1. *If process z becomes fully corrupt at i then of all the nodes of Σ that end with z only nodes of round i and $i+1$ can be fully corrupt.*

PROOF. By definition of fully corrupt, all correct processes will have $z \in \mathcal{F}$ in round $i+2$. So in that round and later all nodes will put $\perp$ in $\mathcal{RT}$ for z. □

Let $\mathcal{CT}$, the *corrupt tree*, be a dynamic tree structure. $\mathcal{CT}$ is the tree of all fully corrupt nodes (note that due to coloring, the set of fully corrupt nodes is indeed a tree). We denote by $\mathcal{CT}[i]$ the state of $\mathcal{CT}$ at the end of round i. By the definition of fully corrupt, at round i we add nodes of length $i-2$ to $\mathcal{CT}$.

We *label* the nodes in $\mathcal{CT}$ as follows: a node $\sigma z \in \mathcal{CT}$ is a *regular* node if process z becomes fully corrupt at $|\sigma z|$ and $\sigma z \in \mathcal{CT}$ is a *special* node if process z becomes fully corrupt at $|\sigma z| - 1$.

Let α_i denote the distinct number of processes that become fully corrupt at round i. For convenience, define $\alpha_0 = 0$ (this technicality is useful in Lemma 7). Let $A = \alpha_0, \alpha_1, \ldots$ be the sequence of counts of process that become fully corrupt in a given execution.

Following the approach of [GM98], we define $waste_i = (\sum_{j \le i} \alpha_j) - i$. So $waste_i$ is the number of processes that became fully corrupt till round i minus i (the round number). The following claim connects $waste_i$ to $\cap_{p \in G} \mathcal{FA}[i+3]_p$ the set of fully detected corrupt processes at round $i+3$.

CLAIM 2. *For any round $4 \le r \le t+1$, and any correct process we have $|\mathcal{FA}[r]| \ge \sum_{j \le r-3} \alpha_j$.*

PROOF. By the definition of z becoming fully corrupt at i, all correct processes will have $z \in \mathcal{F}$ in round $i+2$. Due to the gossiping of $\mathcal{F}$, all correct processes will have $z \in \mathcal{FA}$ in round $i+3$. □

So if $waste_i \ge 6$ then in round $r = i+3$ we will have $\sum_{j \le i} \alpha_j - i \ge 6$ so by Lemma 2 for each correct process we have $|\mathcal{FA}[r]| \ge r+3$. In this case all correct processes will start in the associated monitor sequence the next protocol with initial value BAD and the protocol and monitor sequence and global protocol will reach agreement and halt on BAD by round $i+6$ (by Lemma 1).

We will now show that if the adversary maintains a small waste (less than 6 by the argument above, but this will work for any constant) then the $\mathcal{CT}$ tree must remain polynomial sized.

The following key lemma shows that the adversary cannot increase the number of leaves by "cross contamination". In more detail, if the adversary causes two fully corrupt processes at round i_1 followed by a sequence of rounds with exactly one fully corrupt process at each round followed by a round with no fully corrupt process at that round then this action essentially keeps the tree $\mathcal{CT}$ growing at a slow (polynomial) rate. We note that the focus on "cross contamination" follows the approach of [GM98]. But they only verify the case of two fully corrupt followed by a round with no fully corrupt. We have identified a larger family of adversary behavior that does not increase the waste (in the long run). Our proof covers this larger set of behaviors and this requires additional work.

LEMMA 5. *Assume $0 < i_1 < i_2$ such that $\alpha_{i_1} = 2$, $\alpha_{i_2} = 0$ and for all $i_1 < i < i_2$, $\alpha_i = 1$ then for any $\sigma \in \Sigma_{i_1 - 1} \cap \mathcal{CT}$ it is not the case that there exists $\sigma p \tau \in \Sigma_{i_2+1} \cap \mathcal{CT}$ and there exists $\sigma q \tau \in \Sigma_{i_2+1} \cap \mathcal{CT}$ (so there is at most one extension). Moreover the size of the subtree starting from σp or σq and ending in length $i_2 + 1$ is bounded by $O((i_2 - i_1)^2)$.*

See the additional analysis in Section 5.1.

To bound the size of $\mathcal{CT}$, we partition the sequence $A = \alpha_0, \alpha_1, \ldots$ by iteratively marking subsequences using the following procedure. For each subsequence we mark, we prove that it either causes the tree to grow in a controllable manner (so the ending tree is polynomial), or it causes the tree to grow considerably (by a factor of $O(n)$) but at the price of increasing the waste by some positive constant. Since the waste is bounded by a constant, the result follows.

1. By Lemma 7 we know that if A contains a $0(1)^*0$ (a sequence starting with 0 then some 1's then 0) then it contains it just once as a suffix of A. Moreover, this suffix does not increase the size of the tree by more than $O(n)$. Let A_1 be the resulting unmarked sequence after marking such a suffix (if it exists).
2. Mark all subsequences in A_1 of the form $2(1)^*0$ (a sequence starting with 2 then some 1's then 0). By Lemma 5 each such occurrence will not increase the number of leafs in $\mathcal{CT}$ (but may add branches that will close whose total size is at most n^2 over all such sequences). Let A_2 be the remaining unmarked subsequences.
3. Mark all subsequences in A_2 of the form $X(1)^*0$ where $X \in \{3, \ldots, t\}$ (a sequence starting with 3 or a larger number followed by some 1's then 0). By Lemma 8 each occurrence of such a sequence may increase the size of the tree multiplicatively by $O(n)$ leafs and $O(n^2)$ non-leaf nodes, but this also increases the *waste* by $c - 1 > 1$ (where c is the first element of the subsequence). Observe that the remaining unmarked subsequences do not contain any element that equals 0. Let A_3 be the remaining unmarked subsequences.
4. Mark all subsequences of the form $Y(1)^*$ where $Y \in \{2, \ldots, t\}$ (a sequence whose first element is 2 or a larger number followed by some 1's but no zero at the end). Again, by Lemma 8 each such occurrence may increase the size of the tree by $O(n)$ leafs and $O(n^2)$ non-leafs, but this also increases the *waste* by $c > 1$. Let A_4 be the remaining unmarked.
5. Since A_3 contains no element that equals zero and we removed all subsequences that have element of value 2 or larger as the first element then A_4 must either be empty or A_4 is a prefix of A of the form $(1)^*$ (a series of 1's). Since it is a prefix of A then a sequence of 1's keeps at most one leaf. So the tree remains small.

Thus, the size of $\mathcal{CT}$ is polynomial, which by Lemma 6 bounds the size of $\mathcal{IT}$. This completes the proof of Theorem 1.

5.1 Additional Analysis

The following lemma bounds the size of $\mathcal{IT}$ as a function of the size of $\mathcal{CT}$ times $O(n^7)$.

LEMMA 6. *If $\sigma \in \mathcal{IT}$ and $|\sigma| > 7$ then there exists $\sigma' \prec \sigma$ with $|\sigma'| \ge |\sigma| - 7$ such that $\sigma' \in \mathcal{CT}$.*

The following lemma shows that the protocol stops early if the adversary causes two rounds with no new fully corrupt and only one fully corrupt per round between them.

LEMMA 7. *If exists* $0 \leq i_1 < i_2$ *such that* $\alpha_{i_1} = 0$, $\alpha_{i_2} = 0$ *and for all* $i_1 < i < i_2$, $\alpha_i = 1$ *then all processes will halt by the end of round* $i_2 + 5$.

The following lemma shows that having a large number (3 or more) of processes becoming fully corrupt at a given round, followed by a sequence of 1's and then maybe followed by 0 does increase the number of leafs considerably. Note that if $\alpha_{i_1-1} + \alpha_{i_1} \geq 6$ then the monitor process will cause the protocol to reach agreement and stop in a constant number of rounds. So we only look at the case that $\alpha_{i_1-1} + \alpha_{i_1} < 6$.

LEMMA 8. *If* $2 < \alpha_{i_1}$, $\alpha_{i_1-1} + \alpha_{i_1} < 6$, $\alpha_{i_2} \in \{0, 1\}$ *and for all* $i_1 < i < i_2$, $\alpha_i = 1$ *then for any* $\sigma \in \Sigma_{i_1-1} \cap \mathcal{CT}$ *there are at most* $O(i_2 - i_1)$ *nodes of the form* $\sigma\tau \in \Sigma_{i_2+1} \cap \mathcal{CT}$. *Moreover the size of the subtree starting from* σ *and ending in length* $i_2 + 1$ *is bounded by* $O((i_2 - i_1)^2)$.

6. CONCLUSION

In this paper we resolve the problem of the existence of a protocol with polynomial complexity and optimal early stopping and resilience. The main remaining open question is reducing the complexity of such protocols to a low degree polynomial. Another interesting open problem is obtaining unbeatable protocols [CGM14] (which is a stronger notion than early stopping).

We would like to thank Yoram Moses and Juan Garay for insightful discussions and comments.

7. REFERENCES

[BG93] Piotr Berman and Juan A. Garay. Cloture votes: n/4-resilient distributed consensus in t+1 rounds. *Mathematical Systems Theory*, 26(1):3–19, 1993.

[BGP92] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Optimal early stopping in distributed consensus. In Adrian Segall and Shmuel Zaks, editors, *Distributed Algorithms*, volume 647 of *Lecture Notes in Computer Science*, pages 221–237. Springer Berlin Heidelberg, 1992.

[BNDDS92] Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and H. Raymond Strong. Shifting gears: changing algorithms on the fly to expedite byzantine agreement. *Inf. Comput.*, 97:205–233, April 1992.

[CGM14] Armando Castañeda, Yannai A. Gonczarowski, and Yoram Moses. Unbeatable consensus. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2014.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[DRS90] Danny Dolev, Ruediger Reischuk, and H. Raymond Strong. Early stopping in byzantine agreement. *J. ACM*, 37:720–741, October 1990.

[DS82] Danny Dolev and H. Raymond Strong. Polynomial algorithms for multiple processor agreement. In *ACM Symposium on Theory of Computing*, pages 401–407, New York, NY, USA, 1982. ACM.

[FL82] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett.*, 14(4):183–186, 1982.

[FM88] Paul Feldman and Silvio Micali. Optimal algorithms for byzantine agreement. In *ACM Symposium on Theory of Computing*, pages 148–161, 1988.

[FM97] Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM J. Comput.*, 26(4):873–933, 1997.

[GM93] Juan A. Garay and Yoram Moses. Fully polynomial byzantine agreement in t + 1 rounds. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 31–41, New York, NY, USA, 1993. ACM.

[GM98] Juan A. Garay and Yoram Moses. Fully polynomial byzantine agreement for processors in rounds. *SIAM J. Comput.*, 27:247–290, February 1998.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

[KM13] Dariusz R. Kowalski and Achour Mostéfaoui. Synchronous byzantine agreement with nearly a cubic number of communication bits. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 84–91, New York, NY, USA, 2013. ACM.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982.

[PSL80] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

Distributed Resource Discovery in Sub-Logarithmic Time

Bernhard Haeupler*
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
haeupler@cs.cmu.edu

Dahlia Malkhi
VMware Research
Palo Alto, CA 94304, USA
dmalkhi@vmware.com

ABSTRACT

We present a new distributed algorithm for the resource discovery problem introduced by Harchol-Balter, Leighton, and Levin in PODC'99.

The resource discovery problem consists of a synchronous network with n machines in which at any timestep any machine v can PUSH or PULL a message to/from any other machine u whose (IP) address is known to v. Messages can contain addresses which then change the "topology". The goal of a distributed resource discovery problem is to enable all machines to learn the addresses of all other machines as fast as possible while keeping the number of messages sent low.

We present a randomized distributed algorithm that achieves this goal in $O(\log D \log\log n)$ rounds using $O(n)$ messages, where D is the strong diameter of the initial topology. Up to the $\log\log n$ factor, our round complexity is best possible given the trivial $\Omega(\log D)$ lower bound. For many typical networks with $D = o(n)$, our running time is a drastic improvement over prior $O(\log n)$ round algorithms. In particular, for networks with polylogarithmic diameter an $O(\log^2 \log n)$ running time and thus an almost exponential speedup is obtained.

Categories and Subject Descriptors

F.2.2 [**Theory of Computation**]: ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY—*Nonnumerical Algorithms and Problems*; F.1.2 [**Theory of Computation**]: COMPUTATION BY ABSTRACT DEVICES—*Modes of Computation* Parallelism and concurrency

General Terms

Algorithms, Theory, Performance, Reliability

*Research supported in part by NSF award "AF:Small:Distributed Algorithms for Near-Planar Networks".

PODC'14, July 21 – 23, 2015, Donostia-San SebastiÃąn, Spain.

ACM 978-1-4503-3617-8/15/07 ...$15.00.
http://dx.doi.org/10.1145/2767386.2767435.

Keywords

resource discovery; direct addressing; gossip; information dissemination; rumor spreading

1. INTRODUCTION

We give a new distributed algorithm for the classical *resource discovery* problem introduced in [11] that features drastically improved running time on typical networks while maintaining an optimal message complexity.

Resource discovery is a basic task in distributed networks that is concerned with informing all participants in a network about each other in a fast and communication efficient way. It can be used as a coordination mechanism, e.g., in networks with somewhat frequent joins and leaves, and often forms the first step for implementing further distributed functionalities. As such, it has been intensely studied from both a practical and theoretical perspective.

1.1 Formal Problem Definition

The mathematical definition as a distributed problem goes back to the work of Harchol-Baler et al. [11] and arose in the context of building a large-scale distributed cache for their newly founded company Akamai:

Machines are abstracted as nodes in a directed graph $G = (V, E)$ with n nodes in which a directed edge $(u, v) \in E$ means that machine u knows (the IP address of) v and can thus contact v. We write $\Gamma(u)$ to denote set of nodes that node u can contact. Communication proceeds in synchronous rounds in which each node u can PUSH or PULL a message to/from any nodes in $\Gamma(u)$. Each such operation gets counted as one message. In a message a node can communicate any desired information but typically a node simply sends the set of IDs it knows and possibly some small amount of control information, e.g., $O(\log n)$ bits. Once a node has received a message containing addresses it can contact the respective nodes in subsequent rounds and the network graph G changes accordingly. This network model has also been called the DIRECT ADDRESSING or DA model [8].

The goal of distributed resource discovery algorithms in the DA model is to guarantee that for a given initial network topology all nodes learn about each other as quickly as possible without sending too many messages.

1.2 Prior Work on Resource Discovery

The trivial resource discovery algorithm in the DA model is flooding. Flooding simply has every node forward its knowledge to *all* nodes it knows about. It is easy to see that this flooding algorithm terminates in $\log D + 1$ time where D is the (weak) diameter[1] of the initial network topology and that this running time bound is optimal. On the other hand, the flooding algorithm has a horrible message complexity and sends up to $\Theta(n^2)$ messages in a single round. This is even true for sparse networks as they become dense quickly.

The idea of Harchol-Balter et al. was to leverage randomization in achieving more communication efficient information dissemination. They introduced the NameDropper gossip algorithm in which each node sends the addresses it knows solely to a uniformly random contact. They proved that $O(\log D \log n)$ rounds each using n messages are sufficient for this algorithm to solve the resource discovery problem. Subsequently Law and Siu [18] gave a simple randomized resource discovery algorithm requiring $O(\log n)$ rounds and Kutten, Peleg, and Vishkin [17] building on a connectivity algorithm of Shiloach and Vishkin [20] gave a deterministic algorithm that uses $O(\log n)$ rounds and sends only $O(n \log n)$ messages in total.

1.3 Our Result

While these $O(\log n)$ running times match the trivial $\Omega(\log D)$ lower bound for networks with a linear (or polynomial) diameter, such networks are arguably not the setting of interest. Indeed the networks topologies one would expect to encounter in a resource discovery setting are typically well connected and have a small, e.g., at most (poly-)logarithmic, strong diameter[2]. In those networks, the $\Omega(\log D)$ lower bound merely excludes the existence of sub-log-logarithmic algorithm thus leaving open the question whether such exponentially faster algorithms are achievable. In this work we answer this question in the affirmative by giving an algorithm that matches the $\Omega(\log D)$ lower bound for the strong diameter up to an $O(\log \log n)$ factor:

THEOREM 1. *Suppose a network in the DA model with n nodes and a strong diameter of D. There is a distributed randomized algorithm that, with high probability, completes a resource discovery in any such network in $O(\log D \cdot \log \log n)$ rounds using a total of $\Theta(n)$ messages.*

Remarks:

- Like all previous works we do not attempt to give any formal failure model and prove robustness results for our algorithm with respect to any permanent or temporary failure or interleaved join and leaves. This said, we believe that the randomized GOSSIP nature of our algorithm (similar to [11]) actually makes the algorithm quite robust at least to random failures or leaves (which, e.g., are unlikely to specifically delete cluster-centers). More over the (almost) exponentially faster running time guarantee on typical networks compared to prior approaches drastically reduces the timeframe in which the network needs to be (sufficiently) stable and allows the rebuilding process to be run more frequently.
- Like [18] but in contrast to [11, 17] we give running time guarantees with respect to the strong diameter. We postulate this diameter to be small for instances of interest. We also remark that it is generally impossible to have GOSSIP algorithms that perform well in graphs with low weak-diameter while having nodes send at most c messages per round. In fact such algorithms cannot run in less than $\log_c n = \log n / \log c$ rounds on the star-network in which one center node knows about all other n nodes (but not vice versa) even though this network has a weak diameter of 2. Thus the only way to even achieve sub-logarithmic running time in this network is to have a single node send at least polynomially many messages in one round.

1.4 More Prior Work

In addition to the results of [11, 17, 18] on the resource discovery problem (see Section 1.2) the techniques used in our algorithm are also closely related to [8, 2]. These works show that one can achieve sub-logarithmic running times for the global broadcast problem if one considers a communication model that allows for DIRECT ADDRESSING and also for contacting a random participating node in each round. In particular, Avin and Elsässer[2] gave an algorithm with $O(\sqrt{\log n})$ round complexity while Haeupler and Malkhi [9] showed that even a $\Theta(\log \log n)$ running time is achievable and optimal. These results can be interpreted as running times of resource discovery algorithms on random topologies in which each node has many directed links to independently chosen nodes. Since such topologies have a logarithmic diameter Theorem 1 implies an $O(\log^2 \log n)$ running time for this setting as well. The setting in this paper however is significantly more involved because no randomness or even expansion in the topology is assumed. The one (and only) crucial idea we managed to carry over from [8] is to organize nodes in clusters and try to achieve a doubly exponential growing process by having clusters of size s reach out to $O(s)$ clusters for merging thus leading to s^2 size clusters in one step.

We also remark that there are many parallels between this work and gossip algorithms for the LOCAL model [19], which features a fixed undirected topology. There too the resource discovery or global broadcast problem can be solved via flooding in an optimal time complexity, which is $O(D)$ in the LOCAL model. This however requires up to $\Theta(n^2)$ messages per round. Demers et al. [6] were the first to introduce randomized gossip algorithm to achieve fast and reliable information dissemination. Since then uniform gossip has been shown to work well on complete graphs [12] and random and expanding topologies [4, 7]. Similar to this work more recent non-uniform gossip algorithms [3, 8] even achieve $O(D + \text{poly} \log n)$ running times, which are close to the trivial $\Omega(D)$ lower bound achieved by flooding albeit

[1]The weak diameter of a network is the smallest integer for which every node has a path of length at most D to any other node when directions of edges are ignored.

[2]The strong diameter of a network is the smallest integer for which every node has a directed path of length at most D to any other node.

while using only $O(1)$ communication (instead of $O(n)$) per node.

Beyond these works, owing to the high practical relevance of the topic, many works have studied variants of the resource discovery problem. We mention [15, 1, 16] which consider resource discovery in an asynchronous setting, [10, 13] which considers communication efficient resource discovery gossip with small messages and refer to [14] and the recent PhD thesis of Davtya [5] for a broader overview and further references.

2. RESOURCE DISCOVERY

2.1 Clusters

In this section we introduce clusters, a simple tool used in this paper to achieve efficient exploration. Briefly, a cluster is a set of nodes that have a designated cluster-center known to all nodes in the cluster. All nodes within a cluster can exchange information via the center using only a constant number of rounds and messages per node. This allows a cluster to act in a coordinated manner.

More precisely, a cluster is a set S of nodes, with each node $v \in S$ containing a variable $follow_v = \ell_S$ set to the address of the cluster leader. Initially all nodes form their own cluster and have $follow$ set to their own address. For a node v, we denote its cluster by $C(v)$ (or simply C when clear from context).

Having a central leader which is known to all nodes in a cluster allows a cluster to coordinate using only a constant number of communication rounds. In particular, in one communication round followers may PUSH information to the leader and thus have the leader collect inputs from the entire cluster. In a second round, followers may PULL a response from the leader. Throughout the rest of this paper we implicitly assume that information is exchanged within each cluster before external communication steps as needed. This allows us to treat clusters as indivisible entities which act as a unit and share full information, while omitting the details concerning PUSHing and PULLing messages.

We furthermore use the following notation: For two clusters C_1, C_2, if there is an edge from a node $v \in C_1$ to a node $w \in C_2$ we say that C_2 is a *cneighbor* of C_1. Clusters generally do not know which clusters are cneighbors. For this reason any cluster C maintains a set $A(C)$ in which it keeps track of clusters that it knows to be current cneighbors.

Remark:
A cluster can easily find out, using two cluster internal PUSH/PULL rounds, which nodes $\Gamma(C)$ the nodes in C are connected to. However, it is much harder (or message intensive) to find out what its distinct cneighbors are, i.e., which nodes in $\Gamma(C)$ belong to the same cluster. In particular, as clusters merge these clusterings change and it is generally too message expensive to keep a complete list of cneighbors updated. This is the reason for having $A(C)$. It also means that after every round of cluster merging the list $A(C)$ becomes outdated and is thus set to be the empty set.

2.2 The Resource Discovery Algorithm

We next give a high level description of our algorithm:

The algorithm first calls the INITIALIZATION sub-routine which reduces the number of clusters by a polylogarithmic factor, bringing the average cluster size to $s = \log^{O(1)} n$. This is done in $\Theta(\log\log n)$ rounds via a merging process that guarantees singly exponential reduction process in the number of clusters.

The main loop of the algorithm then employs iterations with a doubly exponential growth progress by roughly squaring the average size s of a cluster in each iteration. For this, each iteration begins with using the DISCOVERCNEIGHBORS routine to have clusters explore their neighborhood and discover at least $d \approx s$ cneighbors as potential merging partners.

Clusters that do not have enough cneighbors use INFLATEOUTDEGREE to perform a local flooding procedure until at least d clusters are reached. This requires at most $\log D$ flooding steps and does not lead to a large message overhead because only clusters with less then d cneighbors participate.

The SAMPLEMERGE routine then randomly sub-samples roughly a $1/d \approx 1/s$ fraction of clusters to be the new cluster-centers and assigns each non-selected cluster to a cluster center. With high probablity, the number of clusters is thus reduced roughly by factor s, which leads to a new average cluster size of about $s \cdot d \approx s^2$.

This doubly exponential merging process is performed for $O(\log\log n)$ iterations until the average cluster size reaches at least $\sqrt{n}\log n$. A final INFLATEOUTDEGREE guarantees that every cluster becomes aware of any other cluster. The cluster centers then inform all their nodes about all other nodes, which completes the resource discovery process.

Algorithm 1 Log-logarithmic resource discovery algorithm

1: INITIALIZATION($\Theta(\log\log n)$)
2: $\epsilon \leftarrow 1/2$
3: $s \leftarrow \log^{2/\epsilon} n$
4: Clusters of size $s' > s$ split into $\lceil s'/s \rceil$ clusters of size between $s/2$ and s

5: **repeat**
▷ explore/expand neighbors to discover $s^{1-\epsilon}$ cneighbors
6: DISCOVERCNEIGHBORS($s, 2/\epsilon$)
7: $d \leftarrow s^{1-\epsilon}$
8: INFLATEOUTDEGREE(d)

▷ sample-merge phase to reduce number of clusters by roughly $1/d$
9: SAMPLEMERGE(s, d)
10: $s \leftarrow \Theta(s \cdot d/\log n)$ ▷ $\Theta(s \cdot d/\log n) = \Omega(s^{2-1.5\epsilon})$
11: Clusters of size $s' > s$ split into $\lceil s'/s \rceil$ clusters of size between $s/2$ and s

12: **until** $s \geq \sqrt{n}\log n$ ▷ $\Theta(\log\log n)$ repetitions

13: INFLATEOUTDEGREE($\sqrt{n}$) ▷ all clusters become aware of all clusters
14: Cluster centers inform all nodes in their cluster about all other nodes

2.3 Analysis

Next we prove our main theorem:

THEOREM 1 (repeated). *Suppose a network in the DA model with n nodes and a strong diameter of D. There is a distributed randomized algorithm that, with high probability, completes a resource discovery in any such network in $O(\log D \cdot \log\log n)$ rounds using a total of $\Theta(n)$ messages.*

The proof relies on the following guarantees about the subroutines referenced in Algorithm 1. We give their statements here and provide their implementations and proofs in subsequent subsections.

LEMMA 2. *Assume that there are at least $\Omega(\log n)$ clusters before a call to* INITIALIZATION. *For every r, with high probability,* INITIALIZATION*(r) reduces the number of clusters by a factor of $c = e^{\Theta(r)}$ using $O(r)$ rounds and $\Theta(n)$ messages in total.*

LEMMA 3. *Assume each cluster is of size at most s and suppose that $s \geq \log^2 n$. With high probability after a call of* DISCOVERCNEIGHBORS*(s, r) each cluster C knows either all or at least $|A(C)| \geq \Theta(s^{1-(2/r)})$ distinct cneighbors. The routine furthermore requires at most $O(r)$ rounds with at most $O(rs^{1-(1/r)})$ messages being sent by each cluster.*

LEMMA 4. *Assume that each cluster has outgoing edges to either to all of its cneighbors or to at least d of them. Assume further that the underlying network has a strong diameter of at most D. Then, at the end of* INFLATEOUTDEGREE*(d) a cluster C is aware of either all clusters or at least d cneighbors, i.e, $|A(C)| \geq d$. Furthermore, only $O(\log D)$ rounds and $O(d \cdot \log D)$ messages per cluster are required.*

LEMMA 5. *Assuming that at the beginning of* SAMPLEMERGE*(d) every cluster C is aware of at least $d \geq \log^2 n$ cneighbors, i.e., $|A(C)| \geq d$, then with high probability* SAMPLEMERGE*(d) reduces the number of clusters by a factor of $\Theta(\log n/d)$. Furthermore, only $O(1)$ rounds and $O(d)$ messages per cluster are required.*

PROOF OF THEOREM 1. We first prove the correctness of Algorithm 1. Initially each node forms its own cluster so there are n clusters. Theorem 2 now guarantees that the initialization step ends with at most $n/\log^{2/\epsilon} n = n/s$ clusters. The splitting of clusters of size larger than s leads to at most n/s further clusters for a total of at most $2n/s$ clusters each of size at most s.

We prove by induction that this guarantee of at most $2n/s$ clusters of size at most s holds whenever the beginning or end of the main loop is reached.

In particular, Theorem 3 guarantees for all *eps* > 0, that during DISCOVERCNEIGHBORS$(s, 2/\epsilon)$ every cluster discovers at least $d = s^{1-\epsilon}$ or all its cneighbors and Theorem 4 guarantees that after INFLATEOUTDEGREE(d) every node has at least d cneighbors in its $A(C)$ set. Theorem 5 shows that SAMPLEMERGE(d) then reduces the number of clusters by a $\Theta(\log n/d)$ factor thus decreasing the number of clusters to at most $\Theta(n/(\frac{s \cdot d}{\log n}))$ or n/s after the parameter s is updated. Splitting clusters larger than s again at most doubles the number of clusters to $2n/s$ and leads to clusters of size at most s as postulated.

Once the main loop terminates this leads to at most $2\sqrt{n}/\log n$ clusters. A final call to INFLATEOUTDEGREE$(\sqrt{n})$, according to Theorem 4, then leads to all clusters being aware of all other clusters which allows the cluster centers to inform all nodes in their cluster about all other nodes. The algorithm thus terminates correctly.

Next we analyze the number rounds and number of messages required for this process. Theorem 2 guarantees that the initialization takes $\Theta(\log\log n)$ rounds and $\Theta(n)$ messages. For the main loop we claim that there are at most $4\log\log n$ iterations of the main loop of Algorithm 1 and that each such iteration uses only $O(\log D)$ rounds and at most $O(n/\log n)$ messages. The number of iterations follows simply from the observation that each iteration increases s to $\Theta(s^{2-\epsilon}/\log n) \geq \Theta(s^{2-\epsilon}/s^{\epsilon/2}) = \Theta(s^{2-1.5\epsilon})$. Even for $\epsilon = 1/2$ having $i = 4\log\log n$ iterations s would increase s from $\log^2 n$ to $(\log^2 n)^{1.25^i} \geq 2^{2^{\log\log n}} = n$ which implies that the loop terminates after less than $4\log\log n$ iterations. Regarding the number of messages and rounds, Theorem 3 states that DISCOVERCNEIGHBORS$(s,2/\epsilon)$ uses at most $O(2/\epsilon s^{1-\epsilon/2})$ messages for each of the $2n/s$ clusters for a total of at most $O(n/s^{\epsilon/2}) = O(n/\log n)$ messages per iteration. It also requires only $O(1)$ rounds. Similarly, SAMPLEMERGE(d) uses a total of $2n/s \cdot ds = O(n/s^{\epsilon}) = O(n/\log^2 n)$ messages and $O(1)$ rounds.
Lastly, INFLATEOUTDEGREE(d) uses

$$2n/s \cdot d \log D = O(n \log n/s^{\epsilon}) = O(n/\log n)$$

messages and $\log D$ rounds. This shows that the main loop requires overall at most $O(\log D \log\log n)$ rounds and $o(N)$ messages.

To complete the analysis, note that the final call of INFLATEOUTDEGREE$(\sqrt{n})$ also requires at most $O(\log D)$ rounds and at most $O(\log D\sqrt{n}) \cdot 2n/(\sqrt{n}\log n) = O(n)$ messages. □

2.4 Initialization

The initialization sub-routine brings down the number of clusters in a network by a polylogartihmic factor by having $O(\log\log n)$ rounds in which a constant fraction of the clusters merges.

In a round, the goal is to break symmetry into joiners and joinees, such that sufficiently many joiners merge with joinees. In a full topology, it would suffice to flip a coin to become joiner/joinee and then select a merging partner at random; this would succeed for a constant fraction of the network. In general topologies this simple rule does not work, because some nodes may be more important than others due to the graph structure. For example, in a star graph, the center determines the success/failure of all merging attempts. To address this, we first use a randomized neighbor-selection step which identifies important nodes. After this step, every node which received more than one neighbor-request chooses to be a joinee and immediately accept any joiner request. There will remain a group of nodes which are neither joinees, nor have their requests accepted already. The nice thing is that each one of these nodes will be matched with at most one incoming neighbor request and one outgoing request. Therefore, within this

set, flipping a coin to become joiner/joinee will succeed in merging a constant fraction with high probability.

Algorithm 2 INITIALIZATION(r)

1: **for** r iterations **do**
2: every cluster C picks one neighbor $v \in \Gamma(C)$ and sends a merge request $C(v)$
3: **if** cluster received more than one merge request **then**
4: retract your own merge request ▷ choose to become joinee
5: send all (non-retracted) incoming requests a joinee signal
6: accept any (non-retracted) incoming requests
7: **end if**
8: with probability 1/2 retract your own merge request
9: **if** did not receive a joinee signal **then**
10: accept any (non-retracted) incoming request
11: **end if**
12: retract any merge requests that have not been accepted yet
13: **end for**

LEMMA 2. *Assume that there are at least $\Omega(\log n)$ clusters before a call to* INITIALIZATION. *For every r, with high probability,* INITIALIZATION*(r) reduces the number of clusters by a factor of $c = e^{\Theta(r)}$ using $O(r)$ rounds and $\Theta(n)$ messages in total.*

PROOF. Suppose that the number of clusters before any of the r iterations is $N = \Omega(\log n)$. We prove that the iteration reduces the number of clusters by a constant factor with high probability. For this we consider the directed request graph which indicates which cluster requested to be merged with which other cluster. The merge requests guarantee that every node has an out-degree of exactly one. Since there are exactly N requests the number of nodes that retract their request because of multiple received requests is at most $N/2$. We want to show that at least a constant fraction of these requests are accepted. Some are already accepted at Line 5 by the nodes that retracted their requests. If these make up $N/8$ requests the claim is proved. If not then there are at least $N/4$ remaining requests.

Since we resolved requests to nodes with an in-degree larger than one we have the guarantee that these remaining requests decompose into directed cycles and directed paths. In these paths and cycles we break symmetries using randomization. In particular, any remaining edge has a probability of 1/4 of being accepted because with probability 1/2 it is not retracted by its requester and with an independent probability of 1/2 it is accepted by the cluster that received the request. Furthermore, due to the paths structure the $N/4$ remaining requests contain at least $N/8$ requests that are accepted independently. Out of these $N/8$ requests we expect $N/32$ to be accepted and a standard Chernoff bound shows that with high probability at least $\Theta(N)$ will be accepted. This proves that every iteration leads with high probability to at least a constant fraction of the clusters merging. Doing this for r rounds leads to the claimed $e^{\Theta(r)}$ overall reduction in the number of clusters.

To determine the message complexity we note that each cluster sends only $O(1)$ messages per iteration. This leads to at most $O(n)$ messages in the first iteration. Since the number of clusters in subsequent iterations decreases geometrically with a factor of $\delta < 1$ the overall message complexity is also $\sum_{i=0}^{r} \delta^i O(n) = O(n)$. □

2.5 Cneighbor Discovery

The cneighbor-discovery sub-routine probes the nodes neighboring a cluster in order to identify sufficiently many distinct cneighbors. At the beginning of sub-routine, a cluster C has a set $\Gamma(C)$ of neighboring nodes which are split, potentially unevenly, among an unknown set of cneighbors. In a round, the cluster probes roughly s neighbors. It stores all the cneighbors it discovered in $A(C)$ and repeats with remaining neighbors. Intuitively, each such round makes progress in one of two ways. One case is that it discovers a large enough number of 'light' cneighbors, which do not contain a lot of neighbors, and we are done. The other case is that it discovers 'heavy' cneighbors, which contain a total large number of neighboring nodes. In this case, in the next round, C has a significantly reduced set of neighbors to work on.

The algorithm is given as Algorithm 3.

Algorithm 3 DISCOVERCNEIGHBORS(s, r)

1: $X \leftarrow \Gamma(C)$
2: $A(C) = \emptyset$
3: **for** $2r$ repetitions **do**
4: select a random set $R \subseteq X$ of size $\min\{s^{1-(1/r)}, |X|\}$
5: PULL from each $v \in R$ the members of $C(v)$
6: $A(C) \leftarrow A(C) \cup \left(\bigcup_{v \in R} \{C(v)\}\right)$
7: $X \leftarrow X \setminus \left(\bigcup_{v \in R} C(v)\right)$
8: **end for**

LEMMA 3. *Assume each cluster is of size at most s and suppose that $s \geq \log^2 n$. With high probability after a call of* DISCOVERCNEIGHBORS(s, r) *each cluster C knows either all or at least $|A(C)| \geq \Theta(s^{1-(2/r)})$ distinct cneighbors. The routine furthermore requires at most $O(r)$ rounds with at most $O(rs^{1-(1/r)})$ messages being sent by each cluster.*

PROOF. Define $t = s^{1-(1/r)}$. Consider one cluster C and denote $C_1, \ldots, C_m$ its cneighbors. We define the degree $\Delta_C(C_i)$ of a cneighbor C_i to be the number of edges going from C to distinct nodes in C_i. We classify these cneighbors according to this degree into two groups. The first group contains all "heavy" cneighbors whose degree makes up at least a $c\log(n)/t$ fraction of X for some constant $c > 1$, i.e., $\Delta_C(C_i) \geq c|X|\log(n)/t$, while the second group contains all other "light" cneighbors of C.

If at any iteration the size $|X|$ of the remaining neighborhood becomes smaller than t then all neighbors are contacted completing the claim. The remainder of this proof is concerned with the case that $|X|$ remains large and t neighbors get selected in each iteration.

Since the t neighbors in R are chosen independently at random each heavy cneighbor is PULLED in expectation $c\log n$ times and with high probability at least once. In

particular the probability of not being PULLed at all is at most $(1 - \frac{c|X|\log(n)/t}{|X|})^t < e^{-c\log n}$.

Conversely, every light cneighbor is hit with at most an expected $c\log n$ PULLs from C and a standard Chernoff bound shows that with high probability at most $O(\log n)$ PULLs go to any such cneighbor.

We consider two cases. If the sum total of degrees going into light cneighbors is at least a $1/t^{(1/r)}$ fraction then the number of edges selected from this set is in expectation and with high probability $\Omega(t^{(1-(1/r))}) = \Omega(s^{(1-(1/r))(1-(1/r))}) = \Omega(t^{(1-(2/r))})$. The number of distinct light cneighbors hit is therefore at least $\Omega(t^{(1-(2/r))})/O(\log n)$ with high probability finishing the claim.

The other possibility is that the total number of edges going into light cneighbors is less than a $1/t^{(1/r)}$ fraction. Since these are the only edges that make up the total degree $|X|$ the new set X' of the next iteration has size $|X'| \leq |X|/t^{(1/r)}$.

Thus each iteration with high probability either discovers enough distinct cneighbors or reduces the total degree by a $t^{(1/r)}$ factor. Furthermore, if the algorithm does succeed in the first round we know that in this iteration at least one cluster of degree $c|X|\log n/t$ existed. Since the degree can be at most the size of the cluster which is assumed to be at most s we get that either the first iteration succeeds or $|X| \leq st/c\log n \leq s^2$. Therefore, failed iterations which reduce $|X|$ by a factor of $1/t^{(1/r)}$ can happen at most $2r$ times before $|X| \leq t$ and all remaining cneighbors are contacted. □

2.6 Degree Inflation

The sub-routine for degree-inflation works to expand the out-degree of all clusters to a desired threshold d. In a round, every 'lean' node, whose degree has not reached the desired threshold, simply PULLs from all of its neighbors their neighbor-sets. This means that a cluster PULLs information about nodes at twice the distance of the previous round. Therefore, in at most $\log D$ rounds, the entire graph can be PULLed.

Algorithm 4 InflateOutdegree(d)

```
for log D iterations do
    if |A(C)| < d then
        pull A(C') from every C' ∈ A(C) into A(C)
    end if
end for
```

Lemma 4. *Assume that each cluster has outgoing edges to either to all of its cneighbors or to at least d of them. Assume further that the underlying network has a strong diameter of at most D. Then, at the end of* InflateOutdegree*(d) a cluster C is aware of either all clusters or at least d cneighbors, i.e, $|A(C)| \geq d$. Furthermore, only $O(\log D)$ rounds and $O(d \cdot \log D)$ messages per cluster are required.*

Proof. For analysis purposes, we fix a distance function $d()$ reflecting the shortest cluster-to-cluster hop length between clusters at the beginning of the procedure. Clearly, $d(C_1, C_2) \leq D$ for every two clusters C_1, C_2. InflateDegree progresses in iterations. At the start of each iteration, a cluster C is called *lean* if $|A(C)| < d$. Clusters continue iterating the loop only as long as they remain lean.

We prove the lemma by induction on the following invariant: For every lean cluster C entering its k'th loop iteration, $A(C)$ contains all clusters W with $d(C, W) \leq 2^{(k-1)}$. The base of the induction for $k = 1$ holds since if C is initially lean then it has outgoing edges to all of its distance-1 cneighbors. Suppose C is lean at the beginning of iteration $k+1$. Consider any cluster W with $D(C, W) \leq 2^k$. There exists some cluster W' on the path from C to W, such that $D(C, W') \leq 2^{(k-1)}$ and $D(W', W) \leq 2^{(k-1)}$. Let's examine $A(C)$ and $A(W')$ at the k'th iteration. When the iteration starts, we know that C is lean, and by the induction hypothesis $W' \in A(C)$. We know that C remains lean at the end of the k'th iteration after it pulls $A(W')$ into $A(C)$. Therefore, when the k'th iteration starts, W' too is lean, and by the induction hypothesis, $W \in A(W')$. Therefore, W is pulled into $A(C)$ at the k'th iteration. This completes the induction.

Clearly, within $\log(D)$ iterations, C learns about all existing clusters and we are done. Each cluster furthermore only uses less than d connections per iterations for a total of at most $d \cdot \min\{\log d, \log D\}$ connections per cluster. □

2.7 Sample and Merge

A sample-and-merge sub-routine starts with at most N/s clusters all having d out-cneighbors. Then, a fraction of roughly $1/d$ are sampled to become center-clusters (more precisely, we sample with probability $c\log n/d$, to guarantee that every cluster has a cneighbor which is sampled). Each non-center cluster selects one sampled cluster among its d cneighbors to join. At the end of the sub-routine, the number of clusters is reduced by a factor of roughly $d/(c\log n))$.

Algorithm 5 SampleMerge(d)

```
with probability c log n/d become a cluster-center
if non-center cluster then
    contact d clusters from A(C) to find a center-cluster
    merge with an arbitrary neighboring center-cluster
end if
```

Lemma 5. *Assuming that at the beginning of* SampleMerge*(d) every cluster C is aware of at least $d \geq \log^2 n$ cneighbors, i.e., $|A(C)| \geq d$, then with high probability* SampleMerge*(d) reduces the number of clusters by a factor of $\Theta(\log n/d)$. Furthermore, only $O(1)$ rounds and $O(d)$ messages per cluster are required.*

Proof. Let x be the number of clusters before SampleMerge. After sampling, the expected number of center-clusters is $\mu = x \cdot (c\log n/d)$. Since $x \geq |A(C_1)| = d$ we have $\mu \geq \log n$. Since each cluster makes an independent decision, a standard Chernoff bound shows that with high probability there are exactly $\Theta(\mu)$ center-clusters which form the new clusters after SampleMerge. A similar argument guarantees that with high probability out of the d neighbors contacted by a non-center cluster at least one center-cluster is found. This guarantees that no non-center clusters survive the SampleMerge procedure. □

3. CONCLUSION AND OPEN PROBLEMS

The work presented in this paper makes a large improvement in the round complexity of resource discovery on instances of interest while maintaining the overall communication optimal. There are a number of compelling problems left to tackle.

First, while the message complexity of our algorithm is $O(n)$ there are nodes that send more than $O(1)$ messages per round. We believe that it is possible to design a gossip algorithm with the same running time guarantees in which nodes send at most one message per round. This requires a modified INITIALIZATION routine which does not just guarantee that the number of clusters is reduced by a polylogarithmic factor but instead guarantees that *each* cluster is of polylogarithmic size. More importantly the SAMPLEMERGE routine needs to be followed by an intricate re-balancing step which guarantees that each cluster-center is joined by sufficiently many clusters. We defer these details to the journal version.

Second, our work leaves open to further improve the network discovery time of graphs with only a weak diameter bound. Our intuition is that a crucial building block for solving this would be a log-logarithmic local-flooding algorithm. Once achieved, it may be repeated $\log D$ times to fold the entire graph into a clique, in a similar manner to the utilization of local broadcast in the gossip techniques of [3].

4. REFERENCES

[1] I. Abraham and D. Dolev. Asynchronous resource discovery. *Computer Networks*, 50(10):1616–1629, 2006.

[2] C. Avin and R. Elsässer. Faster Rumor Spreading: Breaking the $\log n$ Barrier. In *Proceedings of the International Symposium on Distributed Computing*, DISC '13, pages 209–223, 2013.

[3] K. Censor-Hillel, B. Haeupler, J. Kelner, and P. Maymounkov. Global Computation in a Poorly Connected World: Fast Rumor Spreading with no Dependence on Conductance. In *Proceedings of the ACM Symposium on Theory of Computing*, STOC '12, pages 961–970, 2012.

[4] F. Chierichetti, S. Lattanzi, and A. Panconesi. Almost tight bounds for rumour spreading with conductance. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC)*, pages 399–408, 2010.

[5] S. Davtyan. *Resource Discovery and Cooperation in Decentralized Systems*. PhD thesis, University of Connecticut, 2014.

[6] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–12, 1987.

[7] G. Giakkoupis. Tight bounds for rumor spreading in graphs of a given conductance. In *28th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 57–68, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[8] B. Haeupler. Simple, fast and deterministic gossip and rumor spreading. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 705–716, 2013.

[9] B. Haeupler and D. Malkhi. Optimal gossip with direct addressing. In *Proceedings of the 31st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 176–185, 2014.

[10] B. Haeupler, G. Pandurangan, D. Peleg, R. Rajaraman, and Z. Sun. Discovery through gossip. In *Proceedinbgs of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 140–149, New York, NY, USA, 2012. ACM.

[11] M. Harchol-Balter, T. Leighton, and D. Lewin. Resource discovery in distributed networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC'99, pages 229–237, 1999.

[12] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. Randomized rumor spreading. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 565–574, Washington, DC, USA, 2000. IEEE Computer Society.

[13] S. Kniesburges, A. Koutsopoulos, and C. Scheideler. A deterministic worst-case message complexity optimal solution for resource discovery. *Theoretical Computer Science*, 2014.

[14] K. M. Konwar, D. Kowalski, and A. A. Shvartsman. Node discovery in networks. *Journal of Parallel and Distributed Computing*, 69(4):337–348, 2009.

[15] S. Kutten and D. Peleg. Asynchronous resource discovery in peer to peer networks. In *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, pages 224–231. IEEE, 2002.

[16] S. Kutten and D. Peleg. Asynchronous resource discovery in peer-to-peer networks. *Computer Networks*, 51(1):190–206, 2007.

[17] S. Kutten, D. Peleg, and U. Vishkin. Deterministic resource discovery in distributed networks. *Theory of Computing Systems*, 36(5):479–495, 2003.

[18] C. Law and K.-Y. Siu. An $O(\log n)$ randomized resource discovery algorithm. In *Brief Announcements of the 14th International Symposium on Distributed Computing, Technical University of Madrid, Technical Report FIM/110.1/DLSIIS*, pages 5–8. Citeseer, 2000.

[19] D. Peleg. *Distributed computing: a locality-sensitive approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[20] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com