ICT-317756

# TRILOGY2

## TRILOGY2: Building the Liquid Net

Specific Targeted Research Project
FP7 ICT Objective 1.1 The Network of the Future

# D1.4: Building the Liquid Net

Due date of deliverable: 30 December 2015
Actual submission date: 30 December 2015

| | |
|---|---|
| Start date of project | 1 January 2013 |
| Duration | 36 months |
| Lead contractor for this deliverable | Universidad Carlos III de Madrid |
| Version | v0.1 , 30 December 2015 |
| Confidentiality status | Public |

**Abstract**

TRILOGY2 aims to construct the Liquid Net by developing tools and mechanisms that pool Network, Storage and Processing resources. This deliverable highlights the challenges involved in performing multi-resource pooling, especially since the resources exist in tradeoff. We offer a systematic examination of the resources pooled across the various components of the TRILOGY2 framework, drawing attention to the tradeoffs exhibited between them. Finally, we present the design, implementation and evaluation of new features and solutions required for building the Liquid Net.

**Target Audience**

The target audience for this document is the networking research and development community, particularly those with an interest in Future Internet technologies and architectures. The material should be accessible to any reader with a background in network architectures, including mobile, wireless, service operator and datacenter networks.



**Impressum**

| | |
|---|---|
| Full project title | TRILOGY2: Building the Liquid Net |
| Title of the workpackage | WP1 Creating Liquidity |
| Editor | Lynne Salameh, UCL-UK |
| Project Co-ordinator | Marcelo Bagnulo Braun, UC3M |
| **Copyright notice** | © 2016 Participants in project TRILOGY2 |

# Executive Summary

TRILOGY2 has set out to build a new Internet architecture based on the concept of network liquidity, with an emphasis on achieving performance gains from pooling three main resources: Bandwidth, Processing and Storage. Motivated by the observation that, when deployed together, mechanisms for pooling any single one of these resources would likely result in sub-optimal performance, this project has targeted the design of a framework capable of orchestrating, provisioning, and controlling several resource pools.

Pooling more than one resource is, at heart, a form of multi-metric optimization. During the course of the project, we have discovered that performing such optimizations across more than two resource types is fundamentally difficult. The resources pooled usually exist in tradeoff, where only one constraining resource has the highest priority for optimization, leaving the other two as secondary goals.

TRILOGY2 has therefore developed a collection of mechanisms for the creation and management of liquidity, mostly by optimizing across two pooled resources. We examine the resources pooled across the various components of the TRILOGY2 framework, highlighting resources that exist in tradeoff. Additionally, we present some refinements to the aforementioned resource pooling mechanisms in this deliverable, as well as several new tools required for building the Liquid Net.

Both VNF Pool enabled virtual Customer Premises Equipment (CPE), and the Federated Marketplace have been refined and completed. In particular, we have implemented a prototype of VNF Pool enabled virtual CPE, in the form of operator tools that deploy, configure and operate liquid VPN services. The prototype uses an enhanced Quagga BGP protocol deamon to support MPLS VPNs and configure VRFs within the data center. The Federated Market and Cloud.net now incorporate several new features which further help customers buy, sell and trade resources without owning infrastructure. These features include improvements to billing, consistency, and customer metric reporting facilities.

Data center environments give rise to many opportunities for creating and managing liquidity. In this deliverable, we systematically examine the different parameters required for the design of NFV data center topologies. We also build a Layer 4 load balancer that minimizes disruption in the data plane in case of hardware failures or when scaling in/out in the data center. Additionally, we develop a New Data center transport Protocol (NDP), designed to achieve low small flow completion times and good large flow throughput in a wide range of scenarios, including the well studied TCP incast scenario.

Finally, we have developed three solutions that combine various facets of bandwidth, processing and storage liquidity in an effort to reduce network latency. C3 targets storage liquidity, and is an adaptive replica selection tool which aims to reduce the tail latencies a client experiences in the face of performance fluctuations in data stores. Minicache and Jitsu use lightweight VMs to bring functionality closer to the user and away from remote data centers and CDNs. Minicache is an minimal Xen-based virtualized content cache prototype, which facilitates the creation of edge CDNs closer to the user. Jitsu, on the other hand, is a new Xen toolstack which allows the rapid instantiation of services on resource-constrained embedded devices near the user. It provides a directory service that launches unikernels in response to network traffic and masks boot latency.

By providing tools and techniques that optimize across multiple resource pools, the aforementioned solutions realize the goal of building the Liquid Net.

# List of Authors

| | |
|---|---|
| Authors | Marcelo Bagnulo, Francisco Valera, Giacomo Bernini, Gino Carrozzo, Elian Kraja, Vincenzo Maffione, Marco Canini, Olivier Bonaventure, Mark Handley, Lynne Salameh, Felipe Huici, Jon Crowcroft, Anil Madhavapeddy, Magnus Skjegstad, Thomas Leonard, Dragos Niculescu, Vladimir Olteanu, Costin Raiciu, John Thomson, Pedro A. Aranda, Jaime Garcia-Reinoso, Ivan Vidal, Julian Chesterfield, Toby Moncaster. |
| Participants | UC3M, NXW, UCL-BE, UCL-UK, NEC, UCAM, UPB, OnApp, TID |
| Work Package | WP1 - Creating Liquidity |
| Security | Public (PU) |
| Nature | R |
| Version | v0.1 |
| Total number of pages | 87 |

# Contents

# List of Figures



# List of Tables

# 1 Introduction

The goal of TRILOGY2 was to investigate liquid networking across a range of resources. The motivation behind this project was the observation that many partners were building mechanisms that performed resource pooling, but in each case only a single resource was being pooled. Examples include network resource pooling using MPTCP to pool the capacities of multiple paths, and CPU resource pooling using server load-balancing software. It had become clear that if multiple single-resource pooling mechanisms were all deployed simultaneously, these mechanisms were likely to perform suboptimally, as each performs its own local resource pooling optimization. In some cases we envisaged different resource pooling mechanism might even conflict with each other.

In essence, prior to the project, resource pooling mechanisms performed single-metric optimization. We envisaged that by looking at more than one resource, we might co-optimize different resource pools, essentially performing a form of multi-metric optimization. Multi-metric resource pooling is difficult - we not only need to examine how different metrics need to be combined, but we need to factor in that each resource has its own timescale for adaptation - sometimes these timescales are similar, sometimes they are sufficiently disjoint that one can only perform one optimization at a time, and sometimes they're on timescales that only partially overlap.

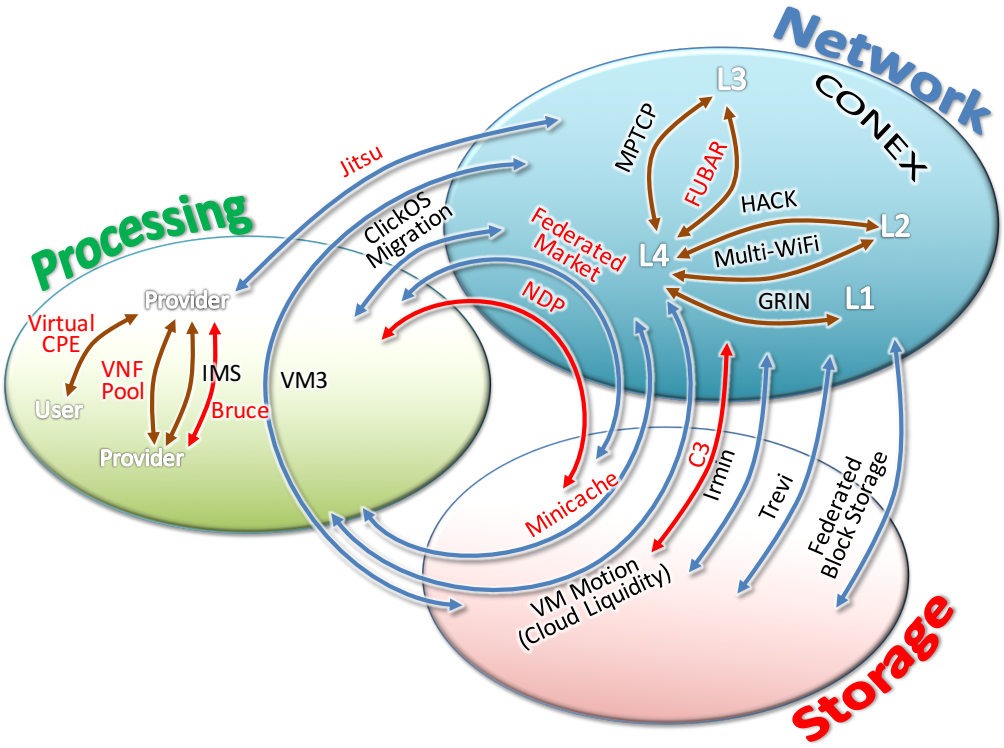



Figure 1.1: TRILOGY2 Overall Framework. Tools which have been newly developed or updated are shown in red.

The main resources we have examined throughout TRILOGY2 are Network, Processing, and Storage. Figure 1.1 provides a high-level overview of the main liquidity mechanisms we have examined throughout the duration of the project. Some of the mechanisms concern all three resource types; some concern liquidity across two, and a number primarily concern one of these resource types but provide liquidity between different entities or perform cross-layer resource pooling within that resource category.

It has become clear during the project that actually providing true liquidity across more than two resource types is both difficult and not usually necessary to get substantial gains. Where our mechanisms concern all of Processing, Networking and Storage, we usually find that only two of the three are actually liquid at a time, with the third being necessary but not liquid in the quite the same manner. The reason for this appears to be that we are examining trade-offs between resources; at any time one of the three resources is the constraining one that is the highest priority for optimization, one is secondary - nice to optimize but

not the highest priority - and one is tertiary. For example, when performing migration of network functions using ClickOS, we may perform the migration primarily to optimize network traffic patterns, while placing functionality on VM hosts subject to CPU load balancing (secondary criteria), but it may not be necessary to optimize RAM usage (tertiary), so long as bounds are respected.
Another rather different example is FUBAR (described in Deliverable 2.4) which is solely a Network liquidity mechanism. FUBAR is a congestion aware (and congestion-control aware) routing/traffic engineering mechanism. It explicitly attempts to co-optimize throughput and latency for traffic flows in an operator's network, which it does by defining a multi-metric utility function that combines learned demands of traffic and operator-imposed constraints. In FUBAR, there are three different timescales for liquidity - the short timescale liquidity imposed by end-to-end congestion control, medium timescale liquidity provided at entry routers attempting to allocate flows to paths to meet capacity constraints, and long-timescale liquidity provided by explicit modeling of both congestion control behavior and flow utility functions so as to allocate those capacity constraints in a near-optimal manner. The aim is to have liquidity between in-network routing at Layer 3 and end-to-end congestion control at Layer 4. Even getting these two forms of network liquidity to play nicely together involves heuristic solutions to more than one NP-hard problem.
This sort of combinatorial explosion is inherent with multi-resource liquidity. We have found throughout the project than many liquidity problems are tractable if we consider one, or at most two, primary resources, but it often rapidly become intractable to consider more resources in detail. For example, allocating virtual machines to a pool of physical machines is relatively easy if one only considers CPU usage. If one also considers memory usage, the problem becomes a version of the knapsack problem [52], which is NP-hard, though there are good heuristic solutions. If we now also consider network usage of those virtual machines, then the problem frequently becomes intractable, so the strategy generally has to be to decide which resource to over-provision and therefore attempt to ignore.
With this in mind, the project has examined a wide range of liquidity mechanisms, most of which concern two forms of liquidity. In such cases there is always a tradeoff between the two resources being pooled. As this fundamental, we have listed in the next section a number of the multi-resouce pooling mechanisms devised within the project. For each, we identify the forms of liquidity and the resources being traded off to gain that liquidity. The remainder of this report contains detailed descriptions of a number of liquidity mechanisms that can be used to build different parts of the liquid net.

# 2 Tradeoffs in the Liquid Net

In light of the observations outlined in the previous chapter, we summarize the solutions for creating and manipulating multi-resource liquidity as presented in the TRILOGY2 framework (Figure 1.1) in the following sections. For each solution, we list the types of resources pooled, and whether any resources were in tradeoff when creating the described liquidity. We also indicate whether the solution provides cross-layer, cross-resource or cross-provider liquidity.
Unsurprisingly, liquidity solutions which optimize the network do so at the expense of performing some extra processing, as seen for example with Software BRAS, GRIN and MPTCP-enabled ClickOS migration. Similarly, mechanisms like Minicache trade off an increased storage capacity with a better network performance. In contrast, creating storage liquidity generates a tradeoff with network and even processing resources, as shown with Federated Block Storage.

| **Resilience for IMS virtualized network functions** | |
|---|---|
| Description | We propose a new architecture to transparently transfer users among the functional elements that implement the IP Multimedia Services (IMS) call session control functions. |
| Resources Pooled | Processing. |
| Resources in tradeoff | The placement of the IMS function tries to optimize the tradeoff between available processing capacity in the server running the IMS function and network capacity available for traffic to reach and leave the selected server. |
| Cross-Liquidity Considerations | This architecture allows the transfer of users between IMS Call-Session Control Functions (CSCFs) transparently to end-user devices, allowing an IMS Liquid Net. |
| Control Tool | When required, users can be reallocated between the over or underloaded CSCF using SIP signalling. |
| Deliverable where it is described. | D3.2 and D3.3 |
| Paper (or standard) where it is described. | J. Garcia-Reinoso, I. Vidal, P. Bellavista, I. Soto, P. A. Aranda, “Transparent Reallocation of Control Functions in IMS Deployments”. IEEE Communications Magazine. January 2016. |

| **Software BRAS** | |
|---|---|
| Description | This tool leverages the ClickOS framework to build high performance, virtualized, software-based Broadband Remote Access Servers (BRAS). By using ClickOS, the BRAS can be implemented in a highly-modularized fashion facilitating easy software-only updates. By leveraging Xen’s virtualization, on which ClickOS is built, deployment of new features is made easier. |
| Resources Pooled | Processing and Network. |
| Resources in tradeoff | Processing and network. Processing at the BRAS can be used to reduce network load via the below-mentioned features. |
| Cross-Liquidity Considerations | This is a cross-resource tool that, among others, handles functionalities such as filtering, rate limiting and compression. |
| Control Tool | No |
| Deliverable where it is described. | D1.1 |
| Paper (or standard) where it is described. | Thomas Dietz, Roberto Bifulco, Filipe Manco, Joao Martins, Hans-Joerg Kolbe, Felipe Huici, “Enhancing the BRAS through Virtualiztion”, IEEE NetSoft 2015 |

| C3: Adaptive Replica Selection in Cloud Data Stores | |
|---|---|
| Description | The C3 tool enables more predictable performance for partitioned and replicated data stores. C3 runs an adaptive replica selection – wherein a client has to make a choice about selecting one out of multiple replica servers to serve a request – to compensate for the presence of many sources of performance variability in changing system dynamics. C3 consists of mechanisms that reduce tail latencies preferring faster replica servers. The mechanisms make use of approximate feedback information while avoiding herd behaviors wherein a large number of clients concentrate requests towards a fast server. We demonstrated that our solution improves Cassandra's latency profile along the mean, median, and the tail (by up to 3× at the 99.9th percentile) while improving read throughput by up to 50%. |
| Resources Pooled | Storage and Network |
| Resources in tradeoff | Processing and network overhead vs. latency. To choose the replica with the best latency metric, C3 requires replicas to provide extra feedback and performs some additional processing. |
| Cross-Liquidity Considerations | C3 enables liquidity by performing adaptive replica selection on a pool of server replicas that present performance fluctuations. Thus it can reduce latencies and improves throughput. The tool is cross layer as it makes use of (1) approximate feedback information from the servers to adapt replica selection decisions at clients and (2) end-to-end measurements to rate limit requests towards overloaded servers. |
| Control Tool | No |
| Deliverable where it is described. | D1.4 |
| Paper (or standard) where it is described. | L. Suresh, M. Canini, S. Schmid, A. Feldmann, "C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection", USENIX Symposium on Networked Systems Design and Implementation (NSDI'15), May 4-6 2015, Oakland, CA, USA |

| Flow Utility Based Routing: FUBAR | |
|---|---|
| Description | FUBAR is a system that reduces congestion and maximizes the utility of the entire network by installing new routes and changing the traffic load on existing ones. FUBAR works offline to periodically adjust the distribution of traffic on paths. It requires neither changes to end hosts nor precise prior knowledge of the traffic matrix. |
| Resources Pooled | Network. |
| Resources in tradeoff | Latency and network, which are components of the network utility function. |
| Cross-Liquidity Considerations | FUBAR is a routing system that increases overall network utility by exploiting path diversity. This results in both bandwidth gains for bandwidth hungry applications and reduced delay for delay sensitive traffic, increasing overall network liquidity. The tool is cross layer as congestion control involves the transport layer and routing involves the network layer and FUBAR harmonizes their behaviour |
| Control Tool | FUBAR allows network operators to define the utility of end-user applications and express high-level traffic engineering traffic policies. |
| Deliverable where it is described. | D2.4 |
| Paper (or standard) where it is described. | Nikola Gvozdiev, Brad Karp, and Mark Handley. FUBAR: Flow utility based routing. In the 13th ACM Workshop on Hot Topics in Networks (HotNets '14), page 12. ACM, 2014 |

| TCP/HACK | |
|---|---|
| Description | TCP/HACK is a cross-layer mechanism that increases the throughput of TCP operating over wireless networks. TCP/HACK, at its core, encapsulates TCP ACKs within WiFi's link-layer acknowledgments, therefore eliminating all medium acquisitions for TCP ACKs in unidirectional TCP flows and increasing the achievable capacity for TCP workloads |
| Resources Pooled | Network. |
| Resources in tradeoff | Network and Processing. Although TCP ACK encapsulation in link-layer ACKs results in an increase in throughput, we are trading this off with extra processing required to compress/decompress TCP ACKs using Robust Header Compression (ROHC) in the wireless drivers. |
| Cross-Liquidity Considerations | TCP/HACK is a cross-layer optimization. By allowing WiFi's link layer to retain some knowledge about TCP, TCP/HACK achieves significant throughput gains. |
| Control Tool | No. |
| Deliverable where it is described. | D1.3 |
| Paper (or standard) where it is described. | Lynne Salameh, Astrit Zhushi, Mark Handley, Kyle Jamieson Brad Karp."HACK: Hierarchical ACKs for Efficient Wireless Medium Utilization", Usenix ATC, 2014 |

| MultiWifi | |
|---|---|
| Description | Pools Access Point capacity by connecting to multiple access points simultaneously, and spreading traffic over them with Multipath TCP. This technique works for Access Points on the same channel or on different channels. In the latter case, clients will dynamically switch the frequency of their NIC to communicate to APs on different channels. |
| Resources Pooled | Network, Cross-Layer, Cross-Provider. |
| Resources in tradeoff | When switching across channels there is a small "dead" period when the NIC cannot send or receive packets for 3-5ms, depending on the card. With channel switching we waste a bit of capacity (max 10%) to ensure smoother experience while mobile. |
| Cross-Liquidity Considerations | Cross layer: we use Multipath TCP (Layer 4) to enable Layer 2 resource pooling. Cross-provider: Our technique only requires client changes, and can be used with existing hotspot deployments by various operators. |
| Control Tool | No |
| Deliverable where it is described. | Deliverable 1.3 |
| Paper (or standard) where it is described. | Towards Wifi Mobility without Fast Handover. A. Croitoru, D. Niculescu and C. Raiciu. In Proc. NSDI 2015. |

| GRIN | |
|---|---|
| Description | Connects servers directly with patch cords in data center racks. When a server sends or receives, it can use its uplink or the uplinks of its neighbors. It pools the capacity of the core network by allowing more throughput for localized traffic patterns such as many-to-one. |
| Resources Pooled | Network, Processing. |
| Resources in tradeoff | We trade Processing (to forwarding packets on behalf of neighbors) to increase throughput. |
| Cross-Liquidity Considerations | Cross resource: see above. Cross-layer: we wire the data center differently to enable resource pooling at higher layers (routing and transport). |
| Control Tool | No |
| Deliverable where it is described. | Deliverable 1.2 |
| Paper (or standard) where it is described. | Increasing Datacenter Network Utilization with GRIN. A. Agache, R. Deaconescu and C. Raiciu. In Proc. NSDI 2015. |

| Minicache | |
|---|---|
| Description | Minicache is based on minimalized virtual machines that can run in the dozens or hundreds on resource-constrained devices. This minimalization is supported not only by a minimalistic OS, but also by a specialized file system and I/O optimizations compared to standard VMs. Furthermore, this minimalization allows instantiation of VMs in less than 100 ms. These VMs can then be used to create an edge CDN (Content Distribution Network) to support caching content such as video streams close to the users. |
| Resources Pooled | Storage, Processing, and Network. |
| Resources in tradeoff | Storage and network. The larger the storage capacity, the larger the cache can be, increasing the potential of caching content and reducing load on the network core. |
| Cross-Liquidity Considerations | This is principally a cross-resource tool, for distribution of content over the network closer to the user. |
| Control Tool | No |
| Deliverable where it is described. | D1.4, D3.3 |
| Paper (or standard) where it is described. | Simon Kuenzer, Joao Martins, Mohamed Ahmed, Felipe Huici, "Towards Minimalistic, Virtualized Content Caches with Minicache", HotMiddlebox 2013 |

| MPTCP-enabled ClickOS Migration | |
|---|---|
| Description | Targets the migration of virtualized ClickOS Middleboxes across networks of arbitrary size, even changing their IP addresses. To ensure that ongoing end-to-end connections are not broken by this migration and address change, MPTCP is employed. Creating subflows to the new host, the migration of the middlebox functions can be initiated; when migration is finished, the subflow to the old host is terminated, leading to seamless migration. |
| Resources Pooled | Processing and Network. |
| Resources in tradeoff | Processing and network. Additional management overhead is spent on conducting the migration, leading to a better-placed server, for example, saving bandwidth. |
| Cross-Liquidity Considerations | This is a cross-resource tool that allows migration of resources from one network to the other. |
| Control Tool | A small control tool manages the migration, taking care of both the VM migration itself as well as the setup and teardown of the MPTCP subflows. |
| Deliverable where it is described. | D3.1 and D3.2 |
| Paper (or standard) where it is described. | Filipe Manco, Joao Martins, Felipe Huici, "Towards the Super Fluid Cloud", SIGCOMM 2014 Demo |

| VNF Pool enabled Virtual CPE | |
|---|---|
| Description | The VNF Pool enabled virtual CPE allows the execution in virtual environments of those network functions traditionally integrated in hardware at customer premises. It is composed by a set of integrated and orchestrated components for the virtualization of CPE functions in the operator's data center to let business customers incorporate new virtual assets (e.g. Virtual Machines) in their MPLS Layer 3 VPNs, providing a liquid VPN service. It includes VNF Pool based resiliency functions to provide: i) high availability for the virtualized BGP routing protocol engine (*i.e.* BGP protocol configuration, status and provisioned VPNs), ii) scaling-in and -out of VNFs with traffic directed to them depending on their current load or on their functional status. |
| Resources Pooled | Processing and Network. |
| Resources in tradeoff | The placement of the virtual CPE function optimizes the tradeoff between available processing capacity in the server running the CPE function and network capacity available for traffic to reach and leave the selected server. Moreover, the CPE function can be placed, executed and operated by the ISP in different physical locations, i.e. PoPs or data centers, introducing a further degree of tradeoff across processing and network resources. |
| Cross-Liquidity Considerations | The VNF Pool enabled virtual CPE is a cross-resource tool since it treats and orchestrates in a seamless way processing and network resources within the operator's data center. It implements the concept of liquid VPN service, enabling network operators to have substantial benefits on their resource pool usage, improving the flexibility at both data center and network levels |
| Control Tool | The VNF Pool enabled virtual CPE leverages on MPLS control plane functionalities to provide isolation and separation of tenants within the virtualized infrastructure. In particular, it includes an extended Quagga BGP routing suite with full support of Multi-Path protocol capabilities and MPLS Layer 3 VPNs |
| Deliverable where it is described. | D1.3 and D1.4 |
| Paper (or standard) where it is described. | G. Bernini, G. Carrozzo, P. A. Gutierrez, D. Lopez, "Virtualizing the Network Edge: Virtual CPE for the datacenter and the PoP", EUCNC 2014 conference, June 23-26 2014, Bologna, Italy. |
| | G. Bernini, V. Maffione, P.A. Gutierrez, D. Lopez, "VNF Pool Orchestration For Automated Resiliency in Service Chains", IRTF NFVRG individual Internet Draft, 11 October 2015. |

| Federated Market | |
|---|---|
| Description | The Federated Market allows cloud operators to buy and sell resources across the wide area. For instance an operator in the US might get a new customer in Europe. Because of the end of the Safe Harbour agreement relating to data protection this means the operator now needs to host the new customer on hardware that is based in Europe. The operator is able to join the Federated Market and look for operators in Europe who have spare resources they are advertising. Having found a suitable set of resources the US operator can then sell these on to its customer in a completely transparent manner. |
| Resources Pooled | Processing. |
| Cross-Liquidity Considerations | This is principally a cross-provider tool. |
| Control Tool | Yes: the Federated Market allows the trading and control of resources across multiple clouds that are managed by different (often competing) companies. |
| Deliverable where it is described. | D2.4 |

| Federated Block Storage | |
|---|---|
| Description | Federated Block Storage allows the replication of block storage across the wide area. vDisks consisting of one or more stripes can be copied across to a new remote location. Meta-data relating to the vDisk is transferred and used to create a new set of stripes equivalent to the original. For each stripe a secure network tunnel is created and nbd is used to transfer the data across. Once the data is in synch then any changes to the local data can be copied across asynchronously in real time. |
| Resources Pooled | Storage. |
| Resources in tradeoff | Network and Processing. Federated Block Storage trades off network resources to enable replication of block storage across the wide area network. This also requires a certain amount of processing to calculate hash values, *etc.* |
| Cross-Liquidity Considerations | This is principally a cross-resource tool, utilizing the network to enable replication of storage across the wide area. |
| Control Tool | No |
| Deliverable where it is described. | D1.3 |

| vM3 | |
|---|---|
| Description | The Virtual Machine Migration for Mobile (vM3) tool is responsible for creating liquidity on end-user mobile devices. It achieves this by migrating applications encapsulated into Virtual Machines. The migration process is a smooth and rapid transition between devices. This is accomplished by liquidizing the network resource of the devices using the MPTCP protocol. |
| Resources Pooled | Storage, Network, Processing. |
| Resources in tradeoff | By migrating seamlessly the VM from mobile devices to other locations, vM3 it able to trade all resources available in the different locations and optimize whatever criteria it prefers. Since all aspects of mobile devices are in generally constrained, usually there will be a preference to move the VM to more resourceful locations. |
| Cross-Liquidity Considerations | The process of migrating an application encapsulated in a VM consumes network during the transfer, but the end result is a destination device with higher resource availability (more CPU, memory, storage, bigger screen). |
| Control Tool | VM migration is handled by a control component - Virtual Machine Migration Manager. This component is part of the tool itself and it is present on both the sending and the receiving device. |
| Deliverable where it is described. | D1.3 and D3.3 |

| Cloud Liquidity | |
|---|---|
| Description | Cloud Liquidity is a mechanism to allow entire Virtual Machines to be migrated between different locations. Moving a VM from one location to another is a three stage process. Firstly a shadow VM is created which is able to act as an end-point for the data transfer. This Shadow VM is provided with all the VM's meta data relating to ownership, state, resources, etc. Next all the storage has to be duplicated to the new location using Federated Block Storage (see above). Finally once the data has been synchronized the original VM is powered off and the shadow VM is upgraded to being a full VM identical to the original and with access to the same data. |
| Resources Pooled | Processing, Storage. |
| Resources in tradeoff | Network. Cloud Liquidity relies on network resources in order to enable true processing liquidity. |
| Cross-Liquidity Considerations | Principally a cross-layer and cross-resource tool. |
| Control Tool | No |
| Deliverable where it is described. | D3.2, D3.3 |

| Jitsu | |
|---|---|
| Description | Network latency is a problem for all cloud services. It can be mitigated by moving computation out of remote data centers and rapidly instantiating local services near the user. This requires an embedded cloud platform on which to deploy multiple applications securely and quickly. We present Jitsu, a new Xen toolstack that satisfies the demands of secure multi-tenant isolation on resource-constrained embedded ARM devices. It does this by using unikernels: lightweight, compact, single address space, memory-safe VMs written in a high-level language. Using fast shared memory channels, Jitsu provides a directory service that launches unikernels in response to network traffic and masks boot latency. Our evaluation shows Jitsu to be a power-efficient and responsive platform for hosting cloud services in the edge network while preserving the strong isolation guarantees of a type-1 hypervisor. |
| Resources Pooled | Network and Processing. |
| Resources in tradeoff | Jitsu makes the tradeoff of some extra processing (for the proxying and booting) in return for more flexibility with scheduling (to place a VM in the right host at boot time). |
| Cross-Liquidity Considerations | Jitsu provides a dynamic scheduling interface for unikernel-based services, and ensures that (when provided with a suitable SLA policy engine) that only the minimal VM resources are used. It thus provides cross-resource liquidity across Processing and Networking resources, and could be extended to dynamically attach storage nodes in the future as well. |
| Control Tool | Controls launching of unikernels, and reduces latency for launch time, and allows for low latency communications between VMs, plus provides memory safe properties. |
| Deliverable where it is described. | D1.4 |
| Paper (or standard) where it is described. | Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, and David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, and Balraj Singh, Jon Ludlam, Jon Crowcroft and Ian Leslie, Jitsu: Just-In-Time Summoning of Unikernels, NSDI 2015. |

| Trevi | |
|---|---|
| Description | Trevi is a storage system that uses fountain coding to encode writes to multiple servers using multicast, therefore it enables reads of data blobs from multiple servers in parallel. Trevi combines loss resilience to load balancing (resource pooling) and to slow (straggler) in a way that is largely oblivious to the topology of the network and the locations of the storage systems. This is far simpler than using systems that actively monitor load on links and stores and attempt to apply some explicit optimization based approach. It is implicit in the codes we use. The trade off now moves to the design and implementation of rate-efficient and computationally affordable codes, and away from the need for any load balancer design at all. At the heart of Trevi is a simple receiver driven flow control mechanism that does away with the need for reliable transports such as TCP. This mechanism leverages the inherent rateless property of fountain codes and avoids many of the chronic problems that beset TCP within data centers (TCP incast, TCP outcast and timeouts to name three). Trevi is able to make use of multipath availability without the need for any end-host optimizations and without the need for complex flow-aware hashing of data across intermediate paths |
| Resources Pooled | Trevi pools storage resources across the network. While it is not directly involved in processing the requirement for encoding of data at the end points does imply that it will need to draw on processing resources. |
| Resources in tradeoff | Network overhead vs. resilience and manual load balancing and extra processing cost from the coding calculations. |
| Cross-Liquidity Considerations | Trevi is a cross-layer tool in that it combines encoding at the application layer with a simplified flow control at the transport layer and runs on top of multicast at the network layer. |
| Control Tool | No. |
| Deliverable where it is described. | D1.3 |
| Paper (or standard) where it is described. | George Parisis, Toby Moncaster, Anil Madhavapeddy, and Jon Crowcroft. “Trevi: Watering down storage hotspots with cool fountain codes”. In Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks, HotNets-XII, pages 22:1–22:7, New York, NY, USA, 2013. ACM. |

# 3 New Liquidity Tools

This chapter introduces several novel mechanisms for controlling and manipulating liquid resources as part of the Liquid Net architecture. We describe several solutions for deploying liquidity in a data center environment. We also revisit VNF Pool enabled virtual CPE and Federation/Cloud.net with the intent of describing new features and prototypes that have been developed. Finally, we describe Liquid Net solutions aimed at reducing network latency, by either allowing the adaptive selection of replicas, or providing light-weight VMs for moving functionality closer to the end user.

## 3.1 Topologies for NFV Datacentres

### 3.1.1 Introduction

Network Function Virtualization (NFV) is a key liquidity tool. By virtualizing network functions, it is possible to decouple the actual function from the resources used to provide it. Therefore, virtualization allows the creation of large resource pools for each type of resource in the operator's infrastructure, namely compute, storage and network. However, in order to fully realize the potential of the resource pools, it is critical to determine how the physical devices providing the resources are inter-connected. In this section, we look into the design considerations for the interconnection network data centers will use for NFV.

A Network Function Virtualization Infrastructure Point of Presence (NFVI PoP) represents a single geographic location where a number of NFVI-Nodes are situated. An NFVI-Node is a physical device deployed and managed as a single entity providing the NFVI functions required to support the execution environment for virtual network functions (VNFs). In other words, an NFVI PoP is the premises where the processing, storage and networking resources (*i.e.* servers and switches) used to execute the VNFs are deployed. The servers and switches in a NFVI PoP will be interconnected forming the NFVI PoP interconnection network. The goal of this document is to explore the different design considerations for the NFI PoP interconnection network topology, including design goals and constrains.

The NFVI PoP is essentially a data center, and the NFVI PoP interconnection network is essentially a data center network. As such, it is only natural to use the current state of the art in data center networking as a starting point for the design of the NFVI PoP network.

### 3.1.2 Considerations for the Design of the NFVI PoP Network Topology

This section describes the different system properties necessary for the design of the NFVI PoP network topology. In some cases, the values of these properties are known (and sometimes readily available), while in others, the values are not known at this stage.

#### 3.1.2.1 External links

The NFVI PoP is part of the operator's infrastructure and as such it is connected to the rest of the operator's network. Information about the number of links used and their respective capacity is a natural requirement for the proper design of the NFVI PoP topology. Different types of PoPs have different number of links with different capacities with which they connect to the rest of the network. In particular, *local* PoPs connect the links from end users (either DSL lines or Fiber to the Home – FTTH, and so forth) and also connect to the rest of the operator's network. The *regional* PoPs, or regional data centers, have links to the local PoPs, to other regional PoPs and also to other parts of the operator's infrastructure.

For instance, a local PoP in a DSL access network can have between 15,000 and 150,000 DSL lines with speeds between 10 Mbps and 100 Mbps. To connect to the operator's core network, the local PoP can have tens of links with speeds between 20 Gbps and 80 Gbps.

#### 3.1.2.2 Number of servers

To design the PoP network topology, knowing the exact number of servers involved is not required. Instead, an estimate of the order of the number of servers can be used to inform the design decisions. If the resulting topology contains tens of servers, then the topology is likely to be be very simple (*e.g.* a tree-like topology with access/aggregation/core switches). On the other hand, if the topology should encompass several hundred or even a few thousand servers, then the problem is more challenging. In these large deployments, we are likely to surpass the available capacity of existing switches, therefore more sophisticated topologies may be required.

The number of servers on a PoP depends on several factors, which include:

- **External links**: Since external links carry the offered load to the PoP, their number and capacity will impact the number of servers used in the topology. In other words, traffic coming through the external links will require processing by the different VNFs hosted in the servers, influencing the number of servers needed.

- **VNFs**: The number of different VFNs provided in the PoP as well as the number and length of service functions chains will also influence the number of servers required. The more demanding a VNF, the more servers will be needed to support it. Similarly, longer service function chains require a large number of servers.

- **Performance of the VNF implementations**: Some VNF implementations are capable of processing at line speed, while other implementations cannot, requiring additional servers to provide the VNF for the same line speed. While there is some initial work assessing the performance of the different VNFs (as reported in Deliverable 1.1), more work is still needed to provide a full picture for different VNFs at different line speeds.

Overall, we need to have a rough estimate of the range of the number servers that will be part of the PoP network in order to provide a successful design and we need to take into account the aforementioned considerations to obtain it.

#### 3.1.2.3 Traffic Patterns

The expected traffic pattern for the NFVI PoP network is, of course, an important ingredient for the proper design of the network topology. In this section we describe the different possible traffic pattern characteristics, and how they influence the choice of network topology.

**3.1.2.3.1 Macroscopic Behavior:** Traffic within a NVFI PoP network may be classified into 4 different types:

- **Cross-through traffic**: which reaches the PoP through an external link. It is processed by a service function chain (*i.e.* it is processed by a number of VNFs inside the PoP) and then is forwarded back onto an external link. Since the PoP is part of the operator's infrastructure whose goal is to forward users' traffic, processing cross-through traffic is one of the main purposes of the PoP.

- **PoP-generated traffic**: which is generated by VNFs located within the PoP. For example, a cache located inside the PoP serves content to users via PoP-generated traffic.

- **PoP-terminated traffic**: external traffic that is terminated by one VNF located inside the PoP, like a firewall for example.

- **Intra-PoP traffic**: which is generated and terminated inside the PoP, in other words the traffic never leaves the PoP. This traffic includes most of the management traffic, traffic deploying and moving virtual machines and VNFs across different servers, and other signaling traffic (*e.g.* voice call signaling).

In order to properly design the PoP network topology, it is relevant to know the distribution of the expected traffic in these categories.

**3.1.2.3.2 Traffic Patterns within the PoP:** The traffic within the PoP will be composed of essentially two types of traffic:

- The traffic served by the PoP. This is the traffic coming from and/or going to external links which traverses a number of servers where the different VNFs are placed. Cross-through traffic, PoP-generated traffic and the PoP-terminated traffic all fall under this category.

- The operation and management traffic. This traffic includes all messages resulting from the virtual machine and VNF management, as well as any signaling traffic required to provide the VNFs. Intra-PoP traffic falls under this category.

Traffic served by the PoP has a pattern basically determined by the location of the input link, the location of the output link and the mapping of the service function chain (SFC) to servers.

**3.1.2.3.3 Mapping of Service Function Chains to Servers:** There are multiple possible strategies to deploy VNFs and SFCs in servers.

- Parallel SFC deployment strategy: deploys all the VNFs of a given service function chain in a single server and deploys as many of these servers in parallel in order to support the different flows. When more flows arrive at the PoP, more servers are used in parallel.
- Sequential SFC deployment strategy: deploys each VNF in a different server and dedicates one (or more) servers to process a particular VNF for all the flows of the PoP. When the number of flows increases, the number of servers providing each VNF is also increased.
- Hybrid strategy: where several VNFs of the SFC are deployed together in a server and other VNFs of the SFC are deployed in separated servers.

Many factors influence the choice of SFC deployment. A VNF implementation's performance in one such factor, especially if the VNF is too demanding to be executed with other VNFs on the same server. Licensing conditions should also be taken into consideration. Some VNF licenses are based on the number of servers deployed, while others may depend on the number of users served, or even the execution time of the VNF.
Given the SFC strategies described above, any PoP topology design must incorporate:

- The number of servers the traffic served by the PoP will traverse (which is determined by the length of the SFCs and the deployment strategy of SFCs in servers).
- The number of different SFCs that will be simultaneously available in the PoP at any point in time. Different flows arriving on an external link at a particular instant can be served by one or more different SFCs. These SFCs can be mapped to different sequences of servers. Therefore, different flows coming from any external link will have to traverse different sequences of servers, affecting the Intra-PoP traffic pattern.

**3.1.2.3.4 Locality:** Two locality aspects affect PoP traffic patterns. The first aspect is the topological proximity of the servers providing the different VNFs of each SFC (*e.g.* whether or not they are in the same rack). If the SFCs that process the majority of the flows can be assumed to be topologically close, topologies that exploit locality can be useful.
Second, the mapping of traffic from input links to output links will influence PoP topology design. Consider the case of a Local PoP, which has "user links" connecting to users (DSL, FTTH, *etc.*) and "core links" connecting to the rest of the provider's network. It is reasonable to assume that most of the traffic coming from a user link will go to a core link and vice-versa. We can expect that the traffic between two user links will be low, and traffic between two core links will be low as well. If we now consider the case of a regional PoP, it is not so clear that we can make a similar assumption. But wherever the assumption holds, it would be possible to deign a PoP topology that pairs a user links with core links to optimize the transit between them.

**3.1.2.3.5 Growth:** The expected growth in terms of offered load to the PoP and also in terms of VNFs in the PoP represents an important property that cannot be overlooked when designing an NFVI PoP topology. We should understand if the capacity of the PoP is expected to increase linearly or exponentially in time. Similarly, we need to understand if the number of VNFs and the length of the SFCs will remain more or less constant or will evolve, and at which pace. Since different topologies support different growth mechanisms, predicting growth requirements can help determine the suitability of various topologies.

### 3.1.2.4 Technological considerations

**3.1.2.4.1 Direct and Indirect networks:** An Indirect network is made up of two node types: nodes that source/sink traffic and nodes that forward traffic. In contrast, every node in a Direct network plays both roles. Usually data center networks are Indirect networks, with switches that forward packets and servers that source/sink packets. While there have been proposals to use both switches and servers to forward packets, the main concern expressed against them maintains that server resources should be used to execute applications (which is the final goal of the data center) rather than forward packets.
In the case of an NFVI PoP network, the actual purpose of the servers is in many cases to forward packets through the VNFs provided by the server, so it may make perfect sense to use servers to forward packets. From this perspective, either Direct networks or networks that use both switches and servers to forward packets may be attractive for NFVI PoPs.

### 3.1.3 Design Goals

In this section we describe the goals for the design of an NFVI PoP network topology. In broad terms, they include scalability, performance, cost, fault tolerance, operation, management and backward compatibility

**3.1.3.0.1 Effective Load:** A first performance parameter that we should take into account when considering different topologies is the effective load supported by the network. The main goal of the NFVI PoP is to forward traffic between the different external links connected to the PoP. The more traffic a PoP can forward, the more effective it is at managing load, resulting in a better performance. In order to assess the effective load supported by the different topologies, we increase the offered load coming to the PoP through the different links and we measure the effective load that the PoP is able to deliver.

The effective load supported by a topology is likely to be affected by multiple factors, including the the different properties we described in the traffic patterns section above (such as the traffic matrix between the different external links, the characteristics of the SFCs and so on). Additionally, the routing inside the PoP, the different locality considerations, and the intra-PoP traffic also come into play. Moreover, in order for the comparison of any two topologies to make sense, they need to be "equal" in some other dimension (*e.g.* cost, number of servers, number of links, number of switches *etc.*).

For example, as a starting point, we can assume a purely random traffic matrix, *i.e.* every packet arriving through an external link is forwarded through $n$ random servers in the topology and exits through a randomly picked external link. We assume shortest-path, equal cost multi-path routing. We can compare different topologies with the same total number of servers $N$ . We perform the comparison by measuring the effective load when increasing the offered load for different values of $N$ and $n$. Of course, these conditions may greatly differ from real operating conditions, emphasizing the importance of having sufficient knowledge of the system properties described in section 3.1.2.

When performing the evaluation above, it is useful to also measure the packet loss and to track the occurrences of hot-spots in the topology, in order to identify the bottlenecks in the topology which can be improved. Correspondingly, examining the bisection bandwidth of the different topologies may also play an important role in their design.

**3.1.3.0.2 Latency:** Another relevant performance indicator is the latency experienced by packets while traversing the PoP network. For a topology of $N$ servers, we'd like to measure the latency of a packet that arrives through an external link, traverses $n$ servers and exits through an external link. Since we only care about the latency induced by the topology itself (in order to assess the topology) we can measure the "latency" as the number of hops that the packet should traverse.

In addition to measuring the mean latency, we must also record the maximum latency, since an upper bound for the time a packet stays in the PoP is also relevant. Again, the latency/hop count depends on the traffic matrix (*i.e.* the mapping of traffic from the input to output links), the routing and the different locality properties, and knowledge of these parameters would be beneficial. Barring that, a purely random configuration similar to the one described in 3.1.3.0.1 could be used as a starting point.

**3.1.3.0.3 Scalability:** Scalability refers to how well the proposed topology supports growth in terms of the number of servers, server line speed and the capacity of the external links. Some topologies require growing some components beyond what is technically feasible (or what is economically efficient) in order to support an increased number of servers. For instance, it is well known that tree topologies require the core switches to grow in order to support more servers, which is not feasible beyond a certain point (or it becomes very expensive). That being said, we should consider scalability in the range of servers that we expect a PoP will have to support within a reasonable time frame.

Another related property is how well the different topologies support incremental growth. It is unclear at this point what the growth rate for the NFVI PoPs will be. In other words, given that we have a PoP with $N$ operational servers, would we need to increase the number of servers to $N+1$, $2*N$ or $N*N$? Different topologies have different growth models. Some support linear growth indefinitely, others can be over-dimensioned in order to support some linear growth, but after a given number of additional servers, they need to grow exponentially.

**3.1.3.0.4 Fault Tolerance:** Fault tolerance is of course paramount for an NFVI PoP network. When choosing the right topology, we must consider its fault tolerance properties. In particular, we care about how well the topology handles link failures, switch failures and server failures.

We can assess the fault tolerance of a topology by measuring the following parameters:

- Node-disjoint paths: The minimum number of paths that share no common intermediate nodes between any arbitrary servers.
- Edge disjoint paths: The minimum number of paths that share no common edges between any arbitrary servers.
- f-fault tolerance: A network is f-fault tolerant if for any $f$ failed components, the network is still connected.
- Redundancy level: A network has redundancy level of $r$ if and only if it remains connected after removing any set of $r$ components, but becomes diconnected after removing a set of $r+1$ components.

**3.1.3.0.5 Cost:** The cost of the resulting network is an important design goal to keep in mind. In order to assess a topology's cost, we can consider the number of switches and the number of interfaces in the topology for the same number of servers. We should also take into account the type of switches required, as we know that the cost of a switch does not scale linearly with the number of interfaces of the switch and with the speed of the interfaces.

**3.1.3.0.6 Backward Compatibility:** Compatibility with existent hardware represents the final design goal. It is unlikely that operators will throw away all their current infrastructure, replacing their specialized hardware with VNFs running on COTS servers. An incremental deployment is more likely, where some functions are virtualized and some functions executed in hardware. It is therefore important to consider how the different topologies will support such hybrid scenarios.

## 3.2 Prototype of the VNF Pool enabled virtual CPE

### 3.2.1 Introduction

The virtual Customer Premises Equipment (CPE) allows the execution of network functions traditionally integrated in hardware at customer premises in virtual environments. Operators are looking to transform future data centers into more dynamic infrastructures with flexible network architectures, on-demand services for their customers, and high elasticity to scale up and down while optimizing performance and resources utilization. Business customers that interconnect their sites (*e.g.* offices, headquarters, labs, *etc.*) through IP/MPLS VPNs services offered by the network operator may want to buy additional sets of Infrastructure as a Service (IaaS) resources (e.g. VMs, storage areas, etc.) from the same operator. Therefore, these customers want to easily and dynamically extend these Layer 3 VPNs to incorporate new virtual assets into a private cloud. To this purpose, we have implemented a set of integrated and orchestrated components for the virtualization of CPE functions and automated liquid VPN service provisioning, including VNF Pool based resiliency functions. This section provides a brief description of the developed proof-of-concept prototype, implemented by Nextworks in collaboration with Telefonica for the BGP routing part.

The VNF Pool enabled virtual CPE leverages on MPLS control plane functionalities to provide isolation and separation of tenants within the virtualized infrastructure. In particular, this virtual CPE implementation integrates an extended Quagga BGP routing suite with full support of Multi-Path protocol capabilities and MPLS Layer 3 VPNs. Thus, it can be considered a cross liquidity tool since it treats and orchestrates in a seamless way processing and network resources within the operator's data centre. The virtual CPE implements the concept of liquid VPN service, enabling network operators to gain substantial benefits from their resource pool usage, and improving the flexibility at both data centre and network levels.

The virtual CPE is also a practical demonstration of how SDN and NFV technologies can be integrated to provide substantial benefits to network operators in terms of robustness, ease of management, control and provisioning of their network infrastructures and services. SDN and NFV are indeed clearly complementary solutions for enabling virtualization of network infrastructures, services and functions while supporting dynamic and flexible network traffic engineering.

### 3.2.2 Reference Architecture

The VNF Pool enabled virtual CPE in the operator's data centre has been presented in deliverable D1.3. Its reference architecture is reported again in this document for sake of completeness of this section. As

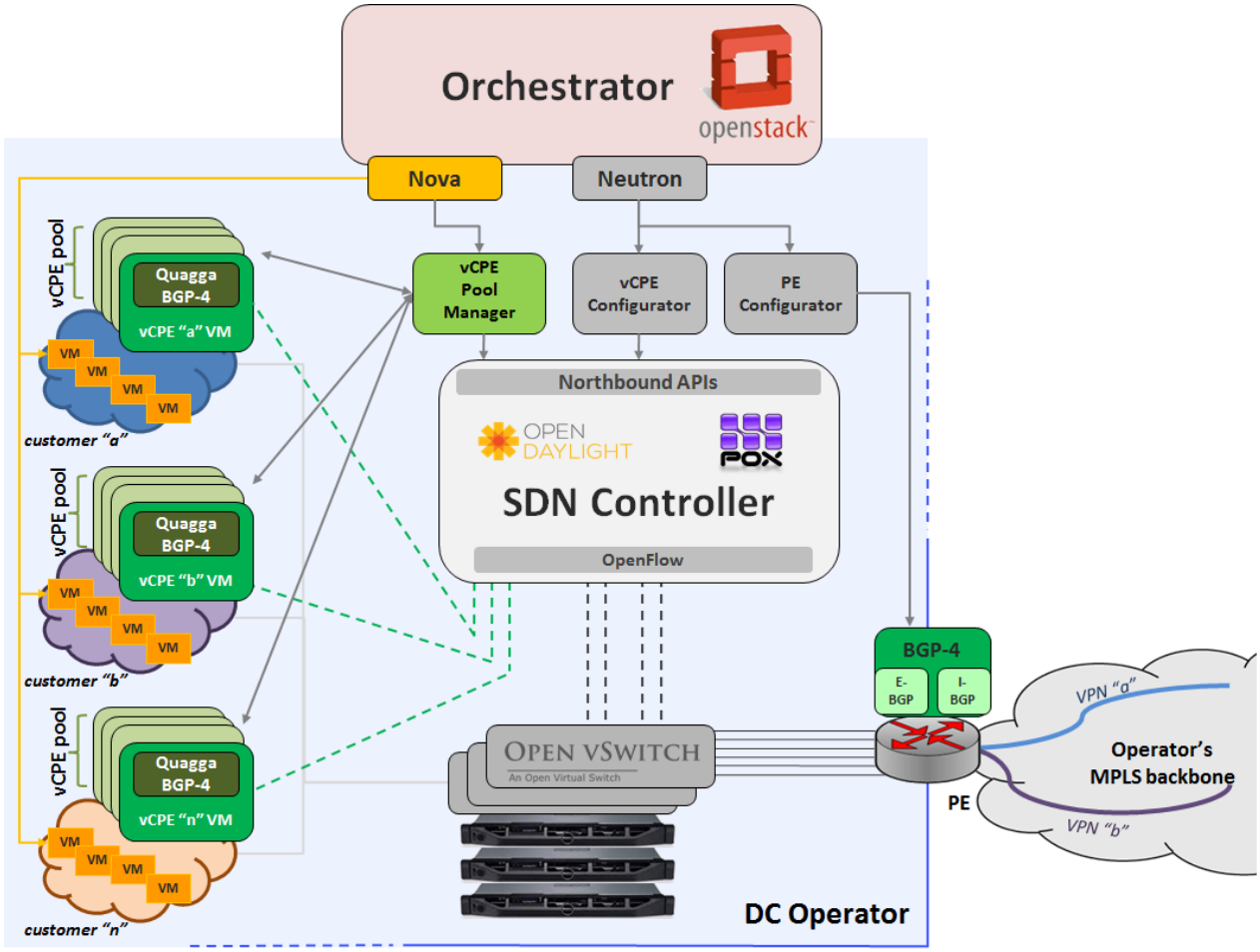



Figure 3.1: VNF Pool Enabled Virtual CPE functional decomposition

described in the introduction, the VNF Pool enabled virtual CPE scenario is aligned with the current trend followed by most of network operators, which are extending their pure network based product and service portfolio (mostly including connectivity offers), towards a more integrated offer including IaaS. They leverage proprietary data centers where specialized NFV and SDN solutions can be deployed to ease the provisioning and orchestration of virtualized network functions and services. In this liquid VPN service implementation, the virtual CPE is basically an additional VM under the control of the operator, deployed on demand to dynamically configure the customer VPN memberships and Virtual Routing and Forwarding (VRF) instances, *i.e.* dynamically join or release customer VMs to the proper VPN. When a customer requests to buy some IaaS resources and include them into a private cloud accessible through its Layer 3 VPNs already in place, the virtual CPE VM is deployed on-demand as part of the IaaS to extend its VPN. This is implemented by running Open virtual Switches (OVS-es) in the data centre hypervisors, which allow to perform VPNs and IaaS service chaining via software, properly managed and controlled by an SDN and NFV enabled orchestration framework. The VNF Pool enabled virtual CPE architecture and functional decomposition is shown in Figure 3.1, where the building blocks are highlighted along with their interactions. The core part is the SDN controller (one for the whole data center). In combination with a set of enhanced network applications, the controller provides on-demand flow and routing tables configuration at the OVS and data center edge router. Additionally, an Orchestrator integrating OpenStack [63] as Virtual Infrastructure Manager is owned by the network operator to control its data centers (possibly more than one). The Orchestrator uses dedicated modules such as OpenStack Nova [57] and Neutron [55] to perform on-demand deployment and configuration of the IaaS resources (mainly VMs, storage, *etc.*) a requested by the business customers. A detailed description for each of the VNF Pool enabled virtual CPE functional module is available in deliverable D1.3.

### 3.2.3 Prototype Implementation

A proof-of-concept prototype of the VNF Pool enabled virtual CPE depicted in Figure 3.1 has been developed by Nextworks, with the collaboration of Telefonica for the routing protocol part, in support of the liquid VPN service scenario described above. The aim has been to implement a set of components integrated in a software tool. The components will dynamically provision an operator's business customer VMs and automatically joining them to existing MPLS VPN services. Isolation and separation of customers (i.e. tenants) is a must for virtualized infrastructures provisioned within data centers. This VNF Pool enabled virtual CPE implementation uses MPLS VPNs and BGP as control technologies to guarantee this separation. The following subsections provide a brief description for the components developed for this proof-of-concept prototype of the VNF Pool enabled virtual CPE.

#### 3.2.3.1 Overview of Components

The VNF Pool enabled virtual CPE has been implemented as a set of software components developed by Nextworks and integrated in a liquidity tool for the network operator. The tools deploys, configures and operates liquid VPN services within data centers owned by network operators, following the approach summarized above and detailed in deliverable D1.3. An overview of the developed software components is presented in Figure 3.2, where all the building blocks of the virtual CPE prototype are depicted, including both control and data plane functionalities and interactions. In the picture, the different LANs depicted represent three customers with VMs belonging to different VPNs deployed in the data center. Three core components form this liquidity tool: the orchestrator, the SDN controller with its virtual router application, and the virtual CPE Virtual Machine (VM). Their main role is respectively to: i) coordinate the liquid VPN service provisioning and configuration (interacting with most of the components), ii) configure the operator's data center network forwarding entities and properly route both control and data traffic, iii) implement the BGP protocol to dynamically configure MPLS VPNs within the data center and isolate customer's VMs traffic while joining them to the proper VPN outside the data center. On top of these software components, the virtual CPE pool manager implements basic high availability and resiliency functionalities for the virtual CPE VM, in particular concerning the BGP protocol configuration, status and provisioned VPNs. While these four components build the control (and orchestration) functions of the virtual CPE prototype, its data plane is composed of the integration of the OVS instance and the OpenStack integration bridge. While the former is the forwarding plane for the virtual CPE VM running the BGP protocol, and is used to route control and customer's VM data traffic, the latter is the OpenStack forwarding entity maintained by Neutron to provide isolation of traffic coming from the customer's VMs. The Provider Edge (PE) router is considered out of scope for the implementation of this virtual CPE prototype. For testing and demo purposes, we deployed the PE as an additional VM running the same BGP protocol software of the virtual CPE VM. In the following subsections, we will briefly describe each of the software components providing control and orchestration functionalities.

#### 3.2.3.2 Orchestrator

The orchestrator is the network operator's entry management point of the virtual CPE. It coordinates the deployment, configuration and provisioning of all the components shown in Figure 3.2. In this virtual CPE prototype, the orchestrator is implemented as a set of Python scripts to automate the liquid VPN service provisioning. Provisioning involves the deployment and configuration of the virtual CPE VM and SDN controller. Additionally, provisioning takes care of the creation or destruction of the VRFs instances, customer LANs and VMs. The orchestrator is integrated with OpenStack, which is used to create VM instances and virtual networks within the operator data center. To this purpose, the orchestrator uses the OpenStack Nova and OpenStack Neutron components respectively. The customer VM's tagged traffic moves between OpenStack's integration bridge and the OVS datapath by means of a patch port, for all the Neutron networks. The orchestrator also maintains all the bindings between virtual CPE, OVS and OpenStack integration bridge ports and VLANs needed to isolate the customer VM's traffic. In particular, as shown in Figure 3.2, VLAN based isolation is used between the OVS (as the forwarding plane of the virtual CPE) and the PE, with a dedicated VLAN $p$ for each VRF (i.e. the MPLS VPN virtual routing instance) configured within the virtual CPE VM. These are kept mapped by the orchestrator (and enforced in the OVS) with the internal VLAN tags $n$ used by the OpenStack integration bridge to separate traffic belonging to different Neutron networks. Moreover, the orchestrator maintains a simple database where all the current configurations (virtual CPE VM and SDN controller in particular), port and VLANs bindings, address spaces, list of active VRFs, *etc.* are

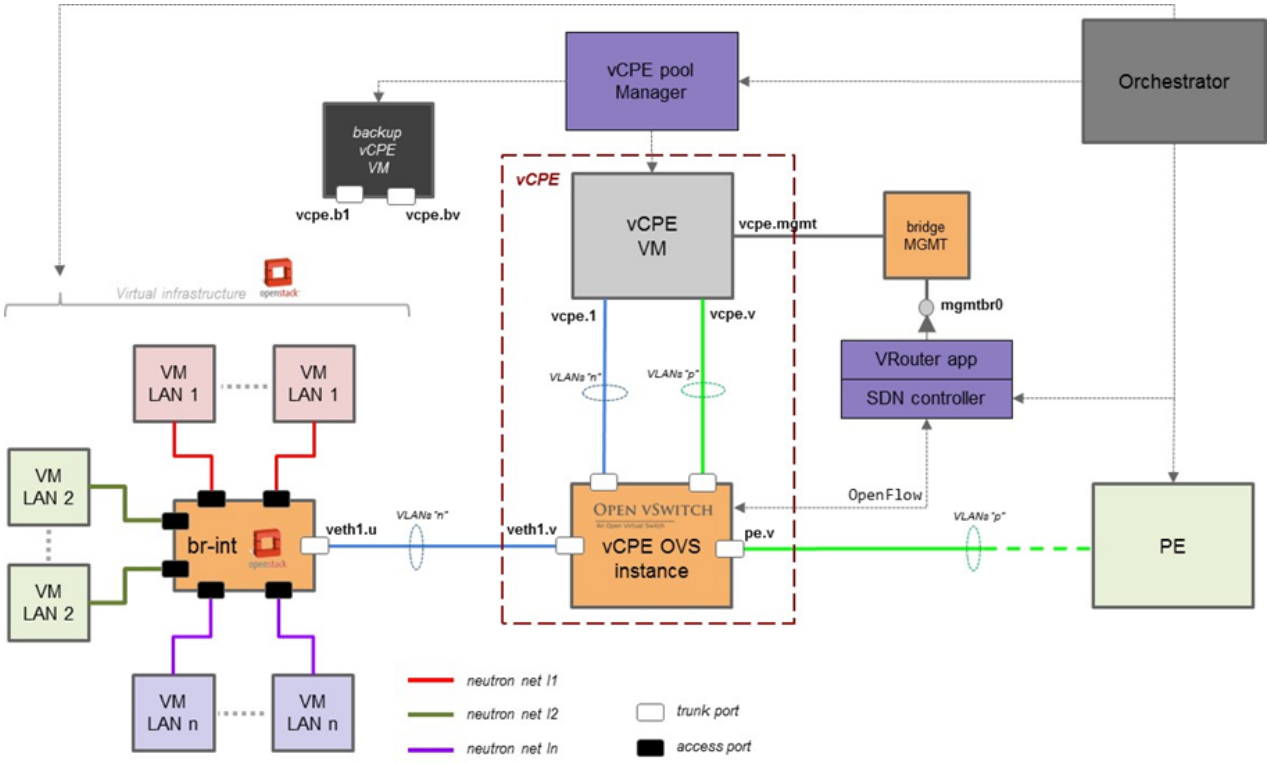



Figure 3.2: VNF Pool Enabled Virtual CPE prototype software components.

stored and can be queried. In practice, the orchestrator offers a set of management primitives in the form of Python scripts which coordinate all the deployment, configuration and provisioning actions to be performed for the liquid VPN service. The full list of available orchestration scripts, including a short description and involved components, is provided in Table 3.1.

#### 3.2.3.3 Virtual CPE VM

The virtual CPE VM is the virtual network function provided by this proof-of-concept prototype and includes a set of software tools and applications. The Quagga routing suite [67] is the most important piece of software running within the VM, and it is the virtual CPE componenent where the colloboration between Nextworks and Telefonica happened. Quagga is an opensource software suite that provides TCP/IP based routing services for UNIX that support of a variety of routing protocols, including RIP, OSPF, ISIS and BGP. The core part of Quagga is the Zebra deamon, which provides an abstraction layer over the underlying Unix kernel. Therefore, it offers a set of APIs over a Unix or TCP stream to the Quagga protocol clients. In this virtual CPE prototype, the Quagga BGP protocol deamon is used to terminate MPLS VPNs and configure VRFs within the data center. In particular, a set of extensions and enhancements to Quagga v0.99.23 have been applied to have full support of Multi-Protocol BGP features, MPLS VPNs and distribution of VRF identifiers:

- Integrated an opensource patch developed by 6Wind to have support of VRFs in Zebra.
- Integrated a patch developed by Telefonica (see next section) to have full Multi-Protocol BGP support in Quagga, with MPLS VPN implementation including Route Target and Route Distinguisher usage.
- Full-implementation of VRF support and distribution in Quagga BGP protocol (including data structures, protocol interfaces, messages).

In addition to these extensions, extensive bug fixing to solve several issues in the Quagga BGP protocol has been carried out by Nextworks. In particular, bugs concerning the distribution of VRF identifiers and IPv4 network prefixes within MPLS VPNs have been tackled. The actual implementation of the VRFs within the virtual CPE VM has been done by using linux network namespaces. In practice, Quagga is used in this prototype for virtual CPE routing table management. The virtual CPE routes can be either installed manually

| Orchestration Script | Description | Arguments | Components Involved |
|---|---|---|---|
| *bootstrap* | Prepares and bootstraps the virtual CPE environment. It starts and configures all the virtual CPE software components, including the virtual CPE VM, the SDN controller with vRouter, the VNF Pool manager, the OVS instance, and OpenStack. It creates the internal management network to let the virtual CPE VM talk to the vRouter application. It also initializes the orchestrator database. | Configuration file (optional): loads a start-up configuration scenario which specifies the VRFs, and the customer's VMs and LANs. | All |
| *teardown* | Cleans the data center environment by tearing down all the deployed VMs and removing all the VRF instances and network configurations. | None | All |
| *vrf-add* | Creates a new VRF. It is the first step to join a set of customer's VMs to an MPLS VPN already established outside the data center. | *name*: the ID of the VRF. *upprefix*: the IPv4 prefix for the VLAN link between the vCPE and PE. | Virtual CPE VM, VNF Pool manager. |
| *vrf-del* | Removes an existing and provisioned VRF. | *name*: the ID of the VRF. | Virtual CPE VM, VNF Pool manager |
| *neutron-lan-add* | Creates a new OpenStack Neutron network for customer VMs to be attached to the integration bridge. | *name*: the ID of the customer LAN *vrf*: The ID of the VRF this LAN will be part of. *prefix*: The IPv4 prefix to use in the LAN | OpenStack, OVS, Virtual CPE VM. |
| *neutron-lan-del* | Deletes of an existing OpenStack Neutron network attached to the integration bridge. | *name*: the ID of the customer LAN | OpenStack, OVS, Virtual CPE VM. |
| *nova-vm-add* | Deploys and starts a new customer VM within an existing customer LAN. The deployment is performed through OpenStack Nova, which automatically assigns an IP to the VM. | *name*: the name of the VM *lan*: the ID of the customer LAN this VM will be part of. | OpenStack, OVS. |
| *nova-vm-del* | Stops running a customer VM within an existing customer LAN. The process is performed through OpenStack Nova. | *name*: the name of the VM | OpenStack, OVS. |

Table 3.1: Virtual CPE orchestrator management scripts.

(*e.g.* using the IP route command, or by adding the routes for the networks directly attached), or learned by Quagga through the BGP session established with the PE. Since the actual virtual CPE routing functions in this prototype are provided by the combination of the vRouter SDN application (*i.e.* the software router function) and the OVS (*i.e.* the forwarding plane), we need to push the routing tables maintained by Quagga to the vRouter to properly configure them in the OVS forwarding plane. To this end, we have developed *fprt*, a Forwarding Plane Manager (FPM) dedicated application deployed within the virtual CPE VM. *fprt* is a Python application that uses Quagga Forwarding Information Base (FIB) push interface to receive route updates from Quagga via a local TCP/IP socket and forward them to the vRouter application via another TCP/IP socket through the `vcpe.mgmt` inteface in Figure 3.2. These route updates, used to add and remove routes within the vRouter, are encoded into `NEWROUTE` and `DELROUTE` commands.

#### 3.2.3.4 SDN Controller and vRouter

In this virtual CPE prototype, the POX SDN controller [66] has been used as the starting point for the implementation of the virtual CPE software router functions. POX is a opensource development platform for SDN control applications. Essentially an OpenFlow controller, POX provides a Python based OpenFlow interface with few additional features, like topology discovery. It communicates with OpenFlow 1.0 switches and includes special support and integration with OVS. For the virtual CPE prototype, a new virtual Router (vRouter) Python application has been implemented from scratch. Additionally, an implementation of the same functionalities on top of the OpenDaylight Java SDN platform [59] is in progress (including porting from Python to Java) and will take most of the effort for the next steps after TRILOGY2 (see Section 3.2.5). The vRouter application is a software virtual router on top of the forwarding plane implemented with OVS, and provides a set of new functions and services within POX:

- ARP Handler.
- Integration with Quagga FPM.
- Mirroring of vCPE routing tables, updated from Quagga.
- On-demand installation of flows into OVS.
- VLAN support.

In particular, a vRouter uses the POX OpenFlow interface to receive "`PACKET IN`" events related to ARP responses and customers' new IP connections, and to send the ARP requests necessary to discover next-hop MAC addresses for customers' traffic. Concerning the control plane functions, the vRouter maintains a mirror of the vCPE VM routing table, receiving updates from the *fprt* application running in the virtual CPE VM, in the form of `NEWROUTE` and `DELROUTE` commands. When a new customer connection comes to the vRouter from OVS (*i.e.* through a `PACKET IN`), if it can be routed using the vCPE routing table a flow is installed in OVS through OpenFlow. Packets waiting for an ARP response to be routed are enqueued in ad-hoc vRouter queues. In addition, for management purposes, vRouter accepts `NEWPORTMAPPING` and `DELPORTMAPPING` commands from the orchestrator to bind or unbind the vCPE VM ports to VLANs towards the customers' LANs and PE. Also, vRouter accepts `NEWVRFMAPPING` and `DELVRFMAPPING` commands to associate or disassociate a VRF, a VLAN and an IP address to a vCPE VM port. This is crucial to implement VLAN based traffic isolation (at the OVS level) across MPLS VPNs belonging to different customers. The vRouter can be started by the orchestrator with a startup configuration describing the `NEWPORTMAPPING` and `NEWVRFMAPPING` to be loaded.

#### 3.2.3.5 Virtual CPE Pool Manager

The virtual CPE pool manager is a Python application providing some basic redundancy and high availability functions. In this proof-of-concept prototype, such functions are limited to the virtual CPE VM as the only virtual network function provided. The virtual CPE pool manager basically acts as a configuration and provisioning manager for the pool of virtual CPE VMs, thus providing redundancy functions transparent to the orchestrator. Each virtual CPE VM control and provisioning action performed by the orchestrator is transparently managed by this pool manager in order to have consistent and aligned active and backup virtual CPE VMs at all times. In the prototype, the pool is composed of two VMs (*i.e.* with only one acting as a backup). The virtual CPE pool manager implements a simple periodic `keepalive` mechanism to check the status of

the active virtual CPE VM. When it crashes, the pool manager takes care to swap the active and backup VMs. The virtual CPE pool manager dynamically manages and adds redundancy to the following configuration actions: the configuration of VLAN interfaces and IP addresses on the VM ports, the configuration of linux namespaces to implement the VRFs, and finally the configuration of Quagga BGP protocols.

### 3.2.4 Components interactions

Figure 3.3 depicts the main interactions between the virtual CPE components. Interactions marked in red refer are management actions mostly triggered by the orchestrator. Some of these management actions create new LANs and VRF instances that need to join the customer's MPLS VPNs, others deploy new customer VMs and configure the virtual CPE components in general for management purposes. Interactions marked in black, on the other hand, represent dynamic updates to routing and forwarding tables spread across the different virtual CPE components, i.e. the OVS instance, the vRouter and the Quagga BGP protocol instance in the virtual CPE VM.

The following list summarizes the interactions which occur when the orchestrator creates a new VRF, a new customer LAN (*i.e.* OpenStack Neutron LAN) and a new customer VM in sequence. Using the *vrf-add*, *neutron-lan-add* and *neutron-vm-add* scripts, the orchestrator:

(i) **Updates the network configuration of the virtual CPE VM**. The orchestrator creates a new linux network namespace to host the new VRF, and creates and configures a new VLAN on the interface facing the PE (*i.e.* the *vcpe.v* interface in Figure 3.2). The new VLAN's ID is chosen to be identical to the VRF ID.

(ii) **Updates the Quagga configuration**, by adding the new VRF in the Quagga BGP configuration, thus connecting to the protocol deamon running in the virtual CPE VM through telnet.

(iii) **Sends a `NEWVRFMAPPING` command to the vRouter** to notify it about the new VRF/VLAN mapping becoming active on the *vcpe.v* interface.

(iv) **Creates a new customer LAN**. The orchestrator adds a new OpenStack Neutron network to the integration bridge, and creates a new Neutron subnet.

(v) **Updates the network configuration of the virtual CPE VM**, by allocating a new vCPE port or selecting an already allocated vCPE port (*i.e. vcpe.1*) using an unused VLAN ID on that port.

(vi) **Updates the OVS instance configuration**. The orchestrator creates the patch port between the OVS instance and the OpenStack Neutron integration bridge, if it hasn't been created yet.

(vii) **Sends a `NEWPORTMAPPING` command to the vRouter** if a new virtual CPE VM port was allocated (*i.e. vcpe.1*), to advertise the new port binding involving the LAN interface.

(viii) **Sends a `NEWVRFMAPPING` command to the vRouter**, to advertise the new VRF/VLAN mapping becoming active on the LAN interface.

(ix) **Updates the Quagga configuration**, and adds the OpenStack Neutron network prefix of the new LAN to the Quagga BGP VRF configuration, once again connecting through telnet to the protocol instance.

(x) **Creates a new customer VM** through the OpenStack Nova service specifying the OpenStack Neutron network to be attached to.

### 3.2.5 Next steps

The prototype described in this section has been implemented by Nextworks as a proof-of-concept for the VNF Pool enabled virtual CPE fully described in deliverable D1.3. Currently several consolidation and extension tracks have been identified and started by Nextworks and Telefonica towards a more complete and comprehensive virtual CPE prototype. First, the plan is to integrate the current virtual CPE VM with additional virtual network functions, like virtual firewall, virtual intrusion detection and prevention systems (vIDS and vIPS), with the goal of obtaining a heterogeneous and orchestrated set of VNFs deployed and operated within the data center. Moreover, the current virtual CPE pool manager resiliency functions will be extended

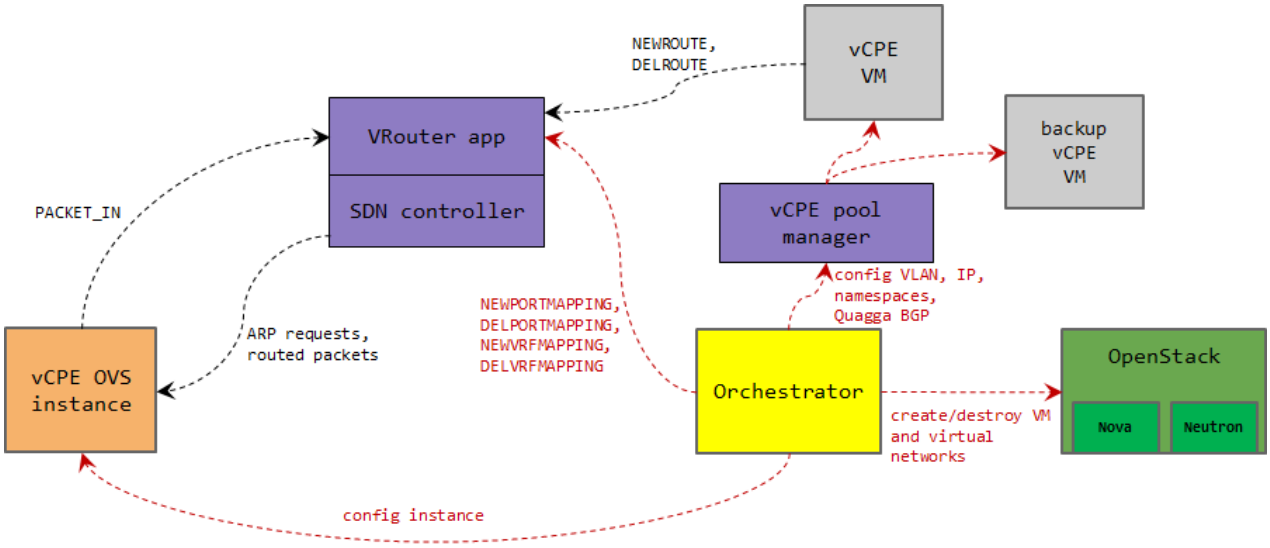



Figure 3.3: Intractions between the virtual CPE components.

to other components (*e.g.* the vRouter and orchestration database) to increase the set of highly available control functions. A further enhancement track (that is already in progress) involves replacing the POX SDN controller with the OpenDaylight platform, currently the de-facto standard for SDN deployments in data center environments. It provides a wide range of southbound and northbound protocols and interfaces, and will therefore ease the integration of the virtual CPE with other NFV components and tools. In this context, the virtual CPE orchestrator will be integrated with existing opensource NFV Orchestrator (NFVO) software tools. Currently, the tools under investigation are: OpenMANO [62], OpenBaton [61] and OpenStack Tacker [82].

## 3.3 MPLS/VPN Embedded Quagga Routing Control Plane

### 3.3.1 Introduction

The previous section describes the integration in the Nextworks virtual CPE implementation of an extended version of the MPLS/VPN enabled Quagga daemon implemented by Telefonica and described in previous deliverables of TRILOGY2. This component has been also used in different proofs-of-concept (PoCs) in relation of the virtual CPE in Telefonica.

### 3.3.2 Reference Architecture

The MPLS/VPN enabled Quagga routing control plane (referred to as *router* in this section) is a VM which exposes a Forwarding Plane Manager (FPM) interface to send routing table updates to an external data plane switching component.
The following figure shows the interfaces exposed by the router:

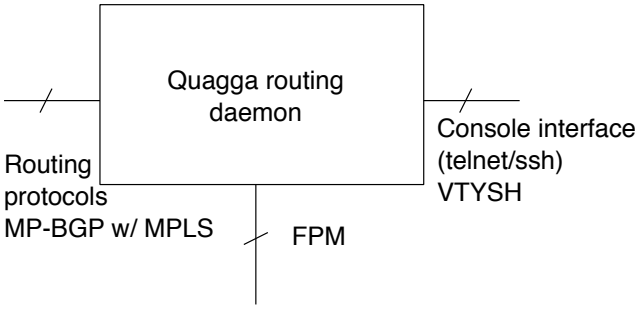



Figure 3.4: Interfaces exposed by the Quagga router.

### 3.3.3 Prototype Implementation

The prototype implementation uses a minimalist Ubuntu 14.04 LTS which only includes the components and libraries needed for the Quagga routing daemon to work. This VM is used as the base for a PoC implementation of a Carrier-Grace NAT (CGNAT) and of an accelerated IPv4 router, which can be used as a virtual network function (VNF).

### 3.3.4 Next steps

The VM is currently the basic component for VNF implementations that are used in different experimental facilities in the scope of VNF activities at Telefonica, I+D. Currently, the VMs use Intel's Dataplane Development Kit (DPDK) [40] to implement accelerated virtual network functions. We intend to continue this development in TRILOGY2 follow-up activities, and, as a result of our exposure to netmap [70] during the project, we plan to migrate the VM to use it.

## 3.4 Federation and Cloud.net

### 3.4.1 Introduction

During the course of TRILOGY2 Project, OnApp has investigated the ability to federate Cloud resources across the wide-area. In Deliverable D1.1 the initial ideas behind the OnApp Federation were discussed along with the motivation for generating a testbed platform. Initial platform work was carried out to determine what would be needed to allow disparate cloud sites to communicate with each other and allow resource sharing across providers. A limited testbed was set up between partners to determine how resource sharing could be enabled and an OCCI compliant connector for the CompatibleOne project was created.

The early limitations of the platform were soon discovered and a more scalable architecture was decided upon that required a re-write of the platform. This work was carried out in the second-half of the first year of the project and resulted in the Federation being released as an alpha version on 28th January 2014 and a limited beta on 1st July 2014. Customers have been taking advantage of the Federation since then for provisioning, through buying/selling/trading, and utilizing compute resources in multiple locations. In addition a Virtual Service Provider (VSP) offering was produced in the form of Cloud.net. This platform allows customers without infrastructure to buy resources and provide services on top of infrastructure that they do not own. This is a key realization of the Liquid Net - where the resources can be distributed with the service providers and business logic as a separate layer to the infrastructure.

The Federation, see Figure 3.5, has allowed customers to sell excess resources to other providers and has been exploited as an interesting value-added proposition for choosing OnApp cloud over other offerings. The buying of resources presented by the Federation has been more complicated than expected: customers want a different set of assurances than initially expected and offered. Metrics that expose some of the underlying workings of the platform to allow customers to make a choice have not been used in the expected way. These findings will be discussed later in Section 3.4.5.

'Disaster Recovery-as-a-Service' (DRaaS), a service that has been developed in the course of TRILOGY2, exposes the features of the Federation and enables storage liquidity. DRaaS allows continuous block replication between cloud sites that are owned by different providers, which in turn allow fail-over and fail-back. DRaaS is a real-world business use-case of the Liquid Net as promoted by TRILOGY2. More of the details regarding DRaaS, its implementation and its deployment with relation to the Federation is described in the scope of Work Package 3 (WP3).

In this Deliverable we highlight the new features that help to enable the TRILOGY2 vision of the Liquid Net in Section 3.4.2. We then describe the Cloud.net features that allow customers to buy, sell and trade resources without owning infrastructure, which have been researched and developed as part of TRILOGY2 in Section 3.4.3. The linkage of the use-case, developed in WP3, DRaaS to the Federation is described in Section 3.4.4. Some of the new features and expected work that will be carried on beyond TRILOGY2 as well as the lessons learned are described in Section 3.4.5. We then describe the conclusions in Section 3.4.6.

### 3.4.2 Features introduced to Federation

#### 3.4.2.1 Billing Engine Integration

One of the key items for enabling liquidity in a commercially acceptable manner is the ability to tie the Federation to billing systems that can track the utilization of resources by different users. OnApp has a separate resource tracking system for each of its licensed customers for purposes of internal billing. For the

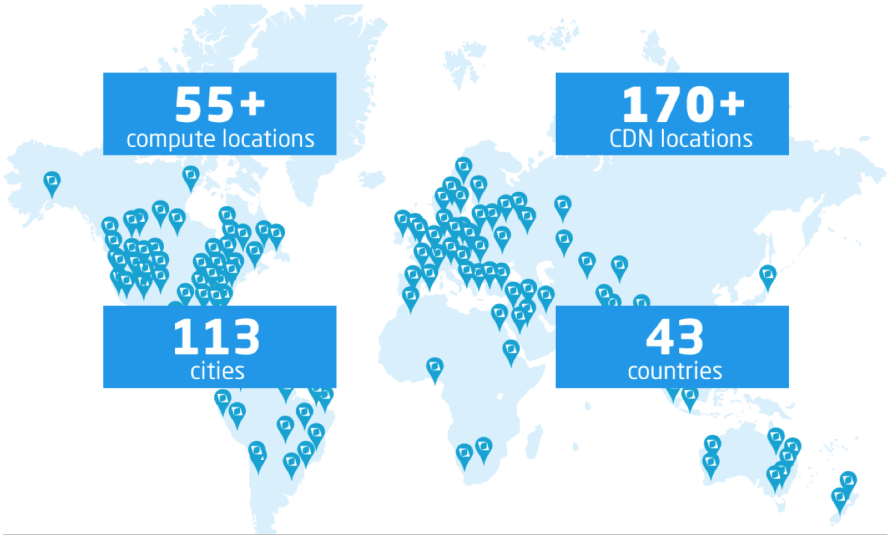



Figure 3.5: OnApp Federation map showing the number of compute and CDN locations distributed across the world.

Federation, it has been important to have a more finely-grained resource tracking system to capture real-time resource usage and in particular utilization of VMs. The buyer and seller of resources can independently keep records of the resource utilization that they expect, but the OnApp Market as a trusted entity will report the figures directly to Zuora[1] (a billing platform). The seller can then use the billing figures produced by Zuora to invoice the buyers for the resources utilized. Separately the buyers will have a billing plan associated with what they charge VM owners and this can also be used to generate reports based on the information presented to Zuora. A PostgreSQL database is used to maintain the billing information locally.

#### 3.4.2.2 Additional Consistency Logic

Having the addition of a remote site has led to quite a few consistency check points being needed along with additional transactions. These are needed such that the state of the cloud platforms is consistent between the buyer, market and sellers. As such, quite a few dependencies on performing operations on the buyer or seller first, prior to the market have been developed. These include:

- Removing firewall rules from the market only after they have been removed from the supplier.
- Removing backup from the market only after it has been removed from the supplier.
- Sending pricing update messages from the market to the buyer if the seller pricing has changed.
- Removing VM resources and meta-data from the market only once they have been removed from the supplier.
- Removing all associations with the VMs when a particular VM is removed.
- Flushing billing data before removing a VM on a seller.

Additionally, the Federated Market has an extra dependency that is needed due to the billing system integration, which is to assure that the Zuora account is created when a new market user is added in the Dashboard licensing system. The dashboard also has to assure that the Zuora account is linked correctly with the subscription details from the Market for being able to receive usage data.

#### 3.4.2.3 Increased Functionality in Federation

As the Federation grows in popularity, more functions are being requested from customers that are available in the main platform but are not currently present on the Federation. Some of these features have required re-designing to be idempotent in order to avoid repeated signals causing race conditions. The model of the

[1] https://www.zuora.com/

VM on the market has also had to be expanded to take into consideration the extra functionality that has been required.
Some of the new features include:

- The ability to reboot VMs in recovery mode, so that the VMs can be for repaired.
- The ability to rebuild a VM instance on a seller site.
- Adding a Virtual Disk to a VM.
- Deleting a Virtual Disk from a VM.
- The ability to rebuild the network associated with a VM.

#### 3.4.2.4 More Accurate Display of Resource Zone Availability

In previous incarnations of the Federated Market, the time between updates was quite long. There have been some discussions as to whether an event driven model that incorporates some form of buffer of queued operations would be workable in comparison to the periodic updates. The update time has been shortened and the update signals have been split into different types of messages.

- Resource data - CP version, CPU available, Memory utilization and usage of the platform - synchronized every hour
- Template trackers - Templates that are available on the market - synchronized every hour.
- Datastore - Disk utilization statistics and information - synchronized twice an hour.
- VM - number of VMs and utilization - synchronized once an hour.

All data is synchronized periodically to help update the search functionality and ensure that the market doesn't differ too much from the real state of the resources. When a request is made for a resource, a transaction is created at the same point, and the actual results of the request for operations are reported via the Market. The only case where commands are not sent through the Market instance is for creating a remote viewer through the use of HTML and VNC that links the buyer and seller directly to avoid extra latency and bandwidth constraints.

#### 3.4.2.5 Model Made Reactive to Events Sent out by Sellers

In addition to the more frequent polling updates by the Market instance, one of the changes that has been partly implemented to improve consistency is to make the Market react to events sent by the sellers and update the model accordingly. When responding to queries, sellers can add information about the current state in the 'event' messages that need to go through the Market and back to the buyer. The additional overhead from the extra information costs more in bandwidth, latency and message handling, but is a way of making the Market temporally more consistent without decreasing the polling time.

#### 3.4.2.6 Improve the Manageability of the Platform

There are also a set of controllability and manageability features that have been added to the Federation that are described in D2.5. There however exists a set of new features and tools that are not used for control but are instead used for clean-up and automation processes. These include:

- Market timeout exception reporting for handling queries or events that take too long to receive, after a number of retries.
- Tools to aid removal of a buyer's VM and resources until the full dependency tree is generated.
- Highlight support and problem views to filter standard messages and just show errors and serious issues.

### 3.4.3 Features Introduced to Cloud.net

Cloud.Net is a platform for buying, selling and trading compute resources across geographically diverse regions. Management is through a single-pane-of-glass dashboard available at `http://www.cloud.net`. It has been previously described in D1.3 in Section 4.1. The architecture has not diverged much from the original platform and is still used by customers for providing services. Cloud.net can also be combined with the OnApp CDN platform to allow Points of Presence (PoPs) to be managed by the same single dashboard. The combination of CDN and compute resources allows for interesting possibilities where companies can generate content from compute units close to customers and then use CDN to distribute the content to PoPs. In the third year of TRILOGY2, several new features and refinements have been made with respect to what was reported in D1.3.

#### 3.4.3.1 Metrics for Clouds

Cloud metrics were seen as a way of differentiating and ranking Clouds that offered their services on Cloud.net. A combination of price, location, uptime and performance metrics could then be used by potential customers to decide which resources to utilize.

The price is currently a value that can be set by potential customers who would like to offer their resources on the Federation. The location is detected from a reverse Geo-IP lookup that is based on the IP to which the Cloud is licensed and can be modified by an administrator if required. The name of the service can also be chosen independently, but is usually the same name as the company. These attributes are relatively fixed (OnApp does allow pricing changes but this is done at a controlled rate to avoid speculative pricing).

The uptime is currently calculated by pinging the Control Panel at regular intervals and checking that the site is able to provide a response. Depending on the number of pings that get through at fixed time intervals, a percentage is calculated and this becomes the uptime score. This is not an ideal way to measure the uptime and changes are being made in order to provision a VM onto a randomly assigned Hypervisor and then check the availability of that VM. This process is more reflective of what an actual end-user would be looking for.

The cloud index is an initial way of comparing the performance of the different cloud platforms. To analyze the performance of a Cloud provider, a helper VM is created from the Market onto the Cloud. A set of benchmarks from the ServerBear[2] benchmark suite are run on the helper VM. To ensure fairness of testing, the same VM profile is created on each cloud site to be evaluated – a Debian 7.0 x64 instance with VM profile of 2 cores, 2GB RAM, 40GB storage, full network. The following benchmarks are run;

- CPU performance is calculated with UnixBench[3].
- Disk performance calculated with IOPs.
- Average bandwidth/throughput performance via CDN tests on various global edge servers.

The provider score index is calculated for CPU/Disk/Bandwidth parameters individually. The provider score for an individual parameter is calculated as

> Provider's parameter score: (Provider Raw Score / Max Raw Score) * 100%

The overall provider index is calculated as follows:

> Provider's overall Cloud index: Sum (CPU/Disk/Bandwidth parameter scores) / 3

The results are visually displayed to users as a score as seen in Figure 3.6.

#### 3.4.3.2 Revamped U/I and Website

Visually displaying Liquid Resources, especially for services that link compute, storage and bandwidth as is done with the Federation, is not a trivial effort. For the purpose of the Federation and allowing users to choose compute locations it has been determined that the most relevant visualization is location centric. Cloud sites are shown on a map, allowing users to drill down and find more information about a particular site's resources. Filters can also be used to hide results that are not appropriate, depending on the end-users choice of VM attributes. The Cloud metrics as shown in Figure 3.6 and described in the previous section now form a centre stage for the Cloud.net dashboard.

[2] `http://www.serverbear.com`
[3] `https://github.com/kdlucas/byte-unixbench`

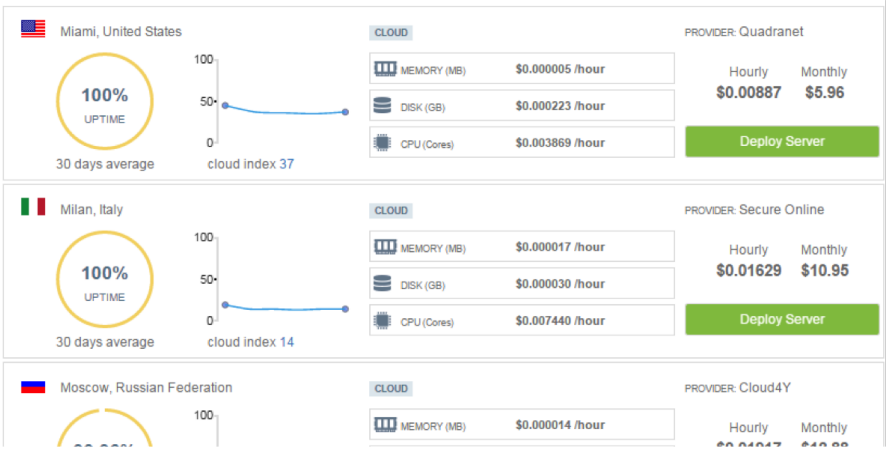



Figure 3.6: Cloud.net statistics showing the uptime and resource utilization on a cloud by cloud basis.

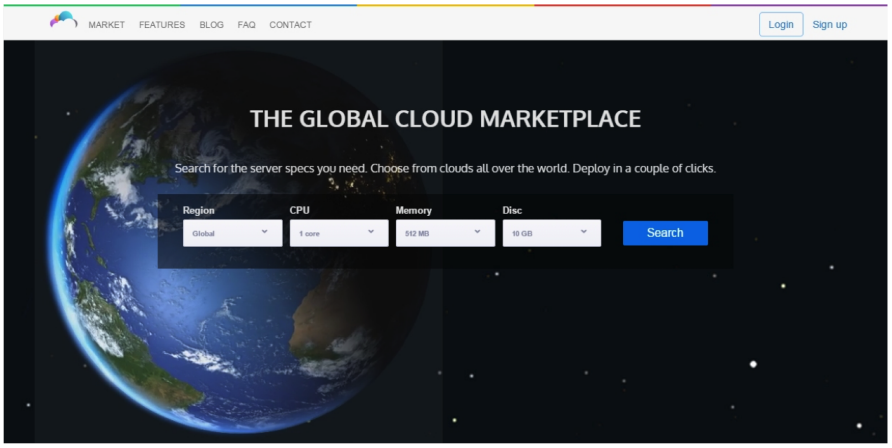



Figure 3.7: The front-page Cloud.net dashboard.

Another important concept developed during the course of TRILOGY2 is displaying and managing the resources through a single-pane-of-glass. Making modifications at each Cloud site individually and then tying them together with a dashboard is not appropriate for the majority of users, hence the centralized management system is displayed via the dashboard page. A screenshot of the updated dashboard page for Cloud.net is captured in Figure 3.7. From this dashboard, users can create and manage new Cloud.net accounts that will then tie into the Federation and show the resources that a buyer can acquire and conversely the resources that are being sold by a seller.

To assist the end-user, several steps were undertaken to make the process of enabling liquidity much simpler. The U/I was refined and the number of steps cut down for provisioning resources based on repeated operations that had to be carried out. For instance, if a payment credit card was on file, then it could be used when buying additional resources without having to re-enter the details each time. Similarly, for support operations tied to an account, information about the Cloud site, the user and details about the errors are captured in requests built into the system. As Cloud.net manages the support actions and interacts with data center owners on behalf of users, this helps to provide users with a more streamlined process. When errors are detected by the platform, an error report is automatically generated in the background, with an associated support item that can then be investigated to make the system more robust.

#### 3.4.3.3 Billing

All the billing of Cloud.net is managed internally. Customers have limited visibility and access to the Cloud sites on which services are provisioned via Cloud.net as part of the centralized management system. Support queries, billing *etc.*. are all handled by Cloud.net, which presents a single point of access to the Federated resources.

Various fixes have been made to make the billing platform consistent with the Federated Market and allow

changes in pricing. The pricing model has also been simplified to use a single credit system rather than multiple credit notes that were becoming unmanageable.

#### 3.4.3.4 Docker Integration

One of the promising features that is being researched and developed is Docker[4] integration into Cloud.net. Docker is a popular container service for deploying web-based applications quickly to Cloud sites. Docker is external to the project, but it represents a movement towards the Unikernel and minimalistic VM's that have been presented in previous Deliverables of TRILOGY2. By being able to quickly provision applications, we are further enabling liquidity of resources by being able to provision resources on-demand, as required.

#### 3.4.3.5 Manageability of the Platform

As more users have started using Cloud.net platform, there are a number of administrative functions that are not needed for control or management but for cleaning up and automating certain repetitive processes. One particular repetitive operation involved testing a new Cloud site to ensure that it was fully configured and working with Cloud.net. The operation previously involved several manual steps performed by the administrators of Cloud.net for checking that all the processes were working.

To assist the business administrators of Cloud.net, additional views that display the current status of the platform were created. These views link in the current resource trends and allow some forecasting to be made based on previous historical values.

To make the platform more available and resilient to outages, Cloud.net was scaled to two geographical sites that manage resources in U.S.A and Americas and a separate location in Europe for managing the EMEA region. The choice of location when creating an account is determined by the account manager, depending on where the majority of the business will be carried out and also where the Service provider is located.

### 3.4.4 Link with DRaaS Use-Case

The Disaster Recovery-as-a-Service platform is a commercially motivated use-case that is described in more detail in WP3. It allows Cloud sites to provide a disaster recovery system whereby disk content is continuously replicated from a provider to another cloud service provider (CSP). As soon as the primary cloud site fails, this outage can be detected by a management dashboard that allows an administrator to fail-over the VMs, thus allowing the VMs to run on the other CSP system. Once the primary site has come back up after whatever fixes are needed, the VMs can then be failed-back to the primary site. The VM on the secondary site is halted and replication continues as before.

The DRaaS platform utilizes a number of the primitives and features available in the Federation platform. The Liquidity of resources, in this case - storage, is exposed and then mapped across a permanently established network connection. There is a contractual agreement from both the user (buyer) of DRaaS and the supplier (seller) in the form of SLAs that denote how long the VMs can be failed over for and the cost of providing a fail-over standby site. The third party in this system is OnApp, which continually monitors the availability of resources on the primary cloud site and then triggers an offline alert at the time resources cannot be detected. DRaaS is the first step towards full Cloud resource liquidity that allows VMs to be migrated between Cloud sites. Further work is required to get DRaaS working within the scope of the Federation platform, see Section 3.4.5.

### 3.4.5 Findings and Further activities

One of the major findings of Cloud.net and the Federation is that Cloud owners are very willing to sell excess resources. Nevertheless, it is very difficult to incentivize buyers and end-users to make use of such a platform. One of the main reasons for this is that the Cloud services, being disparate and controlled by different providers, offer different levels of service and SLAs. Single providers that have control over the complete infrastructure, such as AWS from Amazon, Google Cloud from Google and Azure from Microsoft, offer a single consolidated Cloud infrastructure that is more appealing to customers.

Another of the main findings is that the Cloud metrics in the form of the Cloud Index are not useful for customers. We originally thought that presenting a large amount of detailed information would allow the end-users to make a more informed decision. Based on interviews and feedback from real customers, it appears that what they actually want is an assurance of a certain minimum standard for each provider that can then be combined together to form a service offering of their own. The Cloud Index results have been

[4] http://www.docker.com

considered good metrics for the Service Providers to differentiate themselves and compare their performance and as such, the index is a useful tool but for a different set of stakeholders than originally intended.
Based on these two findings, OnApp is going to be investigating whether a tiered service model will be more appealing for customers. With this tiering, service providers will need a certain minimum performance to be allowed entry to certain tiers. The metrics would still be available for those that want them, but the products would be packaged and displayed without the metrics and just on price, location and 'level'. Customers would then be able to select from a set of low/medium/high/extreme tiers (to be decided) that allow them to quickly select a set of resources and provide an offering, with the assurance that if a provider fails to deliver certain criteria to be in a level, that they would be dropped. This policy pushes the monitoring and enforcement activity away from the service provider to the platform provider, OnApp who is in a better suited for this observer and enforcer role.
A finding from Cloud.net feedback is that simplification is a large requirement for customer adoption. Tools and APIs can make use of the metrics on the platform but ultimately the customers want to be able to select an adequate offering quickly from a large set of providers. This is a counter-intuitive finding and should be taken into consideration by other EC projects that want to offer Liquid resources through brokerage systems. The ability to access details based on finer grain metrics should still be present in such a system. The level of detail that resources in a resource pool should expose is extremely dependent on the target audience.
One of the activities that will be continued beyond the scope of TRILOGY2, given that the Federation is now a commercial offering that is available, will be to offer further services. Abstracting the workloads from the VM containers is one area that will be continued. Docker has shown that having manageable resource units providing a single application is commercially viable and as such it makes sense to continue work on getting Unikernel support and light-weight VMs into the Liquid Net. Additionally, the incentives and enforcement work that is described in D2.5 will be continued. Getting the commercial balance tied with the technical implementation is very important for enabling true Liquidity. Finally, to support the deployment of the Cloud VM mobility use-case, the DRaaS use-case will be expanded to allow for the VM state to be transferred across the wide-area in addition to the disk content that is already continuously replicated.

### 3.4.6 Conclusion

The Federation and the Cloud.net system that is overlayed on the Federation allows for the liquidity of resources. Network resources that were available as part of the OnApp CDN platform have been supplemented with the addition of compute resources that are available at different geographical regions operated by various providers. We have incorporated the buying, trading and selling of compute resources into a commercially viable platform. The underlying technology has been researched and moulded by findings in TRILOGY2. Although there are still numerous areas for improvements, Federation and Cloud.net as a platform demonstrate the effectiveness of the Liquid Net, especially due to their wide adoption by customers. More details regarding the findings are covered in the dissemination and exploitation activities recorded in D4.3.

## 3.5 Load Balancing at the Data Center Scale

### 3.5.1 Introduction

Popular present-day web services can expect to have thousands of simultaneous users. Running one such service would typically require multiple servers in order to cope with the volume of incoming requests.
Cloud services are an ideal candidate for hosting web services. Tenants can dynamically increase the number of leased machines during peak hours and decrease them when the servers are under-utilized. To make running a web service in such environments feasible, a Layer 4 load balancer is needed.
Building upon current designs (*e.g.* [65]), we have implemented a Layer 4 load balancer that minimizes disruption in the data plane in case of hardware failures or when scaling in/out.

### 3.5.2 Architecture

In order for the load balancer to work across multiple broadcast domains and to take advantage of a technique called direct source return (see section 3.5.2.1), we use the three-tiered architecture proposed by Patel et al. [65]:

- The lower layer is made up of the back-end servers running the service (*e.g.* the Apache HTTP server). Each server has a host agent (HA), which is a traffic processing module interposed between the NIC and the server's network stack.

- The middle layer is made up of a number of multiplexers (or Muxes, for short). When hit with traffic from a client, a multiplexer picks which back-end server to direct it to. The handling of return traffic will be discussed in section 3.5.2.1.
- The top layer is made up of routers with ECMP capabilities. The routers spread traffic across the Muxes.

#### 3.5.2.1 Direct Source Return

Typically, clients download more data than they upload. As such, it would be advantageous if return traffic (traffic from the servers to the clients) does not hit the Muxes. IP-in-IP encapsulation is therefore used to avoid the Muxes, in addition to some form of network address translation.
Forward traffic from the client to the servers is processed in the following way:

- The client (with a Client IP address, or CIP) sends a packet addressed to the service's Virtual IP (VIP). It eventually makes its way to one of the routers in the top tier of our architecture.
- The router has several routes for the VIP (one route for each Mux). It uses ECMP to pick one of the routes and forwards the packet to the Mux.
- By some mechanism, the Mux determines which backend server the packet should go to. It encapsulates the packet using IP-in-IP. The outer IP header has the VIP as its source and the server's Direct IP (DIP) as its destination.
- Before reaching the server's network stack, the packet is decapsulated and translated by the Host Agent. The packet's destination IP is set to the servers Direct IP (DIP). In other words, the translated packet looks as if it were addressed directly to the server.

Return traffic receives a simpler treatment:

- The server acts as if it were talking to the client directly. Return traffic will have the DIP as its source and the CIP as its destination.
- The HA intercepts outbound packets before they reach the wire and sets their source address to the service's VIP.
- Packets leaving the server reach the client via regular IP routing.

### 3.5.3 Design Hurdles

As seen in section 3.5.2, the architecture features three tiers. At each of the two topmost tiers, a lower-tier machine (Mux or server, respectively) must be selected for each packet. The selection scheme must be such that all packets belonging to the same TCP connection must consistently reach the same backend server.

#### 3.5.3.1 Strawman Approach

Barring any failures (and/or deliberate changes in the pool of Muxes or servers), a simple hashing scheme works well. A hash of the packets' 5-tuple modulo the number of machines on the next tier ensures that all packets belonging to a flow follow the same path. However, churn will occur when the number of machines changes, due to crashes, scaling in/out *etc.*
One of the worst-case scenarios is a single backend server crash. Assuming the initial number of servers is $n$ and that one of them crashes, only around $1/n$ of 5-tuples will still reach the same server as before (this is because all consecutive numbers are coprime). This means that the vast majority of TCP connections will be severed.

#### 3.5.3.2 Proposed fixes

Ananta [65] proposes using the strawman hash-based approach described above with a few additions.
Churn caused by changes in the pool of servers is dealt with at the Muxes. The first time a 5-tuple is seen by a Mux, the backend server indicated by the hashing algorithm is recorded; all subsequent packets will be dealt with according to the recorded choice. In essence, the hashing scheme is used per-flow, rather than per-packet.

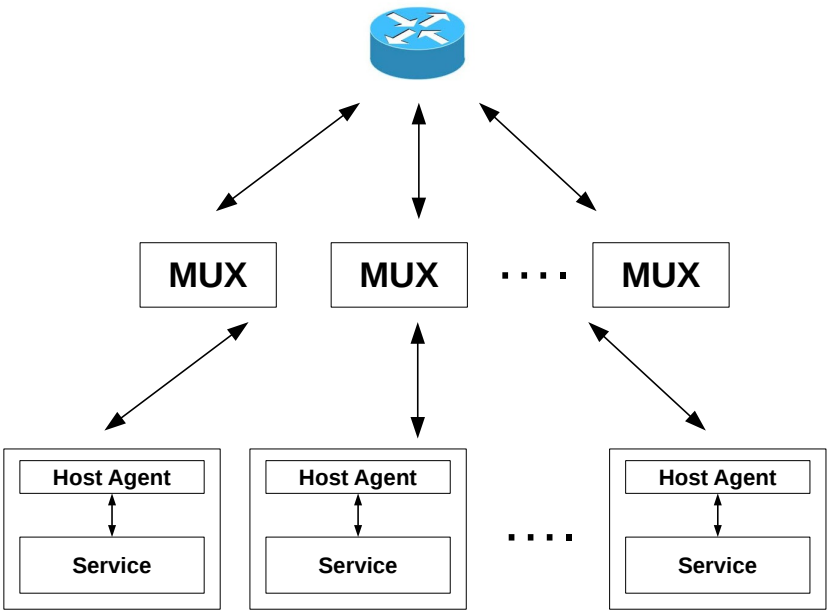



Figure 3.8: Load balancer architecture.

This design decision moves the problems caused by churn up one tier. Rather than having to make sure that all packets reach the same server, now we have to make sure that they reach the same Mux. While the authors of Ananta do not solve this issue in the paper, they suggest using a DHT to share state across the Muxes. We expect using a DHT will have a significant negative impact on packet rates.

### 3.5.4 Design

Our design follows the three-tier architecture mentioned in section 3.5.2. Rather than trying to mitigate or work around churn, our approach is to avoid it altogether.
In our design, the Muxes are stateless. For the purpose of correctness, it does not matter which Mux is hit by any given packet. Our design choice lets us focus solely on what happens between the two bottom tiers of the architecture – the Mux pool and the pool of servers.

### 3.5.5 Consistent Hashing

In our design, the Muxes use consistent hashing to determine where packets should go. We have decided to use a strategy proposed by DeCandia et al. [25]. On a ring we place a number of tokens which is (at least) an order of magnitude greater than the maximum number of backend servers we expect to have. All tokens are equidistant. Next, we statically assign tokens to servers.
All we need to do for the system to run correctly is to keep Muxes up-to-date with token-to-DIP mappings. Since all tokens are equidistant, all DIP lookups are in $O(1)$ (rather than $O(n)$, as is usual when tokens take up arbitrary positions on the ring [80]).

### 3.5.6 Token Reassignment

Whenever the number of servers changes, tokens must be reassigned in order to keep the load even. As far as Muxes are concerned, a token reassignment simply means changing a token's associated DIP. Obviously, there could still be some open connections matching the reassigned token. It is up to the servers' host agents to keep pre-existing connections from dying off.
When reassigning a token from server A to server B, B is notified of the reassignment. B's HA has access to B's list of open connections. As such, it can tell if an incoming packet belongs to a connection that is newer than the reassignment. All packets belonging to older connections are forwarded to A, while the rest are sent to B's stack. Eventually, older connections will die off.
One of the main benefits of this approach is that it is now possible to perform graceful shutdowns. When scaling in during periods of lower traffic, machines that are due to be shut down can be kept powered on after having all of their tokens transferred elsewhere, while waiting for existing connections to finish.

### 3.5.7 Implementation

We have implemented the Mux as a Click [54] element. We use Apache Zookeeper to keep the token-to-DIP mappings up-to-date. The host agent is a simple Linux kernel module that uses the netfilter framework to intercept and alter packets as described in section 3.5.2.1.

## 3.6 NDP

In our earlier work on Multipath TCP we examined how the multipath topologies of modern data centers lent themselves to bandwidth resource pooling. For example, many modern data centers are built around Clos topologies such as FatTree, providing many parallel paths between each pair of endpoints.

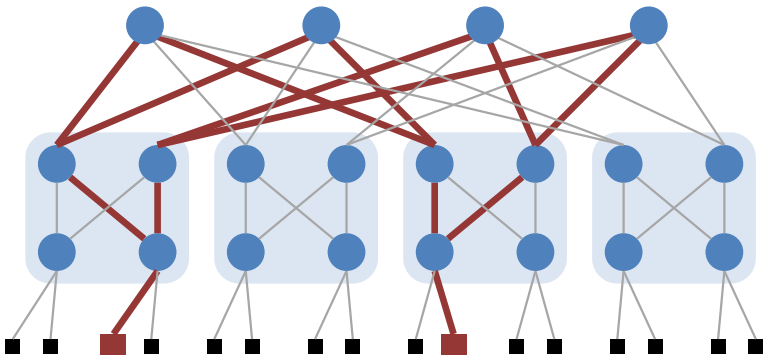


Figure 3.9: FatTree data center topology.

Traditionally ECMP routing, hashing on source and destination ports and addresses to decide a path, is normally used in such topologies to spread load. We showed that throughput in such data centers depended strongly on whether more than one flow randomly hashed to the same link at any instant, giving significantly reduced throughput and resulting in unpredictable performance. Using MPTCP instead of TCP in such topologies gives substantial throughput improvements, as MPTCP spreads traffic over several paths, allowing it to shift traffic off of links where it collides with other flows. We showed that MPTCP could triple throughput on Amazon's EC2 data centers when hosts had multiple paths available between them.

However, it has since become clear that although some flows in modern data centers demand high throughput for significant lengths of time (especially transfers between storage servers), more often data centre applications are latency bounded rather than throughput limited, or suffer from incast problems when the results of tasks farmed out to many workers implode back at the originator. MPTCP is not optimized to minimize latency, nor is it especially helpful for incast scenarios. These problems required us to take another look at whether we can achieve both high-throughput transport, pooling traffic across many paths, and low latency, even in the presence of incast traffic patterns. The work described below is a preliminary study into how this can be achieved, based on the lessons learned in TRILOGY2 from the experience of talking to operators about the deployment of MPTCP in data centers.

Our system, NDP[5] has only been evaluated in simulation at this point in rather constrained scenarios, so the work is very preliminary. However, in such cases it achieves both near-optimal small flow completion times and near-optimal large flow throughput in a wide range of scenarios, including Incast. It does this by combining techniques including packet spraying [26], extremely small switch buffers, a new randomized variant of CP[19] which trims packets back to just their headers when congested, a novel use of priority queuing and a highly tuned receiver-driven ACK clock that triggers retransmissions so rapidly that lost packets don't hurt performance. These pieces build upon previous work, but all need to be modified to work together effectively in pursuit of extremely low delay.

### 3.6.1 Design

To achieve both low latency and high data center throughput, we need three things: we need to keep queues short while not losing time with retransmissions, we need a way to use all the capacity the underlying topology

[5] New Data center Protocol - we could do with a more imaginative name.

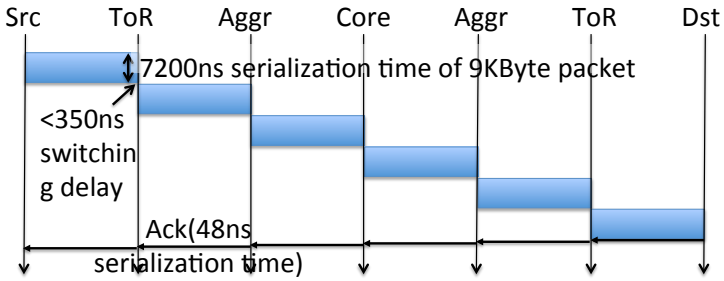



Figure 3.10: Latency in an unloaded 10Gb/s network is dominated by serialization.

makes available in the "core", and we need a way to deal with incast at access links. We address each of these in turn.

#### 3.6.1.1 Core Capacity

Modern data center topologies such as FatTree [3] and VL2 [36] provide full bisectional bandwidth: enough capacity for every host to transmit to any other host simultaneously, so long as traffic is not concentrated at a receiver. In reality, to achieve full cross-sectional bandwidth with practical transport protocols is not so simple. Flow-based ECMP does not spread traffic sufficiently widely, so hot spots emerge where more flows than average concentrate on a link, significantly reducing overall throughput. Multipath TCP mitigates this by sending over multiple paths simultaneously, actively moving traffic off the more congested paths using a modified TCP congestion control algorithm. However, MPTCP's congestion control needs large buffers at switches, increasing per-packet latency which, in turn, impacts the completion times of short competing flows.

To fully utilize available capacity so that hot spots are not created, we would like to perform per-packet ECMP, also known as *packet spraying* [26]. In a Clos topology such as FatTree, packet spraying ensures traffic cannot concentrate on any subset of the paths, except as traffic fans in to the receiver. This allows high throughput, but the challenge is also to cope with asymmetric paths that may appear due to failures or congestion hotspots, as well as to minimize transfer times for the small flows that dominate in data centers. To do so, we must take an aggressive approach to transport.

In a data center the end systems can know the network topology and link speeds in advance because the network is both stable and under one administrative organization. In such a topology, if a sender wishes to send to a receiver, all it needs to do is to burst at the minimum of the sender and the receiver's link speeds. Packet spraying and the nature of the Clos topology guarantee that there will be enough capacity (at least on average) to cope, except if multiple flows leave one sender or converge at one receiver simultaneously.

#### 3.6.1.2 Incast

The remaining problem is that of *incast*, where flows converge on one receiver. If each sender bursts at line-rate, many of the packets will be lost on the last link. Others have worked on probing-based approaches to avoid incast; we take a much more radical approach: the sender should always send the first RTT of data at line rate, to ensure short flows fully utilize the capacity. Instead of avoiding incast, we design mechanisms to detect it in minimal time and retransmit as fast as possible. The key observation is that retransmissions do not cost bottleneck bandwidth; if we can minimize their latency cost, then better to optimize retransmission than try too hard to avoid it.

In principle, the latency across a data center network can be extremely small. Figure 3.10 shows that in an unloaded network consisting of 10Gb/s Ethernet links with standard store-and-forward switching, the latency of sending a 9 KByte packet and receiving an ACK is dominated by the 7.2 $\mu s$ serialization delay at each hop. Control messages can traverse the network in less than one data packet serialization time, so long as they don't encounter queues. With careful protocol design we can take advantage of this low delay so that retransmissions don't significantly hurt flow completion time.

With incast, we want the receiver to know the instantaneous demand, and, after the first RTT of uncontrolled traffic, it can pace incoming traffic including any retransmissions for the remainder of the connections' life-

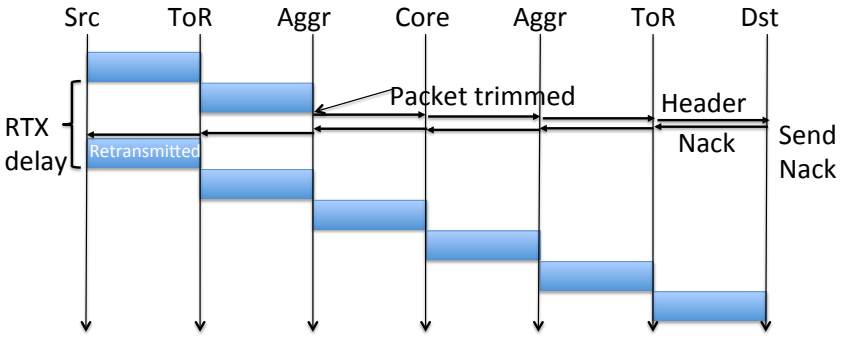



Figure 3.11: Trimmed packets can be quickly resent.

time. How can the receiver know instantaneous demand if most packets being sent to it are lost? The innovation required is packet trimming (aka Cut Packet [19]): when a switch queue fills, the switch does not drop the packet, but rather trims off the payload leaving only the header. For typical 9KByte data center Ethernet packets, trimming to just a 60Byte header provides 150:1 compression, allowing 150 headers to be forwarded in the time that a single data packet would take. If the receiver gets the headers of *all* the packets that were sent, it knows precisely the instantaneous demands of flows sending to it and precisely which packets are missing.

Packet trimming is the key insight allowing very low delay data center networking, but to get the most from it, we need to take a different approach from [19] where headers are queued behind regular data packets. As Figure 3.11 illustrates, a key advantage of trimming is that a trimmed packet can be retransmitted extremely quickly - the retransmission can be on its way before the original packet would have arrived.

Such extremely low delay retransmissions are not possible unless we manage latency very carefully. First, instead of just trimming a packet, we place all trimmed packets in a priority queue at each switch output port, so they overtake waiting data packets. We also priority-queue ACK and NACK packets, so the sender can know the fate of a packet within a bounded time: in the worst case an ACK or NACK will have to wait for only one data packet that it cannot preempt at each hop, as delays due to other packets in the priority queue are negligible. Senders know their packets' fate in under 45$\mu$s.

The majority of any variable latency in the control loop will be on the forward path due to queues. What is the minimum buffer size that switches can use that still gives good utilization? In our experiments we operate switches with eight packet buffers and can still achieve greater than 95% of full bisection bandwidth. Six packet buffers are also feasible, but there is a throughput penalty from doing so.

Clearly with such tiny switch buffers, the queues will sometimes fill due to random variation caused by packet spraying. However, because all the delays, including queuing delay, are so small, we can retransmit any missing packets fast enough to have minimal impact on flow completion times.

Such a data center, with very short queues, packet trimming and priority-queued control messages, is a very deterministic environment. We observe that it is prone to strong phase effects, especially with incast, where one flow consistently loses and another consistently wins. To break up these phase effects we must add randomness. When a data packet arrives at a full queue, rather than always trimming the arriving packet, instead we trim the arriving packet with probability 50%, otherwise we trim the last packet that was already queued. This is both very simple and sufficient to completely eliminate such phase effects.

#### 3.6.1.3 Clocking Data

The final and most crucial element of NDP is the data clocking algorithm run between a receiver and the senders sending to it. A NDP sender will only burst at line-rate for one RTT - typically ten packets in a Fat Tree topology. After this it waits for the receiver to tell it what to do next, because only the receiver knows the instantaneous demand. To permit very low delay, we split the multiple roles of a traditional ACK-clock [41] into multiple stages, as follows:

- For every data packet that arrives, the receiver immediately sends an ACK. ACKs allow a sender to free

buffers, but never trigger data transmissions.

- For every trimmed data header that arrives, it immediately sends a NACK. NACKs do not cause the sender to retransmit, but cause the missing packet to be added to a high-priority retransmit queue at the sender.
- For every ACK or NACK the receiver sends, it enqueues a *pull packet* on a FIFO queue. The occupancy of the pull queue therefore accurately tracks the instantaneous demand of all the connections flowing to that receiver. Pull packets have a monotonically increasing per-connection *pull number* and a monotonically increasing per-receiving-host *pacer number*.
- The receiver dequeues pull packets at the pace it can handle data packets arriving - for example, every 7.2$\mu$s in the case of 9KB data packets on 10Gb/s links.
- When a pull packet arrives at the sender, it triggers the transmission of as many data packets as the pull number increases by. Normally this will be one, but as packets are often reordered, it can step by more than one, or briefly decrease. The cumulative nature of pulls means that if one is delayed, a later one that beats it through the network can perform its role, keeping the data clock flowing.
- The sender preferentially sends retransmitted packets in response to a pull; it sends new data when no retransmitted packets are available.
- The sender echoes the *pacer number* from the pull packet in the data packets it triggers. The receiver compares the echoed pacer number in incoming data packets to that of the most recent pull it sent. This difference is a measure of flight size, but due to reordering it is noisy.

Under all but the most extreme circumstances, the combination of arriving data packets/headers and the pull messages they elicit keep the correct amount of data circulating. NACKs add packets to the retransmit queue at the sender. ACKs allow the sender to free buffers very quickly. In the first RTT of incast, many packets are trimmed, but they can be retransmitted fast enough that the first of them join the queue at the link to the receiver before it goes idle. After the first RTT, the receiver pulls packets from all senders at precisely the rate that it can sustain. These packets are rarely trimmed, except due to transient overload due to imperfect load balancing.

At each switch the priority queue size is bounded to the same size as that for data packets. With an eight 9KB data packet queue, this means 72KB of data and 72KB of headers (1200 packet headers) can be queued. It is possible that data packets are really lost, not just trimmed, either because the priority queue fills with headers from a really large incast or due to corruption. NDP handles this gracefully. The sender knows that it can expect an ACK or a NACK within approximately 45$\mu$s of the intended arrival time, and strictly within approximately 400$\mu$s, so it can time out missing data packets with a very short RTO and add them to the retransmit queue. At the same time the receiver can use the pacer number difference to detect that pulled data has gone missing, and use it to generate additional pull messages to compensate and keep the data clock flowing.

#### 3.6.1.4 NDP in Action

Figure 3.12 is a time/sequence plot of NDP in action.[6] Three flows shown in black, blue and grey start almost simultaneously each sending a 180KB file to one common receiver. Each enqueues ten packets (one RTT's worth) immediately at startup. Every dequeue event at the sender or a switch is shown as a small blob. From 4–70 $\mu s$ the three senders' queues drain at their line rate, and packets traverse the network. The first black packet arrives at 50 $\mu s$ (solid triangle). At 75 $\mu s$ the queue feeding the receiver's link overflows, and two thirds of the packets are trimmed (shown as stars) over the next 50 $\mu s$. NACK packets (not shown) enqueue the first retransmission (empty box) at 80 $\mu s$ and the last is enqueued by 135 $\mu s$. These are pulled by the receiver starting at 105 $\mu s$; pulls continue for the rest of the transmissions clocking the remainder of the data and no further packets are trimmed. The first retransmission (black packet 4) arrives at 180 $\mu s$ and the last (black packet 12) arrives at arrives at 320 $\mu s$. The first new packet to be pulled is black packet 11, at 60 $\mu s$. The last data packets from each of the three flows arrive back-to-back, with the final packet arriving at 482 $\mu s$. It is not possible for any protocol to do significantly better than this - the earliest theoretical arrival time for the last packet in this scenario is 477 $\mu s$.

Figure 3.13 shows the pull-clock in action for the black flow from Figure 3.12. Due to pacing, pull packets can be queued for some time until the receiver is ready for the data they will elicit, but when a pull is released,

[6] The topology is a 128-node Fat Tree, with all links 10Gb/s, 9KB packets, 1 $\mu s$ switching and speed of light delay per hop

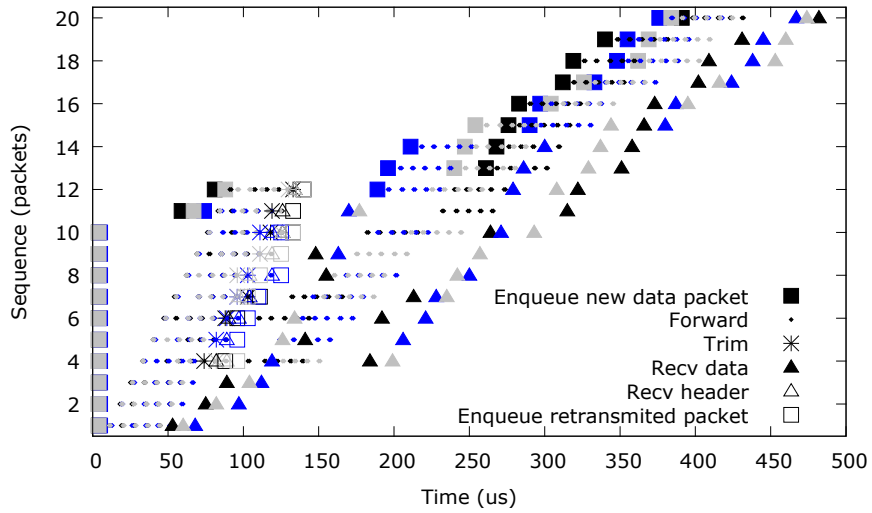



Figure 3.12: Time/sequence plot of three incast flows.

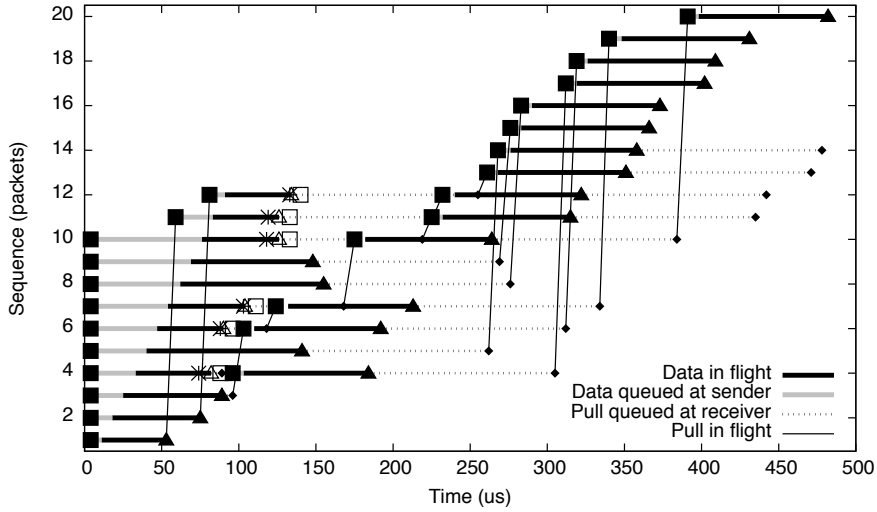



Figure 3.13: Pull-clock of the black flow in Fig.3.12

it triggers the sending of data rapidly. The goal is to have enough pulled data in flight between all flows to keep the link to the receiver occupied.

### 3.6.2 Evaluation

Data center traffic is a mix of numerous mice flows (short flows of around 50KB-1MB) and few elephant flows that carry most bytes [36]. Common traffic patterns include one-to-one transfers (such as file system reads or writes), scatter-gather traffic (e.g. one-to-many query distribution and many-to-one replies with answers), and many-to-many traffic as in the shuffle phase of map-reduce. For our evaluation, we use the *htsim* simulator running over a 10Gbps FatTree topology, with jumbo frames. We run a mix of experiments focusing on different aspects of performance: high utilization for elephant flows, resilience to network asymmetry, short flow completion times and incast traffic.

We compare NDP to Multipath TCP and Packet Spraying, the state of the art solutions to achieve high network utilization. Packet Spraying is a version of TCP modified to cope with packet reordering due to per-packet load balancing: we set the duplicate ACK threshold to be $3 + \#paths$. This change ensures that TCP does not fast retransmit needlessly when packets are reordered, and we chose it experimentally: choosing smaller values led to spurious retransmissions, while larger values affect performance of short flows, as we discuss later. When measuring flow completion times, we contrast NDP to the theoretical optimal.

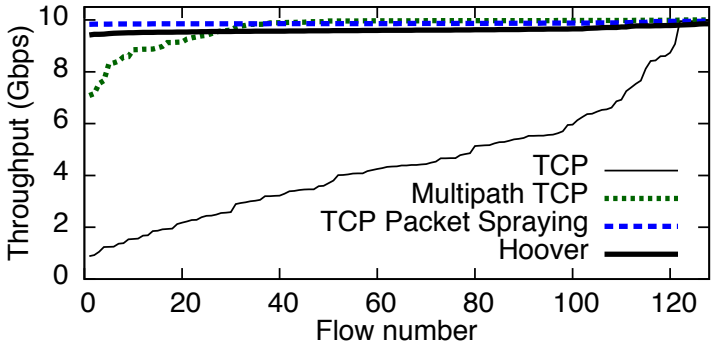



Figure 3.14: Permutation traffic throughput

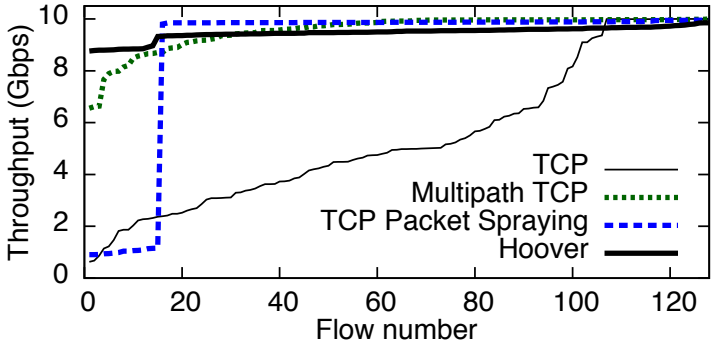



Figure 3.15: Asymmetric topology: a core link dropped to 1Gbps

### Packet-level load balancing algorithms.

Per-packet ECMP uses a hash function to decide the next hop for each packet quasi randomly; this can cause mild traffic concentration. If we use partial source-routing, as in VL2, round-robin load balancing across core switches can spread load more evenly, though hosts might synchronize with each other. To achieve better balancing, we use a simple routing scheme which does randomized round-robin balancing (RRB): each source places packets on the available $p$ paths in a per-host random order, then shuffles the order every $p$ packets. RRB gives 5–10% higher utilization than ECMP. We use RRB for both NDP and Packet Spraying in this evaluation.

### Utilization for elephant flows.

As observed in previous work [68, 4], the worst-case traffic pattern for the FatTree is a permutation: each host sends a single connection and receives a single connection. As the topology is full-bisection, and there is no traffic concentration at the edges, each connection should run at 10Gbps. In Figure 3.14 we plot the individual flow throughputs for different protocols running over a K=8 FatTree containing 128 servers; larger topologies give similar behavior. All TCP variants were run with 100 packet buffers for each switch port. NDP ran with 8 packet buffers, with payload trimming on overload.

As in previous work [4, 68], we find that regular TCP running over per-flow ECMP suffers because flows are randomly hashed to the same link and capacity elsewhere is wasted: the average flow throughput is just 4Gbps. TCP over Packet Spraying gives near-optimal throughput and Multipath TCP with 8 subflows gives perfect throughput for 70% of flows, yet some flows get only 7Gbps. NDP comes within 96% of optimal for all flows using buffers one twelfth the size; this is key to reducing small flow completion times.

### Switch buffer sizing.

The 8-packet buffer size for NDP was chosen experimentally. In our 128-server topology 4-packet buffers give a mean utilization of 75%; 6 packets increase this to 90%. Increasing the buffer to 10 packets only marginally improves performance, so we chose 8 for our experiments. For larger topologies a small increase is probably recommended: a 1024-server network achieves 88% with 8 packets and 92% with 10 packets per

port.

**Asymmetric topologies.**

Asymmetries may arise because of failures or due to persistent traffic concentration over certain links, such as caused by external TCP traffic. To examine how this affects the protocols we downgrade one of the links between a core-switch and an aggregation-switch to 1Gb/s and rerun the permutation experiment.

Figure 3.15 shows that Packet Spraying suffers badly: it is agnostic to the paths its packets take and backs off when some of them are dropped on the slow link, despite there existing capacity elsewhere. Multipath TCP keeps per subflow congestion state, so it only backs off on the affected subflows. NDP is near-optimal because it does not do sender-based congestion control; the only effect is that a larger fraction of its packets are trimmed, NACKed, and quickly resent.

**Short flow experiments.**

Small flow completion times are critical to many data center applications facing clients: responses are composed of small pieces of data gathered from many servers, and delays lead to lost revenue. To understand small flow completion times, we run two experiments.

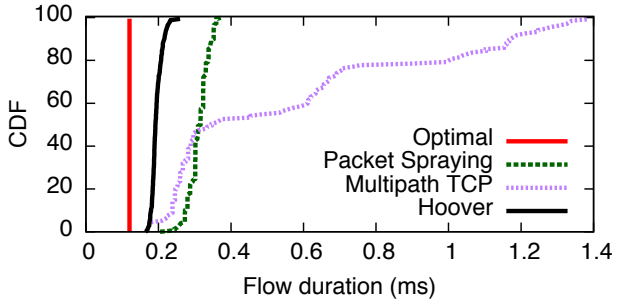



Figure 3.16: 90KB flow completion times in fully utilized 128 server FatTree

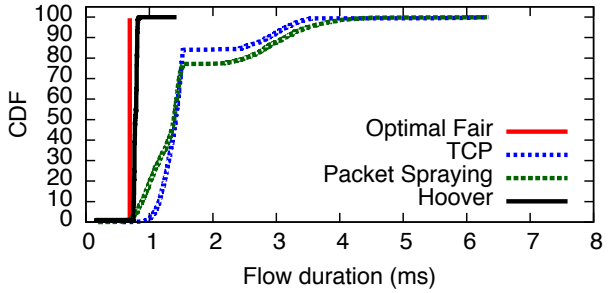



Figure 3.17: 90KB flow completion times competing with many-to-one traffic

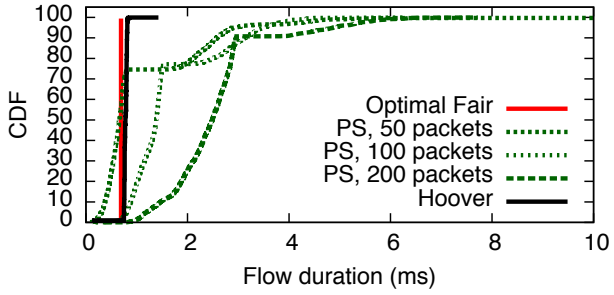



Figure 3.18: Buffer sizing affects flow completion times for Packet Spraying

We have a source repeatedly send a small flow (90KB) to a chosen destination and measure its completion time. If the network is idle, each of these flows finishes in 120 $\mu$s. Next, we load the network up with 127 other long-running flows. With perfect load balancing of long flows, there should still be spare capacity for the small flows to finish in 120 $\mu$s.

The CDF of small flow completion times is shown in Figure 3.16. When the long flows use Multipath TCP,

the completion time of short flows is seriously affected: the tail is 11 times worse than the theoretical best. This is because Multipath TCP does not perfectly spread load across the network, filling some of the 100 packet buffers on certain switch ports. Both Packet Spraying and NDP do a much better job at spreading the load from the elephant flows across the network: the max flow completion time is 4 times the optimal for Packet Spraying and twice the optimal for NDP.

Next, we want to see the effect of buffer contention on small flows. We set up nine long running flows to one destination and have a tenth source repeatedly send a 90KB file to the same destination. If the small flows share the capacity equally with the long flows, each should finish in approximately 0.72ms. Figure 3.17 shows the results. NDP is close to optimal, with a maximum flow completion time of 1.1ms and median very close to the optimal.

Packet Spraying's median transfer time is twice as long, with some flows taking as much as 7ms to finish. We initially thought that this was due to the 100 packet buffers used, but this was not the root case. Packet Spraying's higher duplicate ACK threshold, increased to avoid spurious retransmissions in the face of reordering, is the main reason for this tail: when the small flow loses a packet there aren't enough duplicate ACKs to trigger fast retransmit, leading to timeouts. Regular TCP, with a smaller duplicate ACK threshold, suffers less from excessive completion times in this scenario.

### Buffer sizing for TCP Packet Spraying.

To reduce maximum flow completion times for Packet Spraying in the experiment above, we can increase the switch buffer sizes. If we use 200 packet buffers, for instance, the maximum flow completion time is reduced to around 6ms. However, the median flow completion time grows to 2.5ms as shown in Figure 3.18. Using 50 packet buffers decreases the median flow completion time close to optimal, but increases the tail to more than 10ms. 100 packet buffers appear to be a sweetspot and are close to what is being run in practice; this is why we chose them for our experiments.

### Incast Flows.

Incast is a common scenario in data centers, typically caused by farming out work to many workers and them all completing simultaneously. In this scenario, traffic is concentrated on the last link to the receiver. Figure 3.19 shows how NDP performs under incast - 64 hosts simultaneously transmit a file to one receiver in a 128 node Fat Tree. One result is shown for each file size from 9000 bytes (1 packet) to 1 MByte. For each file size, we plot the min, median and max flow completion times from the flows competing in that incast. For comparison we show the theoretical best completion time - it is not possible for the last flow to complete earlier than this.

Some flows get lucky, have few packets trimmed, and finish ahead of the rest. In general though, the flow completion times are closely clustered around the median, indicating good fairness. Many applications that suffer incast care about the completion time of the last flow, as they cannot aggregate the responses until they have them all. The last flow completion time is within 1% of the theoretical optimum completion time, regardless of flow size.

How does incast affect nearby traffic? Figure 3.22 shows the throughput of one long NDP flow when a 64-flow incast arrives, each sending 900KB to a host on the same ToR switch as the long flow. The long flow suffers a dip in throughput lasting less than 1 $\mu s$ due to the initial burst; after this the incast flows and the long flow get full throughput.

In Figure 3.19, the ten-packet initial window of the 64 flows combine to overflow the data queue on the link to the receiver, and many of them are trimmed. However, the priority queue for headers can accommodate 1200 headers, so it does not overflow. What happens if the incast is so large that the priority queue overflows? In Figure 3.20, 200 senders in a 432 node FatTree send simultaneously to one receiver. With such a large incast, if the file size is greater than 54KB then even the priority queue for headers overflows and headers are dropped. It might be thought that these packets that cannot be NACKed would increase the flow completion time, but this is not the case. Although these packets will only be resent when a retransmit timeout fires, we can safely use a very short RTO timer without risk of spurious retransmission because NDP separates Acking from Pulling. These retransmissions will be added to the sender's retransmit queue before the receiver has finished pulling retransmissions of trimmed packets, so they will be pulled immediately, with no additional delay.

TCP with Packet Spraying's incast performance is shown in Figure 3.21. We reduced TCP's initial RTO from

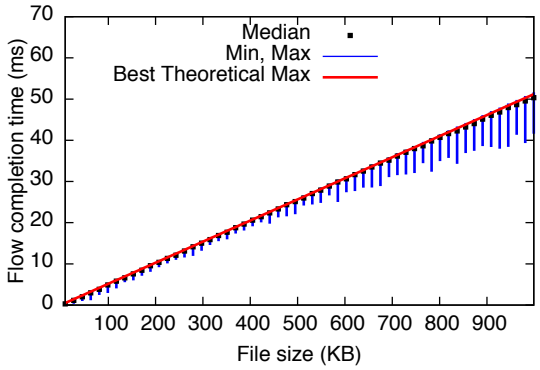



Figure 3.19: Incast flow completion time as a function of file size: NDP flows, 64:1 incast, 128 node FatTree.

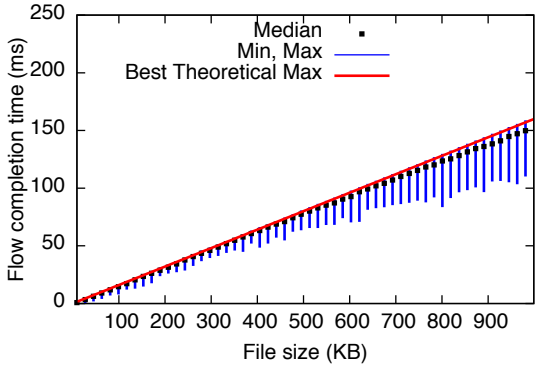



Figure 3.20: Incast flow completion time as a function of file size: NDP flows, 200:1 incast, 432 node FatTree.

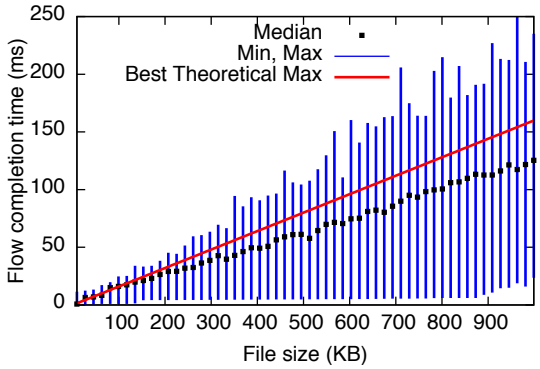



Figure 3.21: Incast flow completion time as a function of file size: TCP+PS, 200:1 incast, 432 node FatTree

3s to 10ms, and set RTO min to 1ms as suggested by [85][7]. Even so, the last TCP/PS flow typically completes about 30–50% slower than NDP for all flow sizes. TCP's performance can be improved slightly by reducing its initial window from 10 to 1 or by disabling Packet Spraying, but these hurt other results. MPTCP does not help with incast.

### Outgoing Flows.

NDP is designed around being optimal for incast, but it must also work well for all other traffic patterns. We have run experiments sending various file sizes to from one sender to many receivers simultaneously. With NDP, the last flow always completes at precisely the minimum theoretical completion time.

### Oversubscribed topologies.

NDP works well in a full bisection FatTree, but we wanted to find out how it fares in oversubscribed ones. We ran permutation experiments in a 512-server FatTree oversubscribed 4:1 at the ToR level. 75% of all packets were trimmed in this setup, yet the bandwidth overhead was negligible (¡1%) and throughput was very fair,

[7] TCP's default parameters perform substantially worse.

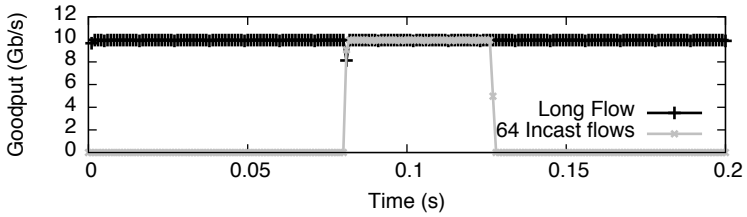



Figure 3.22: Collateral damage caused by incast is minimal.

with only a 10% difference between min and max flows.

### 3.6.3 Discussion

Packet spraying, payload trimming, receiver-driven protocols and small buffers are well known partial solutions to the problem of achieving low delay, high throughput data center transport protocols. Packet spraying gives high utilization but requires perfectly symmetric topologies and impacts small flow completion times. Small buffers are a recipe for small flow completion times, but they hurt the utilization of elephant flows and don't work in incast scenarios.

With NDP we recognize the limitations of each of these parts and combine them in a solution that is tailored to multipath topologies such as FatTrees. By giving up on sender congestion control, NDP is robust to asymmetric topologies, and payload trimming coupled with receiver pull pacing gives excellent incast behavior even with very small queues.

This is preliminary work. We still need to understand how NDP behaves in host-centric data center topologies such as BCube [37], or in topologies with unequal path lengths such as Jellyfish [76]. If it behaves well in such topologies then it seems likely that NDP will be well-suited to rack-scale computing, where switches are integrated onto each CPU's die. Implementing NDP will also raise challenges regarding the granularity of receiver pacing.

## 3.7 C3

The interactive nature of modern web applications necessitates low and predictable latencies because people naturally prefer fluid response times [35], whereas degraded user experience directly impacts revenue [16, 77]. However, it is challenging to deliver consistent low latency — in particular, to keep the tail of the latency distribution low [24, 87, 42]. Since interactive web applications are typically structured as multi-tiered, large-scale distributed systems, even serving a single end-user request (*e.g.* to return a web page) may involve contacting tens or hundreds of servers [25, 42]. Significant delays at any of these servers inflate the latency observed by end users. Furthermore, even temporary latency spikes from individual nodes may ultimately dominate end-to-end latencies [5]. Finally, the increasing adoption of commercial clouds to deliver applications further exacerbates the response time unpredictability since, in these environments, applications almost unavoidably experience performance interference due to contention for shared resources (like CPU, memory, and I/O) [90, 89, 44].

Several studies [24, 89, 42] indicate that latency distributions in Internet-scale systems exhibit long-tail behaviors. That is, the $99.9^{th}$ percentile latency can be more than an order of magnitude higher than the median latency. Recent efforts [31, 78, 64, 5, 24, 42, 91] have thus proposed approaches to reduce tail latencies and lower the impact of skewed performance. These approaches rely on standard techniques including giving preferential resource allocations or guarantees, reissuing requests, trading off completeness for latency, and creating performance models to predict stragglers in the system.

A recurring pattern to reducing tail latency is to exploit the redundancy built into each tier of the application architecture. We will show that the problem of *replica selection* — wherein a *client* node has to make a choice about selecting one out of multiple *replica servers* to serve a request — is a first-order concern in this context. Interestingly, we find that the impact of the replica selection algorithm has often been overlooked. We argue that layering approaches like request duplication and reissues atop a poorly performing replica selection algorithm should be cause for concern. For example, reissuing requests but selecting poorly-performing nodes to process them increases system utilization [87] in exchange for limited benefits.

As we show in Section 3.7.1, the replica selection strategy has a direct effect on the tail of the latency

distribution. This is particularly so in the context of data stores that rely on replication and partitioning for scalability, such as key-value stores. The performance of these systems is influenced by many sources of variability [24, 46] and running such systems in cloud environments, where utilization should be high and environmental uncertainty is a fact of life, further aggravates performance fluctuations [44].
Replica selection can compensate for these conditions by preferring faster replica servers whenever possible. However, this is made challenging by the fact that servers exhibit performance fluctuations over time. Hence, replica selection needs to quickly adapt to changing system dynamics. On the other hand, any reactive scheme in this context must avoid entering pathological behaviors that lead to load imbalance among nodes and oscillating instabilities. In addition, replica selection should not be computationally costly, nor require significant coordination overheads.
We present C3, an adaptive replica selection mechanism that is robust in the face of fluctuations in system performance. At the core of C3's design, two key concepts allow it to reduce tail latencies and hence improve performance predictability. First, using simple and inexpensive feedback from servers, clients make use of a replica ranking function to prefer faster servers and compensate for slower service times, all while ensuring that the system does not enter herd behaviors or load-oscillations. Second, in C3, clients implement a distributed rate control mechanism to ensure that, even at high fan-ins, clients do not overwhelm individual servers. The combination of these mechanisms enable C3 to reduce queuing delays at servers while the system remains reactive to variations in service times.
Our study applies to any low-latency data store wherein replica diversity is available, such as a key-value store. We hence base our study on the widely-used [23] Cassandra distributed database [9], which is designed to store and serve larger-than-memory datasets. Cassandra powers a variety of applications at large web sites such as Netflix and eBay [10]. Compared to other related systems (Table 3.2), Cassandra implements a more sophisticated load-based replica selection mechanism as well, and is thus a better reference point for our study. However, C3 is applicable to other systems and environments that need to exploit replica diversity in the face of performance variability, such as a typical multi-tiered application or other data stores like as MongoDB or Riak.
In summary, we make the following contributions:

(i) Through performance evaluations on Amazon EC2, we expose the fundamental challenges involved in managing tail latencies in the face of service-time variability (Section 3.7.1).
(ii) We develop an adaptive replica selection mechanism, C3, that reduces the latency tail in the presence of service-time fluctuations in the system. C3 does not make use of request reissues, and only relies on minimal and approximate information exchange between clients and servers (Section 3.7.5).
(iii) We implement C3 (Section 3.7.9) in the Cassandra distributed database and evaluate it through experiments conducted on Amazon EC2 (for accuracy) (Section 3.7.10) and simulations (for scale) (Section 3.7.11). We demonstrate that our solution improves Cassandra's latency profile along the mean, median, and the tail (by up to a factor of 3 at the $99.9^{th}$ percentile) whilst improving read throughput by up to 50%.

### 3.7.1 The Challenge of Replica Selection

In this section, we first discuss the problem of time-varying performance variability in the context of cloud environments. We then underline the need for load-based replica selection schemes and the challenges associated with designing them.

| | |
|---|---|
| Cassandra | *Dynamic Snitching*: considers history of read latencies and I/O load |
| OpenStack Swift | Read from a single node and retry in case of failures |
| MongoDB | Optionally select nearest node by network latency (does not include CPU or I/O load) |
| Riak | Recommendation is to use an external load balancer such as Nginx [69] |

Table 3.2: Replica selection mechanisms in popular NoSQL solutions. Only Cassandra employs a form of adaptive replica selection (Section 3.7.4).

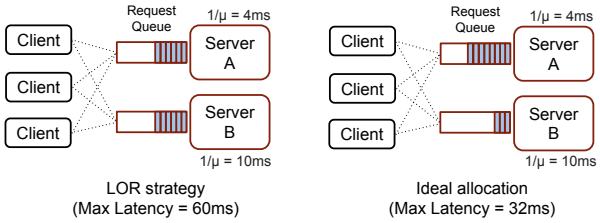



Figure 3.23: *Left*: how the least-outstanding requests (LOR) strategy allocates a burst of requests across two servers when executed individually by each client. *Right:* An ideal allocation that compensates for higher services time with lower queue lengths.

### 3.7.2 Performance Fluctuations are the Norm

Servers in cloud environments routinely experience performance fluctuations due to a multitude of reasons. Citing experiences at Google, Dean and Barroso [24] list many sources of latency variability that occur in practice. Their list includes, but is not limited to, contention for shared resources within different parts of and between applications (further discussed in [44]), periodic garbage collection, maintenance activities (such as log compaction), and background daemons performing periodic tasks [75]. Recently, an experimental study of response times on Amazon EC2 [89] illustrated that long tails in latency distribution can also be exacerbated by virtualization. A study [42] of interactive services at Microsoft Bing found that over 30% of analyzed services have $95^{th}$ percentile of latency 3 times their median latency. Their analysis showed that a major cause for the high service performance variability is that latency varies greatly across machines and time. Lastly, a common workflow involves accessing large volumes of data from a data store to serve as inputs for batch jobs on large-scale computing platforms such as Hadoop, and injecting results back into the data store [81]. These workloads can introduce latency spikes at the data store and further impact on end-user delays.

As part of our study, we spoke with engineers at Spotify and SoundCloud, two companies that use and operate large Cassandra clusters in production. Our discussions further confirmed that all of the above mentioned causes of performance variability are true pain points. Even in well provisioned clusters, unpredictable events such as garbage collection on individual hosts can lead to latency spikes. Furthermore, Cassandra nodes periodically perform *compaction*, wherein a node merges multiple SSTables (the on-disk representation of the stored data) [9, 18] to minimize the number of SSTable files to be consulted per-read, as well as to reclaim space. This leads to significantly increased I/O activity.

Given the presence of time-varying performance fluctuations, many of which can potentially occur even at sub-second timescales [24], it is important that systems gracefully adapt to changing conditions. By exploiting server redundancy in the system, we investigate how replica selection effectively reduces the tail latency.

### 3.7.3 Load-Based Replica Selection is Hard

Accommodating time-varying performance fluctuations across nodes in the system necessitates a replica selection strategy that takes into account the load across different servers in the system. A strategy commonly employed by many systems is the *least-outstanding requests* strategy (LOR). For each request, the client selects the server to which it has the least number of outstanding requests. This technique is simple to implement and does not require global system information, which may not be available or is difficult to obtain in a scalable fashion. In fact, this is commonly used in load-balancing applications such as Nginx [56] (recommended as a load-balancer for Riak [69]) or Amazon ELB [8].

However, we observe that this technique is not ideal for reducing the latency tail, especially since many realistic workloads are skewed in practice and access patterns change over time [12]. Consider the system in Figure 3.23, with two replica servers that at a particular point in time have service times of 4 ms and 10 ms respectively. Assume all three clients receive a burst of 4 requests each. Each request needs to be forwarded to a single server. Based on purely local information, if every client selects a server using the LOR strategy, it will result in each server receiving an equal share of requests. This leads to a maximum latency of 60 ms, whereas an ideal allocation in this case obtains a maximum latency of 32 ms. We note that *LOR* over time will prefer faster servers, but by virtue of purely relying on local information, it does not account for the existence of other clients with potentially bursty workloads and skewed access patterns, and does not explicitly adapt

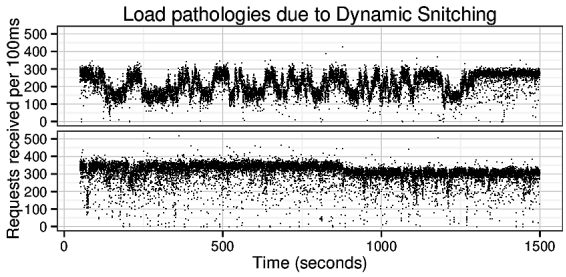



Figure 3.24: Example load oscillations seen by a given node in Cassandra due to Dynamic Snitching, in measurements obtained on Amazon EC2. The y-axis represents the number of requests processed in a 100 ms window by a Cassandra node. Even under stable conditions (*bottom*), the number of requests processed in a 100 ms window by a node ranges from 0 up to 500, which is symptomatic of herd behavior.

to fast-changing service times.

Designing distributed, adaptive and stable load-sensitive replica selection techniques is challenging. If not carefully designed, these techniques can suffer from "herd behavior" [53, 72]. Herd behavior leads to load oscillations, wherein multiple clients are coaxed to direct requests towards the least-loaded server, degrading the server's performance, which subsequently causes clients to repeat the same procedure with a different server.

Indeed, looking at the landscape of popular data stores (Table 3.2), we find that most systems only implement very simple schemes that have little or no ability to react quickly to service-time variations nor distribute requests in a load-sensitive fashion. Among the systems we studied, Cassandra implements a more sophisticated strategy called Dynamic Snitching that attempts to make replica selection decisions informed by histories of read latencies and I/O loads. However, through performance analysis of Cassandra, we find that this technique suffers from several weaknesses, which we discuss next.

### 3.7.4 Dynamic Snitching's Weaknesses

Cassandra servers organize themselves into a one-hop distributed hash table. A client can contact any server for a read request. This server then acts as a *coordinator*, and internally fetches the record from the node hosting the data. Coordinators select the best replica for a given request using *Dynamic Snitching*. With Dynamic Snitching, every Cassandra server ranks and prefers faster replicas by factoring in read latencies to each of its peers, as well as I/O load information that each server shares with the cluster through a gossip protocol.

Given that Dynamic Snitching is load-based, we evaluate it to characterize how it manages tail-latencies and if it is subject to entering load-oscillations. Indeed, our experiments on Amazon EC2 with a 15-node Cassandra cluster confirm this (the details of the experimental setup are described in Section 3.7.10). In particular, we recorded heavy-tailed latency characteristics where the difference between the $99.9^{th}$ percentile latencies are *up to 10 times* that of the median. Furthermore, we recorded the number of read requests individual Cassandra nodes serviced in 100 ms intervals. For every run, we observed the node that contributed most to the overall throughput. These nodes consistently exhibited synchronized load oscillations, example sequences of which are shown in Figure 3.24. Additionally, we confirmed our results with the Spotify engineers, who have also encountered load instabilities that arise due to garbage-collection induced performance fluctuations in the system [47].

A key reason for Dynamic Snitching's vulnerability to oscillations is that each Cassandra node re-computes scores for its peers at fixed, discrete intervals. This interval based scheme poses two problems. First, the system cannot react to time-varying performance fluctuations among peers that occur at time-scales less than the fixed-interval used for the score recomputation. Second, by virtue of fixing a choice over a discrete time interval (100 ms by default), the system risks synchronization as seen in Figure 3.24. While one may argue that this can be overcome by shortening the interval itself, the calculation performed to compute the scores is expensive, as it is also stated explicitly in the source code; a median over a history of exponentially weighted latency samples (that is reset only every 10 minutes) has to be computed for each node as part of the scoring

process. Additionally, Dynamic Snitching relies on gossiping one second averages of `iowait` information between nodes to aid with the ranking procedure (the intuition being that nodes can avoid peers who are performing compaction). These `iowait` measurements influence the scores used for ranking peers heavily (up to two orders of magnitude more influence than latency measurements). Thus, an external or internal perturbation in I/O activity can influence a Cassandra node's replica selection loop for extended intervals. Together with the synchronization-prone behavior of having a periodically updated ranking, this can lead to poor replica selection decisions that degrade system performance.

### 3.7.5 C3 Design

C3 is an adaptive replica selection mechanism designed with the objective of reducing tail latency. Based on the considerations in Section 3.7.1, our design of C3 must be:

$i$) *Adaptive:* Replica selection must *cope and quickly react to heterogeneous and time-varying service times* across servers.

$ii$) *Well-behaved:* Clients performing replica selection must *avoid herd behaviors* where a large number of clients concentrate requests towards a fast server.

At the core of C3's design are the two following components that allow it to satisfy the above properties:

(i) **Replica Ranking:** Using minimal and approximate feedback from individual servers, clients rank and prefer servers according to a scoring function. The scoring function factors in the existence of multiple clients and the subsequent risk of herd behavior, whilst allowing clients to prefer faster servers.

(ii) **Distributed Rate Control and Backpressure:** Every client rate limits requests destined to each server, adapting these rates in a fully-distributed manner using a congestion-control inspired technique [38]. When rate limits of all candidate servers for a request are exceeded, clients retain requests in a backlog queue until at least one server is within its rate limit again.

### 3.7.6 Replica Ranking

With replica ranking, clients individually rank servers according to a scoring function, with the scores serving as a proxy for the latency to expect from the corresponding server. Clients then use these scores to prefer faster servers (lower scores) for each request. To reduce tail latency, we aim to minimize the product of queue-size ($q_s$) and service-time ($1/\mu_s$, the inverse of the service rate) across every server $s$.

**Delayed and approximate feedback.** In C3, servers relay feedback about their respective $q_s$ and $1/\mu_s$ on each response to a client. The $q_s$ is recorded after the request has been serviced and the response is about to be dispatched. Clients maintain Exponentially Weighted Moving Averages (EWMA) of these metrics to smoothen the signal. We refer to these smoothed values as $\bar{q}_s$ and $\bar{\mu}_s$.

**Accounting for uncertainty and concurrency.** The delayed feedback from the servers lends clients only an approximate view of the load across the servers and is not sufficient by itself. Such a view is oblivious to the existence of other clients in the system, as well as the number of requests that are potentially in flight, and is thus prone to herd behaviors. It is therefore imperative that clients account for this potential concurrency in their estimation of each server's queue-size.

For each server $s$, a client maintains an instantaneous count of its outstanding requests $os_s$ (requests for which a response is yet to be received). Clients calculate the queue-size estimate ($\hat{q}_s$) of each server as $\hat{q}_s = 1 + os_s \cdot w + \bar{q}_s$, where $w$ is a weight parameter. We refer to the $os_s \cdot w$ term as the *concurrency compensation*.

The intuition behind the concurrency compensation term is that a client will always extrapolate the queue-size of a server by an estimated number of requests in flight. That is, it will always account for the possibility of multiple clients concurrently submitting requests to the same server. Furthermore, clients with a higher value of $os_s$ will implicitly project a higher queue-size at $s$ and thus rank it lower than a client that has sent fewer requests to $s$. Using this queue-size estimate to project the $\hat{q}_s/\bar{\mu}_s$ ratio results in a desirable effect: a client with a higher demand will be more likely to rank $s$ poorly compared to a client with a lighter demand. The estimate therefore provides a degree of robustness to synchronization. In our experiments, we set $w$ to the number of clients in the system. This serves as a good approximation in settings where the number of clients is comparable to the expected queue lengths at the servers.

**Penalizing long queues.** With the above estimation, clients can compute the $\hat{q}_s/\bar{\mu}_s$ ratio of each server and rank them accordingly. However, given the existence of multiple clients and time-varying service times, a function linear in $\hat{q}$ is not an effective scoring function for replica ranking. To see why, consider the example

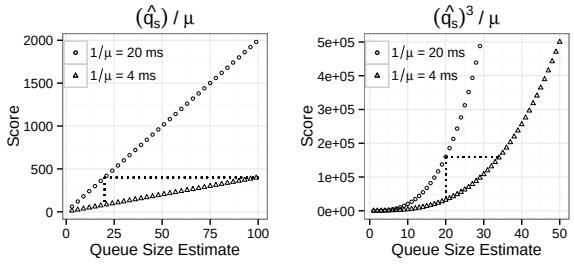



Figure 3.25: A comparison between linear (**left**) and cubic (**right**) scoring functions. For differing values of $1/\mu$, the difference in queue-size estimates required for the scores of two replicas to be equal is smaller for the cubic function (thus penalizing longer queues).

in Figure 3.25. The figure shows how clients would score two servers using a linear function: here, the service time estimates are 4 ms and 20 ms, respectively. We observe that under a linear scoring regime, for a queue-size estimate of 20 at the slower server, only a corresponding value of 100 at the faster server would cause a client to prefer the slower server again. If clients distribute requests by choosing the best replica according to this scoring function, they will build up and maintain long queues at the faster server in order to balance response times between the two nodes.

However, if the service time of the faster server increases due to an unpredictable event such as a garbage collection pause, *all requests* in its queue will incur higher waiting times. To alleviate this, C3's scoring function *penalizes longer queue lengths* using the same intuition behind that of delay costs as in [84, 15]. That is, we use a non-decreasing convex function of the queue-size estimate in the scoring function to penalize longer queues. We achieve this by raising the $\hat{q}_s$ term in the scoring function to a higher degree, $b$: $(\hat{q}_s)^b/\bar{\mu_s}$. Returning to the above example, this means the scoring function will treat the above two servers as being of equal score if the queue-size estimate of the faster server ($1/\mu = 4$ ms) is $\sqrt[b]{20/4}$ times that of the slower server ($1/\mu = 20$ ms). For higher values of $b$, clients will be less greedy about preferring a server with a lower $\mu^{-1}$. We use $b = 3$ to have a cubic scoring function (Figure 3.25), which presents a good trade-off between clients preferring faster servers and providing enough robustness to time-varying service times.

**Cubic replica selection.** In summary, clients use the following scoring function for each replica:

$$\Psi_s = R_s - 1/\bar{\mu_s} + (\hat{q}_s)^3/\bar{\mu_s}$$

where $\hat{q}_s = 1 + os_s \cdot n + \bar{q_s}$ is the queue-size estimation term, $os_s$ is the number of outstanding requests from the client to $s$, $n$ is the number of clients in the system, and $R_s$, $\bar{q_s}$ and $\bar{\mu_s}^{-1}$ are EWMAs of the response time (as witnessed by the client),[8] queue-size and service time feedback received from server $s$, respectively. The score reduces to $R_s$ when the queue-size estimate term of the server is 1 (which can only occur if the client has no outstanding requests to $s$ and the queue-size feedback is zero). Note that the $R_s - \mu_s^{-1}$ term's contribution to the score diminishes quickly when the client has a non-zero queue-size estimate (see Figure 3.25).

### 3.7.7 Rate Control and Backpressure

Replica selection allows clients to prefer faster servers. However, replica selection alone cannot ensure that the combined demands of all clients on a single server remain within that server's capacity. Exceeding capacity increases queuing on the server-side and reduces the system's reactivity to time-varying performance fluctuations. Thus, we introduce an element of rate-control to the system, wherein every client rate-limits requests to individual servers. If the rates of all candidate servers for a request are saturated, clients retain the request in a backlog queue until a server is within its rate limit again.

**Decentralized rate control.** To account for servers' performance fluctuations, clients need to adapt their estimations of a server's capacity and adjust their sending rates accordingly. As a design choice and inspired by the CUBIC congestion-control scheme [38], we opt to use a decentralized algorithm for clients to estimate and adapt rates across servers. That is, we avoid the need for clients to inform each other about their demands

[8] Note $R_s$ implicitly accounts for network latency but we consider that network congestion is not the source of performance fluctuations.

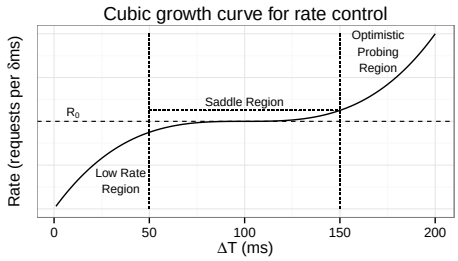



Figure 3.26: Cubic function for clients to adapt their sending rates

for individual servers, or for the servers to calculate allocations for potentially numerous clients individually. This further increases the robustness of our system: clients' adaptation to performance fluctuations in the system is not purely tied to explicit feedback from the servers.

Thus, every client maintains a token-bucket based rate-limiter for each server, which limits the number of requests sent to a server within a specified time window of $\delta$ ms. We refer to this limit as the *sending-rate* (*srate*). To adapt the rate limiter according to the perceived performance of the server, clients track the number of responses being received from a server in a $\delta$ ms interval, that is, the *receive-rate* (*rrate*). The rate-adaptation algorithm aims to adjust *srate* in order to match the *rrate* of the server.

**Cubic rate adaptation function.** Upon receiving a response from a server $s$, the client compares the current *srate* and *rrate* for $s$. If the client's sending rate is lower than the receive rate, it increases its rate according to a cubic function [38]:

$$srate \leftarrow \gamma \cdot \left( \Delta T - \sqrt[3]{(\frac{\beta \cdot R_0}{\gamma})} \right)^3 + R_0$$

where $\Delta T$ is the elapsed time since the last rate-decrease event, and $R_0$ is the "saturation rate" — the rate at the time of the last rate-decrease event. If the receive-rate is lower than the sending-rate, the client decreases its sending-rate multiplicatively by $\beta$. $\gamma$ represents a scaling factor and is chosen to set the desired duration of the saddle region (see Section 3.7.9 for the values used).

**Benefits of the cubic function.** While we have not fully explored the vast design space for a rate adaptation technique, we were attracted to a cubic growth function because of its property of having a saddle region. The functioning of the *cubic* rate adaption strategy caters to the following three operational regions (Figure 3.26): (1) *Low-rates:* when the current sending rate is significantly lower than the saturation rate (after say, a multiplicative decrease),the client increases the rate steeply; (2) *Saddle region:* when the sending rate is close to the perceived saturation point of the server ($R_0$), the client stabilizes its sending rate, and increases it conservatively, and (3) *Optimistic probing:* if the client has spent enough time in the stable region, it will again increase its rate aggressively, and thus probe for more capacity. At any time, if the algorithm perceives itself to be exceeding the server's capacity, it will update its view of the server's saturation point and multiplicatively reduce its sending rate. The parameter $\gamma$ can be adjusted for a desired length of the saddle region. Lastly, given that multiple clients may potentially be adjusting their rates simultaneously, for stability reasons, we cap the step size of a rate increase by a parameter $s_{\max}$.

### 3.7.8 Putting Everything Together

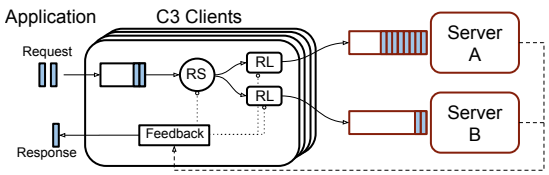



Figure 3.27: Overview of C3. **RS:** Replica Selection scheduler, **RL:** Rate Limiter of server $s \in [A, B]$.

**Algorithm 1** On Request Arrival (Request $req$, Replicas $\mathcal{R}$)

**repeat**
  $\mathcal{R} \leftarrow sort(\mathcal{R})$ ▷ sort replicas by cubic score function
  **for** Server $s$ in $\mathcal{R}$ **do**
    **if** $s$ within $srate_s$ **then**
      $consume\_token(srate_s)$
      $os_s \leftarrow os_s + 1$ ▷ update outstanding requests
      $send(req, s)$ ▷ send to server $s$
      **return**
  **if** $req$ **not sent then**
    **wait** until token available ▷ Backpressure
**until** $req$ is sent

C3 combines distributed replica selection and rate control as indicated in Algorithms 1 and 2, with the control flow in the system depicted in Figure 3.27. When a client issues a request, it is directed to a replica selection scheduler. The scheduler uses the scoring function to order the subset of servers that can handle the request, that is, the replica group ($\mathcal{R}$). It then iterates through the list of replicas and selects the first server $s$ that is within the rate as defined by the local rate limiter for $s$. If all replicas have exceeded their rate limits, the request is enqueued into a backlog queue. The scheduler then waits until at least one replica is within its rate before repeating the procedure. When a response for a request arrives, the client records the feedback metrics from the server and adjusts its sending rate for that server according to the cubic-rate adaptation mechanism. After a rate increase, a hysteresis period is enforced (Algorithm 2, line 3) before another rate-decrease so as to allow clients' receive-rate measurements enough time to catch up since the last increased sending rate at $T_{inc}$.

**Algorithm 2** On Request Completion (Request $req$, Server $s$)

1: $os_s \leftarrow os_s - 1$ ▷ update outstanding requests
2: **update** EWMA of $q_s$, $\mu_s^{-1}$ feedback
3: **if** ($srate_s > rrate_s$ **&&** $now() - T_{inc} > hysteresis\_period$) **then**
4:  $R_0 \leftarrow srate_s$
5:  $srate_s \leftarrow srate_s \cdot \beta$
6:  $T_{dec} \leftarrow now()$
7: **else if** ($srate_s < rrate_s$) **then**
8:  $\Delta T \leftarrow now() - T_{dec}$
9:  $T_{inc} \leftarrow now()$
10:  $R \leftarrow \gamma \cdot \left(\Delta T - \sqrt[3]{(\frac{\beta \cdot R_0}{\gamma})}\right)^3 + R_0$
11:  $srate_s \leftarrow \min(srate_s + s_{\max}, R)$

### 3.7.9 Implementation

We implemented C3 within Cassandra, for Cassandra's internal read request routing mechanism. This means that every Cassandra node is both a C3 client and server (specifically, coordinators in Cassandra's read path are C3 clients). In vanilla Cassandra, every read request follows a synchronous chain of steps leading up to an eventual enqueuing of the request into a per-node TCP connection buffer. For C3, we modified this chain of steps to control the number of requests that would be pushed to the TCP buffers of each node. Recall that C3's replica scoring and rate control operate at the granularity of replica groups. Given that in Cassandra there are as many replica groups as nodes themselves, we need as many backpressure queues and replica selection schedulers as there are nodes. Thus, every read-request upon arrival in the system needs to be asynchronously routed to a scheduler corresponding to the request's replica group. Lastly, when a coordinator node performs a remote read, the server that handles the request tracks the service time of the operation and the number of pending read requests in the server. This information is piggybacked to the coordinator and serves as the feedback for the replica ranking.

There are challenges in making this implementation efficient. For one, since a single remote peer can be part of multiple replica sets, multiple admission control schedulers may potentially contend to push a request from their respective backpressure queues towards the same endpoint. Care needs to be exercised that this does not lead to starvation. To handle this complexity, we used the Akka framework [2] for message-passing concurrency (*Actor* based programming). With Akka, every per-replica-group scheduler is represented as a

single actor, and we configured the underlying Java thread dispatcher to schedule between the actors fairly. This design of having multiple backpressure queues also increases robustness, as one replica group entering backpressure will not affect other replica groups. The message queue that backs each Akka actor implicitly serves as the backpressure queue per replica group. At roughly 600 bytes of overhead per actor, our extensions to Cassandra are lightweight. Our implementation amounted to 398 lines of code.[9]

For the rest of our study, we set the cubic rate adaptation parameters as follows: the multiplicative decrease parameter $\beta$ is set to 0.2, and we configured $\gamma$ to set the saddle region to be 100 ms long. We define the rate for each server as a number of permissible requests per 20 ms ($\delta$), and use a hysteresis duration equal to twice the rate interval. We cap the cubic-rate step size ($s_{max}$) to 10. We did not conduct an exhaustive sensitivity analysis of all system parameters, which we leave for future work. Lastly, Cassandra uses read-repairs for anti-entropy; a fraction of read requests will go to all replicas (10% by default). This further allows coordinators to update their view of their peers.

### 3.7.10 System Evaluation

We evaluated C3 on Amazon EC2. Our Cassandra deployment consisted of 15 m1.xlarge instances. We tuned the instances and Cassandra according to the officially recommended production settings from Datastax [17] as well as in consultation with our contacts from the industry who operate production Cassandra clusters.

On each instance, we configured a single RAID0 array encompassing the four ephemeral disks which served as Cassandra's data folder (we also experimented on instances with SSD storage as we report on later). As we don't have production workloads, we used the industry-standard Yahoo Cloud Serving Benchmark (YCSB) [21] to generate datasets and run our workloads while stressing Cassandra up to its maximum attainable throughput. We assign tokens to each Cassandra node such that each node owns an equal segment of the keyspace. Cassandra's replication factor was set to 3. We inserted 500 million 1KB size records generated by YCSB, which served as the dataset. The workload against the cluster was driven from three instances of YCSB running in separate VMs, each running 40 request generators, for a total of 120 generators. Each generator has a TCP connection of its own to the Cassandra cluster. Generators create requests for keys distributed according to a Zipfian access pattern prescribed by YCSB, with Zipf parameter $\rho = 0.99$, drawing from a set of 10 million keys. We used three common workload patterns for Cassandra deployments to evaluate our scheme: read-heavy (95% reads – 5% writes), update-heavy (50% reads – 50% writes) and read-only (100% read). These workloads generate access patterns typical of photo tagging, session-store and user-profile applications, respectively [21]. The read and update heavy workloads in particular are popular across a variety of Cassandra deployments [29, 43]. Each measurement involves 10 million operations of the workload, and is repeated five times. Bar plots represent averages and $95^{th}$ percentile confidence intervals.

In evaluating C3, we are interested in answering the following questions across various conditions:

(i) Does C3 improve the tail latency without sacrificing the mean or median?
(ii) Does C3 improve the read throughput (requests/s)?
(iii) How well does C3 load condition the cluster and adapt to dynamic changes in the environment?

**Impact of workload on latency:** Figure 3.28 shows the read latency characteristics of Cassandra across different workloads when using C3 compared to Dynamic Snitching (DS). Regardless of the workload used, C3 improves the latency across all the considered metrics, namely, the mean, median, $99^{th}$ and $99.9^{th}$ percentile latencies. Since the ephemeral storage in our instances are backed by spinning-head disks, the latency increases with the amount of random disk seeks. This explains why the read-heavy workload results in lower latencies than the read-only workload (since the latter causes more random seeks). Furthermore, C3 effectively shortens the ratio of tail-latencies to the median, leading to a more predictable latency profile. With the read-heavy workload, the difference between the $99.9^{th}$ percentile latency and the median is $24.5$ ms with C3, whereas with DS, it is $83.91$ ms: *more than 3x improvement*. In the update-heavy and read-only scenarios, C3 improves the same difference by a factor of 2.6 each. C3 also improves the mean latency by 3-4 ms across all scenarios.

**Impact of workload on read throughput:** Figure 3.29 shows the measured throughputs for C3 versus DS. By virtue of controlling waiting times across the replicas, C3 makes better use of the available system capacity, resulting in an increase in throughput across the considered workloads. In particular, C3 improves the throughput by between 26% and 43% across the considered workloads (update-heavy and read-heavy

[9] Based on a Cassandra 2.0 development version.

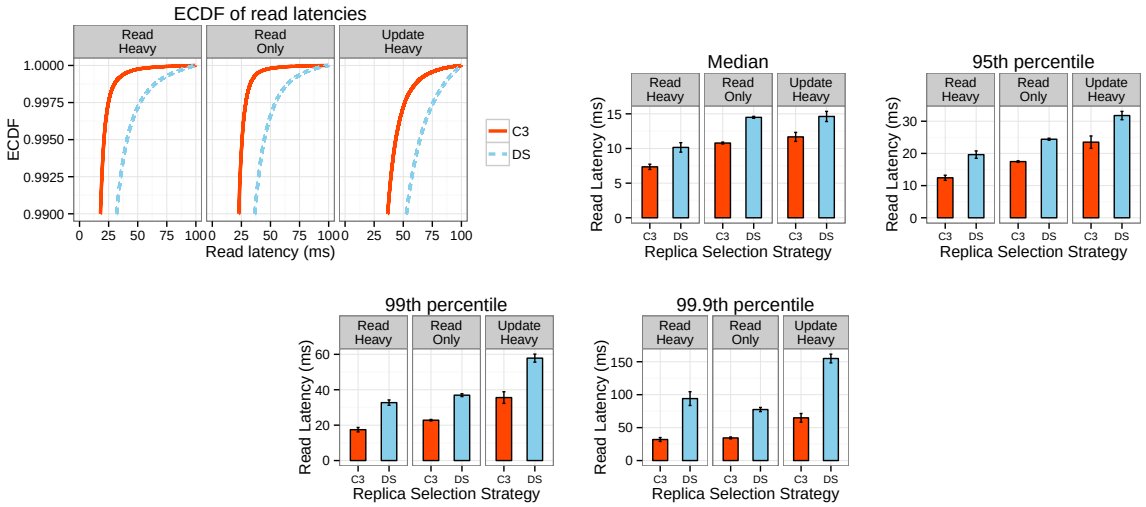



Figure 3.28: Cassandra's latency characteristics when using Dynamic Snitching and C3. C3 significantly improves the tail latency under different workloads without compromising the median.

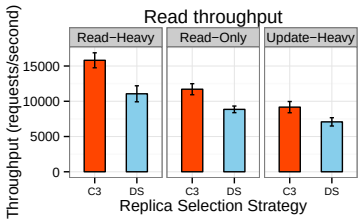



Figure 3.29: Throughput obtained with C3 and with Dynamic Snitching. C3 achieves higher throughput by better utilizing the available system capacity across replica servers.

workloads respectively). We also note that the difference in throughput between the read- and update-heavy workloads of roughly 75% (across both strategies) is consistent with publicly available Cassandra benchmark data [29].

**Impact of workload on load-conditioning:** We now verify whether C3 fulfills its design objective of avoiding load pathologies. Since the key access pattern of our workloads are Zipfian distributed, we observe the load over time of the node that has served the highest number of reads across each run, that is, the most heavily utilized node. Figure 3.30 represents the distribution of the number of reads served per 100 ms by the most heavily utilized node in the cluster across runs. Note that *despite improving the overall system throughput*, the most heavily utilized node in C3 serves fewer requests than with DS. As a further confirmation of this, we present an example load profile as produced by C3 on highly utilized nodes (Figure 3.31). Unlike with DS, we do not see synchronized load-spikes when using C3, evidenced by the lack of oscillations and synchronized vertical bursts in the time-series. Furthermore, given that C3's rate control absorbs and distributes bursts carefully, it leads to a smoother load-profile where samples of the load in a given interval are closer to the system's true capacity unlike with DS.

**Performance at higher system utilization:** We now compare C3 with DS to understand how the performance of both systems degrade with an increase in overall system utilization. We increase the number of workload generators from 120 to 210 (an increase of 75%). Figure 3.32 presents the tail latencies observed for the read-heavy workload. For a 75% increase in the demand, we observe that C3's latency profile, even at the $99.9^{th}$ percentile, degrades proportionally to the increase in system load. With DS, the median and $99.9^{th}$ percentile latencies degrade by roughly 82%, whereas the $95^{th}$ and $99^{th}$ percentile latencies *degrade by factors of up to 150%*. Furthermore, the mean latency with Dynamic Snitching is *70% higher* than with C3 under the higher load.

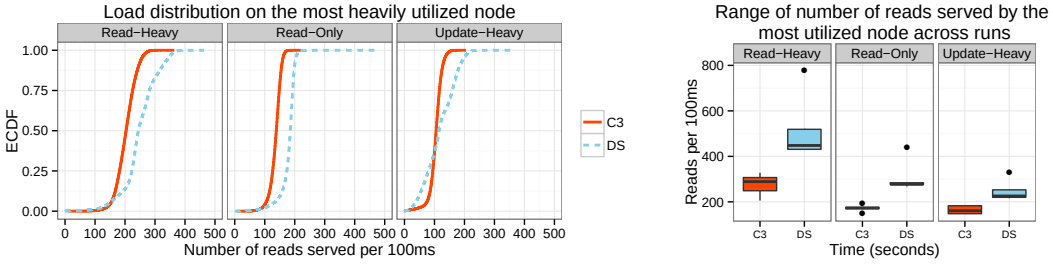



Figure 3.30: Aggregated distribution of number of reads serviced per 100 ms, by the most heavily loaded Cassandra node per run. With C3, the most heavily utilized node has a lower range in the load over time, wherein the difference between the 99th percentile and median number of requests served in 100 ms is lower than with Dynamic Snitching.

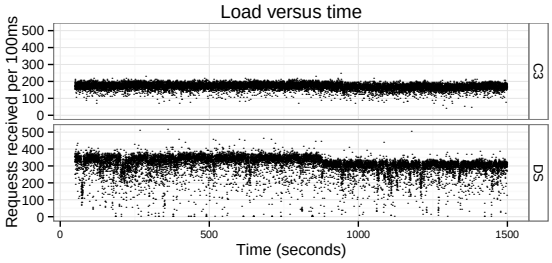



Figure 3.31: Example number of reads received by a single Cassandra node, per 100ms. With C3 (top), Cassandra coordinators internally adjust sending rates to match their peers' perceived capacity, leading to a smoother load profile free of oscillations. The per-server load is lower in C3 also because the requests are spread over more servers compared to DS (bottom).

**Adaptation to dynamic workload change:** We now evaluate a scenario where an update-heavy workload enters a system where a read-heavy workload is already active, and observe the effect on the latter's read latencies. The experiment begins with 80 generators running a read-heavy workload against the cluster. After 640 s, an additional 40 generators enter the system, issuing update-heavy workloads. We observe the latencies from the perspective of the read-heavy generators around the 640 s mark. Figure 3.33 indicates a time-series of the latencies contrasting C3 versus DS. Each plot represents a 50-sample wide moving median[10] over the recorded latencies. Both DS and C3 react to the new generators entering the system, with a degradation of the read latencies observed at the 640 s mark. However, in contrast to DS, C3's latency profile degrades gracefully, evidenced by the lack of synchronized spikes visible in the time-series as is the case with DS.

**Skewed record sizes:** So far, we considered fixed-length records. Since C3 relies on per-request feedback of the service times in the system, we observe whether variable length records may introduce any anomalies in the control loop. We use YCSB to generate a similar dataset as before, but where field sizes are Zipfian distributed (favoring shorter values). The maximum record length is 2KB, with each record consisting of the key and ten fields. Again, C3 improves over DS along all the considered latency metrics. In particular, with C3, the $99^{th}$ percentile latency is just under 14 ms, whereas that of DS is close to 30 ms; *more than 2x improvement*.

**Performance when using SSDs:** As a further demonstration of C3's generality, we also perform measurements with m3.xlarge instances, which are backed by two 40 GB SSD disks. We configured a RAID0 array encompassing both disks. We reduced the dataset size to 150 million 1KB records in order to ensure that the dataset fits the reduced disk capacities of all nodes. Given that with SSDs, the system can sustain a higher workload, we used 210 read-heavy generators (70 threads per YCSB instance). Figure 3.34 illustrates the latency improvements obtained when using C3 versus DS with SSD backed instances. Even under the higher

[10] A moving median is better suited to reveal the underlying trend of a high-variance time-series than a moving average [11]

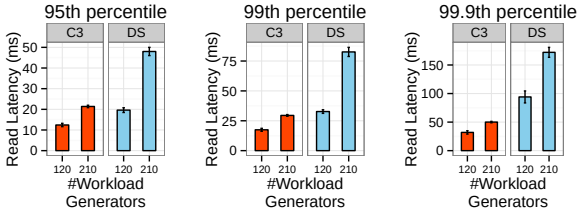



Figure 3.32: Overall performance degradation when increasing the number of workload generators from 120 to 210.

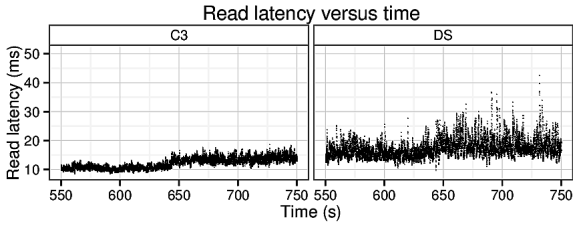



Figure 3.33: Dynamic workload experiment. The moving median over the latencies observed by the read-heavy generators from a run each involving C3 (**left**) and DS (**right**). At time 640 s, 40 new generators join the system and issue update-heavy workloads. With C3, the latencies degrade gracefully, whereas DS fails to avoid latency spikes.

load, both algorithms have significantly lower latencies than when using spinning head disks. However, C3 again improves the $99.9^{th}$ percentile latency by *more than 3x*. Furthermore, the difference between the $99^{th}$ and $99.9^{th}$ percentile latencies in C3 is *under 5 ms*, whereas with DS, it is on the order of 20 ms. Lastly, C3 also improves the average latency by roughly 3 ms, and increases the read throughput by 50% of that obtained by DS.

**Comparison against request reissues:** Cassandra has an implementation of speculative retries [24] as a means of reducing tail latencies. After sending a read request to a replica, the coordinator waits for the response until a configurable duration before reissuing the request to another replica. We evaluated the performance of DS with speculative retries, configured to fire after waiting until the $99^{th}$ percentile latency. However, we observed that latencies actually degraded significantly after making use of this feature, up to a factor of 5 at the $99^{th}$ percentile. We attribute this to the following cause: in the presence of highly variable response times across the cluster (already due to DS), coordinators potentially speculate too many requests. This increases the load on disks, further increasing seek latencies. Due to this anomaly, we did not perform further experiments. However, we would like to caution that speculative retries are not a silver bullet when operating a system at high utilization [87].

**Sending rate adaptation and backpressure over time:** Lastly, we turn to a seven-node Cassandra cluster in our local testbed to depict how nodes adapt their sending rates over time. Figure 3.35 presents a trace of

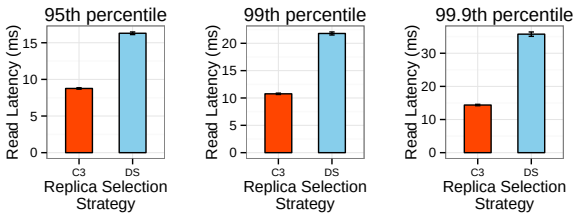



Figure 3.34: Results when using SSDs instead of spinning-head disks.

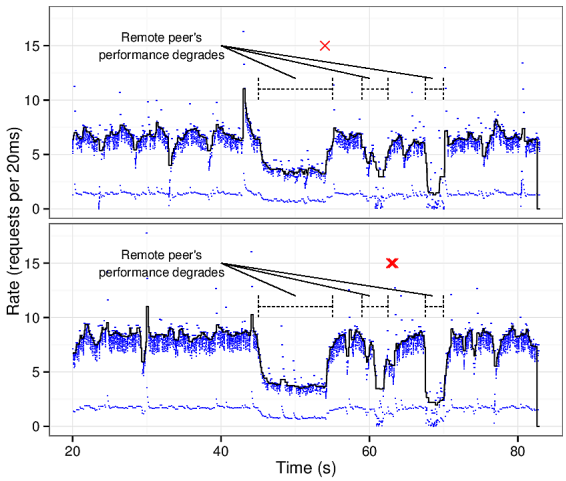



Figure 3.35: Sending rate adaptation performed by two coordinators against a third server. The receiving server's latency is artificially inflated three times. The blue dots represent the sending-rates as adjusted by the cubic rate control algorithm, the black line indicates a moving median of the sending rates, and the red X marks indicate moments when affected replica group schedulers enter backpressure mode.

the sending rate adaptation performed by two coordinators against a third node (*tracked node*). During the run, we artificially inflated the latencies of the tracked node thrice (using the Linux `tc` utility), indicated by the drops in throughput in the interval (45, 55) s, as well as the two shorter drops at times 59 s and 67 s. Observe that both coordinators' estimations of their peer's capacity agree over time. Furthermore, the figure depicts all three rate regimes of the cubic rate control mechanism. The points close to 1 on the y-axis are arrived at via the multiplicative decrease, causing the system to enter the low-rate regime. At that point, C3 aggressively increases its rate to be closer to the tracked saturation rate, entering the saddle region (along the smoothened median). The stray points above the smoothened median are points where C3 optimistically probes for more capacity. During this run, the backpressure mechanism fired 4 times (3 of which are very close in time) across both depicted coordinator nodes. Recall that backpressure is exerted when *all* replicas of a replica group have exceeded their rate limits. When the tracked node's latencies are reset to normal, the YCSB generators throttle up, sending a heavy burst in a short time interval. This causes a momentary surge of traffic towards the tracked node, forcing the corresponding replica selection schedulers to apply backpressure.

### 3.7.11 Evaluation Using Simulations

We also study the C3 scheme independently of the intricacies of Cassandra and draw more general results. For this purpose, we use large-scale simulations under different scenarios.

Our results show that C3 performs better than Least-Outstanding Requests (LOR) and Round-Robin (RR) under various service-time fluctuation frequencies. We also note that C3's performance remains relatively close to that of an oracle solution. The paper published at NSDI 2015 describes the results in details.

### 3.7.12 Summary

We highlighted the challenges involved in making a replica selection scheme explicitly cope with performance fluctuations in the system and environment. We presented the design and implementation of C3. C3 uses a combination of in-band feedback from servers to rank and prefer faster replicas along with distributed rate control and backpressure in order to reduce tail latencies in the presence of service-time fluctuations. Through comprehensive performance evaluations, we demonstrate that C3 improves Cassandra's mean, median and tail latencies (by up to 3 times at the $99.9^{th}$ percentile), all while increasing read throughput and avoiding load pathologies.

## 3.8 Jitsu: Just-In-Time Summoning of Unikernels

### 3.8.1 Introduction

The benefits of cloud hosting are clear: dynamic resource provisioning, lower capital expenditure, high availability, centralized management. Unfortunately, all services architected and deployed in this way inevitably suffer the same problem: latency. Physical separation between remote data centers where processing occurs and users of these services, imposes unavoidable minimum bounds on network latency. Recent developments in augmented reality (e.g., Google Glass) and voice control (e.g., Apple's Siri) particularly suffer in this regard. Concurrent with the move of services to the cloud, we are now seeing uptake of the 'Internet-of-Things' (IoT), giving rise to our second concern: integrity. These devices often rely on the network for their operation but many of the devices we use daily suffer from an unrelenting stream of security exploits, including routers, buildings and automobiles. The future success of IoT platforms being deployed in edge networks depends on the convenience of secure multi-tenant isolation that the public cloud utilizes.

The widely deployed Xen hypervisor enforces isolation between multiple tenants sharing physical machines. Xen recently added support for hardware virtualized ARM guests, opening up the possibility of building an embedded cloud: a system of distributed low-power devices, deployed near users, able to host applications delivering real-time services directly via local networks. There has been a steady increase in ARM boards featuring a favorable energy/price/speed trade-off for constructing embedded systems (e.g., the Cubieboard2 has 1GB RAM, a dual-core A20 ARM CPU and costs £39).

We present Jitsu, a system for securely managing multi-tenant networked applications on embedded infrastructure. Jitsu re-architects the Xen toolstack to lower the resource overheads of manipulating virtual machines (VMs), overcoming current limitations that prevent Xen from becoming an effective platform for building embedded clouds. Rather than booting conventional VMs, Jitsu services network requests with low latency using unikernels as the unit of deployment. These are small enough to be booted in a few hundred milliseconds, a latency that Jitsu further masks through connection hand-off. The MirageOS unikernels that we use are also secure enough to survive inexpertly managed network-facing deployment. Jitsu uses the virtual hardware abstraction layer provided by the Xen type-1 hypervisor, adding a new control toolstack that eliminates bottlenecks in VM management (Figure 3.36). Although developed with unikernels in mind, it preserves sufficient compatibility that many of its benefits apply equally to generic (e.g., Linux or FreeBSD) VMs targeting either ARM or x86. Since Jitsu provides an untra-low latency, flexible control plane for Unikernels, end-system containers like Docker or (mainly) data plane NFV/Forwarding systems like ClickOS can be coordinated and controlled through Jitsu, providing liquidity in the overall system management.

The remainder of this section will contain the following: a description of how to build efficient, secure unikernels on the new open-source Xen/ARM (Section 3.8.2) and an explanation of the Jitsu Xen toolstack architecture (Section 3.8.5).

Building software for embedded systems is typically more complex than for standard platforms. Embedded systems are often power-constrained, impose soft realtime constraints, and are designed around a monolithic firmware model that forces whole system upgrades rather than the upgrade of constituent packages. To date, general purpose hypervisors have not been able to meet these requirements, though microkernels have made inroads. Several approaches to providing application isolation have received attention recently. As each provides different trade-offs between security and resource usage, we discuss them in turn, motivating our choice of unikernels as our unit of deployment. We then outline the new Xen/ARM port that uses the latest ARM v7-A virtualization instructions and provide details of our implementation of a single-address space ARM unikernel using this new ABI.

### 3.8.2 Application Containment

Strong isolation of multi-tenant applications is a requirement to support the distribution of application and system code. This requires both isolation at runtime as well as compact, lightweight distribution of code and associated state for booting. We next describe the spectrum of approaches meeting these goals, depicted in Figure 3.37.

**OS Containers (Figure 3.37 (a))**: FreeBSD Jails and Linux containers both provide a lightweight mechanism to separate applications and their associated kernel policies. This is enforced via kernel support for isolated namespaces for files, processes, user accounts and other global configuration. Containers put the entire monolithic kernel in the trusted computing base, while still preventing applications from using certain

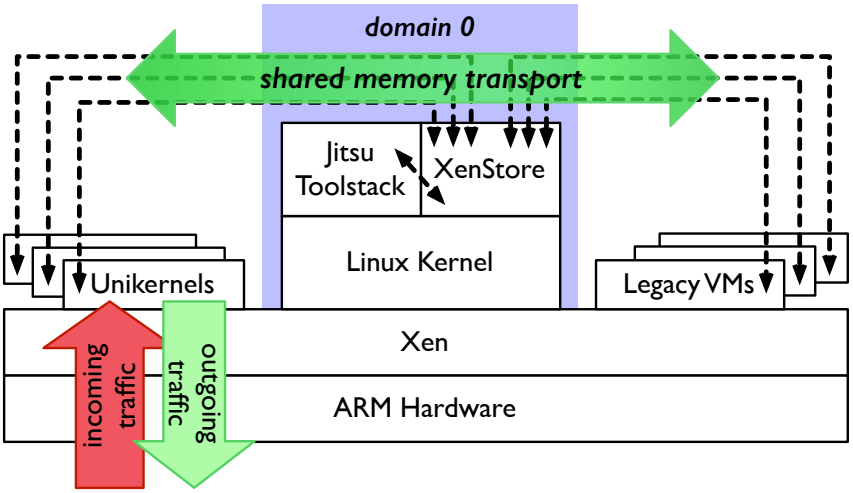



Figure 3.36: The Jitsu Architecture: external network connectivity is handled solely by memory-safe unikernels connected to general purpose VMs via shared memory.

functionality. Even the popular Docker container manager does not yet support isolation of root processes from each other (`https://docs.docker.com/articles/security/`).
Both the total number and ongoing high rate of discovery of vulnerabilities indicate that stronger isolation is highly desirable.
An effective way to achieve this is to build applications using a library operating system (libOS) to run over the smaller trusted computing base of a simple hypervisor. This has been explored in two modern strands of work.

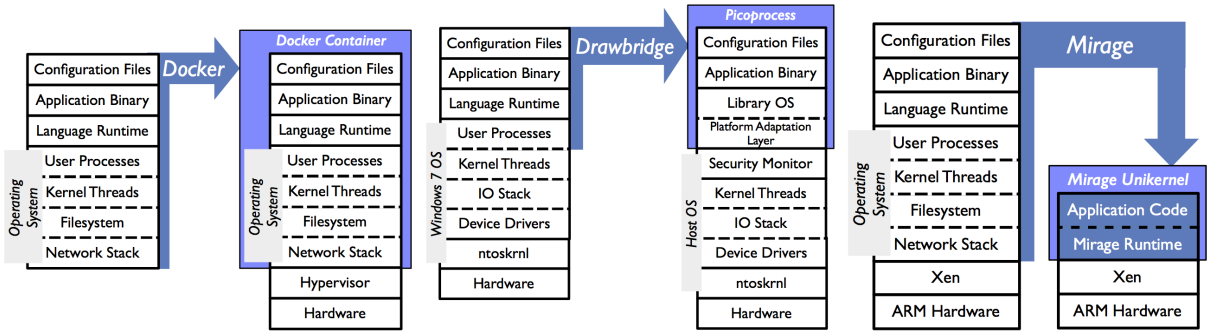
**Picoprocesses (Figure 3.37 (b))**: Drawbridge demonstrated that the libOS approach can scale to running Windows applications with relatively low overhead (just 16MB of working set memory). Each application runs in its own picoprocess on top of a hypervisor, and this technique has since been extended to running POSIX applications as well. Embassies refactors the web client around this model such that untrusted applications can run on the user's computer in low-level native code containers that communicate externally via the network.

**Unikernels (Figure 3.37 (c))**: Even more specialized applications can be built by leveraging modern programming languages to build unikernels. Single-pass compilation of application logic, configuration files and device drivers results in output of a single-address-space VM where the standard compiler toolchain has eliminated unnecessary features. This approach is most beneficial for single-purpose appliances as opposed to more complex multi-tenant services. Unikernel frameworks are gaining traction for many domain-specific tasks including virtualizing network functions, eliminating I/O overheads, building distributed systems and providing a minimal trust base to secure existing systems. In Jitsu we use the open-source MirageOS2 written in OCaml, a statically type-safe language that has a low resource footprint and good native code compilers for both x86 and ARM. A particular advantage of using MirageOS when working with Xen is that all the toolstack libraries involved are written entirely in OCaml, making it easier to safely manage the flow of data through the system and to eliminate code that would otherwise add overhead.

### 3.8.3 ARM Hardware Virtualization

Xen is a widely deployed type-1 hypervisor that isolates multiple VMs that share hardware resources. It was originally developed for x86 processors, on which it now provides three execution modes for VMs: paravirtualization (PV), where the guest operating system source is directly modified; hardware emulation (HVM), where specialized virtualization instructions and paging features available in modern x86 CPUs obviate the need to modify guest OS source code; and a hybrid model (PVH) that enables paravirtualized guests to use these newer hardware features for performance.
The Xen 4.4 release added support for recent ARM architectures, specifically ARM v7-A and ARM v8-A.




(a) Containers, e.g., Docker. (b) Picoprocesses, e.g., Drawbridge. (c) Unikernels, e.g., MirageOS.

Figure 3.37: Contrasting approaches to application containment.

These include extensions that let a hypervisor manage hardware virtualized guests without the complexity of full paravirtualization. The Xen/ARM port is markedly simpler than x86 as it can avoid a range of legacy requirements: e.g., x86 VMs require qemu device emulation, which adds considerably to the trusted computing base. Simultaneously, Xen/ARM is able to share a great deal of the mature Xen toolstack with Xen/x86, including the mechanics for specifying security policies and VM configurations. Jitsu can thus target both Xen/ARM and Xen/x86, resulting in a consistent interface that spans a range of deployment environments, from conventional x86 server hosting environments to the more resource-constrained embedded environments with which we are particularly concerned, where ARM CPUs are commonplace.

### 3.8.4 Xen/ARM Unikernels

Bringing up MirageOS unikernels on ARM required detailed work mapping the libOS model onto the ARM architecture. We now describe booting MirageOS unikernels on ARM, their memory management requirements, and device virtualization support.

**Xen Boot Library** The first generation of unikernels such as MirageOS (OCaml), HaLVM (Haskell) and the GuestVM (Java) were constructed by forking Mini-OS, a tiny Xen library kernel that initializes the CPU, displays console messages and allocates memory pages. Over the years, Mini-OS has been directly incorporated into many other custom Xen operating systems, has had semi-POSIX compatibility bolted on, and has become part of the trusted computing base for some distributions. This copying of code becomes a maintenance burden when integrating new features that get added to Mini-OS. Before porting to ARM, we therefore rearranged Mini-OS to be installed as a system library, suitable for static linking by any unikernel.

An important consequence of this is that a libc is no longer required for the core of MirageOS: all libc functionality is subsumed by pure OCaml libraries including networking, storage and unicode handling, with the exception of the rarely used floating point formatting code used by printf, for which we extracted code from the musl libc. Removing this functionality does not just benefit code size: these embedded libraries are both security-critical (they run in the same address space as the type-safe unikernel code) and difficult to audit (they target a wide range of esoteric hardware platforms and thus require careful configuration of many compile-time options). Our refactoring thus significantly reduced the size of a unikernel's trusted computing base as well as improving portability.

**Fast Booting on ARM** We then ported Mini-OS to boot against the new Xen ARM ABI. This domain building process is critical to reducing system latency, so we describe it here briefly. Xen/ARM kernels use the Linux zImage format to boot into a contiguous memory area. The Xen domain builder allocates a fresh virtual machine descriptor, assigns RAM to it and loads the kernel at the offset 0x8000 (32KB). Execution begins with the r2 register pointing to a Flattened Device Tree (FDT). This is a similar key/value store to the one supplied by native ARM bootloaders and provides a unified tree for all further aspects of VM configuration. The FDT approach is much simpler than x86 booting, where the demands of supporting multiple modes (paravirtual, hardware-assisted and hybrids) result in configuration information being spread across virtualized BIOS, memory and Xen-specific interfaces.

Some assembler code then performs basic boot tasks:

- Configuring the MMU, which handles mapping virtual to physical memory addresses.
- Turning on caching and branch prediction.
- Setting up the exception vector table, which defines how to handle interrupts and deal with various faults such as reading from an invalid memory address.
- Setting up the stack pointer and jumping into the C `arch_init` function for the remainder of execution. The early C code sets up the virtual logging console and interrupt controllers.

After this, unikernel-specific C code binds interrupt handlers, memory allocators, timekeeping and grant tables into the language runtime. The final step is to jump into the OCaml code section5 and begin executing application logic. The application links memory-safe OCaml libraries to perform the remaining functions of device drivers and network stacks.

**Modifying Memory Management** Once the MirageOS/ARM unikernel has booted, it runs in a single address space without context switching. However, the memory layout under ARM is significantly different from that for x86. Under the ARM Virtualization Extensions, there are two stages to converting a virtual memory address (used by application code) to a physical address in RAM, both of which go through translation tables. The first stage is under the control of the guest VM, where it maps the virtual address to what the guest believes is the physical address – the Intermediate Physical Address (IPA). The second stage, under the control of Xen, maps the IPA to the real physical address. MirageOS' memory needs are very simple compared with traditional guest OSs. Most memory is provided directly to the managed OCaml heap which is grown on demand. Unikernels will typically also allocate a few pages for interacting directly with Xen as these must be page-aligned and static, and so cannot be allocated on the garbage collected OCaml heap.

Although Xen does not commit to a specific fixed address for the IPA, the C code does need to run from a known location. To resolve this, the assembler boot code uses the program counter to detect where it is running and sets up a virtual-to-physical mapping that will make it appear at the expected location by adding a fixed offset to each virtual address. The physical address is always at a fixed offset from the virtual address and addresses wrap around, so virtual address `0xC0400000` maps back to physical address `0` in this example.

The stack, which grows downwards, is placed at the start of RAM so that an overflow will trigger a page fault that can be caught, and can also be grown in size later the boot process when all of the RAM is available. The 16KB translation table is an array of 4-byte entries each mapping 1MB of the virtual address space, so the 16KB table is able to map the entire 32-bit address space (4GB). Each entry can either give the physical section address directly or point to a second-level table mapping individual 4KB pages; MirageOS implements the former as this reduces possible delays due to TLB misses. The kernel code is followed by the data section containing constants and global variables, then the bss section with data that is initially zero and thus need not be stored in the kernel image, and finally the rest of the RAM under control of the memory allocator.

**Device Virtualization** On Xen/x86 it is possible to add virtual devices by two means: pure PV devices that operate via a split-device model, and emulated hardware devices that use the qemu device emulator to provide the software model. Xen/ARM does not support the more complex hardware emulation at all, instead mandating (as a new ABI) that VMs support the Xen PV driver model to attach virtual devices. MirageOS includes OCaml library implementations of the Xen PV protocols for networking and storage. The only modifications required from their x86 versions were the architecture-dependent memory barrier assembly instructions that differ between x86 and ARM, accessed via the OCaml foreign function interface. The result of this work is to bring the benefits of MirageOS unikernels (compact, specialized appliances without excess baggage) to the resource-constrained ARM platform, providing an alternative to running full Linux or FreeBSD VMs. While we have described the specifics of the MirageOS port here, other teams have already picked up our work for their respective projects and are adapting it for other runtimes such as Click and Haskell.

### 3.8.5 The Jitsu Toolstack

We turn now to the Jitsu toolstack which supports the low-latency on-demand launching of the unikernels in response to network traffic. Our goal is to ensure that services listening on a network endpoint are always available to respond to traffic, but are otherwise not running to reduce resource utilization. Jitsu is the Xen equivalent of the venerable `inetd` service on Unix, but instead of starting a process in response to incoming traffic, it starts a unikernel that can respond to requests on that IP ad- dress. While there have been wide-area versions of this approach in the past, we believe this is the first time it has been implemented with such low latency in a single embedded host without sacrificing isolation. We describe Jitsu in three phases, each of which progressively reduces end-to-end latency. First, the traditional Xen toolstack is highly serialized across multiple blocking internal components, leading to large boot times due to long pauses between actual boot activity. We thus reduce these boot times by reducing this blocking behavior and speeding up various boot components. Jitsu preserves the existing boot protocol so that the many millions of existing Xen VM images will continue to work.

Second, we describe optimization of the inter-VM communications protocol via conduits, a Plan9-like extension to support direct shared memory communication between named endpoints (Section 3.8.5.2). Conduits eliminate the need to use local networking to communicate between Jitsu and unikernels, further driving down latency.

Third, we introduce the *Synjitsu* directory service that masks boot latency to external clients by handling the initial stages of TCP handshake, only to hand-off the resulting state via a local conduit while the unikernel service completes booting and attaches to the network bridge (Section 3.8.6).

The net result is that a service VM can "cold boot" and respond to a TCP client in around 300–350ms, and an already-booted service can respond to local traffic in around 5ms. In all cases, all network traffic is handled via memory-safe code in an unprivileged Xen VM.

#### 3.8.5.1 Optimizing Boot Times

Jitsu builds on the existing Xen toolstack by extending XenStore, a storage space shared between all VMs running on a physical host. XenStore is a hierarchical, transactional key-value store where keys describe a path down a tree, and values store configuration and live status information for domains. Each running domain on a Xen instance has its own subtree, and so communication between domains can be coordinated via XenStore. There are several stages to a VM booting that are triggered by XenStore: *(i)* a domain builder process loads the guest kernel image and configures it within a Xen data structure before launching it; *(ii)* the new VM boots and attaches to its virtual devices, most notably a logging console and a network device; *(iii)* the remote end of the network and console device rings are attached to the backends that bridge them; and finally, *(iv)* the userspace starts and applications begin serving traffic. Jitsu's utility relies on the ability to launch new VMs very quickly. Using the vanilla Xen toolstack, VM boot times are far too high for this, typically 3–5 seconds with high CPU usage for a Linux VM – hardly "just in time" when trying to start a network service with imperceptible client delay.

Jitsu applies three optimizations to significantly reduce this, achieving lowest latency when booting a specialized unikernel instead of a generic VM.

(i) **Domain building**. Xen's domain builder creates the initial VM kernel image. Most of its work is to initialize and zero out physical memory pages, thus guests with less memory are naturally built more quickly. As unikernels require such small amounts of memory to boot (8MB is plenty), they have an advantage over modern Linux distributions which typically require at least 64MB and are often recommended 128MB or more.

(ii) **Parallel device attachment**. While modern Linux parallelizes much of its boot process, individual devices still have a serialization overhead. The console device, for example, attaches to a dom0 `xenconsoled` service that drains the VM output and logs it. More significantly, attaching the network driver requires the backend domain to create a `vif` device in dom0, and to add it to a network bridge so that it can receive traffic. This blocks the VM while a slew of RPCs go back-and-forth between it and dom0, where hotplug shell scripts are executed. This can be further sped up by parallelizing the entire device attachment cycle with the domain builder itself. Jitsu starts the `vif` creation process before the domain builder runs, resulting in the two running in parallel. Although we could eliminate this overhead entirely by pre-creating domains and attaching them to the bridge (making VM

launch simply a matter of attaching a unikernel to a domain before unpausing it), we prefer not to pay the cost of increased memory usage that would result from the pre-created domains.

(iii) **Transaction Deserialization**. As the domain is built, a series of XenStore operations coordinates the multiple components involved in booting a VM. Building just one domain involves many transactional operations, and it becomes a latency bottleneck if they do not parallelize well. There are two XenStore implementations provided by upstream Xen: the default is a C implementation with filesystem-based transactions, and the other an alternative written in OCaml that uses in-memory transactions with merge functions that reduce the number of conflicts. We further improved the OCaml XenStore transaction handling in a Jitsu-specific fork by providing a custom merge function that handles common directory roots in parallel transactions.

Figure 3.38 shows the dramatic differences in VM start time when doing VM start/stop operations in parallel, with the OCaml implementations clearly more efficient than the default daemon in C. This is due to the reduced number of conflicts which otherwise cause the toolstack to cancel and retry a large set of domain building RPCs.
Figure 3.39 breaks down the impact of the domain creation optimizations. The test builds the VM image with a console and network interface and starts it. As this measures only VM construction time, not boot time, it applies to both unikernels and Linux VMs. Memory usage is a significant factor in domain creation, with a 256MB domain taking a full second to create, and a 16MB domain (suitable for a unikernel) still taking a significant 650ms. Rewriting the networking hotplug scripts to use the lightweight dash rather than the default bash reduces boot time to 300ms, and eliminating forking by invoking ioctl calls directly rather than running shell scripts further reduces boot time to 200ms. The final two optimizations to parallelize `vif` setup and asynchronously attach the console give the end result of 120ms to boot on ARM.
Jitsu is fully compatible with x86 as well as ARM, and so we ran the same tests on a 2.4GHz quad-core AMD x86 64 server to compare boot times against ARM. The most optimized VM creation time was just 20ms on x86 – around 6 times faster than the lower powered ARM board. Although we are focused on embedded deployments, it is worth noting that such fast boot times are possible in situations where power consumption is less of a concern.

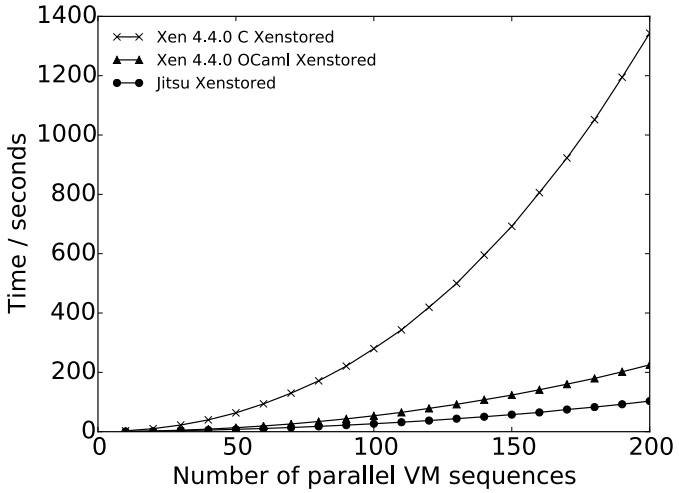



Figure 3.38: Comparison of different transaction reconciliation implementations during VM start/stop.

### 3.8.5.2 Communication Conduits

Coordinating a set of running unikernels requires some means to communicate between them. For conventional VMs, all such communication passes via shared memory rings to real hardware running in a privileged VM. Device-specific RPC protocols are built over these rings to provide traditional abstractions such as `netfront` (network cards) or `blkfront` (mass storage). This is a convenient abstraction when virtualizing existing OS kernels to run under Xen, as each protocol fits into the existing device driver framework.

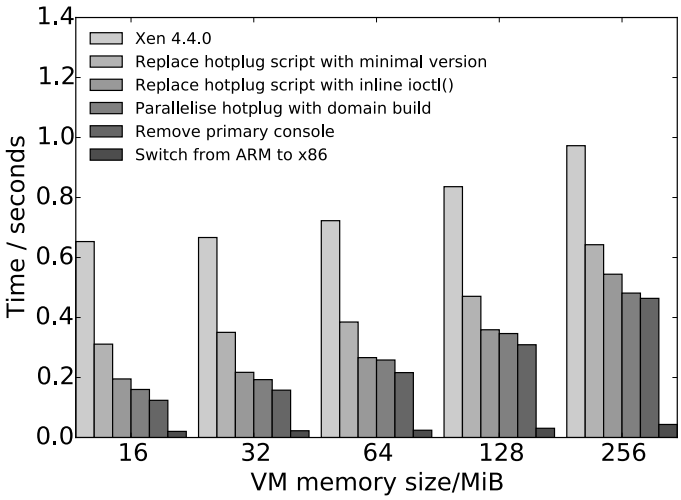



Figure 3.39: Optimizing Xen/ARM domain build times.

However, the lack of user/kernel space divide in a unikernel means that it links in device drivers as normal libraries: there is no need to fit the protocols into any existing abstraction. It becomes easy to construct custom RPC layers for communication between unikernels, whether instantiated as VMs on Xen or as Linux processes. Jitsu provides an abstraction over such a shared-memory communication protocol called *Conduit*, which *(i)* establishes shared-memory pages for zero-copy communication between peers; *(ii)* provides a rendezvous facility for VMs to discover named peers; and *(iii)* hooks into higher level name services like DNS. Conduit is designed to be compatible with the `vchan` library for interVM communication.

#### 3.8.5.3 Establishing a Fast Point-to-Point Connection

A `vchan` is a point-to-point link that uses Xen grant tables to map shared memory pages between two VMs, using Xen event channels to synchronize access to these pages. Establishing a `vchan` between two VMs requires each side to know its peer's domain id before the shared memory connection can be established. This allows vchan to work early in Xen's bootcycle before XenStore is available (*e.g.* within a disaggregated system). Unlike previous inter-VM communication proposals, `vchan` remains simple by not mandating any rendezvous mechanism across VMs, focusing solely on providing a fast shared memory datapath. Modern Linux kernels provide userspace access to grant mappings (`/dev/gntmap`) and event channels (`/dev/evtchn`), so we implemented the `vchan` protocol in pure OCaml using these devices. This required fixing several bugs in upstream Linux arising from the many ways to deadlock the system when interacting between user and kernel space. The lack of such a divide in unikernels made implementing this protocol for MirageOS far simpler. The resulting code allows unikernel and Linux VMs on the same host to communicate without the overhead of a local network bridge.

#### 3.8.5.4 Listening on Named Endpoints

For convenience, Conduit provides a higher-level rendezvous interface above `vchan` by using the existing XenStore metadata store. It extends the XenStore namespace in two places: the existing `/local/domain` tree for per-VM metadata, and a new `/conduit` tree for registering endpoint names and tracking established flows. Consider a XenStore fragment with an HTTP client VM connecting to a HTTP server. When the server VM boots, it registers a name mapping from its domain id to `/conduit/http_server`. It then watches the listen key for any incoming connections. The client VM similarly registers `/conduit/http_client` when it starts. The `http_client` picks a unique port name and attempts to "resolve" the `http_server` target by writing the port name to the listen queue for the server (*e.g.* `/conduit/http_server/listen/conn1`). The server VM receives a watch event and reads the remote domain id and port name from its listen queue, giving it sufficient information to establish a `vchan`. The connection metadata is written into `/local/domain/<domid>/vchan`, and contains the grant table and event channel references through which both sides obtain their shared memory pages and virtual inter-

rupts. The server also updates the /flows table with extra metadata such as per-flow statistics that can be read by management tools.

#### 3.8.5.5 Access Control and Transactions

XenStore already has an access control model that allows per-domain access control over keys and their child nodes. This is a good fit for Conduit except during initial setup where the client domain must write directly into the listen directory published by the server. Although the directory is open for writing from any other VM, new keys must be restricted to only be readable by the directory owner and the creator of the key. This is analogous to setting the setgid and sticky bits in POSIX filesystems. With this extension added to XenStore, domains cannot observe or interfere with the creation of conduits that do not concern them, and only XenStore itself is required for rendezvous.

As XenStore is already a filesystem-like interface, this protocol is similar to the Plan 9 network model, with a few notable differences: *(i)* although connection establishment goes through XenStore, established channels are zero-copy shared memory endpoints that no longer require any interaction with XenStore; and *(ii)* XenStore provides a transactional interface to let batch updates be committed atomically. This eliminates potential inconsistencies arising from having state metadata spread over several keys.

The Conduit interface enables us to write unikernel code without having to know in advance where the remote peer is running. For example, the http_server might be a Xen unikernel or a normal Linux guest VM listening from a userspace Unix binary. Unikernels also need not trust each other as they act as a distributed system on a single host, communicating via a bytestream rather than directly sharing pointers into each other's address spaces.

### 3.8.6 The Jitsu Directory Service

Our goal is to ensure that unikernels are launched and halted in real-time in response to network requests. This role is similar to that performed by inetd on Unix, and is fulfilled by the Jitsu Directory Service that maps external DNS requests onto unikernel instances. When the unikernel for a service has launched, it can serve as many requests as a single VM can handle – we typically launch a VM per registered service, not one per TCP connection.

A Jitsu VM is launched at boot time with access to the external network and handles name resolution, invoked either by a local unikernel over a conduit, or through DNS protocol handlers listening on the network bridge. In the former case, the Jitsu resolver is discovered via a well-known jitsud Conduit node, while in the latter it is discovered through the usual process in DNS (*e.g.* resolving ns.domain.name). If a name resolution request is received that maps onto a running unikernel, Jitsu just returns an appropriate IP address or vchan endpoint.

If the name requested does not correspond to a running unikernel, Jitsu launches the desired unikernel while simultaneously returning an appropriate endpoint (again, IP address or vchan) against which the client can start the higher level protocol interaction (*e.g.* a TCP threeway handshake). However, while the VM is starting it will not be ready to respond to network traffic as the network bridging subsystem connects asynchronously. This opens a race condition where the DNS response has been sent to the client, but the unikernel is not yet listening for the TCP SYN packet that will follow (likely very quickly as the client is typically local). The SYN packet is dropped, and the client retransmits after 1s – well outside our low-latency requirement.

#### 3.8.6.1 Connection Proxying via Synjitsu

We could remove this race condition by delaying the initial DNS response until the unikernel network is fully established. Instead we take advantage of the high-level libOS network stack available to us to provide a lower latency solution: we explicitly handle incoming connections in a proxy unikernel, and hand off the state to the full unikernel once it has finished plugging its network device in. This helps Jitsu to mask any latency associated with booting the target unikernel, as well as making it more robust in the face of TCP connections arriving unexpectedly outside of DNS resolution (*e.g.* because a client did not respect the TTL in a DNS response and attempted to connect to the service directly).

Figure 3.40 shows the packet flow with the synjitsu unikernel performing this connection proxying. When a DNS request comes in, the unikernel boot process starts, returning a DNS response as soon as the VM resource allocation is complete (resource exhaustion can thus be returned in the DNS response as a SERVFAIL to indicate the client should go elsewhere). As unikernel boot (20ms on x86, 350ms on ARM) takes longer than the RTT of a packet on a local network (5ms), it is likely that a TCP SYN would follow and be lost before

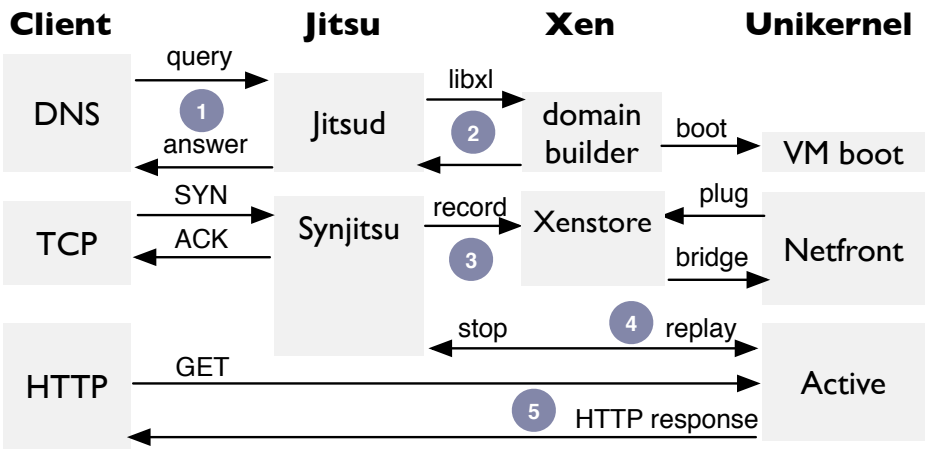



Figure 3.40: How Jitsu masks boot latency: ① A DNS request triggers the unikernel launch; ② Response sent when domain building completes but before networking is active; ③ TCP requests are buffered into XenStore until bridging is setup; and ④ the active unikernel replays the buffered connections before ⑤ directly serving traffic.

the unikernel has booted, triggering a slow TCP client retransmission.

`synjitsu`, built using the same OCaml TCP stack as the booting unikernel, removes this race entirely by listening on the external network bridge and an internal conduit for TCP packets destined for a unikernel that is still booting. When it receives a `SYN`, it writes entries into a special area in the conduit XenStore tree for the booting unikernel. Figure 3.41 shows two examples; *(i)* where a `SYN` has been received but not responded to, and *(ii)* where a `SYN_ACK` has been sent by the proxy and the TCP data stream buffered up. When the unikernel finishes booting and has an active network interface, it signals to `synjitsu` that it is ready for traffic via a two-phase commit in XenStore, ensuring only one of them ever handles any given packet. The unikernel then reconstructs the TCP state descriptors based on the recorded state, and handles subsequent traffic on the bridge directly, with no further interference from `synjitsu`.

Splitting state across a dormant kernel and a proxy is not a new technique, but the high-level nature of the OCaml TCP/IP stack makes implementation a simple matter of (de)serializing values across XenStore. As only one of `synjitsu` or the unikernel ever replies to a packet, we avoid the complexity and latency increase from building a distributed network stack within the host. It is also relatively easy to extend to higher-level protocols such as SSL/TLS, *e.g.* to perform the 7-way initial key exchange in one VM before it hands off the connection to another unikernel that has no access to the private keys for the remainder of its lifetime.

```
conduit
└── http_server = "3"
    └── tcpv4 ..... Connection state from proxy
        ├── 1 ..... Received SYN but not responded
        │   ├── state = "SYN"
        │   └── tcb = "(src ...)(port ...)"
        └── 2 ..... Established and buffering packets
            ├── state = "SYN_ACK"
            ├── tcb = "((port ...)(isn ...))"
            └── packets = "((data ...)(...))"
```

Figure 3.41: The `synjitsu` proxy registers embryonic TCP connections to mask unikernel startup time.

### 3.8.7 Service Configuration

Consider a client wishing to access one of a set of low-traffic websites, such as a set of personal homepages and photographs. Hosting each of these relatively low traffic sites in the cloud would be a waste of money, while a typical small home router or similar is unlikely to have sufficient resources to keep them all simultaneously live yet isolated. An ARM device using the Jitsu toolstack is registered in the public DNS as `ns.family.name`, the nameserver for the `family.name zone`. When a DNS request comes in for `alice.family.name`, Jitsu returns the local external IP address configured for Alice's unikernel and performs connection proxying while Alice's unikernel launches. Conventional failover models are supported – multiple ARM boards could be registered in the DNS and return `SERVFAIL` responses if they do not have resources to serve the traffic.

In our current implementation, the Jitsu services are statically configured via OCaml code to map their unikernel with an IP address, protocol and port. We expose publication of running services via the DNS, as either an authoritative server or recursive resolver. More dynamic configurations, where launched unikernels may themselves alter the name-address-unikernel mappings and can publish using, *e.g.* Dynamic DNS are possible to build over this lower-level interface.

### 3.8.8 Discussion

Jitsu solves the problems of supporting low-latency deployment of code requiring strong isolation to resource-constrained embedded platforms. Although we focus on use of MirageOS unikernels in that specific problem domain, the techniques embodied within Jitsu have a number of attendant benefits which we discuss here.

**General Jitsu**. Although we have focused here on using Jitsu with unikernels, we have not made any changes to the Xen guest ABI. As a result Jitsu works as described with legacy VMs (e.g., Linux, FreeBSD) on both ARM and in traditional x86 datacenter environments. This contrasts with systems such as ClickOS which modifies the ABI to achieve very dense, highly parallel deployments of 10,000s of VMs in an x86 64 datacenter. We anticipate that both of these approaches will converge in upstream Xen in the future through a revision of the XenStore protocol. The one thing that Jitsu cannot provide with legacy VMs is guaranteed latency, due to the inherent boot overheads of such VMs. Tests on x86 point to the intriguing possibility of very fast 20–30ms response times in datacenter environments as well.

Jitsu can easily be extended to support other VM life-cycle operations such as live relocation or VM forking in response to network requests. However, Jitsu is particularly well-suited to Xen/ARM through its use of explicit state transfer in `synjitsu` rather than depending on these hypervisor-level features. Forking or migrating entire VMs is more resource intensive than protocol state transfer, and is not yet fully supported by Xen/ARM 4.5. Simple TCP connection handover as in `synjitsu` is also easily extensible, and we are currently applying it to a full seven packet SSL/TLS handshake to support encrypted connections.

Finally, as noted previously, use of the Conduit stack for coordinating communication between VMs is not limited to unikernels. The basic principle of providing a name-based resolver to shared memory endpoints that does not depend on either a network (*e.g.* TCP/IP) or a process model (*e.g.* SysV `shmem`) can be used to interface conventional VM software stacks with unikernels.

**Modularity**. Jitsu both exploits and enables extensive use of modularity, which is very useful in building reliable distributed systems. The first form of modularity is found the way MirageOS is implemented – a set of lightweight OCaml libraries fulfilling module type signatures. Type-checking these signatures makes it easy to ensure that, when picking and choosing the features to be included in a particular unikernel, the basic system requirements are satisfied. As a result, developing features such as Conduit was far more straightforward than would have been for a traditional OS: during development it never crashed the MiniOS kernel, and almost every error was caught and turned into an explicit condition or a high-level OCaml exception. Similarly, the `synjitsu` proxy uses the same OCaml TCP/IP library as found in the unikernels, simply with very different runtime policies.

The new Conduit capability also directly addresses one of the key criticisms of the MirageOS approach: lack of multilingual support through the dependence on OCaml. With Jitsu – specifically the combination of Synjitsu and Conduit's low latency high throughput inter-VM communication – it is entirely feasible to launch a TCP/IP MirageOS unikernel that will proxy incoming traffic to another unikernel (*e.g.* in Ruby or PHP) that need only implement the Conduit protocol and so need not expose, or even include, a TCP/IP

implementation.

**Use cases**. We envisage Jitsu being useful in a wide range of situations. For example, where legacy software that may be difficult to upgrade (*e.g.* embedded device firmware) must be run, Jitsu can be used to provide a very narrow, application specific firewall that can filter and groom incoming traffic from the public Internet limiting the exposure of the legacy software. Another useful scenario would be to contain application code that would normally run as a cloud service so that it can be run on a platform, such as the home router, inside the home. For example, consider the latency sensitive applications noted earlier, Google Glass and Apple's Siri. By implementing the cloud services that support these applications as unikernels, they could be downloaded to run locally on the home router, providing significantly lower latency for common operations while still having the full power of the cloud at their disposal.

Yet other application scenarios include those where the data to be processed by the cloud-hosted service might be considered particularly personal, such as a family's photos. Photos might be hosted encrypted on the home router, and then unikernel versions of services such as Apple's iPhoto and Google's Picasa might be instantiated on-demand on the home router and given access to decryption keys held locally. Access to photos is then more directly controlled within the home without giving up all the personal data to the cloud providers

## 3.9 Minicache

### 3.9.1 Introduction

Nowadays, video streaming is the Internet's killer app, with reports stating that it will account for up to 90% of all consumer traffic by 2019. Most of this content is delivered via Content Delivery Networks (CDNs), a trend apparent in the fact that metro traffic will surpass long-haul traffic in 2015 [20].

CDNs do a great job of improving important user experience metrics such as buffering times, video quality and buffering ratios, but depending on features such as the region of the world that the client is in, time of the day, or the overall volume of requests (including overload conditions such as flash crowds), a particular CDN may not provide the best service possible. A recent measurement study of 200 million video sessions confirms this, stating that more than 20% of sessions had a re-buffering ratio greater than 10% and more than 14% had a startup greater than 10 seconds [49]. These amounts matter: in [27], the authors report that a 1% increase in buffering can reduce viewing time by more than 3 minutes. Another report declares that an increase of 100 ms in page load time decreased Amazon sales by 1%, and a 500 ms increase in Google search results reduced revenue by 20% [71].

These performance shortcomings have prompted a number of works on CDN multi-homing (*i.e.* using multiple CDNs in order to improve video QoE and reduce provisioning costs), including proposals for building CDN control planes [34, 49]. The authors of [49], in particular, point out that simply by choosing CDNs more intelligently, the re-buffering ratio can be reduced by a factor of two. Further work states that multi-homing can reduce publishing costs by up to 40% [48], and that federated CDNs would do so by 95% [14].

All of this work constitutes a big step in the right direction, showing, among other things, that the more choice in terms of delivery sites, and the more a CDN can dynamically re-act to changing loads, the better the quality of video delivery and the lower the costs. Thankfully, there are a number of trends (some recent, some less so) that can make a larger range of network sites available for video delivery:

- **Micro Datacenters.** A number of major network operators are deploying *micro-datacenters* (*e.g.* a rack of commodity servers) at Points-of-Presence (PoPs) in their networks, initially to run their own services but in the longer run to rent this infrastructure out to third parties [32].
- **Mobile Edge Computing.** A recent ETSI white paper [30] calls for the deployment of servers at the edge of the network, in RANs (Remote Access Networks) next to base stations and radio network controllers. Big players such as Intel are also getting in on the act, arguing for deployment of servers in so-called "smart cells" [39]. Along this trend, a survey of 2,000 telco industry professionals states that 78% of respondents consider video content streaming as one of the most lucrative LTE services [83].
- **Federated CDNs** aim to combine CDNs operated by various telecom operators to be able to compete with traditional CDN companies such as Akamai, Limelight or MaxCDN [13, 22, 73]. There are even brokers bringing together infrastructure providers and (virtual) CDN operators [60], and the same survey cited above reports that 50% of respondents stated that video delivery would be the first network function they would virtualize [83].

- **Public Clouds** can be leveraged as additional sites to deliver content from. Netflix, for instance, uses Amazon Web Services for both services and delivery of content [7].
- **Pay-as-you-go CDNs** like Amazon's CloudFront [6], Akamai's Aura [1] or Alcatel-Lucent's Velocix [86] can provide additional deployment sites.

Given this landscape, we ask a simple, key question: can such infrastructure be leveraged to improve the performance of video delivery, and if so, what would the quantitative effects of doing so be? Beyond this, we make the case for ephemeral CDNs: the ability to build large-scale, VoD (Video on Demand) and live streaming virtual content distribution networks on shared infrastructure *on-the-fly*. Such virtual CDNs (vCDNs) could be built to deliver a single piece of content like a live event or a VoD series episode, could take into account parameters such as users' geographical locations (*e.g.* ensuring that there's always a nearby cache nearby) and demand (*e.g.* scaling the number of caches in a particular high-load location), and could be quickly adapted as these vary throughout the lifetime of the stream.
We believe this would bring benefits to a number of players. End users would see improvements in video quality (*e.g.* better join times and buffering ratios); CDN operators could expand their service to cope with fluctuations in load and demand from specific regions by dynamically building a vCDN to complement their existing infrastructure; and network operators would derive additional revenue from acting as CDN operators and from renting out their infrastructure.
Towards the vision of ephemeral CDNs, we make a number of specific contributions:

- The development of a custom-built simulator to show the large scale effects that ephemeral CDNs could have if deployed.
- The implementation and evaluation of MiniCache, a virtualized, specialized, high performance content cache that runs on Xen and KVM. MiniCache contains a purpose-built, tiny HTTP server and filesystem, and runs on top of minimalistic operating systems such as OSv [45] and MiniOS [88]. MiniCache can be instantiated in just 23 ms on x86, ready to serve content from its cache.
- The porting of MiniCache to ARM, and an evaluation of it on a microserver (*i.e.* a single-board PC) well suited to edge/mobile deployments. On such a platform, MiniCache is able to boot in about 140 ms..
- The development of Tinyx, a minimalistic Linux kernel and distribution that results in a small ($<$ 2 MB compressed, $<$ 6 MB uncompressed), Tinyx-based MiniCache image. Further, since it is essentially Linux, Tinyx allows easier integration with existing content delivery applications.
- A number of optimizations to Xen including a fast I/O mechanism called *persistent grants* that does not require modifications to guests; improvements to MiniOS and lwIP; and a number of small binaries to speed up functions such as VM boot times.

**Cross-resource pooling relevance:** Like any CDN, MiniCache allows us to trade additional storage in terms of the content caches versus network bandwidth, *i.e.*, traffic can be served locally and so does not need to traverse core networks. The advantage of it is that it can do so *on-the-fly*, building virtual CDNs as and when they are needed.

### 3.9.2 System Requirements

The high-level goal is to be able to build on-the-fly, virtual CDNs on third-party, shared infrastructure for video streaming. In greater detail, we would like to quickly instantiate virtual content caches on servers (and microservers) distributed close to video consumers, in essence dynamically generating overlay multicast video streaming trees.
To achieve this, each cache should be able to download content from an origin server or an upstream cache, cache the content, and serve it to any number of clients or downstream caches. Note that the focus of this section is not on designing optimal algorithms to decide what the overall topology for all of these caches should be. Instead, in we carry out simulations that employ a number of simple topology and content cache placement strategies to quantify their effects on QoE metrics, and concentrate on the design, implementation and optimization of MiniCache (a few simulation results can be found in Deliverable 3.3). Before going into these details, however, we first define a number of basic requirements.

**Wide deployment**: MiniCache strives to leverage as many deployment sites and infrastructure as possible by running on a number of different platforms (from powerful x86 servers all the way down to

| Requirement | How Addressed |
|---|---|
| Wide deployment | Support for x86, ARM, Xen, KVM, MiniOS, OSv, Tinyx |
| Strong isolation | Use of hypervisor technologies. |
| Fast scaling | Optimized boot ($<$ 100 ms) boot/destroy times. |
| High consolidation | Specialization allows running of hundreds of concurrent instances. |
| High performance | Optimized packet I/O, file access, network and block drivers. |
| Support kernel/stack optimization | MiniCache's upper layers can run on different kernels (in our case OSv, MiniOS and Linux/Tinyx). |
| Reliability | MiniCache can detect a crash and restart a VM in $<$ 100 ms. |

Table 3.3: Requirements in support of ephemeral CDNs and how MiniCache addresses each of them.

resource-constrained, small-sized, ARM-based microservers), guest operating systems (MiniOS, OSv and Linux/Tinyx) and virtualization technologies (Xen and KVM).

**Strong isolation**: Each server will be potentially multi-tenant, running concurrent instances of content caches belonging to different commercial entities. Isolation is thus needed to ensure the integrity of applications, system code and client data.

**Fast scale out/in**: The system should be able to scale to quickly build vCDNs according to demand, and to adapt to changing conditions (*e.g.* flash crowds as happened with the release of Apple's iOS 8 [74], a sudden increase in demand from a particular region as in local sport events [14], or a high decay rate in required volume when interest in content such as news stories wanes [14]). Because it is a small, custom-built content cache, MiniCache can be instantiated in as little as 23 ms on Xen on x86, 140 ms on Xen on ARM32, and in 400 ms on OSv/KVM. Tearing down a Xen-based instance takes 5 ms on x86 and 26 ms on ARM32.

**High consolidation**: The more MiniCache instances we can concurrently fit on a given server the better the system will be for the infrastructure provider's bottom line. Because MiniCache is specialized and based on minimalistic OSes it has a small memory footprint (32 MBs), meaning that a server with a relatively small amount of memory (*e.g.* 16GB) can support hundreds of instances. Clearly memory is not the only requirement; in Deliverable 3.3 we evaluate other metrics such as throughput and number of requests in the presence of an increasing number of instances.

**High performance**: CDN caches have to cope with large requests per second rates, many simultaneous connections, and high cumulative throughput, especially for high definition video such as Netflix's 6Mb/s SuperHD streams [33]. We optimize various portions of MiniCache and its underlying systems in order to match these requirements.

**Support for kernel and network stack optimization**: To achieve high throughput between caches (and between caches and origin servers), many of the major CDN operators use optimized transport stacks and/or kernel settings. For example, Netflix uses optimized TCP settings, an I/O model that reduces the number of system calls (sendfile), and support for in-kernel TLS[79]. Similarly, Akamai uses custom TCP connection control algorithms [58]. These optimizations, among other reasons explained later in this section, made us opt for virtual machines as opposed to other technologies such as containers.

**Reliability**: Finally, it would be ideal if the stream delivery infrastructure could cope with a content cache crash without it affecting end users. In MiniCache we optimize a mechanism to detect VM crashes and quickly and automatically re-launch an instance. We evaluate this mechanism in Deliverable 3.3.

Table 3.3 gives an overview of how MiniCache addresses each of these requirements. In the next section we outline MiniCache's architecture and implementation in detail.

### 3.9.3 Architecture and Implementation

One of the basic requirements for MiniCache is to be able to provide strong isolation to support multi-tenancy on third-party infrastructure. This naturally points to a virtualization technology, but which one should be used? We rule out containers because (1) the number of security issues [28] related to the fact that they put the entire kernel in the trusted computing base make them less than ideal for multi-tenant environments and (2) they violate our requirement to support customization of kernels and network stacks. As a result, we opt for hypervisor-based solutions (Xen and KVM).

The reader may ask whether, because of this choice, we are sacrificing performance. We note that the op-

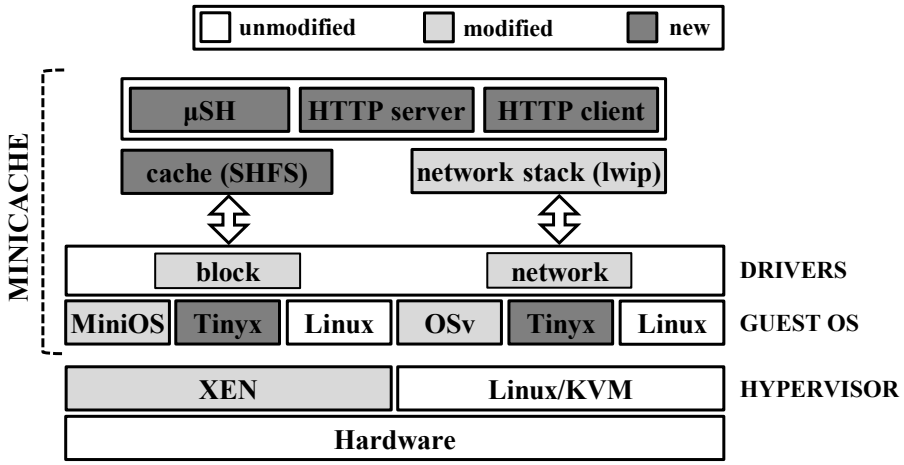



Figure 3.42: MiniCache architecture showing components developed from scratch, modified/optimized ones, and unmodified ones.

timizations we will present throughout this section allow our specialized MiniCache instances to perform on-par or better than containers (*e.g.* the work in [50] reports Docker container boot times of 1.1 seconds on an ARM board, considerably higher than specialized VMs). In a sense, we can have the best from both worlds: we can take advantage of the strong isolation warranted by hypervisor-based virtualization technologies, while having the performance more commonly found in lighter virtualization mechanisms.

#### 3.9.3.1 Overall Architecture

At a high level, MiniCache consists of content cache code (*e.g.* an HTTP server, or a filesystem) on top of a minimalistic operating system. We choose this type of OS since it provides a number of advantages. First, it is a single process with a single address space, meaning that there is no kernel/user-space divide, and so no expensive system calls. Second, it typically uses a co-operative scheduler, avoiding context switch overheads. Finally, since it is minimalistic, providing the basis for building MiniCache instances that can be spun up quickly (as we show in Deliverable 3.3 in as little as 23 ms) and that have a low memory footprint (*e.g.* 23 MB). Thanks to these characteristics and coupled with a number of optimizations described later on, we are able to meet the high consolidation, high performance and fast scaling requirements previously outlined.

Figure 3.42 shows MiniCache's overall architecture, including which components we developed from scratch and which we modified (we will give more details on this throughout this section). In order to meet the requirement of supporting wide deployment, MiniCache can run on both Xen and KVM, two of the most widely-deployed virtualization technologies. On Xen, we leverage MiniOS [88], a minimalistic, paravirtualized OS distributed with the Xen sources. MiniOS has a single address space, a cooperative scheduler and supports multi-threading (but not SMP, nor processes). Further, it presents a POSIX-like API, which, coupled with the fact that we compile it with `newlibc` and lwIP (a small, open source TCP/IP stack for embedded systems), allows applications written in C/C++ to be compiled into a single VM image relatively effortlessly.

MiniOS does not support KVM, and since it is paravirtualized, it would require a non-trivial amount of effort to port it. Instead, we opt for OSv [45], another minimalistic OS that runs on KVM [11]. OSv also uses a single address space and single process, supports SMP and most of the Linux ABI, and building an application on top of it is as easy as compiling it as a shared library.

Beyond these minimalistic OSes, and in further support of wide deployment, we introduce Tinyx. Tinyx consists of a stripped down Linux kernel (1.4MB compressed, 6.7MB uncompressed on Xen and 1.6MB and 7.3MB on KVM) along with a minimalistic distribution that has essentially a single process for the application running (in our case MiniCache), and little else (e.g., `init` and a shell). Compiled with only BusyBox (i.e., no MiniCache), Tinyx has a size of 2.4MB compressed and 9.2MB decompressed.

[11] OSv also supports Xen, although in the slower, HVM mode.

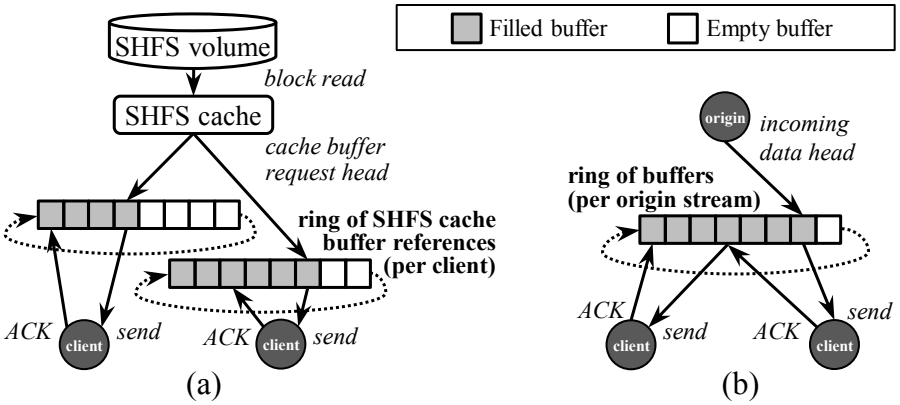



Figure 3.43: MiniCache HTTP server architecture. The server can serve content from local storage in which case it uses one ring per HTTP client (a). If serving from an upstream source (b), it clones the stream to multiple clients using one ring per stream.

#### 3.9.3.2 Cache Node Components

In terms of the content cache code (*i.e.* the application), MiniCache has a number of components. First, the actual cache is based on SHFS, a purpose-built, hash-based filesystem optimized for performing look-ups of content objects based on their IDs. SHFS has a flat hierarchy (no folders) and is organized into multi-block, 4-32 KiB areas called chunks, configurable at format time. A hash digest (SHA, MD5, or a function provided by the user) is used as the file's name, and the filesystem's meta data is loaded into memory at mount time. In addition, SHFS supports concurrent access from multiple readers (*i.e.* MiniCache instances). Further, SHFS includes a block cache that helps speed up accesses; in Deliverable 3.3 we show that SHFS can perform 23.6 file open/close operations per second. Finally, SHFS includes support for keeping per-file content cache statistics such as hit/miss counts, last access timestamps and download progress as a percentage of total file size.

The second component is the MiniCache HTTP server, which is implemented on top of SHFS' block cache, leverages lwIP's callback-based RAW API and uses the Joyent HTTP parser [12]. When a client sends an HTTP request, the server calculates the hash digest of the requested object and uses this ID to perform an SHFS file open (essentially a table lookup). If an object exists, the server next checks if its type is "file" or "link". For the former, this means that the object can be found locally, and so a read request is issued to the SHFS block cache, which may, in turn, have to perform a read from a physical storage device (Figure 3.43). For the latter, a TCP connection is set-up to an upstream node (another cache or origin server) and the object is streamed down.

Stream cloning (copying an incoming stream into multiple outgoing ones) is supported by using a common ring buffer per object/stream, and keeping two pointers per client, one to the last buffer ACKed and another one to the next buffer to be sent (Figure 3.43). In addition, it is worth noting that the HTTP server supports zero-copy from SHFS cache buffers all the way to the sending of TCP packets by having lwIP packet buffers directly reference SHFS cache buffers.

The third component is a stripped-down HTTP client that is used to download content from upstream caches and/or origin servers. Finally, we also include $\mu$SH, a simple shell that allows the CDN operator to control the cache node (*e.g.* to insert and delete objects and to retrieve statistics about them).

Taken together, these tailored components constitute 9,760 LoC (SHFS is 3,757, the HTTP server and client code 4,811 and $\mu$SH 1,192) and allow us to meet the high performance and high consolidation requirements described earlier.

#### 3.9.3.3 ARM Port

Video content is increasingly being accessed from mobile devices so that, as mentioned in the introduction, it would be ideal to be able to deploy content caches in edge/RAN networks, potentially at sites that have space and/or power constraints. As described in the Jitsu section, there has been an emergence of low-power ARM based computing platforms that are resource constrained, power-efficient and cheap. To this end, we

[12] https://github.com/joyent/http-parser

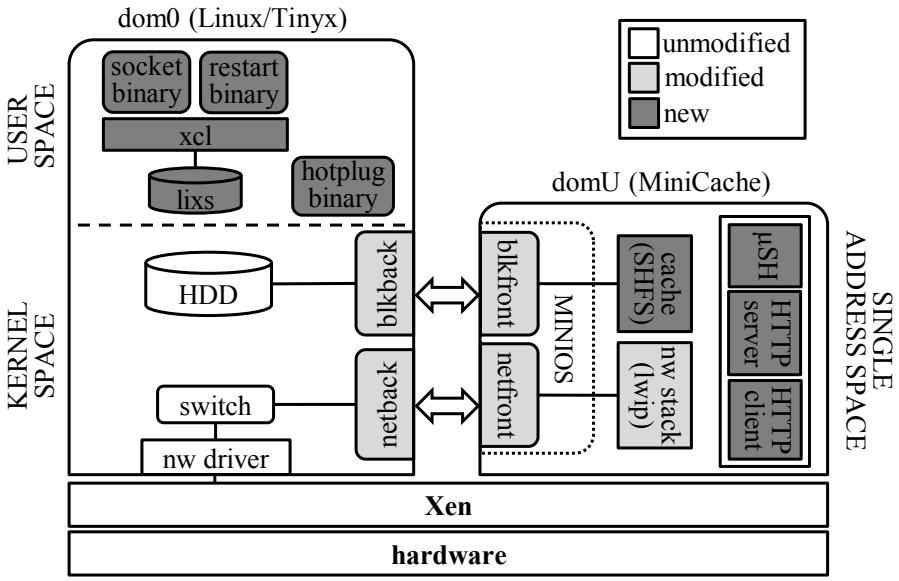



Figure 3.44: MiniCache Architecture on Xen.

leverage *microservers* (*i.e.* single-board PCs such as the Raspberry Pi), since their small physical size, low cost (typically $50-$200) and low energy consumption (many of them do not even have active cooling) make them ideally suited for our purposes.

We port MiniCache to the ARM architecture (on Xen), and in particular to the Cubietruck (ARM Cortex A7 dual core CPU, 2GB DDR3 RAM and 1Gb Ethernet, $100), an attractive offering in terms of CPU power, cost and power consumption. While we settled on the Cubietruck, in principle MiniCache could run on a number of other ARM-based platforms such as the Raspberry Pi 2 and the Odroid-XU3 (and of course x86 ones like the Intel NUC or the Gizmo 2).

In greater detail, we first took the MiniOS ARM port described in [50]. To get MiniCache to compile on MiniOS, we modified a number of compilation flags and had to fix types definitions. Additionally, we added the ability to receive boot arguments from the DTB (Device Tree Binary), a mechanism we use for setting a MiniCache VM's IP address and instructing it to mount an SHFS volume, among other things.

Finally, because this platform is CPU and memory-constrained, we further built a Xen dom0 (the privileged VM used to manage Xen) on Tinyx. In Deliverable 3.3 we evaluate the difference between running a full-fledged Linux distribution as dom0 versus using our Tinyx-based one.

#### 3.9.3.4 Xen, MiniOS and lwIP Optimizations

Towards achieving high consolidation, fast scaling (*i.e.* fast instantiation and destroy times) and high performance (*e.g.* high throughput, high requests/sec rates) we carry out a number of optimizations to Xen, MiniOS, and lwIP.

As brief background, a typical Xen deployment consists of the hypervisor and a privileged virtual machine called domain0 or dom0 for short (see Figure 3.44). In essence, dom0 is used as a management domain (*e.g.* to create and destroy VMs) and as a driver domain, meaning that it contains device drivers and the back-end software switch used to connect network devices to VMs. Further, Xen uses a split-driver model: a back-end driver (netback) runs in the driver domain, and the other domains/guests implement a hardware-agnostic front-end driver (netfront). On the control side, Xen relies on the XenStore, a `proc`-like database containing information about VMs such as virtual interfaces and which CPUs they run on; and the toolstack, which along with the `xl` command-line tool, provide users with a control interface.

We optimize a number of these components (shown in Figure 3.44), as well as the OS and network stack that MiniCache uses:

**Persistent Grants**: Xen uses *grants* to allow inter-VM memory sharing, a mechanism used for, among other things, sharing buffer rings to perform network and block I/O between a guest and the driver domain. By default, a grant is requested, mapped and unmapped for every transaction, an expensive operation which

requires a hypercall (essentially a system call to the hypervisor) and causes a TLB shootdown in the unmap operation which heavily decreases throughput. We implement *persistent grants* in the network drivers (*i.e.* `netback`, and Linux's and MiniOS' `netfront`) as well as the MiniOS block frontend driver (*i.e.*, `blkfront`). These modifications allow us to push throughput to as high as 40 Gb/s (Rx) and 32 Gb/s (Tx). We have submitted the implementation of persistent grants to the Linux kernel (still under review[13]).

**MiniOS**: Our MiniOS netfront driver's implementation of `select/poll` uses a busy-poll approach which is inefficient in terms of CPU usage. Instead, we improve `select/poll` by switching to a sleep model, and by polling only on active devices, rather than adding all MiniOS devices to `select`'s read/write file descriptors. These changes improve CPU usage when working at high throughput rates. Furthermore, we port DPDK's implementation of the `memcpy` function, which uses SSE4/AVX instructions, to MiniOS. We show evaluations results for these modifications in Deliverable 3.3.

**GSO/GRO and checksum offloading**: Modern NICs come with a number of features that can significantly improve throughput and in particular TCP throughput. We modify the MiniOS netfront driver to support TCP checksum offloading, Generic Segmentation Offloading (GSO, for transmit), and Generic Receive Offload (GRO, for receive). GSO also required changes to lwIP's TCP write function to prevent the stack from splitting segments by the MTU.

**Control and hotplug binaries**: We introduce three binaries to further optimize MiniCache on Xen: (1) the hotplug binary replaces the script in charge of attaching virtual interfaces to the back-end software switch; (2) the socket binary allows fast MiniCache instantiation and tear-down from a remote host; and (3) the restart binary monitors MiniCache instances, detects crashes, and quickly re-starts them.

**Toolstack and XenStore**: We leverage both the optimized Xen toolstack described in [51] which minimizes the number of per-guest XenStore entries, and `lixs`, the minimalistic XenStore also presented in that work.

#### 3.9.3.5 KVM/OSv Port

MiniCache on OSv is a prototypical port of MiniCache for KVM, meaning that it does not yet incorporate the optimizations features found in the Xen/MiniOS VM (*e.g.* no TSO nor checksum offloading). MiniCache is built as a `.so` library that OSv executes in its single memory address space. We replace bindings to MiniOS calls with POSIX/OSv equivalents (*e.g.*, we replace `now()` with `gettimeofday()` and semaphore operations such as `up`/`down`/`trydown` with `sem_post`/`sem_wait`/`sem_trywait`). We also include lwIP configured to act as an IP/TCP stack, meaning that the OSv stack handles Layer-2 (*e.g.*, ARP resolution) and lwIP Layers 3-4. Finally, since MiniCache has its own block caching mechanism, we modify SHFS to directly submit block requests to OSv's BIOS layer, thus bypassing OSv's functionality.

[13] `http://lists.xenproject.org/archives/html/xen-devel/2015-05/msg01498.html`

# 4 Conclusions

This deliverable has outlined the challenges of performing multi-metric optimization to obtain liquidity across Network, Processing and Storage resources in TRILOGY2. Because these resources are in tradeoff, we have presented solutions that usually target one or two resources to be optimized. Nevertheless, cross-liquidity mechanisms presented in this deliverable, and in the past as part of the TRILGOY2 architecture (Figure 1.1), have demonstrated substantial gains through thorough evaluation. For example, solutions like vCPE and Minicache, *etc.* have highlighted the necessity of moving from operator controlled functions in the core network to the utilization of resources at the Edge, leading to a reduction of latency and operator costs. Jitsu and Minicache, on the other hand, avoid the overhead of heavy-weight VMs and traditional hypervisor architectures. With regards to the three liquid resources, these traditional hypervisors lead to the unnecessary consumption of the third tertiary resource and an over-use of the other two resources. More light-weight platforms that present the resources in a much more efficient manner form a crucial building block for enabling Liquidity.

In essence, the gains we describe indicate that ensuring a global liquidity across more than two resource types is in fact not necessary. The components of the TRILOGY2 Liquid Net have established cross-resource, cross-layer and cross-provider liquidity which have yielded pronounced performance benefits, which support a much more Liquid future of the Internet.

# Bibliography


[1] Akamai. Aura Licensed CDN. http://www.akamai.com/html/solutions/aura_licensed_cdn.html, 2013.

[2] Akka. http://akka.io/, accessed Sept 25, 2014.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proc. SIGCOMM 2010*, 2010.

[4] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proc. NSDI'10*.

[5] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, , and Murari Sridharan. Data center TCP (DCTCP). In *Proc. SIGCOMM 2010*, 2010.

[6] Amazon. Amazon CloudFront. http://aws.amazon.com/cloudfront/, June 2015.

[7] Amazon. AWS Case Study: Netflix. http://aws.amazon.com/solutions/case-studies/netflix/, May 2015.

[8] Amazon ELB. http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/TerminologyandKeyConcepts.html, accessed Sept 24, 2014.

[9] Apache Cassandra. http://cassandra.apache.org/, accessed June 10, 2013.

[10] Apache Cassandra Use Cases. http://planetcassandra.org/apache-cassandra-use-cases/, accessed Sept 25, 2014.

[11] Gonzalo R Arce. *Nonlinear Signal Processing: A Statistical Approach*. Wiley, 2004.

[12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *SIGMETRICS*, 2012.

[13] B. Niven-Jenkins and F. Le Faucheur and N. Bitar. Content Distribution Network Interconnection (CDNI) Problem Statement. https://tools.ietf.org/html/draft-ietf-cdni-problem-statement-08, June 2012.

[14] Athula Balachandran, Vyas Sekar, Aditya Akella, and Srinivasan Seshan. Analyzing the potential benefits of cdn augmentation strategies for internet video workloads. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 43–56, New York, NY, USA, 2013. ACM.

[15] Carlos F Bispo. The single-server scheduling problem with convex costs. *Queueing Systems*, 73(3), 2013.

[16] Jake Brutlag. Speed Matters, accessed Sept 24, 2014. http://googleresearch.blogspot.com/2009/06/speed-matters.html.

[17] Cassandra Documentation. http://www.datastax.com/documentation/cassandra/2.0, accessed Sept 25, 2014.

[18] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.

[19] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Catch the whole lot in an action: Rapid precise packet loss notification in data centers. In *Proc. NSDI'14*.

[20] Cisco Systems. Cisco Visual Networking Index: Forecast and Methodology, 2014-2019. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html, 2015.

[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[22] D. Rayburn. Telcos And Carriers Forming New Federated CDN Group Called OCX . http://goo.gl/abB9hQ, June 2011.

[23] DB-Engines Ranking of Wide Column Stores. http://db-engines.com/en/ranking/wide+column+store, accessed Sept 25, 2014.

[24] Jeffrey Dean and Luiz Andr Barroso. The Tail At Scale. *Communications of the ACM*, 56:74–80, 2013.

[25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[26] A Dixit, P Prakash, Y.C. Hu, and R.R Kompella. On the impact of packet spraying in data center networks. In *Proc. IEEE INFOCOM 2013*, 2013.

[27] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. *SIGCOMM Comput. Commun. Rev.*, 41(4):362–373, August 2011.

[28] Docker Inc. Docker Security. https://docs.docker.com/articles/security/, June 2015.

[29] Jonathan Ellis. How not to benchmark Cassandra: a case study, 2014. http://www.datastax.com/dev/blog/how-not-to-benchmark-cassandra-a-case-study.

[30] ETSI Portal. Mobile-Edge Computing - Introductory Technical White Paper. https://portal.etsi.org/Portals/0/TBpages/MEC/Docs/Mobile-edge_Computing_-_Introductory_Technical_White_Paper_V1%2018-09-14.pdf, September 2014.

[31] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *EuroSys*, 2012.

[32] Benjamin Frank, Ingmar Poese, Yin Lin, Georgios Smaragdakis, Anja Feldmann, Bruce Maggs, Jannis Rake, Steve Uhlig, and Rick Weber. Pushing cdn-isp collaboration to the limit. *SIGCOMM Comput. Commun. Rev.*, 43(3):34–44, July 2013.

[33] Frost and Sullivan. Netflix Doubles Video Quality Making 6Mbps SuperHD Streams Available To Everyone. http://www.frost.com/reg/blog-display.do?id=3100186, 2013.

[34] Aditya Ganjam, Faisal Siddiqui, Jibin Zhan, Xi Liu, Ion Stoica, Junchen Jiang, Vyas Sekar, and Hui Zhang. C3: Internet-scale control plane for video quality optimization. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 131–144, Oakland, CA, May 2015. USENIX Association.

[35] W. D. Gray and D. Boehm-Davis. Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior. *Journal of Experimental Psychology: Applied*, 6, 2000.

[36] Albert Greenberg el al. VL2: a scalable and flexible data center network. In *Proc. ACM Sigcomm 2009*, 2009.

[37] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: A high performance, server-centric network architecture for modular data centers. In *Proc. SIGCOMM '09*.

[38] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Oper. Syst. Rev.*, 42(5), 2008.

[39] Intel. Smart cells revolutionize service delivery. `http://www.intel.de/content/dam/www/public/us/en/documents/white-papers/smart-cells-revolutionize-service-delivery.pdf`.

[40] DPDK: Dataplane Development Kit. `www.dpdk.org`, nov 2015.

[41] Van Jacobson and Michael J. Karels. Congestion avoidance and control. In *In Proceedings of the Sigcomm '88 Symposium*, Stanford, CA, August 1988.

[42] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up Distributed Request-Response Workflows. In *SIGCOMM*, 2013.

[43] Christos Kalantzis. Revisiting 1 Million Writes per second, 2014. `http://techblog.netflix.com/2014/07/revisiting-1-million-writes-per-second.html`.

[44] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A. Kim. Measuring Interference Between Live Datacenter Applications. In *SC*, 2012.

[45] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association.

[46] Jialin Li, Naveen Kr Sharma, Dan R K Ports, and Steven D Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *SoCC*, 2014.

[47] Axel Liljencrantz. How Not to Use Cassandra, talk at Cassandra Summit, 2013. `https://www.youtube.com/watch?v=0u-EKJBPrj8`.

[48] Hongqiang Harry Liu, Ye Wang, Yang Richard Yang, Hao Wang, and Chen Tian. Optimizing cost and performance for content multihoming. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 371–382, New York, NY, USA, 2012. ACM.

[49] Xi Liu, Florin Dobrian, Henry Milner, Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. A case for a coordinated internet video control plane. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 359–370, New York, NY, USA, 2012. ACM.

[50] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie. Jitsu: Just-In-Time Summoning of Unikernels. In *NSDI*, 2015.

[51] F. Manco, J. Martins, K. Yasukata, S. Kuenzer, and F. Huici. The case for the superfluid cloud. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Cloud Computing (to appear)*, HotCloud '15. ACM, 2015.

[52] Silvano Martello and Paolo Toth. *Knapsack problems: Algorithms and computer interpretations.* Wiley-Interscience, 1990.

[53] Michael Mitzenmacher. How Useful Is Old Information? *IEEE Trans. Parallel Distrib. Syst.*, 11(1), January 2000.

[54] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. *SIGOPS Oper. Syst. Rev.*, 33(5):217–231, December 1999.

[55] OpenStack Neutron. lhttps://wiki.openstack.org/wiki/Neutron.

[56] Nginx. http://nginx.org/en/docs/http/load_balancing.html, accessed Sept 24, 2014.

[57] OpenStack Nova. http://docs.openstack.org/developer/nova/.

[58] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, August 2010.

[59] OpenDaylight SDN platform. ,http://www.opendaylight.org/.

[60] OnApp. OnApp CDN: Build your own content delivery network. http://onapp.com/platform/onapp-cdn, May 2015.

[61] OpenBaton Network Function Virtualization Orchestrator (NFVO). http://openbaton.github.io/.

[62] OpenMano - Open NFV MANO. https://github.com/nfvlabs/openmano.

[63] OpenStack Cloud software. http://www.openstack.org/.

[64] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *SOSP*, 2013.

[65] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 207–218, New York, NY, USA, 2013. ACM.

[66] POX SDN controller. https://github.com/noxrepo/pox.

[67] Quagga Routing Suite. http://www.nongnu.org/quagga/.

[68] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with Multipath TCP. In *Proc. SIGCOMM '11*.

[69] Riak. Load Balancing and Proxy Configuration, accessed Sept 24, 2014. http://docs.basho.com/riak/1.4.0/cookbooks/Load-Balancing-and-Proxy-Configuration/.

[70] Luiggi Rizzo. The netmap project. http://info.iet.unipi.it/~luigi/netmap/, aug 2015.

[71] Ron Kohavi and Roger Longbotham. Online Experiments: Lessons Learned. http://www.exp-platform.com/Documents/IEEEComputer2007OnlineExperiments.pdf, 2007.

[72] Mema Roussopoulos and Mary Baker. Practical Load Balancing for Content Requests in Peer-to-Peer Networks. *Distributed Computing*, 18(6), 2006.

[73] S. Puopolo and M. Latouche and F. Le Faucheur and and J. Defour. Content Delivery Network (CDN) Federations. https://www.cisco.com/web/about/ac79/docs/sp/CDN-PoV_IBSG.pdf, October 2011.

[74] Sandvine Inc. Sandvine global internet phenomena report 2H 2014. https://www.sandvine.com/downloads/general/global-internet-phenomena/2014/2h-2014-global-internet-phenomena-report.pdf, 2014.

[75] Salvatore Sanfilippo. Redis latency spikes and the 99th percentile, 2014. http://antirez.com/news/83.

[76] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *Proc. NSDI'12*.

[77] Steve Souders. Velocity and the Bottom Line, 2009. http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html.

[78] C. Stewart, A. Chakrabarti, and R. Griffith. Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs. In *USENIX ICAC*, 2013.

[79] Randall Stewart, John-Mark Gurney, and Scott Long. Optimizing TLS for High-Bandwidth Applications in FreeBSD. https://people.freebsd.org/~rrs/asiabsd_2015_tls.pdf, April 2015.

[80] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[81] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving Large-scale Batch Computed Data with Project Voldemort. In *FAST*, 2012.

[82] OpenStack Tacker - Open NFV Orchestrator. https://wiki.openstack.org/wiki/Tacker.

[83] Telecoms.com. Telecoms.com intelligence annual industry survey 2015. http://telecoms.com/intelligence/telecoms-com-annual-industry-survey-2015/.

[84] Jan A van Mieghem. Dynamic Scheduling with Convex Delay Costs: The Generalized $c|mu$ Rule. *The Annals of Applied Probability*, 5, 1995.

[85] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *Proc. SIGCOMM '09*.

[86] Velocix. CDN Federation/Interconnect. http://www.velocix.com/vx-portfolio/solutions/cdn-federation-interconnect, 2015.

[87] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low Latency via Redundancy. In *CoNEXT*, 2013.

[88] Xen.org. Mini-OS. http://wiki.xen.org/wiki/Mini-OS, 2015.

[89] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *NSDI*, 2013.

[90] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI$^2$: CPU performance isolation for shared compute clusters. In *EuroSys*, 2013.

[91] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *SoCC*, 2014.