ICT-317756

TRILOGY2

Trilogy 2: Building the Liquid Net

Specific Targeted Research Project

FP7 ICT Objective 1.1 – The Network of the Future

D1.3 Advanced Cross Liquidity Tools

Due date of deliverable: 1 January 2015

Actual submission date: 15 January 2015

Start date of project	1 January 2012
Duration	36 months
Lead contractor for this deliverable	Intel
Version	v1.0, January 15, 2015
Confidentiality status	Public

Abstract

Trilogy 1 successfully defined a set of mechanisms to provide optimal fault tolerant transport across an IP network by creating and exploiting multiple simultaneous path across the network. Trilogy 2 sets out to extend the results of Trilogy 1 to be cross-resource, cross-layer, and cross-provider. Cross-resource seeks a generalization to all IT resources including processing and storage. This document describes more advanced liquidity tools, focusing on enabling liquidity across providers. It focuses on the design, implementation and evaluation of solutions that allow pooling different resource types across different providers.

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	D1.3 Advanced Cross Liquidity Tools
Editor	George Milescu, Intel
Project Co-ordinator	Marcelo Bagnulo Braun, UC3M
Copyright notice	© 2015 Participants in project TRILOGY2

Executive Summary

As the Internet started to connect different systems together, it became a bridge where storage, processing, bandwidth and energy were shared for services worldwide. These resources, however, were stand-alone islands that could not be grouped together but accessed separately each within its own domain.

Trilogy 2 aims to create a framework capable of orchestrating, provisioning, and controlling the usage of heterogeneous resource pools as demanded by the applications. Within the project, WP1 develops the set of tools that the framework is based on. The focus of this document is on the advanced cross-liquidity mechanisms that have been implemented in the second year of the project in the context of WP1 activities. Work for D1.3 continued on the basis set during the first year of the project, and described in D1.2 - Initial cross-liquidity tools.

The target of this deliverable is to describe the above mentioned advanced cross-liquidity tools and the way they interconnect different resources across providers. We group our descriptions into cross-resource, cross-layer and cross-provider tools.

The cross-resource tools provide mechanisms to allow resource trading to happen. Resources can be exchanged between entities by cross-trading, allowing a balance to be obtained between the resources that are spent to gain access to more valuable resources. As an example, in vM3 bandwidth is exchanged to access CPU and storage. Storage liquidity tools make use of both network and CPU to facilitate accessing the storage resources. These tools are enhanced with the use of Mirage, a unikernel system, and Irminsule, a distributed database designed for static type-safety for security and reliability. Trevi, based on fountain coding, allows the design of a storage service that is tolerant to packet loss, but without retransmissions or timeouts. For cloud deployments, Federated Block Storage offers a wide-area replication architecture. In terms of NFV, the Customer Premises Equipment (CPE) is emerging as one of the primary applications of the NFV architecture. Sharing resources between the operator data center and the edge equipment brings extra flexibility in designing infrastructure architectures and efficiently exploiting available resources.

The cross-layer tools open access to resources controlled at different levels in the system or network architecture. HACK [34] performs a cross-layer optimization of TCP and WiFi acknowledgements, by combining them and significantly improving the medium utilization. Multi WiFi takes the handoff problem from layer 2 and brings it to layer 4, presenting itself as a very good solution for device mobility. The concept of Polyversal TCP (PVTCP) is improved by adding a security mechanism for key exchange that is used on cross-layer encryption. In datacenters, Multi-path TCP (MPTCP) [11] is a valuable tool for exploiting all available network interfaces, and improving the network overlay between nodes. GRIN [1] presents such an architecture that covers both layer 2 and layer 3 in the networking stack.

Cross-provider liquidity orchestrates the resources coming from different providers or resource owners. In this context, the Federated Market is a management system permitting different cloud systems to share resources among them. In terms of bandwidth liquidity, MPTCP has proven to be a flexible tool for accessing

bandwidth across network providers. In addition to the basic MPTCP mechanisms, this deliverable contains tools for enabling applications to control the MPTCP sub-paths and react to network events by implementing their own Path Management.

List of Authors

Authors	Pedro Aranda, Marcelo Bagnulo, Giacomo Bernini, Olivier Bonaventure, Julian Chesterfield, Jaime Garcia-Reinoso, Felipe Huici, Fabien Duchêne, Benjamin Hesmans, Toby Moncaster, George Milescu, Cătălin Moraru, Valentin Ilie, Dragoș Niculescu, Costin Raiciu, John Thomson, Lynne Salameh
Participants	Intel, NEC, NXW, OnApp, TID, UC3M, UCAM, UCL-BE, UPB, UCL-UK
Work-package	WP1
Security	Public (PU)
Nature	R
Version	1.0
Total number of pages	95

Contents

List of Figures

1 Introduction

1.1 Deliverable context

The development of the Internet led to a new set of computing paradigms where resources were shared between devices and were used to provide the services requested by the clients. In this design resource sharing is limited to stand-alone islands of compute, storage and bandwidth with boundaries shaping the separate usage of each resource. Technologies existing today efficiently exploit hardware resources but maintain a separation between resource types and architecture layers. As a very simple example, bandwidth available on a network interface cannot be easily used to compensate for an overloaded second interface. At the same time, the bandwidth available from separate network providers can not easily be aggregated together and is seen as two separate resources.

The aim of Trilogy 2 is to provide tools and to create a framework that takes individual resources and puts them together in a pool, facilitating resource allocation towards application demand and making the resources liquid. Trilogy 2 focuses on creating liquidity for three types of resources: CPU, storage and network.

CPU resources are allocated and used both as stand-alone pools and as a part of accessing storage and network. The same applies for network resources, where the traffic starting from a node or originating to a node is associated with data processing or data storage, while traffic passing through a node may be regarded as a stand-alone network resource. Storage resources, however, rely on CPU and network to be accessed. This results in network being a common denominator for resource sharing. The Trilogy project introduced the MPTCP protocol and created network liquidity. The protocol allows bandwidth resources to be aggregated and efficiently allocated to application requirements.

Creating liquid resource pools and using them to respond to the application demand is the first step towards making resource utilization more efficient. The real advantages of the liquidity pools arise from placing together heterogeneous resources, both in terms of resource types and in terms of resource providers. This creates cross-liquidity tools that govern the interactions between existing resource pooling techniques.

Initial work in this area was presented in D1.2 - Initial Cross-Liquidity tools that covered the work performed during the first year of the project. D1.2 mainly focused on the bandwidth domain as bandwidth provides topology to the network and allows all other resources to be pooled together and managed in an unified way. This Deliverable, D1.3, presents a more complete set of tools that create liquidity from three different perspectives: cross-resource, cross-layer and cross-provider.

Cross-resource liquidity puts together different types of resources and introduces mechanisms to trade between them, trying to meet the application demand in the most effective way.

Cross-layer liquidity increases the flexibility of the network liquidity tools by bridging the gap between different layers of the stack.

Cross-provider liquidity enables the coordination of heterogeneous resources offered by different providers.

These three dimensions of liquidity determine the types of resource pools that can be created. Some of the tools presented in this Deliverable clearly fit into one of the above dimensions, for example the VM Migration for Mobiles (vM3) is cross-resource since it trades the usage of bandwidth for compute and storage. Other tools extend across multiple dimensions and can be seen from different perspectives, for example the Federated Market can be both cross-resource and cross-provider. We have presented such tools in the section that is closest to the fundamental idea of the tool. This discussion is included at the beginning of each of the following sections and will further detail the separation between the three liquidity domains.

1.2 Structure of the document

Section 1 presents an introduction and the overview of the Deliverable.

Section 2 presents the cross-resource liquidity tools. Since bandwidth allows other resources to be pooled together, this section includes mechanisms for trading bandwidth for compute and for storage.

Section 3 details cross-layer liquidity. The tools in this section cross the boundaries of the networking stack and create closer links between different layers.

Section 4 introduces cross-provider liquidity. Placing together resources governed by different providers increases the flexibility of resource allocation and created new market opportunities.

Conclusions are presented in section 5.

2 Cross resource liquidity

Cross resource liquidity aims at creating heterogeneous resource pools that can be managed in an efficient way to reduce the overall costs in running services in the current cloud centric world. The applications running in this paradigm can trade some of their resources (e.g. storage) to obtain other types of resources (e.g. compute, bandwidth).

VM Migration in Mobile (vM3) is a mechanism that allows applications to be moved between mobile devices or even pushed to the cloud. This mechanism allows, for example, pushing the backend CPU intensive task of an app to another nearby device or the cloud, allowing the phone to save battery, at the cost of a small delay in getting the computed results back. MPTCP helps vM3 to keep the network connections alive during the migration, making this solution a solid candidate for migrating applications that have a critical dependency on their Internet connection (e.g. Skype).

Federated block storage deals with wide area block replication systems, addressing use-cases where traditional mechanisms fail to work. The main expected use-case scenarios are disaster recovery as a service and virtual machine migration with the migration use-case detailed in Deliberable D3.2. Virtual machine migration benefits from federated block storage by having its attached virtual storage synchronized between relevant locations. Federated block storage utilises network resources in order to give improved storage reliability and redundancy.

Irmin uses MirageOS[23], a library operating system, to build high-performance network applications. MirageOS runs on top of Xen[2], benefiting from its stable hardware platform, avoiding the need for device drivers found in a traditional OS. Its small footprint make it an optimal candidate for auto-scaling web-servers with minimal latency.

Trevi [29], presented in previous deliverables, has been concluded in this Deliverable, with work related to a storage transport service that is tolerant to packet loss. This solution trades bandwidth to obtain a reliable channel with no retransmissions or timeout.

VNF Pool virtual CPE is a method of moving network functions that traditionally are implemented on specialized hardware, to virtualized computing platforms. This flexibility allows huge cost savings by avoiding the need to deploy new network boxes and replacing hardware that has reached its end of life. Virtual CPE can also be used in datacenters, by implementing liquid VPN services that can be used in conjunction with OpenStack.

2.1 VM Migration for Mobile (vM3)

The Virtual Machine Migration for Mobile (vM3) tool is responsible for creating liquidity on the end devices, allowing the user to share resources among nearby computing resources. The tool targets two main areas:

- Mobile to mobile liquidity
- Mobile to cloud liquidity

In the first case the liquidity tools provide enhanced user mobility. They enable the building blocks for a new user-interaction paradigm: the application follows the user. By breaking the hardware boundaries between devices, the context of applications can be moved to the device that best suits the user's needs. Figure 2.1 highlights the main components of the architecture.

2.1.1 Migration Service

The Migration Service (MS) is the management part of the tool. The The Migration Service handles multiple, migratable applications during their life cycle. Migratable applications are applications that run in a specialized container called a VMSlot. The Migration Service is an always-on agent (a service) that handles three main events relative to an application: start, stop and migrate.

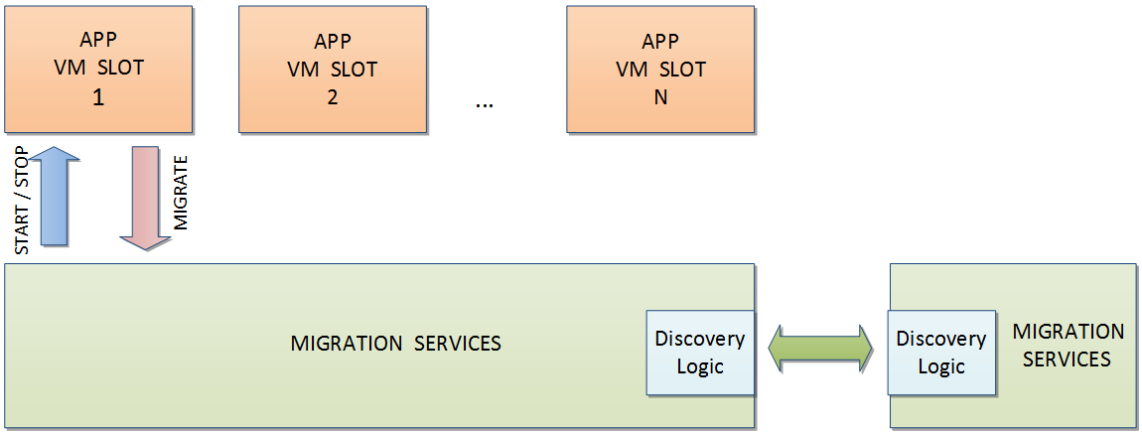


Figure 2.1: Migration Service Architecture

Because this vM3 tool must run on both Linux and Android, the number of VMSlots is fixed. The main goal of Migration Services is to enable apps to migrate from one VMSlot to a VMSlot located on another device. Because of this each migration-capable device must run a "Migration Service".
Migration slot actions:

- Start App inside VMSlot
- Discover other migration-enabled devices
- Initiate migration
- Handle migration request

2.1.2 Device discovery

In order to migrate an application, the tool must first discover peer devices that can handle migration. A device becomes aware of neighbor migration-capable devices by using a discovery logic. This discovery logic must first establish contact between devices and must result in establishing a communication migration-able channel. Currently migration can only be achieved over IP networking, but the contact channel can be established in a number of other ways, like Bluetooth and NFC.

The current version of the vM3 tool uses Multicast DNS [7] in order to discover other migration-able devices in the same network. Every device uses mDNS to maintain and update a list of neighbors that advertise the same service. The neighbors are not actually contacted until a migration process is initiated.

The advantages of mDNS are zero configuration and portability. Multicast DNS implementations exist in both Linux and Android, the main operating systems our tool is targeting. The disadvantages of using mDNS are delays in broadcasting peer state updates (when a peer goes online/offline) and sporadic failures of reporting the presence of a new peer in the system.

2.1.3 VM slot management

The vM3 tool design imposes a fixed number of VMSlots per device. A VMSlot is a context that runs a Virtual Machine containing an application. VM Slot management refers to allocating running applications to existing VMSlots.

There are multiple reasons why the number of VMSlots is fixed per device. The main reason is that, on Android, each application can have only one running instance. Thus, each VMSlot must be seen as a different application. Also, an application must be first installed prior to running (excluding the case when the device is rooted). As a result, it is very difficult to instantiate applications on demand for the Android ecosystems. As a solution, the vM3 tool on Android uses a fixed number of pre-installed VMSlots that differ only by name. Another reason for having a static number of devices is to simplify the management logic.

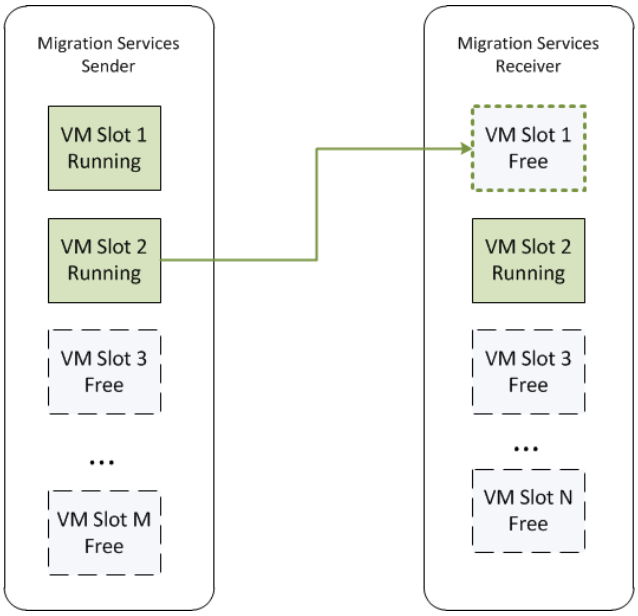


Figure 2.2: VMSlot management

Figure 2.2 shows VMSlot management in a typical migration scenario. The receiving MS has to find an empty slot before confirming the migration. The sending MS has to free the slot only after the migration is signalled as successful.

Migration Service has to do more maintenance work than just handling VM slots. A receiving MS has to check the requirements of the incoming VM (like RAM usage or used devices). Only after it made sure

that a matching VMSlot is fired up, can the receiving MS agree to the transfer. There are cases when the migration destination device cannot meet all the source requirements and instead negotiates a trade-off set of requirements. The sending VMSlot initiates the transfer but keeps the VM running. Only after the migration process is successful and the VM has resumed on the destination device, can the sending MS shut down the VM and free the VMSlot.

2.1.4 Injection of context information inside the VM

By definition a VM is a separate environment that is loosely connected to the outside world. But the vM3 design covers three key elements in the life of a VM:

- Starting the VM
- Starting the application
- Migration of the VM

In the current vM3 implementation, starting the application inside the VM takes place immediately after the VM starts. In the case of starting a VM, general configurations like IP address, VMSlot ID or connected devices must be established. These configurations are all local settings and depend on the host device. The parameters are passed directly to the hypervisor upon starting the VM.
Starting the application however requires information about external application resources, like the link for retrieving the application. For example an HTML5 video streaming application requires a link to the web location of the stream. Passing parameters to the application at VM startup can be done in a number of different ways. The vM3 solution prefers opening a socket to the VM and directly passing the parameters to a listening daemon. The parameters are encoded as a text string are interpreted by the guest OS.
The VM start and application start events are not necessarily connected. Starting a VM is a time consuming operation, so in the future we plan to have pre-booted VM's that can be resumed when a migratable application needs to be started. This way, the VM start event occurs a significant amount of time before the application start event.
The third and most important event is the signal to migrate sent by the Migration Services instance to a VMSlot. This event can be subdivided into two components:

- MigrationStart (the moment when the memory transfer starts between the source and the destination device)
- MigrationEnd (the moment when the VM is resumed on the destination device)

It is important to notice that while MigrationStart can be easily signalled to the VM (through a socket or a char device communication channel), MigrationEnd is harder to detect. After migration, the VM is transported to a different VMSlot setup and must be reinitialized with almost the same set of parameters as the VMStart event. Only this time the parameters cannot be passed directly to the hypervisor and all the usual communication

channels (socket, char devices) are severed. So, even if the signal of MigrationEnd reaches inside the VM, it is still hard to pass the new configuration. The VM could perform an action like DHCP request to a predefined server, but this is time consuming and in the end it gets to the point where TCP connections are severed.
To resolve this issue, we mount a partition from the destination host into the newly received VM. The mount action generates a HW signal that marks the end of the migration process. Also, the mounted partition contains configuration data related to the new VMSlot used to host the migrated VM. This approach is fast and resolves the two issues at once: signalling and reinitialization. The downside is that the VM migration process cannot be restarted immediately after the VM has been copied on the destination device. The system needs to wait an amount of time for the partition to be mounted and the data stored in it to be interpreted.

2.1.5 App Wrapper

The AppWrapper is the environment required to run a migratable applications. This environment is composed of two major ingredients:

- Image binary - consisting of a very slim ISO image that is meant to be booted with the application. This binary is used by all applications. The size of the image is around 80 MB.
- App binary - containing only app specific data that is needed to start that app on the VM image. The app binary is on a partition that is mounted prior to starting the app.

An interesting observation is that on Android, those binaries must be installed as APK Expansion Files and not with the rest of the application (they are represented by Opaque Binary Blobs (OBB's) and are installed in a different path altogether).

2.1.6 Application I/O

In this section we discuss the way vM3 handles I/O. The app runs in a virtualized container powered by the QEMU and KVM software stack. This stack is developed mainly for the Linux kernel and desktop/server Linux-based operating systems. Android OS on the other hand uses a very different software stack on top of a modified Linux kernel. In order to extend the I/O features to mobile devices, the vM3 tool had to extend the QEMU and KVM I/O capabilities to mobile devices.
One of the main I/O functions to be ported is video output. At first we tried using the kernel framebuffer directly. The upside is good performance. The main downside of the approach is that it is too intrusive and requires profound changes to the Android image, changes that have a low probability of being accepted into the upstream Android tree. The other downside is that the approach is a process of reinventing the wheel on Android.
The other solution was to look at existing projects that could, at least partly, do what we wanted. The most well suited project was Simple Direct Media Layer, SDL. SDL is a hardware abstraction library that offers a generic API to access I/O devices like video, sound, keyboard. It covers a wide range a platforms including Linux, Android and Windows. It even has QEMU binding.

We integrated SDL along side our tweaked version of QEMU in order to integrate with the Android framework. We opted for maintaining the binding with Android framework as the main event source/destination in order to envelope the vM3 applications as a normal Android application.

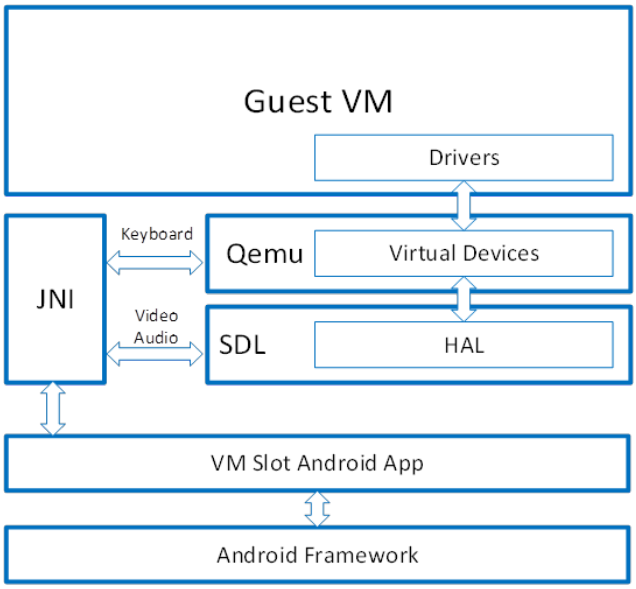


Figure 2.3: App I/O with QEMU and SDL on Android

The full software stack of the VMSlot part of the vM3 tool can be seen in Figure 2.3. The components specific only to the Android version of the tool are the VM Slot Android App and the JNI bindings to the emulator. QEMU, SDL and GuestVM parts are mostly the same as their Linux counterparts. This way the vM3 tool brings together pieces in order to create new liquidity capabilities.

2.2 Federated Block Storage

2.2.1 Introduction

Resource liquidity lies at the heart of Trilogy 2 and storage is a key element of this. Data replication in general and block replication in particular are key enablers for storage liquidity and underpins other functionality such as Virtual Machine Migration.

Storage replication is not a new idea and is already used for a number of reasons within datacenters. Key among these are high availability and redundancy to protect against hard disk failure; Creation of a snapshot (or backup) of data to a more reliable storage media; Duplication of data across nodes to allow for efficient parallelism. SLAs and working practices determine what data is replicated and how. Regulation also requires that certain types of financial and health related data are located at different physical sites to increase cross-site resilience.

For certain types of work-loads, reverting to a previous snapshot of data from a backup is an acceptable recovery mechanism. However other types of transactional workloads such as financial or retail systems cannot rely on backups and need an active transaction system that is consistent and verifiable. Currently local replication through the use of RAID and self-monitoring systems has been sufficient for ensuring data

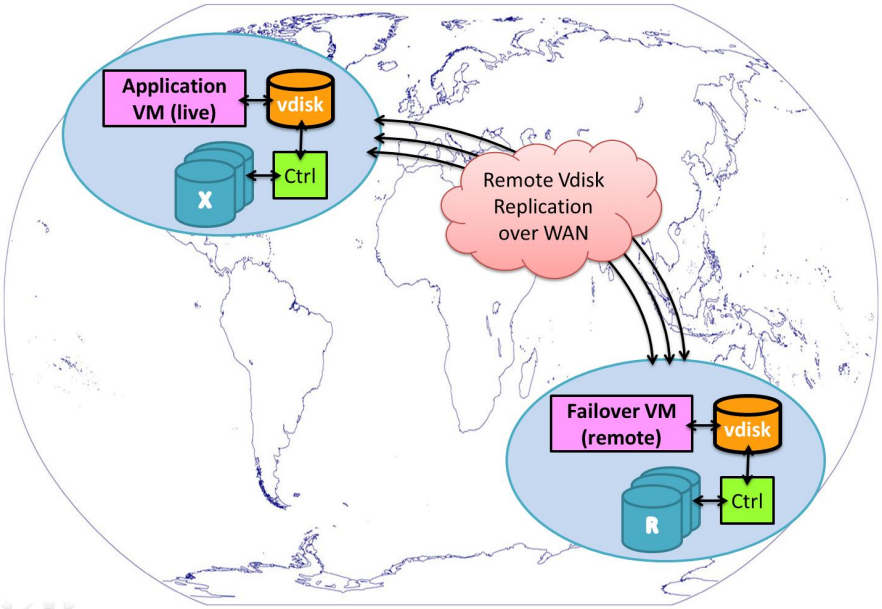


Figure 2.4: Block replication can provide a remote hot spare VM

consistency. However as workloads scale there is an increased need to maintain redundancy across hardware units within a Datacenter and between Datacenters. This requires wide area block replication across and between datacenters.

This section describes the wide area block replication system designed as part of the Trilogy 2 project. D3.2 then gives details of how this helps enable Virtual Machine Migration.

2.2.1.1 Description of the issues faced

A number of issues must be addressed for any wide area block replication system to work efficiently. These are summarized below.

I/O Limitations Currently replication of data is handled within a single system device such as a SAN, locally within a server or in between storage elements within a rack. Depending on the mechanisms and choices made there are a set of possible configurations for replication in both hardware and software that will ultimately lead to a trade-off between performance, reliability and availability. In the case of wide-area replication the main issue is the slow write path to the remote storage location. Local I/O performance through storage controllers and RAID devices is limited by the physical I/O speed of the device, the communication channel bandwidth, the controller bandwidth or the host server. Once you are looking at replication across physically separate racks or between datacenters you have to include the network bandwidth within the datacenter, the upstream bandwidth of the datacenter, the link bandwidth and congestion between datacenters and the congestion control systems on the local and remote systems. The overall I/O speed will be limited to the slowest point in the path and may not be linear with changing load.

Control Path Latency The increased latency due to a longer path there also creates issues in the communication between the sender and the receiver. Control messages will take longer to be received and re-assembled at each end slowing down error checking and correction. Round trip times (RTT) for messages can also increase

with each additional software layer in the end-to-end path. These are known issues that affect distributed file-systems and distributed databases. Various mechanisms have been proposed to handle slow replication paths using a combination of local caching, eventually consistent best-effort mechanisms (BASE). However remote block replication provides a much faster replication mechanism.

Transaction Synchronicity Reads by nature are usually asynchronous and are normally only used in a synchronous configuration if data validation and verification is required. Read performance benefits from this asynchronous behaviour. With careful selection of different blocks from different replica locations the bandwidth for reads will be close to the sum of all the bandwidths of the individual paths to the blocks. Latency can also be improved at the cost of read bandwidth by using data from the fastest path - the read bandwidth is reduced as both paths will have to read the same blocks. Writes however require validation from both paths that the data has been successfully written before the system can be sure that the write is consistent and so require synchronous paths. This means writes see the slowest latency of all the available paths and the write performance is dominated by the bandwidth of the slowest path. Local caching can be used at both sender and receiver (only the fastest path will use the local cache). This can be used to allow bursty data to be written to at least one path and allow other I/O operations to continue. However at the point that the local cache is full, the rate will then be limited to that of the slowest path.

2.2.1.2 Utilising Wide Area Block Replication

Within the context of Trilogy 2 there are two main use cases for wide area block migration. These are Disaster Recovery as a Service (DRaaS) and Virtual Machine Migration. Virtual Machine Migration forms one of the key Trilogy 2 Use Cases. As such it is described in some detail in D3.2 'Use Case Selection'. The following is a brief summary of how it makes use of wide area block replication.

Virtual Machine Migration (or VMM) describes the ability to be able to migrate live VMs between different sites for purposes such as dealing with planned maintenance, balancing workloads according to demand, responding to cost/energy savings (including 'follow the sun' requirements), Business continuity and disaster recovery. Current approaches for migrating VMs rely on stopping the local VM instance, taking a snapshot to capture its state, then spawning a new remote VM using that system state. Wide area block replication offers the chance to do this in a much more efficient manner with no loss of service (see Figure 2.5). Upon identifying the need to migrate a VM the workflow might progress like this:

(i) The local Cloud manager selects a VM to migrate.

(ii) Once selected the manager chooses a migration target and sets up a VM on that target.

(iii) The new VM will use block replication to create an exact copy of the current VDisk belonging to the local VM.

(iv) Once all data is synchronised between the two VMs the new VM can be brought up (or alternatively can function as a "hot spare" ready for any failover).

(v) Any writes to the original disk are automatically transferred to the remove VDisk, thus capturing any current transaction state. If the WAN link between the two VDisks is unpredictable these changes may not propagate immediately.

(vi) When the system is ready to transfer the old VM can be paused and the load balancer transfers any connections to the new VM so it takes over from the old one.

(vii) Finally the old VM can be turned off.

This highlights a few of the issues that need to be addressed by the wide area block replication tool. These include the need to replicate only the blocks that have changed, the requirement to secure the data being replicated and the desirability of asynchronous write behaviour if the workload is highly bursty.

2.2.2 Architecture

The proposed wide-area replication architecture is based on the federated OnApp cloud architecture that includes the OnApp storage platform, so before delving into the design of the wide-area storage replication, we present a brief introduction to the architecture of the OnApp storage platform.

2.2.2.1 The OnApp storage architecture

The OnApp storage architecture is a distributed block storage system that uses existing commodity cloud hardware to present a reliable, scalable storage system as an alternative to traditional storage area networks (SANs).

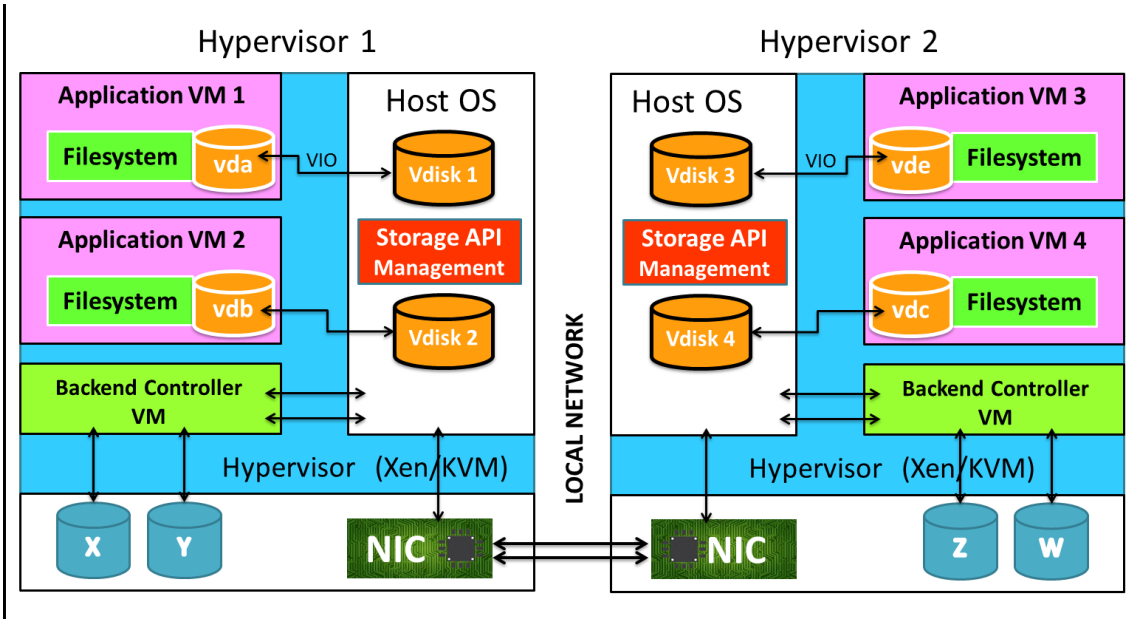


Figure 2.6: Overview of the Onapp storage architecture

At the lowest level of the storage platform, the storage devices (disk or solid-state drives) are accessed by back-end instances that perform network communication with the front end, either locally or within the OnApp Control Panel, a web-based interface to manage resources within a local cluster of hypervisors (HVs).

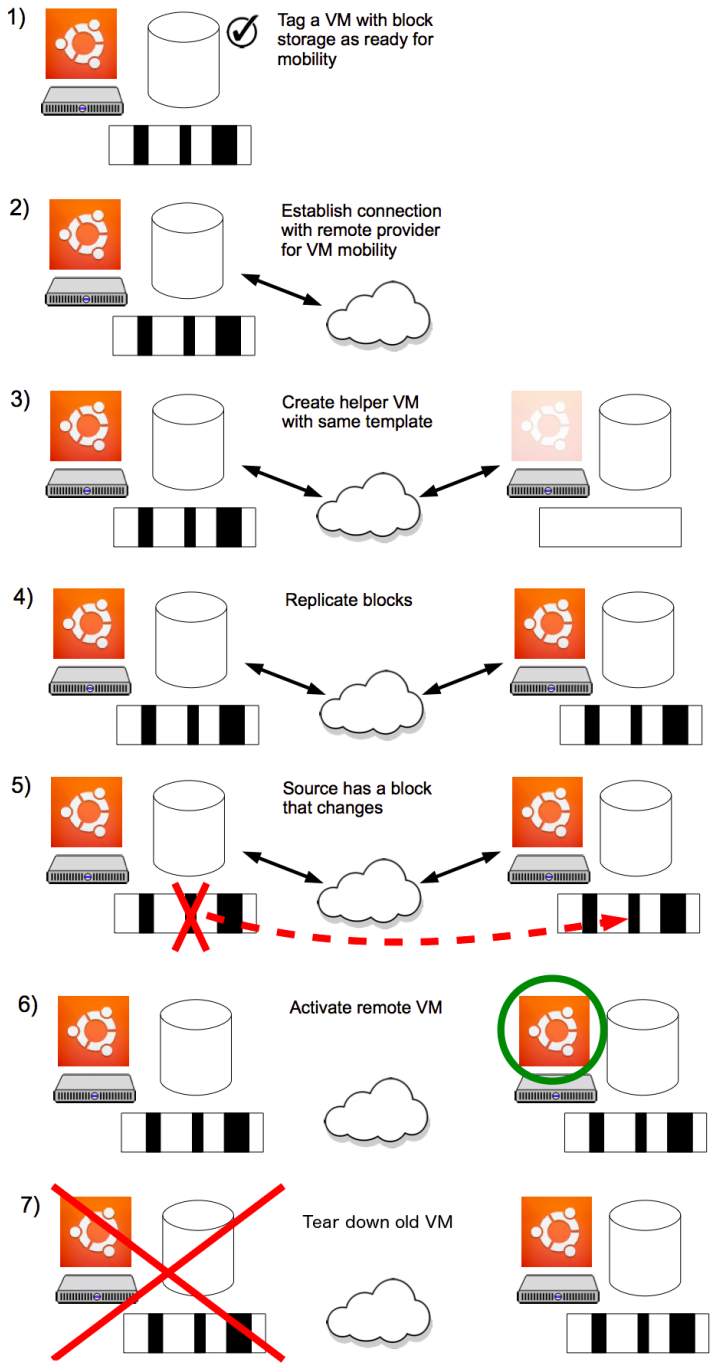


Figure 2.5: Migrating a VM using remote block replication

A virtual disk (VDisk) is part of a data store. Each VDisk replica has an individual handler that connects it with the front end; the back end also handles the access to the storage drives. Once a VDisk has been successfully created, it becomes available through the device mapper as a block based drive.

Drives that are connected to the hypervisor are displayed in the OnApp Management user interface, a web-console which manages the OnApp Cloud Platform and Storage system. After the back end reports about the storage drives, they will be displayed in the OnApp user interface.

OnApp Storage uses multiple front ends (2+ hypervisors) that communicate via back-ends to avoid a single point of failure. As long as there is an active back end with access to a replica, the data can be accessed. If a hypervisor that contains a replica fails, the failed data replica will become out of date as soon as data writes are performed. This leads to the VDisk degrading. To fix the degraded disk, you need to manually perform the disk repair operation. During the disk repair, the disk volume is repaired using good available replicas. However, if the disk drive has completely failed and cannot be repaired, it can be forgotten via the UI. Then it can be replaced with the new drive after the rebalancing operation.

The OnApp Storage system is also location-aware. Having detected where the application virtual server is, the Storage system will attempt to keep and use a replica on the back end system which is local to that server. This feature allows to optimize data placement, reduce the amount of network traffic and improve the performance. If the virtual server is migrated to another location, the Storage system will detect changes and migrate data to the new VS location.

Virtual server live migration is available on Xen and KVM hypervisors and storage migration is fully supported across the data store to any hypervisor drives within the same zone.

The OnApp Storage architecture has been designed to use existing cloud hardware. There are many different types of storage drives connected to the hypervisor servers. The Storage system divides the drive performance into low, medium and high. For example, most Solid State Drives (SSDs) will be classified as high performance. Standard Hard Disk Drives (HDDs) can be either of low or medium performance. The performance metrics are calculated when the storage is activated to check the read and write behaviour. You can also manually set disk performance in the OnApp UI. Different drives are then detected and enabled through a multicast channel local to a single Control Panel and divided by hypervisor zones. The division by hypervisor zones helps to separate the storage channels for different types of underlying hypervisor types (Xen/KVM etc).

2.2.2.2 Wide-area replication over OnApp's storage architecture

With reference to the scenario described in Section 2.2.1.2. Initially the local VM has access to one or more local VDisks (indicated as a block store in Figure 2.4). If the "prefer local storage" option has been chosen then these will be on the same hypervisor. When the new VM is created on the remote provider an identical VDisk arrangement needs to also be created. The data is replicated to these VDisks via the WAN. This replication happens at the back-end node level with writes to the "nbd servers" being sent to the remote

VDisks (see Figure 2.7).

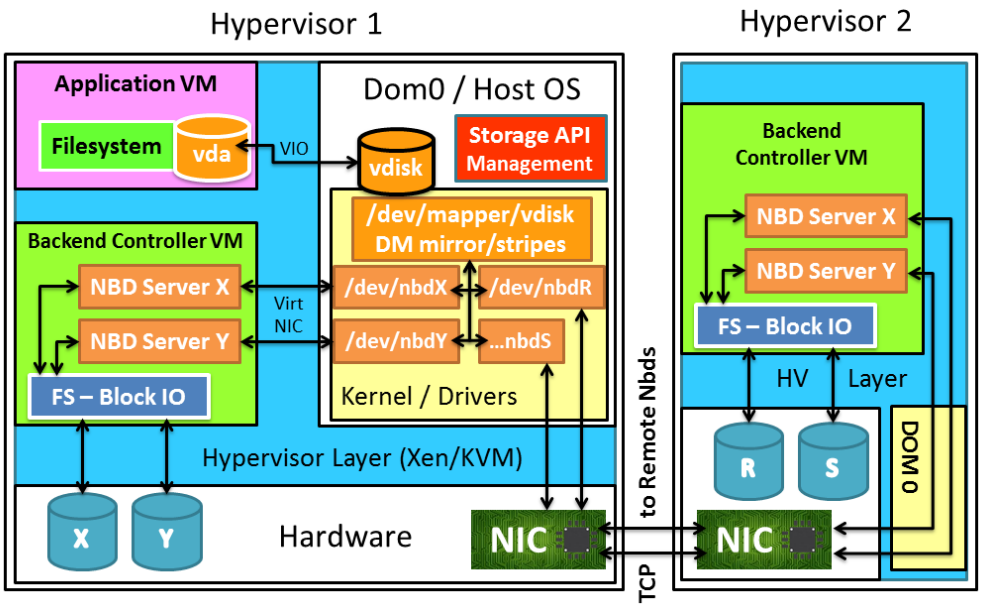


Figure 2.7: The backend and frontend nodes can be remote

Due to the high latency over the WAN writes replicating the data remotely will be buffered and coalesced locally before being sent. This allows the system to verify the write asynchronously (which is important across a wide area network with variable latency). It is also possible for writes to be sent as staggered chunks to the remote replicas. If a striping scheme is employed on the primary VDisk (with alternate stripes of data being written to alternate VDisks), there would be need to be a similar de-striping mapping on the secondary VDisk setup to allow proper data placement for reconstruction, since data is copied at the stripe level. The aim of this system is to keep the remote replicas as near identical to the state of the primary VDisks as possible. The goal is for the remote replication to avoid introducing any overhead that may affect the performance of the primary VDisk. If the WAN connecting the primary and remote replicas has adequate throughput to handle the volume and rate of writes then there should be no noticeable performance degradation. However if the level of writes is too high and exceeds the capacity of the buffer to level them out then the system has to choose between two options:

(i) The primary VDisk is throttled to match the performance level of the replication

(ii) The replication stops temporarily and is performed in lower frequency intervals. This second approach is more appropriate for temporary problems, such as temporary network outages

2.3 Irmin

Work arround this topic was presented, in previous Trilogy 2 deliverables, as Irminsule. Currently the name of the tool was changed to Irmin and this section together with future deliverables will reflect this change.

2.3.1 Mirage, an OCaml-based unikernel system

(based on the `openmirage.org` website)

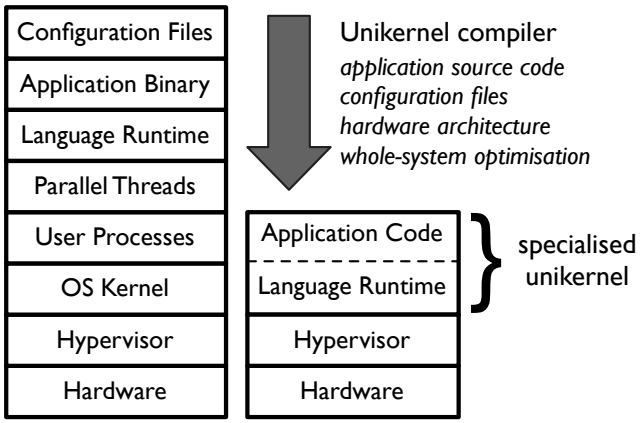


Figure 2.8: Contrasting software layers in existing VM appliances vs. unikernel's standalone kernel compilation approach.

Most applications that run in the cloud aren't optimised to do so, ecause they rely on an underlying operating system. Compartmentalisation of large servers into smaller virtual machines has enabled many new businesses to get started and achieve scale. But many of those virtual machines are single-purpose and yet they contain largely complete operating systems, including their vulnerabilities and bloat. Mirage represents a new approach where only the necessary components of the OS are included and compiled along with the application into a unikernel. This results in highly efficient appliances that can be deployed directly to the cloud and embedded devices, with the benefits of reduced costs and increased security and scalability. Figure 2.8 describes the difference between unikernels and current approaches.

Mirage is a *library operating system*[22] designed for constructing secure, high-performance network applications across a variety of cloud computing and mobile platforms. It is based around the Xen hypervisor and the OCaml language. Xen provides a stable hardware platform[3], which avoids the need to support the thousands of device drivers found in a traditional OS. Thanks to OCaml, code can be developed in a high-level functional programming language which avoids many of usual flaws[21]. It is then compiled into a fully-standalone, specialised unikernel which can run directly on Xen hypervisor APIs. Since Xen powers most public clouds, Mirage lets you deploy your servers more cheaply, securely and faster in any of them.

Due to the much smaller size, it is possible for example to create auto-scaling web-servers with very small footprints. If a sudden spike in traffic occurs, the web-servers can be configured to create and deploy copies of themselves to serve the demand. This auto-scaling can happen so quickly that an incoming connection can trigger the creation of new server and the new server can then handle that request before it times out (which is on the order of milliseconds). When the demand dies down again, these web-servers can automatically shut themselves down. This elasticity results in the ability to precisely match the demand and avoid wasting resources.

2.3.1.0.1 Towards a distributed database As a cloud oriented operating system, Mirage has to deal with all the problems arising in distributed systems. One of them is to share data between several devices spread out over a network. In the example of self-scaling web-servers, newly created instances should first fork an image of the web-server's global current state. Then they can modify it independently according to the requests they receive. Finally, when the traffic is decreasing and returning to a normal rate, the forked web-servers have to somehow merge back all their local modifications.

2.3.2 Design

Irmin is a portable distributed database written in OCaml, and designed with the requirements of static type-safety for security and reliability. Designing a distributed database is a difficult problem, mainly because of the CAP theorem [14] which states that:

> It is impossible for a distributed system to simultaneously provide all three of the following guarantees:
>
> (i) *Consistency*: all nodes see the same data at the same time
>
> (ii) *Availability*: a guarantee that every request receives a response about whether it was successful or failed
>
> (iii) *Partition tolerance*: the system continues to operate despite arbitrary message loss or failure of part of the system.

The modern answer to solve this paradox is to maintain availability but relax the consistency model. Most of the large-scale distributed systems assume there is no unique global state of the system, which results in a class of system said to be *eventually consistent* [5]. Irmin's approach to relax the consistency model is inspired by distributed version control systems such as Git and Mercurial. Every actor owns a branch representing a partial replica of the global database. Modifications are local and happen only on the current branch. Branches can explicitly be merged in order to recover the consistency property, using application-defined merge policies between replicas. Such classes of systems have been called *branch-consistent* models. Data replication is a key technology in distributed systems that enables higher availability and performance. It is possible to distinguish two kinds of replication: *pessimistic* [36] [20] [28] methods where master election and synchronous locking are used to block the system while changes are propagated; and *optimistic* [33] methods where the changes are propagated in the background and where special techniques handle supposed rare conflicts. Irmin chooses to use an optimistic replication system because it improves availability, does not need knowledge about the underlying network, and can easily scale because it does not need synchronisation. The drawback is that the users have to handle conflicts.

Conflicts can appear in two different situations: when two nearby users are modifying the same value at the same time; and when a value has been changed in two distant locations, with the background propagation resulting in a conflict. Irmin allows the application developer to deal with these situations using several tools:

- Conflict-free replicated data types [37]
- Type of data with custom merge operator
- Callback functions applied every time a conflict happen

The study and the design of such mechanisms is one of the main goals of my internship.

2.3.3 Architecture

Irmin provides a high-level interface built upon two user-provided stores.

The *block store* is a low-level key/value append-only store, where values are a sequence of bytes and keys are deterministically computed from the values (for instance using SHA algorithms). This mean that:

- if a value is modified, a new key/value pair is created: the resulting data-store is *immutable*
- if two data-stores share the same values, they will have the same keys: the store is *consistent*, the overall structure only depend on the stored data

The block store contains serialized values from application contents, but also structured data, like prefix-tree nodes and history meta-data. As the store is append-only, there is no remove function. The store is expected to grow forever, but garbage-collection and compression techniques can be used to manage its growth. This is not an issue as commodity storage steadily becomes more and more inexpensive.

```
module type S = sig
  type t
  type path
  type contents

  val read: t -> path -> contents
  (** Read the content at [path] in [t]. *)
  val update: t -> path -> contents -> unit
  (** Replace the contents at [path] in [t] by [contents] if [path] is already
  defined and create it otherwise. *)
  val remove: t -> path -> unit
  (** Remove the given [path] in [t]. *)
  ...
end
```

Figure 2.9: High-level prefix-tree interface of Irmin, generated over the block store, the tag store and the application contents description.

The tag store is the only mutable part of the system. It is a key/value store, where keys are names created by users and values are keys from the block store, and can be seen as a set of named pointers to keys in the block store. This store is expected to be local to each replica and very small. The tag store is central for higher-level algorithms such as synchronisation and garbage collection.

The high-level interface is generated over the block store, the tag store and the application contents description. It lifts immutable operations on the block store into a mutable prefix-tree, whose signature is given in

figure 2.9. The prefix tree `path` is usually a list of strings and node values are the user-defined mergeable `contents`.

2.3.4 Weakly consistent data structures

As mentioned earlier, Irmin allows the application developer to deal with conflicts using several tools. One of them is the use of data structures with custom merge operators. The idea is to give an abstraction of the Irmin low-level store as a high-level data structure. In addition to their classical operations, these data structure include the merge operation mentioned earlier. The OPAM repository for mergeable queues is available on `github.com/mirage`.

These data structures and their associated operations have already been widely studied, even in a pure functional context [27]. The purpose of this work is not to compete with existing implementations of such data structures, but to extend them with an efficient and consistent merge operation.

2.3.4.1 Queues

There are several efficient implementations of queues that can perform enqueuing (or *push*) and dequeuing (or *pop*) operations in $O(1)$ time. However, Irmin is built on an append-only low-level store. Such representations of the memory matches well with the functional programming model where the memory is immutable. For this reason these mergeable queues are designed as functional queues.

Operation	Read	Write	Big O
Push	0	2	$O(1)$
Pop	2 on average	1 on average	$O(1)$
Merge	n worst case	1	$O(n)$

Figure 2.10: Cost of queue operations where n denotes the length of the queue. Read and write are expressed in number of memory access.

A functional queue is composed of two simple linked lists. The first one contains elements that have been pushed onto the queue, and the second one those that will be popped. When the pop list is empty, the push list is flushed into the pop one. This operation is called *normalization*. Event though the normalization is a linear operation, each element of the queue has to be normalized only once. That is why –as it is reminded in figure 2.10– the amortized cost of operations on the queue is $O(1)$.

2.3.4.1.1 Internal structure The implementation of mergeable queues is based on an Irmin store containing three types of element: `Index`, `Node` and `Elt`. Their type declarations are given in Figure 2.11.

`Index` are queue accessors. They are defined by four fields, `push`, `pop`, `top` and `bottom`. The two first fields, `push` and `pop`, are the number of pushes and pops applied to the queue since its creation. They are useful for the merge operation. The two others, `top` and `bottom`, are keys of the top and bottom element of the queue.

`Node` are elements manipulated by queue operations. They are composed of four optional elements, `next`, `previous`, `elt` and `branch`. `next` and `previous` are keys to a potential preceding or following ele-

ment.In practice, only one of these two can be not empty. `elt` is also an optional key which points to a value of the queue. The last field `branch` is an optional index, used only by the merge operation.

Finally, `Elt` contains elements added to the queue.

```
type index = {
  push: int;
  pop: int;
  top: K.t;
  bottom: K.t;
}

type node = {
  next: K.t option;
  previous: K.t option;
  elt: K.t option;
  branch: index option;
}

type elt = Index of index | Node of node | Elt of V.t
```

Figure 2.11: Type declaration of mergeable queue structuring elements. The Irmin store is specialized in order to containing such elements.

2.3.4.1.2 Major operation

The two first main operations on a queue are `push` and `pop`. The push operation adds a new `Elt` containing the pushed value, and a `Node` pointing to this element and the previous bottom element of the queue. It returns a new `Index` where the bottom element is the new created `Node`. The `pop` operation tries to read the top element of the queue. If the pop list is empty, the queue is normalized. Then it returns the value associated with the reading `Node` and an `Index`, where the top element is the following element of the reading `Node`. On average, there are two reads and three writes in the Irmin store for one `push` and one `pop`.

The other main operation is merging. The merging operation takes three arguments: two queues to be merged, `q1` and `q2`, and a common ancestor to those two queues called `old`. The resulting queue – called `new` – has to reflect the transformations from `old` to both `q1` and `q2`. First, elements of `old` which have been removed from `q2` are removed from `q1`. This is done without accessing these elements by using the `push` and `pop` values. In the same way, all elements of `old` which are still in `q2` are removed. Then, `q2` is concatenated at the end of `q1` by adding a new node where the field `branch` contains the index of `q2`. In the worst case, the merge operation uses a number of reads linear in the size of `old`, but always only one write.

In Figure 2.12 the main queue is accessible through the index `I0`. The index `I1` is pointing to a queue concatenated during a previous merge operation. This queue will be unfolded during the next normalization. Because of the Irmin store behavior, two nodes containing the same element share its physical representation.

2.3.4.2 Ropes

2.3.4.2.1 General description

A rope is a data structure that is used for efficiently storing and manipulating a very long string. A rope is a binary tree having leaf nodes that contain a short string. Each node has an index equal to the sum of the length of each string in its left subtree. Thus a node with two children divides the whole string into two parts: the left subtree stores the first part of the string and the right subtree stores the second part. The binary tree is crossed from the root to leaf each time the string has to be accessed. It can be done in $\log(n)$ time if the tree is balanced.

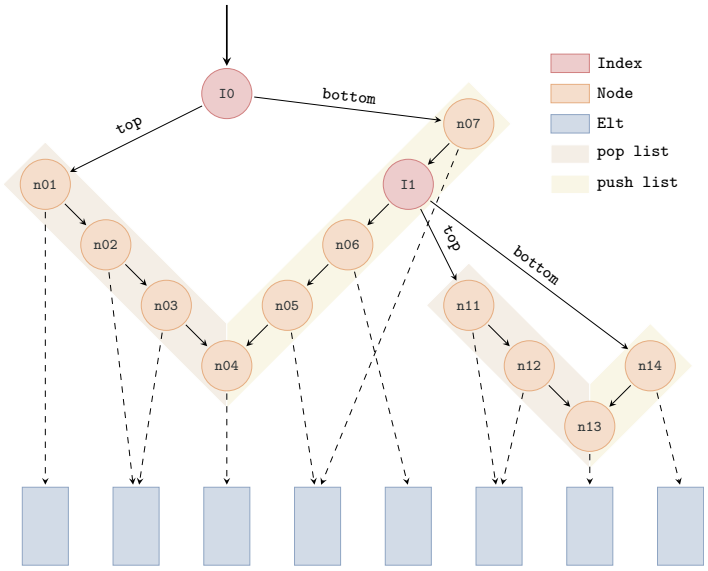


Figure 2.12: Example of a possible queue internal structure.

The main operations on a rope are *set*, *get*, *split*, *concatenate*, *insert* and *delete*: Set and get respectively set or get the character at a given position. Split(t, i) split at the position i the rope t into two new rope. Concatenate(t_1, t_2) return a new rope which is the concatenation of t_1 and t_2. Insert(t, i, s) insert the character chain s in the rope t at the position i. Finally, Delete(t, i, j) delete in the rope t the characters between i and j. Figure 2.13 compares the complexity of these operations for a rope and a string.

If ropes are mainly used to manipulate strings, they can also be used to manipulate any other type of container, as long as they support the above six operations. In fact, I design the ropes with the following idea: "give me a container with a set of operations, and I will return you a rope on this container, supporting the same operations but achieving a better complexity, with the exception of set and get". I therefore request a merge operation on the container in order to implement the merge operation of the mergeable rope. Indeed, because such a rope can be built on any type of container, it is impossible to have a general way to merge it.

Operation	Rope	String
Set/Get	$O(\log n)$	$O(1)$
Split	$O(\log n)$	$O(1)$
Concatenate	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$
Merge	$\log(f(n))$	$f(n)$

Figure 2.13: Comparison of the complexity of several operations on ropes and strings. n denotes the length of the rope/string, and f is the complexity of the merge function provided by the user.

2.3.4.2.2 Implementation overview The implementation of mergeable ropes is quite straightforward, in the sense that it follows the previous description, and its signature is given in binary search tree which keeps a factor of two between its minimal and maximal depth. To implement such tree, the Irmin store is specialized in order to contain three types of elements. As in mergeable queue, `Index` are accessors to the

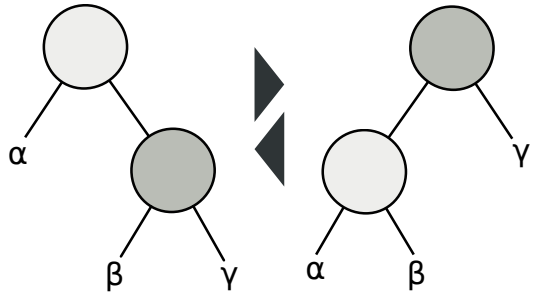


Figure 2.14: Generic tree rotation. A tree rotation moves one node up in the tree and one node down.

data structure. `Node` are intermediate elements of the tree which contain information to improve the binary search. Finally, `Leaf` are the user-defined container on which the rope is built.

The implementations of the six main operations follow more or less the same pattern. The algorithm performs a binary search on the tree in order to determine the leaves concerned by the operation. Then the operation is applied on the containers found in the leaves. In order to achieve the $\log n$ complexity, the size of these containers has to be of the same order as the depth of the tree. Finally, the tree is recursively rebuilt, using a rotation transformation in order to maintain the balancing property.

The merge operation is a bit different. Given two trees to be merged and their common ancestor, their keys in the Irmin store are used to determine the smallest subtrees where modifications occurred. Then these subtrees are linearised in containers on which the user-defined merge operation is applied. On simple trees, this approach is very efficient because the merge operation is applied on the smallest possible container. However, the balancing property reduces this effectiveness. Indeed, the internal structure of a tree can be deeply modified during a rebalancing operation, making it impossible to compare potentially large subtrees. In order to minimize the scope of rebalancing operations, the rotation function is applied as little as possible, and only on the smallest unbalanced subtree. Due to this restriction, the impact on the merge function efficiency is proportional to the volume of modifications produced by an operation.

2.3.5 Data structure analysis

After having implemented those two data structures, it was necessary to assert the correctness of these implementations and to analyse their effective costs.

2.3.5.1 Automatic checking

In the case of the correctness, we especially want to ensure that the classical operations match with their equivalents in other implementations, and that the merge operation follows its specification. The way it works is similar for queues and ropes. An oracle is used to determine a sequence of operations and their result. This sequence is then applied on the tested data structure, and on a witness obtained from another implementation. At each step, the data structure and the witness are required to exactly match. And at the end of the sequence, the two obtained result have to correspond with the result predicted by the oracle.

If this protocol is working fine with classical operations, it cannot be honestly applied on the merge operation. Indeed, to my knowledge no other data structure exists that can be used as a witness. The verification process

is therefore a bit different for this operation. In this case, we chose a result with a easily verifiable property which is preserved after each merge operation. For example, a queue containing an increasing sequence of numbers, or a rope composed by a non-decreasing sequence of characters. This result is decomposed into several subresults that have to be merged in order to recover the initial one. As a merge preserves the aforementioned property, we can verify the correctness of the merge operation at each of its uses.

These tests have successfully guaranteed the good behaviors of the implementation of queues and ropes data structures. But they are not sufficient to ensure that the theoretical complexity is reached. Several benchmark tests have been developed in order to validate this last point. Aside from showing that theoretical complexity is reached, these tests highlight some other interesting facts.

2.3.5.2 Benchmarking

2.3.5.2.1 Performance analysis The most obvious interest of benchmark tests is to measure the needed time of a given operation. Figure 2.15 shows the result of one of these tests. As a first glance, we can see that the theoretical complexity is reached for every operations. Then we can see a general stair behavior. This one is due to the internal tree representation. At each a step a new level is needed in this tree in order to represent the whole rope. Finally, we can see several spikes, which are the consequences of the complex interactions between the balancing property and maintaining a leaf length proportional to the depth of the tree.

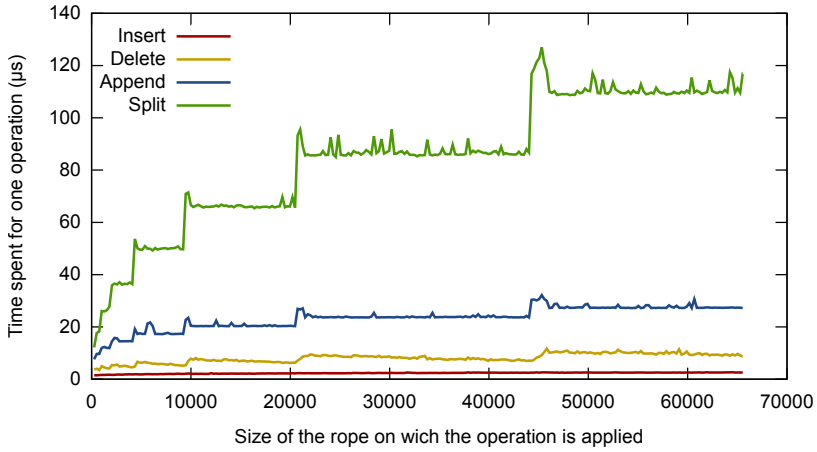


Figure 2.15: Needed time for one operation on a rope of size n on the Obj backend

2.3.5.2.2 Backend comparison As Irmin can be instantiated on different backends, it may be interesting to compare their relative performances. The figure 2.16 shows the result of a same test run on different backends. On this graph, *Memory* refers to the in RAM backend. *GitMem* and *GitDsk* respectively refer to a Git backend, in the first case instantiate in the memory, on the hard drive in the second case.

The two last are a slightly different. *Obj* is backend based on the OCaml module Obj that I have implemented. The idea is to directly manipulate the OCaml heap. In a sense, that gives the raw cost of the algorithm. Its implementation is given in Finally *Core* is not strictly a backend because it refers in fact to the implementation of functional queue in the Core library. It is used here as a sort of optimal case, in a matter of comparison.

As we can see, there is a factor five between Core and the implementation of my queue on the Obj backend.

It is an acceptable price to pay for maintaining a mergeable data structure.

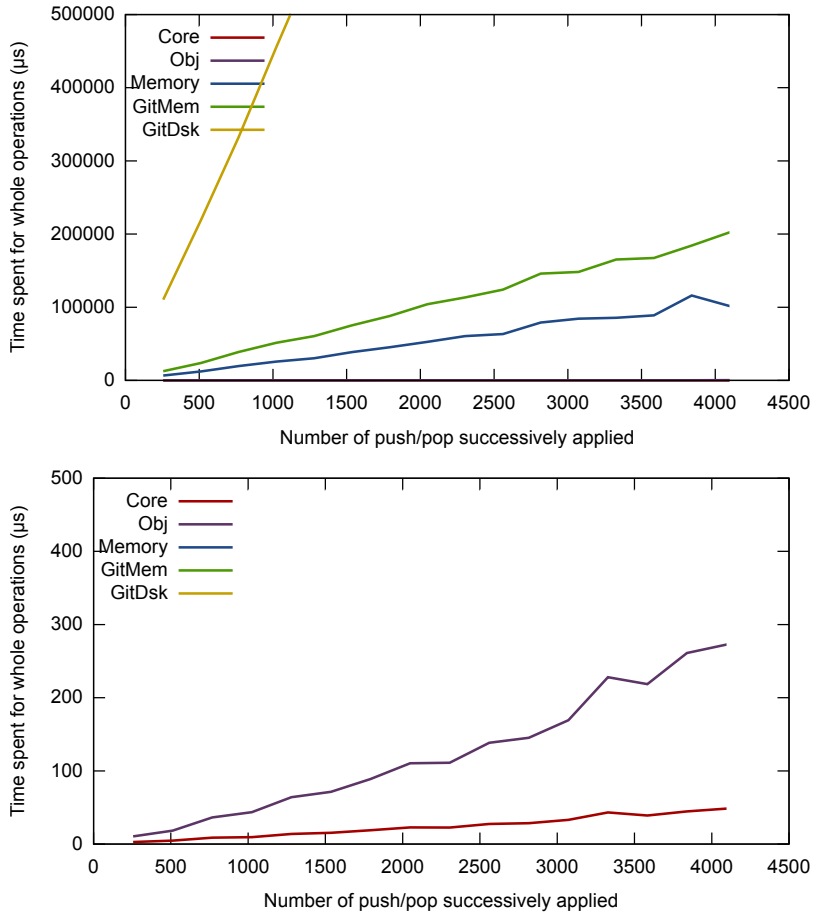


Figure 2.16: Time needed for n push followed by n pop on different backends

2.3.5.2.3 IO cost estimation A last original use of benchmarks is the possibility to determine the time needed for a read and a write on different backends. Indeed, the rope implementation is able to produce some statistics about the number of reads and writes used in each operation. Some results are given in figure 2.17. Aside from highlighting the obvious fact that the cost of an operation is proportional to the number of reads and writes, we can see that the relative proportion of read and write is different in each operation. Knowing the time needed for one operation, these proportions give us a linear system of four independent equations where variables are the time needed for a read and a write, represented in Figure 2.18. The average of the intersection points indicates the solution that we are looking for.

2.3.6 Conclusion and future work

Further work is ongoing on the use of Bloom Filters and DAGs to maintain consistency during concurrent access. We wrote a library[1] of functions enabling a user to synchronize persistent DAGs, with a low cost in term of data exchanged and of complexity. We dealt with the side cases due to false positive in Bloom Filters and tried to find a ”good” hash family function that ensures the user that the false positive rate will be low. The question of choosing an interesting hash function family is very important in our problem because we

[1] https://github.com/samoht/ocaml-bloom-filters

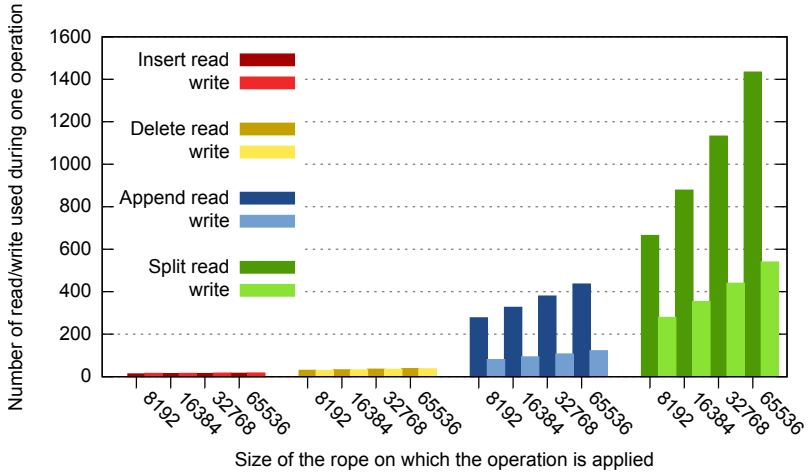


Figure 2.17: Number of read/write used during one operation on a rope of size n.

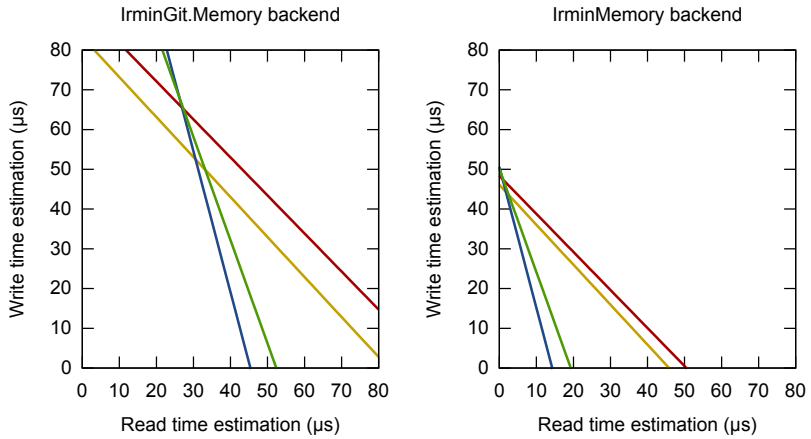


Figure 2.18: Let x be the time needed for one read, y the time needed for one write and d the time needed for one operation on a rope. Then the above curves –where colors match with the previous figure– are the plot of the equation: $x *$ nrb of read $+ y *$ nbr of write $= d$

often have to hash elements, in order to be able to store data or to test the membership of an element in a set. Therefore we want the hash functions to be fast to compute while being independent. In particular we will look for a set of hash functions that hashes integers in the set $\{0, \cdots, m-1\}$:

(i) The multiplication method : we assume that we have a real number generator. We generate the hash family $h_\theta : x \to \lfloor m\text{frac}(x \times \theta) \rfloor$ where $0 < \theta < 1$ is a random real number.

(ii) The less hashing method : we assume that we have two hash functions h_a and h_b hashing integers to the set $\{0, \cdots, m-1\}$, we can generate the hash function family $\mathcal{H} = \{x \to h_a(x) + ih_b(x) \bmod m, i \in \{0, \cdots, k-1\}\}$.

2.4 Trevi

The Trevi work [29] was published in Hotnets last year. Ongoing work is now in a separate project run by George Parisis at Sussex University collaborating with another T2 partner, OnApp.

Fountain coding is central to our proposal as it allows us to provide a storage transport service that is tolerant

to packet loss, but without retransmissions or timeouts. It requires extra computing resources to encode and decode symbols and it has a small penalty in terms of bandwidth due to requiring a slightly larger number of encoded symbols than the initial number of fragments to decode the original information. This overhead can be as low as 5%.

Flow and congestion control

Traditionally, the fountain coding transport model has been push-based. i.e. senders send symbols until all receivers decode the data. In receiver-driven layered multicast, receivers take an active role by subscribing to and un-subscribing from multicast groups that represent different coding layers, according to the network congestion. In future work, the other project will mainly investigate receiver-driven, pull-based communication models where receivers actively manage the rate at which encoded symbols arrive (effectively providing flow and congestion control), by explicitly requesting symbols by receivers. No action is taken when symbols are lost because all symbols equally contribute to the decoding process and, therefore, a new symbol is requested instead, without having to rely on retransmission timeouts. In our proposal senders are requested to send encoded symbols at a rate they can actually cope with, by adjusting the number of symbol requests according to RTT variations and observed loss rates. Congestion can be inferred and avoided. Receivers react when congestion occurs in the network by decreasing the rate at which they request new encoded symbols. Hence, there is no need to extensively buffer packets to avoid congestion-related losses.

2.5 VNF Pool enabled Virtual CPE

NFV aims to evolve the way network operators design, deploy and provision their networks by leveraging standard IT virtualization technologies to move and consolidate a wide range of network functions and services onto industry standard high volume servers, switches and storage. The main idea is therefore to run network functions and services in either datacenters or commodity network nodes, in some cases close to the end user premises. This means that with NFV, network functions that are traditionally implemented on specialized hardware devices are moved to general purpose servers based on virtualized computing platforms, with the possibility to be instantiated or moved to various locations in the network as required, without the need for installation of new equipment. A Virtualized Network Function (VNF) provides the same function as the equivalent network function (e.g. firewall, load balancer) but is deployed as a software instance running on general purpose servers via a virtualization technology. Moreover, operators' networks are populated with a large and increasing variety of proprietary software and hardware tools and appliances. The deployment of new network services in operational environments is often a complex and costly procedure, requiring additional space and power to accommodate new boxes. Moreover, today hardware-based appliances rapidly reach end of life, requiring much of the design integration and deployment cycle to be repeated with little revenue benefit. In this context, the adaptation of network functions and appliances from hardware to software solutions by means of NFV promises to address and overcome these hindrances for network operators.

Here, we focus on the virtualization of the Customer Premise Equipment (CPE), which is emerging as one

of the primary applications of the NFV architecture, with the potential to generate a significant impact on the network operators business. With virtual CPE we refer to the execution in virtual environments of those network functions traditionally integrated in hardware gears at customer premises, e.g. for BGP, Firewall, NAT, etc. In particular, we concentrate on the virtual CPE in the datacenter, used to implement liquid VPN services for business customers using the operators IaaS.

2.5.1 Reference Scenario

Current telco infrastructures are facing the rapid development of the cloud market, which includes a broad range of emerging services and distributed applications. The network edge is under a deep revolution, being stimulated by the recent updates on NFV and SDN. In fact, the operators are looking at their future datacenters as more and more dynamic infrastructures with flexible network architectures, on-demand services for their customers and high elasticity to scale up and down while optimizing performances and resources utilization.

The virtual CPE in the operator's datacenter has been introduced in deliverable D3.1 in its first version. The reference scenario is reported again for sake of completeness of this document in Figure 2.19: a business customer wants to interconnect his sites (e.g. offices, headquarters, labs, etc.) through IP/MPLS VPNs services offered by its network operator and buys also a set of IaaS resources (e.g. VMs, storage areas, etc.) from the same operator. This customer wants to extend these Layer 3 VPNs to incorporate the new virtual assets into a private cloud. This 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 Infrastructure as a Service (IaaS), leveraging proprietary datacenters where latest NFV and SDN solutions can be deployed to ease the provisioning and orchestration of virtualized network functions and services. In this scenario, the virtual CPE, as depicted in Figure 2.19, is an additional VM under the control of the operator, deployed on demand and in an automated way to dynamically control and provision the customer VPN memberships, that is to dynamically join/release customer VMs to the proper VPN. Upon the request from a customer 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 the VPN. This is implemented by running Open virtual Switches (OVS-es) in the datacenter hypervisors, which allow VPNs and IaaS service chaining via software, properly managed and controlled by and SDN and NFV enabled orchestration framework, as detailed in the next section.

With respect to the solution provided in deliverable D3.1, here we present an evolution towards a resilient and high availability virtual CPE version. Indeed, network operators need to be assured that the resilience and availability of their networks are not impaired when VNFs are introduced. Indeed one of the main motivations behind the introduction of NFV technologies in operational environments is to improve, ease, automate and speed up the network functions and services resiliency, by means of flexible high availability mechanisms and deployment of sets of VNF instances grouped in pools to provide the same function in a reliable way. The

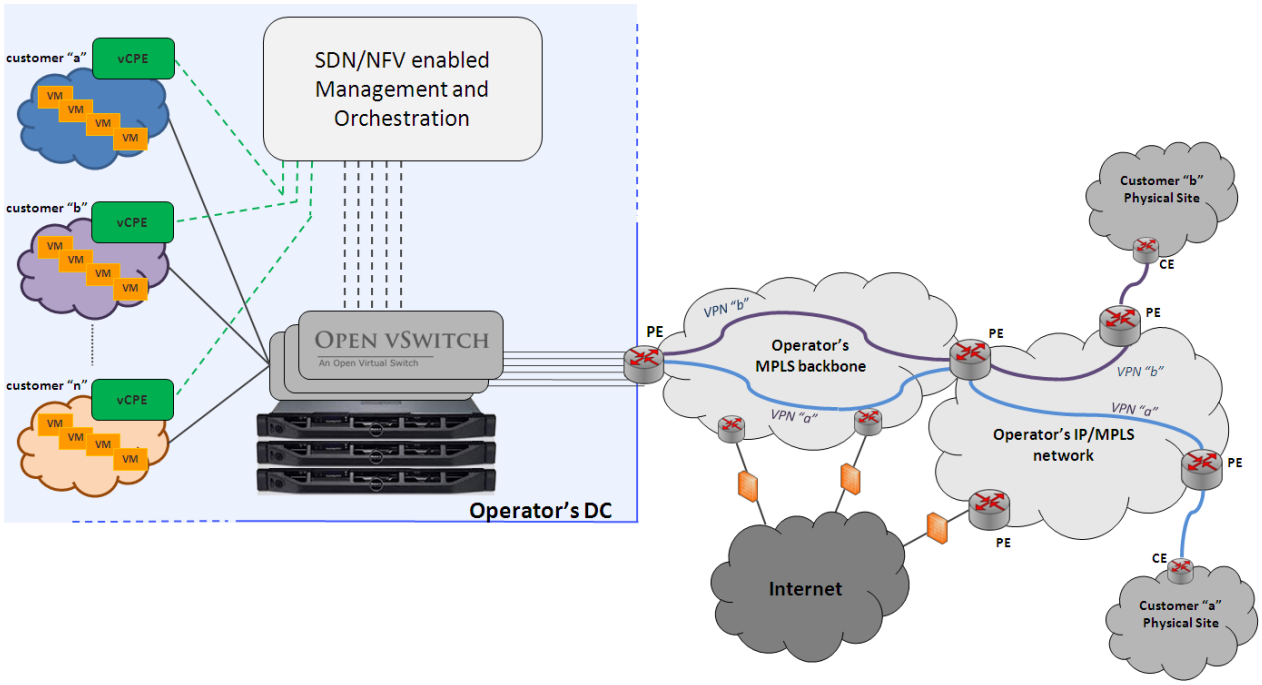


Figure 2.19: VNF Pool Enabled Virtual CPE reference scenario

evolution of the virtual CPE described here goes exactly in this direction: integrate the VNF Pool concepts with the aim of achieving high reliability for virtual CPEs deployed in the operators' datacenters.

2.5.2 Evolution towards a VNF Pool enabled virtual CPE solution

VNF Pool, as defined by IETF and also reported in deliverable D2.3 as a fundamental building block of the Trilogy 2 architecture, aims to provide mechanisms to dynamically manage a set of VNFs (grouped in pools) which provide the same network function in a transparent way for end-hosts and service control entities, and also map the VNF currently in use with the pool it belongs. In the case of the virtual CPE described here, the VNFs are the operator's VMs implementing the virtual CPE network function, the end-hosts are the operator's customers accessing their VMs by means of the extended Layer 3 VPNs, while the service control entity is the SDN and NFV enabled management engine depicted in Figure 2.19 that orchestrates the liquid and highly resilient VPN services offered by the network operator. In this scenario, this evolved VNF Pool enabled virtual CPE is a practical demonstration of how SDN and NFV technologies and concepts 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. On the one hand, SDN focuses on network programmability, traffic steering and multi-tenancy by means of a common, open, and dynamic abstraction of network resources. On the other hand NFV targets a progressive migration of network elements, network appliances and fixed function boxes into VMs that can be run on commodity hardware, enabling the benefits of cloud and datacenters to be applied to network functions. To take full advantage of those benefits, SDN can control the dynamic traffic steering and flexible network provisioning that VNFs VMs require for recovery,

as well as scale-in/out and migration.

2.5.2.1 Extended functional architecture

The VNF Pool enabled virtual CPE described here is an evolution of the virtual CPE introduced in deliverable D3.1. Its functional decomposition is provided in Figure 2.20 where the main functional entities are highlighted along with their interactions. The core part is represented by the OpenDaylight SDN controller (one for the whole datacenter) combined with a set of enhanced network applications provides on-demand flow and routing tables configuration at the OVS and datacenter edge router. The deployment of an SDN controller allows to implement a software driven virtual CPE, with flexible and programmable network functions for VPN and IaaS service chaining. In addition, OpenStack the Cloud Management Platform owned by the network operator and controlling its datacenters (possibly more than one) and is in charge of the on-demand deployment and configuration of the IaaS resources (mainly VMs, storage, etc.) requested by the business customers, by means of the dedicated modules Nova and Neutron. A detailed description for each of the VNF Pool enabled virtual CPE functional module is provided hereunder, in terms of main functionalities, roles and responsibilities.

OpenStack - Cloud Management System

The OpenStack open source cloud management system groups all the sets of hypervisors within a datacenter (or cross multiple datacenters) into pools of resources, which can be used and managed from a single point, i.e. the dashboard of Openstack. It easily provides functions a set of primitives to create virtual machines, configure networks and manage volumes all from a single place. OpenStack is therefore a control layer that sits above all the virtualized layers and provides a consistent way to access everything regardless of the hypervisor technology used (e.g.: KVM, Xen, vmware, etc.). Thus, OpenStack is a framework that enables efficient and dynamic management of virtualization, storage and networking of all the resources with great flexibility and ease, by means of a consistent set of APIs provided by the different sub-systems of OpenStack. Indeed, OpenStack is a composition of several modules, each providing a given function (e.g. Nova for IaaS, Neutron for networking, Ceilometer for monitoring, etc.).

In the context of the VNF Pool enabled virtual CPE, OpenStack is fully controlled and operated by the network operator: it provides the on-demand orchestration of the IaaS resource deployment (the customers' VMs) and the configuration of the virtual CPE functions, in terms of networking and routing. In particular, two OpenStack sub-systems are mostly used for the on-demand and dynamic extension of the customer's VPN to incorporate the IaaS resources in the datacentre: Nova by the OpenStack Heat Orchestration Template (HOT). The HOT provides to the operator a set of REST APIs to seamlessly manage in a coordinated way the provisioning of IaaS and networking resources inside the datacenter, as provided individually by Nova and Neutron respectively. Nova is the project name for OpenStack Compute, a cloud computing fabric controller, the main part of an IaaS system. Nova deploys a set of Compute Controllers to host and manage the computing resources, that in the case of the VNF Pool Enabled virtual CPE are the set of VMs deployed

for either customers' IaaS or virtual CPE functions. Neutron is the networking project in OpenStack, and its main goal is to allow for programmatic management of the virtual networks in multi-tenant environments. It provides REST APIs for programmatic management of virtual networks by the operator and by the cloud tenants, and a pluggable architecture to allow for diverse implementations with extensible design to allow for feature addition and evolution, such as the provisioning of the VPNs configurations for the VNF Pool enabled virtual CPE implementation by means of interaction and integration with the OpenDaylight SDN controller.

OpenDaylight SDN controller

OpenDaylight is an open source platform for network programmability hosted by the Linux Foundation through a combination of components including a fully pluggable controller, interfaces, protocol plug-ins and applications. The core part of the project is the modular, pluggable and flexible controller: it is implemented strictly in software and is contained within its own Java Virtual Machine (JVM) that can run on any platform that supports Java. The northbound and southbound interfaces are clearly defined and documented APIs; their combination allows vendors, service providers, customers and application developers to utilize a standards-based and widely supported SDN platform. The OpenDaylight SDN platform architecture is composed by three main layers: the Network Application Orchestration and Services, the Controller Platform and the Southbound Interfaces and Protocol Plugins.

The Network Application Orchestration and Services is composed by business and network logic applications for network control, provisioning and monitoring. In addition, more complex and advanced orchestration applications needed for cloud, datacenter and virtualization services also reside in this layer. These heterogeneous applications use the open northbound API framework exposed by the SDN controller platform which is implemented by a set of bidirectional REST interfaces. The Controller Platform is the core layer where the SDN abstraction happens. It is composed by a collection of dynamically pluggable modules to perform a variety of network tasks and services. A set of base network services are offered by the SDN controller platform: Topology Manager, Statistics Manager, Switch Manager, Shortest Path Forwarding, Host Tracker. Each of these modules exposes its own northbound APIs to let network applications and orchestration services program the network and controller behaviour. In addition, platform oriented services and other enhanced network services can be implemented as part of the controller platform for enhanced SDN functions. Multiple southbound interfaces are supported in OpenDaylight through an heterogeneous set of dynamically loadable plugins implementing a wide set of southbound protocols, such as OpenFlow 1.0, OpenFlow 1.3, BGP Link State (BGP-LS), NETCONF, Open vSwitch Database (OVSDB), PCEP, etc. These modules are dynamically linked into a Service Abstraction Layer (SAL) which actually provides the SDN controller platform abstraction capabilities. Below the SDN controller platform, a wide range of physical and virtual devices, switches, and routers that provides the actual connective fabric between the network endpoints can be controlled.

OpenDaylight has been selected as the reference controller for the SDN software implementations of the virtual CPE due to its highly extensible and modular architecture that natively deploys a wide set of services,

appliances and northbound API for either datacenter or transport ecosystems. Moreover, the high popularity of the industry-led OpenDaylight project in both scientific and industry communities has also been considered as a crucial term of selection, aiming to generate a high impact in the community. In this VNF Pool enabled virtual CPE implementation, OpenDaylight is mainly used to control and provision the network configurations inside the operator's datacenter: this means that it takes care of configuring the OVSes running in the servers to provision the virtual LANs for each customer (including all its IaaS VMs), allowing the proper isolation across multiple customers. To this purpose OpenDaylight natively supports at its southbound interface a plugin for the OVSDB protocol, that is a management protocol standardized by IETF configuration of virtual switches, including OVS. Also, OpenDaylight enables the virtual CPE VM to peer with other BGP instances to exchange reachability information and Virtual Routing and Forwarding (VRF) tables for the given IaaS VMs.

Virtual CPE Pool Manager

The Virtual CPE Pool Manager is a new function with respect to the virtual CPE described in deliverable D1.1, and its main aim is to provide those high availability and resiliency features according to the IETF VNF Pool architecture, which identifies a VNF Pool Manager as the entity that manage the reliability of the VNF itself, by selecting the active instance and interacting with a Service Control Entity for consistent service provisioning. In the case of the virtual CPE, the Service Control Entity is implemented by the combination of OpenStack (for overall orchestration of the liquid VPN service for operator's customers) and OpenDaylight (for actual provisioning of network resources inside the operator's datacenter). The Virtual CPE Pool Manager is a network application deployed on top of OpenDaylight and triggered by OpenStack to mostly act as a virtual CPE redundancy and reliability entity; after the creation and configuration of the master virtual CPE VM with Nova, it provides some post-configuration functions to instantiate the virtual CPE with the desired degree of reliability and redundancy. This translates into further actions to create and configure additional VMs as backups for the virtual CPE VM, therefore building a virtual CPE Pool. The virtual CPE Pool Manager offers several types and degrees of reliability functions. First, it implements specific functions for the persistence of the virtual CPE configuration, including making periodic snapshots of the whole virtual CPE VM. Moreover, at runtime (i.e. with the liquid VPN service in place), the virtual CPE Pool Manager monitor the operational status and performances of the master virtual CPE VM by registering as a new observer of the same APIs used by the Compute Controllers in Nova to collect notifications about VMs status. The virtual CPE Pool Manager reacts to any failure condition by autonomously replacing the master virtual CPE VM with one of its backup on the pool, implementing a swap of VMs for service recovery purposes. Two options have been identified: cold recovery and hot recovery. In the former, a backup virtual CPE VM, properly configured with the same master configuration, is kept ready (but switched off) to be started when the master VM dies. In this case, the convergence time for the service recovery depends on the BGP convergence time to re-build the routing tables from scratch. In the hot recovery, an active backup virtual CPE VM is kept syn-

chronized with the master one, but not directly connected to the BGP adjacency with the peering PE router. In this case the convergence time for the service recovery is much faster, especially if redundancy methods such as the Virtual Router Redundancy Protocol (VRRP) is used, since the backup VM is already in place and fully synchronized with the master one.

Virtual CPE VM

The virtual CPE VM is allocated on-demand by OpenStack as the master VM to implement the virtual CPE routing functions. It is part of the virtual CPE Pool, that is instead managed by the virtual CPE Pool Manager, and it is created and instantiated on-demand by Nova. In particular, a dedicated master virtual CPE VM is deployed for each customer, along with its associated VMs pool.

The virtual CPE VM hosts an implementation of the BGP protocol, that allows IP/MPLS routers to exchange network reachability information and VRFs: in particular it runs the exterior BGP protocol to define the routing needed to allow customer IaaS VMs traffic to be routed across networks (i.e. from datacenter to customer sites, through the backbone) and join the proper Layer 3 VPNs. This BGP instance inside the virtual CPE VM is implemented with Quagga routing software package that provides TCP/IP based routing services for UNIX with routing protocols support such as RIPv1, RIPv2, RIPng, OSPFv2, OSPFv3, IS-IS, BGP-4, and BGP-4+. Quagga uses an advanced software architecture to provide multi server routing engine with an interactive Cisco-like user interface for each routing protocol that supports common client commands. The core part of Quagga is zebra, which acts as an abstraction layer to the underlying Unix kernel and presents a set of APIs over a Unix or TCP stream to the Quagga protocol clients. Since in our case the virtual CPE, and the BGP protocol instance, are running in a virtual environment (i.e. a VM), the Quagga Forwarding Information (FIB) push interface extensions external component to learn the forwarding information computed by the Quagga routing suite to support our scenario where the virtual CPE router has a forwarding path that is distinct from the kernel, i.e. implemented by the external OVSes and not by the VM kernel. This is provided by a Forwarding Plane Manager (FPM), that is the Quagga component that programs the external forwarding plane; in our virtual CPE implementation, this FPM configures the OVSes by means of the OpenDaylight SDN controller.

Virtual CPE Configurator

The virtual CPE configurator is a network application deployed on top of the OpenDaylight SDN controller and hosts the logic for the provisioning of network resources inside the datacenter (mostly the OVSes), acting as a bridge between OpenStack and OpenDaylight). The Virtual CPE Configurator is triggered by Neutron mostly to provision the network configurations for the customer IaaS and the VPN service chaining. First, by means of the northbound APIs exposed by OpenDaylight, it performs the configuration of the virtual CPE BGP routing daemons running in the virtual CPE VMs, mainly in terms of VRFs to be flooded. Moreover, again through OpenDaylight, it sets flows and routing tables in the OVSes for both VMs and BGP protocol traffic purposes.

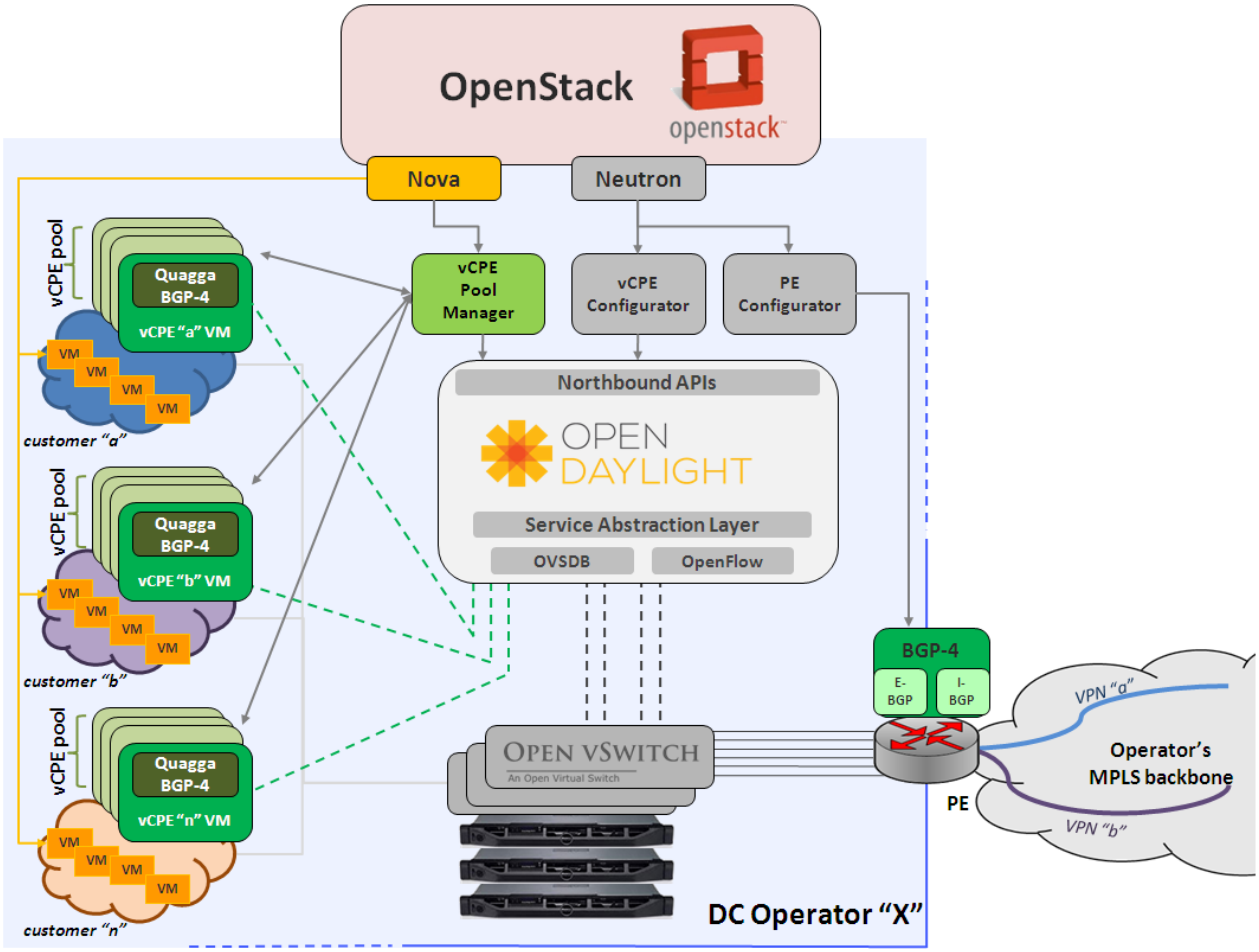


Figure 2.20: VNF Pool Enabled Virtual CPE functional split

PE Configurator

The PE configurator is another network application running on top the OpenDaylight SDN controller. The main role of the PE configurator is to set the VRFs in the datacenter edge router (i.e. the PE router) upon the request coming from the CMS. This allows the traffic generated by the customer's VMs be associated to the proper VPNs. Optionally, the PE configurator may perform some additional MPLS management actions on the PE router to dynamically create or modify MPLS tunnels/LSPs to support new VRFs and establish new VPNs.

3 Cross layer liquidity

This section details the cross-layer liquidity tools from the group of advanced liquidity tools. This research is based on the cross-layer mechanism found in D1.2. Cross-layer liquidity tools explore the possibility to interconnect control over resources at different levels of the system architecture or networking stack.

The tools described in this category are HACK, MultiWifi, PVTCP and GRIN. Every tool has features that could include it in a different cross-liquidity category, we will explain in the following paragraphs why those tools are inherently cross-layer liquidity tools.

HACK, described in section 3.1, is a tool that performs cross-layer optimizations to TCP traffic on Wifi networks by carrying TCP ACKs within WiFi's link-layer acknowledgment. HACK is a cross-layer liquidity tool because it bridges data and transport layers in order to improve the medium utilization.

MultiWifi, see section 3.2, offers a novel way of handling the layer 2 handoff problem of connecting to an Access Point at layer 4. The project improves mobility by enabling a mobile client to lock into a single Wifi channel and associate automatically to all Access Points in it's range without using an explicit layer 2 handover.

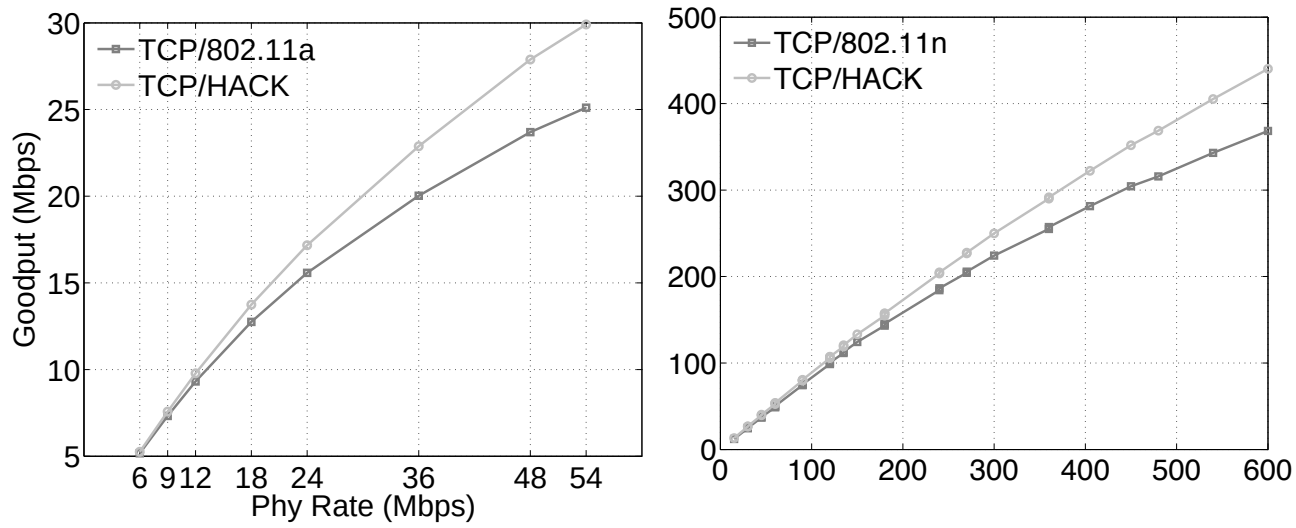
The concept of PVTCP is improved through the Shushi tool, see section 3.3, by adding a security mechanism for key exchange that is used to create a closer link between the application layer and the underlying transport layer.

GRIN, see section 3.4, contributes to cross-layer liquidity by taking advantage of MPTCP in datacenters. MPTCP has the capability of using all network paths in a datacenter and improving the network overlay between nodes.

3.1 HACK

Physical layer speeds have increased in WiFi from 802.11b's 11 Mbps to the Gbps rates of emerging 802.11ac. Despite these gains, WiFi's inefficient MAC layer limits the achievable end-to-end throughput. Namely, the current variants of 802.11 require a sender to first sense the medium to be idle for a randomly chosen interval each time it wishes to transmit, to desynchronize would-be concurrent senders. Unfortunately, the mandatory idle period during each medium acquisition now dominates the duration of a packet's transmission. To use a concrete example, Enhanced Distributed Channel Access (EDCA) in 802.11n [18] enforces an average idle period of 110.5 μs before a frame's transmission, whereas a 1500-byte payload itself lasts only 80 μs at 150 Mbps. Each frame's link-layer acknowledgement (LL ACK) consumes further channel capacity.

As the physical-layer bit-rate increases the pre-transmission idle period remains the same, causing greater inefficiency. If a 600 Mbps 802.11n sender sent single frames in this fashion, it would only achieve 9% of the theoretical channel capacity. Moreover, WiFi senders back off exponentially after a failed transmission, and so incur even longer mean pre-transmission idle periods under contention, further reducing medium efficiency.



(a) (b)

Figure 3.1: Theoretical goodput for 802.11a (a) and 802.11n (b) rates. In (b), theoretical TCP/HACK achieves an 8% improvement on average over TCP/802.11n for physical rates lower than 100 Mbps.

In an effort to amortize the significant overhead of medium acquisition over multiple data frames, 802.11n's MAC protocol batches multiple data frames into a single aggregate MAC protocol data unit (A-MPDU), and incurs only a single medium acquisition for each such batch. 802.11n further aggregates the LL ACKs for the data packets in a received A-MPDU into a single LL Block ACK.

Despite these work-arounds, frame aggregation and LL Block ACKs leave significant medium acquisition overhead for TCP ACKs. TCP traffic is inherently *bidirectional:* a TCP receiver typically transmits a single TCP ACK packet for every pair of TCP data packets it receives. Not only do TCP ACKs incur further expensive medium acquisitions by the TCP receiver—they run the risk of colliding with the TCP data sender's transmissions as well.

In the following sections, we propose TCP/HACK (Hierarchical ACKnowledgment), a system that applies cross-layer optimization to TCP traffic on WiFi networks by carrying TCP ACKs within WiFi's link-layer acknowledgments. LL ACKs are an ideal vessel for carrying TCP ACK information on the reverse path without incurring a costly medium acquisition. By eliminating all medium acquisitions for TCP ACKs in unidirectional TCP flows, TCP/HACK significantly improves medium utilization, and thus significantly increases achievable capacity for TCP workloads. We demonstrate these improvements through simulation of up to 10 competing TCP flows on a 150 Mbps 802.11n network. The simulations illustrate that TCP/HACK improves aggregate throughput by up to 22% over TCP on "stock" 802.11n. Additionally, through an evaluation of a prototype online, wire-speed implementation of TCP/HACK for 802.11a on the SoRa software-defined radio platform [39], we illustrate that TCP/HACK improves aggregate throughput up to 32% over TCP on "stock" 802.11a.

3.1.1 WiFi MAC Overhead

Consider a typical WiFi scenario, where a single 802.11a or 802.11n client downloads a large file from a remote TCP sender. We assume throughout that the TCP receiver uses delayed ACK, and thus generates one

TCP ACK packet for every two TCP data packets it receives.[1]

In Figures 3.1(a) and 3.1(b), the curves labeled "TCP 802.11{a,n}" show analytical predictions of the throughput a single TCP downloader achieves as a function of physical-layer bit-rate on lossless 802.11a and 802.11n networks, respectively. These analytical predictions are based on the parameters of the 802.11a and -n MACs. A detailed derivation of the capacity of the 802.11n MAC layer may be found in [24]; we do not repeat it here. The calculation for 802.11a is similar. (Figures 3.1(a) and 3.1(b) also show the improved throughput achieved by HACK, our modified 802.11 MAC protocol that carries TCP ACKs in link-layer ACKs, which we describe in Section 3.1.2.)

Note that for "stock" 802.11a and -n, the achievable TCP throughput is a progressively smaller fraction of the physical layer bit-rate as the latter increases. Time spent on non-payload overhead for each medium acquisition is to blame. In 802.11a, these overheads include the durations of Distributed Coordination Function InterFrame Space (DIFS) and the contention window (both before a data frame's transmission), the data frame's preamble, the Short InterFrame Spacing (SIFS) interval between data frame and LL ACK, and the LL ACK itself.[2]

As noted earlier, 802.11n aggregates data frames into A-MPDUs so as to amortize medium acquisition overhead over many frames, and combines multiple LL ACKs into Block ACKs in response. The results in Figure 3.1(b) include the application of these techniques, and show that while they reduce 802.11a's overhead, TCP still suffers progressively greater throughput limitations *vs.* the physical-layer rate because of the overhead of medium acquisitions for TCP ACKs.

3.1.2 HACK in Overview

Let us first consider how TCP/HACK works in the simpler case of 802.11a, without batching of packets into A-MPDUs. When a regular TCP client receives a TCP data packet, its network stack generates a TCP ACK and enqueues it for transmission by the WiFi NIC.

Under TCP/HACK, a client does not immediately enqueue a TCP ACK for transmission. Instead, the client compresses each TCP ACK and appends them to a compressed frame that it builds. When the next data packet from the AP arrives, the client encapsulates the compressed TCP ACK frame within the returning LL ACK, *effectively avoiding all medium acquisitions for the corresponding TCP ACKs.* The AP recognizes an "augmented" LL ACK, which it decompresses. It then reconstitutes the encoded TCP ACKs, and forwards them upstream.

Now let us consider 802.11n, where data packets can be aggregated into a single batched A-MPDU, and link-layer ACKs take the form of a Block ACK that includes a bitmap indicating which packets from the A-MPDU were received. On "stock" 802.11n during a TCP download the normal repeating pattern will then be:

[1] Note that this assumption is the best case for the efficiency of the status quo WiFi MAC—were a delayed ACK not used, a TCP receiver would generate twice as many ACK packets, and the WiFi MAC would incur significantly more medium acquisitions.

[2] 802.11n's parameter names and values differ slightly (*e.g.,* AIFS instead of DIFS); the overall scheme of per-medium acquisition overhead does not.

(i) one A-MPDU from AP to client containing TCP data.

(ii) one Block ACK from client to AP.

(iii) one A-MPDU from client to AP containing TCP ACKs.

(iv) one Block ACK from AP to client.

To eliminate medium acquisitions for TCP ACKs in 802.11n, we would like a TCP/HACK client to encapsulate all the TCP ACKs generated in step (iii) in the Block ACK sent in step (ii), and thus avoid step (iv) entirely.

In practice, the arrival of an A-MPDU containing a batch of TCP data packets will cause the client's OS to generate a burst of TCP ACK packets in step (iii) *after* the Block ACK has departed for that A-MPDU. These TCP ACKs arrive at the client's transmit queue where they are compressed and concatenated, waiting for the arrival of the *next* A-MPDU from the AP. The client will then append this compressed frame to the Block ACK it sends the AP in step (ii).

Although the description above is for downloads, the design is in fact symmetric; we envisage TCP/HACK as especially useful for wireless backup to LAN-attached storage, such as Apple Time Capsule.

3.1.3 Cross-Layer Nuances

We now refine our design to handle the subtle cross-layer interactions that arise between TCP and 802.11.

In principle, we would like to encapsulate TCP ACKs on the link-layer ACKs of the TCP packets they acknowledge. For example, if a batch containing TCP packets 1-64 arrives, the client would like to piggyback the TCP ACKs for packets 1-64 on the Block ACK for that batch. However, the 16μs SIFS interval between receiving data and sending the link-layer ACK or Block ACK is too short for the host's TCP stack to turn around the TCP ACKs, compress them, and DMA them to the NIC. For HACK to be practical, the compressed ACKs will have to wait until the next data arrives, and piggyback on its ACK or Block ACK. It turns out that this significantly complicates the dynamics of TCP/HACK and we will explore the consequences.

Figure 3.2 illustrates this process[3]. In response to a batch containing TCP packets 1 and 2, TCP ACKs 1 and 2 arrive at the client transmit queue too late to be carried on that batch's Block ACK. Instead, the TCP ACKs are compressed but not yet sent. When the next batch carrying TCP packets 3 and 4 arrives, its Block ACK can now carry the compressed frame with TCP ACKs 1 and 2. The AP then reconstitutes the full TCP ACKs and passes them up the network stack.

So long as TCP data packets continue to arrive, there is a steady stream of Block ACKs on which to piggyback compressed TCP ACK frames: all TCP ACKs are carried as HACK packets and no vanilla TCP ACK packets need to be sent. But what happens if no further data packets arrive? The client cannot retain the TCP ACKs for too long, or it will cause the TCP sender to time out and retransmit. Thus, after some time period, the

[3]For simplicity it assumes that delayed TCP ACKs are disabled

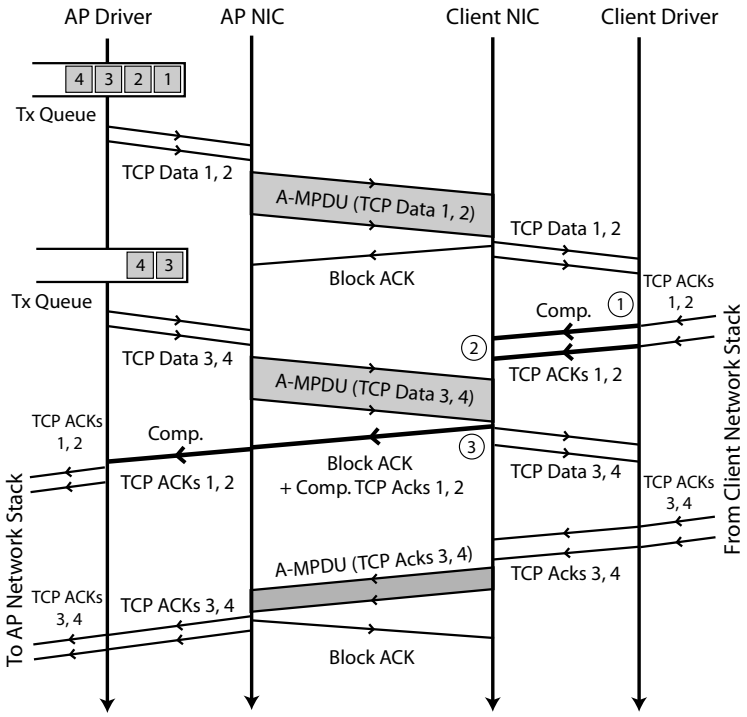


Figure 3.2: Interaction between A-MPDUs, Block ACKs and encapsulated HACK packets

client must send uncompressed vanilla TCP ACKs in the normal way. In Figure 3.2, TCP ACKs 3 and 4 meet this fate, and are in turn Block-ACKed.

Figure 3.1 summarizes the theoretical upper bound on TCP/HACK throughput on 802.11a (Figure 3.1(a)) and 802.11n (Figure 3.1(b)). The curves assume that the sender transmits the largest possible A-MPDUs,[4] that HACK manages to encapsulate all TCP ACKs in TCP Block ACKs, and that the compression is performed using the algorithm in Section 3.1.4. As the bit-rate increases, TCP/HACK significantly improves capacity, with a 20% improvement seen at 600 Mbps on 802.11n. In reality, the improvement can actually exceed that shown in the figure, as TCP/HACK can get closer to its bound than vanilla TCP can. This is due to collisions between TCP data packets and vanilla TCP ACK packets, a problem HACK sidesteps.

To HACK or not to HACK?

To maximize the benefits, TCP/HACK packets should be used whenever possible. But TCP ACKs must not be delayed when no more TCP data packets will arrive. How long should the client retain these TCP ACKs before giving up and sending them natively?

There are several reasons no more packets may arrive, including that the sender has stopped sending, but with 802.11n, the principal reason is the adverse effect of A-MPDUs on TCP's ACK clock. On a busy AP or during slow start, it is common for the entire TCP congestion window to be queued at the AP and then to be sent to the client in a single A-MPDU. An entire congestion window of TCP ACKs is generated and compressed, and these now sit at the client, waiting for the arrival of another incoming data packet so they can be send on

[4] A-MPDU length is limited either by the 64 KByte A-MPDU bound or at lower bitrates by 802.11n's 4 ms transmit opportunity limit.

its Block ACK. As the congestion window is full, this next packet never arrives and the connection stalls until TCP's retransmit timer fires. On 802.11a, which lacks aggregation, we don't often see this problem, but it is normal during slow start when 802.11n batching is used. We consider the following three different designs to address these concerns:

Explicit Timer A naive approach would be to have TCP/HACK time out and fall back to sending regular ACKs after a delay. In practice there is no good delay value that can be chosen, since the client cannot know the RTT and congestion window at the TCP sender, how the sender's packets will be spaced throughout the RTT, nor if the AP will suddenly start sending to another client.

Opportunistic HACK A more adaptive approach is not to explicitly delay TCP ACKs at all, but rather be opportunistic. When the wireless link is the bottleneck, the next downstream data batch will contend with the upstream TCP ACK batch. If the downstream batch wins, HACK can be used, but otherwise vanilla TCP ACKs will be sent. Such a design may often squander the opportunity to use HACK, but it has the virtue of seeming simple—until one considers the complexity of the NIC-network driver interface needed to implement it.

The MORE DATA Bit In Figure 3.2, initially there are four data packets queued at the AP. When the AP forms the first batch containing TCP data packets 1 and 2, it already knows more data will be sent to that client, as it already has packets 3 and 4 in its queue. So long as the AP has more packets queued than will fit in a batch, it knows that it is safe for the client to save up compressed ACKs waiting for the next batch. The AP simply tells the client that there is *more data* coming by setting the MORE DATA bit in the 802.11 header of the A-MPDU.[5] When the client sees this flag, it latches this state and will not transmit any more non-encapsulated TCP ACKs until the next data packet arrives, when it can use HACK to send them.

3.1.4 HACK in Practice

In the preceding sections, we have presented a conceptual description of TCP/HACK, but several questions concerning the practicality of this conceptual design remain unanswered. First, how realizable is TCP/HACK given current systems and hardware? In particular, how should TCP/HACK's functionality be divided between a station's network interface card (NIC) hardware and NIC device driver? Finally, what manner of compression should TCP/HACK employ to reliably encode the TCP ACKs?

3.1.4.1 Driver and NIC Functionality

We realize TCP/HACK (including the MORE DATA mechanism) with very few changes to a station's 802.11 NIC. The main strategy is to implement the bulk of TCP/HACK within the NIC's driver, as we demonstrate using the example shown in Figure 3.2. Our discussion is in the context of a modern Linux wireless driver, such as the Atheros ath9k driver.[6]

AP (data transmission) The only modification needed to the AP when transmitting data packets is to set the

[5] This bit exists in stock 802.11 to assist with power saving. HACK uses this bit irrespective of whether power saving is enabled.
[6] `http://wireless.kernel.org/en/users/Drivers/ath9k`

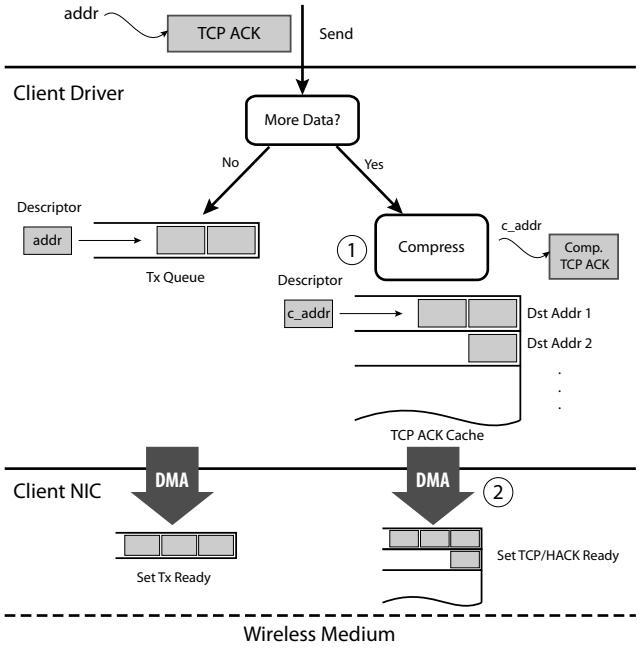


Figure 3.3: Client-side TCP/HACK compressing a TCP ACK, ready to be sent on the link-layer acknowledgment of the next frame.

MORE DATA flag when there are more packets remaining in the transmit queue for the same client.

Client The client's driver needs to determine when it can use TCP/HACK and when it must send TCP ACKs normally. In Figure 3.2, on receiving packets 1 and 2, the client's NIC also passes the MORE DATA state to the driver. The client TCP stack acknowledges the data, generating TCP ACKs 1 and 2, and puts them in the transmit queue at point ①.

Figure 3.3 shows what happens at points ① and ② from Figure 3.2 in more detail. If the driver is not in the MORE DATA state, it simply enqueues these ACKs normally. However, if MORE DATA is set, it compresses the arriving TCP ACKs and creates corresponding buffer descriptors. A separate buffer descriptor chain per destination address is needed to match compressed TCP ACKs with Block ACKs for that destination.

At point ② the driver DMAs the buffer descriptor chain to the NIC. The NIC maintains this table of compressed TCP ACK descriptors separately from normal transmission descriptors. Finally, the driver sets a flag in the NIC to indicate that TCP/HACK is ready.

Figure 3.4 shows what happens when the next batch from the AP arrives at the client. If the TCP/HACK flag indicates "ready," the NIC uses the corresponding descriptors to DMA the compressed TCP ACK frames to the card. It concatenates these frames, and appends them to the returning Block ACK at point ③. Recall that the NIC normally fires an interrupt when it receives data packets. In this case, the interrupt must also indicate whether the NIC succeeded in sending the compressed ACKs.

This design also copes with the race condition where the batch carrying packets 3 and 4 arrives with the MORE DATA flag not set before the driver has succeeded in conveying compressed TCP ACKs 1 and 2 to the NIC. In this case, the TCP/HACK "ready" check will fail. The NIC sends a normal Block ACK and signals to

the driver a TCP/HACK failure in the receive interrupt. The driver now is free to re-enqueue the TCP ACKs on the transmit queue for normal transmission.

AP (ACK reception) Finally, the AP needs to recognize and decompress the "augmented" Block ACKs. The task of recognition falls to the AP's NIC, which extracts the compressed TCP ACK frame from the received Block ACK, adds it to the transmit complete report and interrupts to indicate transmit complete. The driver extracts the compressed TCP ACK frame, decompresses and reconstitutes the TCP ACKs, and forwards them upstream.

3.1.4.2 Compression

A critical component of the design is choosing a compression method for TCP ACKs. As 802.11a and -n transmit LL ACKs at one of the slower basic rates, *e.g.* 6 Mbps, it is desirable to minimize the size of the TCP ACK information appended to LL ACKs. Moreover, the 802.11a and -n MAC protocols' DIFS and AIFS intervals protect "stock" LL ACKs from collisions. Ideally, the compressed ACK information that HACK appends to LL ACKs should be short enough to fit within DIFS and AIFS, to avoid risking a collision.[7] We would like to leverage the redundancy within TCP and IP headers across consecutive TCP ACKs. Since most of the TCP/IP header fields remain static for a particular flow, they can be cached at the compression and decompression endpoints.

To encode TCP and IP header fields reliably, TCP/HACK uses *Robust Header Compression (ROHC)* [35] to efficiently condense TCP/IP segments. ROHC supports the most popular TCP options like Timestamps and Selective Acknowledgments (SACK), and defines the notion of contexts, each with a particular identifier (CID). A context for TCP/HACK's purposes maps nicely to a particular TCP flow. In addition to caching static fields like the TCP/IP five-tuple at the endpoints, ROHC losslessly compresses the dynamic fields like the TCP Sequence and ACK numbers.

TCP/HACK-specific ROHC optimizations Since TCP/HACK applies ROHC in a specific context, we make the following simplifications:

(i) We do not explicitly send Initialize-Refresh (IR) packets from the TCP client to the AP. To initialize a new context, the client can simply send uncompressed TCP ACKs outside of the TCP/HACK mechanism. The AP will consequently store the necessary state for the new context and assign it the correct CID.

(ii) The client and AP need not exchange any messages to agree upon a new CID for an emergent flow. Instead CIDs are computed independently at each endpoint. The client's driver on receiving a TCP ACK for a new flow computes the MD5 [32] hash over the ACK's 5-tuple and selects the lowest byte as the CID.

[7] In our simulations in Section 3.1.5.3, we find that 98.5% of the LL ACKs carrying ROHC-compressed TCP ACKs fit within AIFS for best-effort traffic. For the few that don't fit, the sender may either split the compressed TCP ACKs across multiple LL ACKs (ensuring each LL ACK is fully protected by AIFS) or it may send them all on a single LL ACK (risking a collision with a hidden terminal). Our simulator does the latter; there are no hidden terminals in the scenarios we simulate.

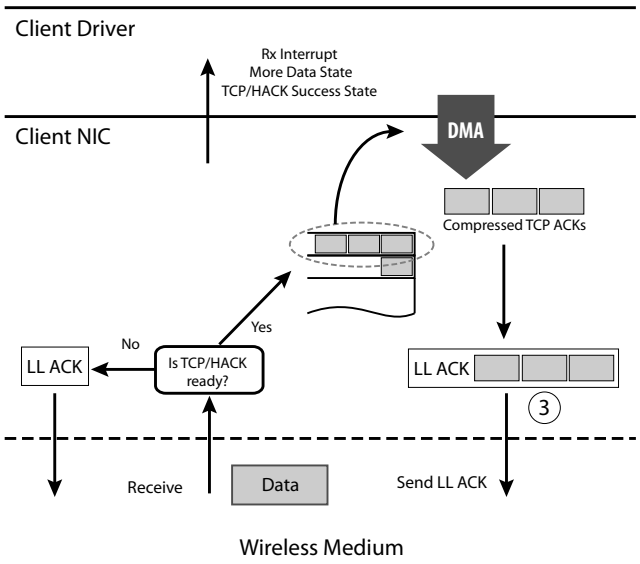


Figure 3.4: Client-side TCP/HACK receiving a batched frame from the air and including compressed TCP ACK frames in the corresponding link-layer acknowledgment.

(iii) Compressed TCP ACK packets encapsulated within link-layer ACKs require a new mechanism to deal with losses outside of sending explicit ROHC feedback packets. We will refrain from going into detail here, but refer the reader to [34] to understand how TCP/HACK handles losses.

With ROHC, a driver can shrink a TCP ACK to about 4 bytes, or even 3 bytes if the associated flow transmits a constant payload size (*e.g.* for large file downloads) [35].

3.1.5 Evaluation

We evaluate TCP/HACK through a combination of simulation in ns-3 [26] and experiments with a real-world implementation for the SoRa software-defined radio platform [39]. We simulate TCP/HACK for 802.11n in ns-3, while our SoRa implementation is for 802.11a, as the public SoRa release does not support 802.11n.

3.1.5.1 SoRa Implementation

We implemented TCP/HACK including the MORE DATA bit and ROHC compression for the SoRa user-level physical layer on Windows 7. Hardware limitations of our SoRa radio boards require us to run 802.11a in the 2.4 GHz band, but this does not affect protocol behavior.

One quirk of the SoRa platform bears mention. We have found that SoRa receivers sometimes return 802.11 link-layer ACKs later than the 802.11 specification's ACK timeout interval, causing spurious link-layer retransmits and backoffs. To avoid this performance hit, we increased the 802.11 ACK timeout to accommodate SoRa's late LL ACKs. The net effect of these delayed LL ACKs is that at 54 Mbps, our SoRa implementation only achieves 87% of the theoretical throughput across all protocols. We confirmed though simulation that this change does not significantly affect the relative benefit of TCP/HACK over regular 802.11a, but the absolute performance numbers are slightly lower.

Testbed Our three wireless nodes each have four-core Intel Core i7 CPUs, between 8–24 GB of RAM, and a PCI Express SoRa radio control board. One acts as the AP and the other two act as clients. We operate the

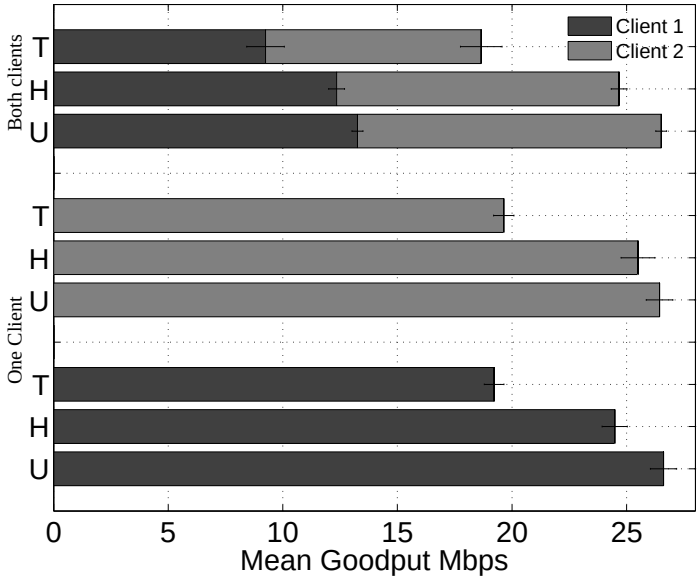


Figure 3.5: TCP throughput with stock 802.11a (T), TCP with HACK (H), and UDP (U) with stock 802.11a, with 1 and 2 clients.

SoRa interfaces in ad hoc mode to eliminate periodic beacon transmission. We run experiments on 802.11g channel 14 (2.484 GHz) in an open-plan office environment. We use *iperf* to generate TCP data streams with a 1500 byte MTU and send at 54 Mbps, the highest 802.11a rate.

3.1.5.2 SoRa Results

Besides demonstrating a successful implementation as evidence of TCP/HACK's practicality, we wish to answer several questions experimentally:

- Are TCP/HACK's capacity benefits in line with theoretical predictions?
- When an AP sends TCP flows to two clients, does TCP over 802.11a suffer collisions between clients' TCP ACKs, and if so, does TCP/HACK offer a performance benefit partly by eliminating such collisions?

Baseline Comparison Figure 3.5 compares the application-level throughput achieved by TCP/802.11a and TCP/HACK for bulk downloads, with UDP/802.11a for comparison. Each bar shows a different experiment: sending to one or both clients, using TCP over HACK, TCP over stock 802.11a or, as a control experiment, unidirectional UDP, which gives an upper bound on usable capacity. The data is the mean over five different 120-second runs; error bars show standard deviation.

Client 1's throughput is slightly less than Client 2's because it suffers a greater packet loss rate, even when only one flow is active. UDP's unidirectional data minimizes medium acquisitions, and achieves the greatest throughput possible on SoRa with link-layer ACKs enabled. In an ideal 802.11 MAC, UDP would achieve 30.2 Mbps; on SoRa, UDP averages 26.5 Mbps across the three experiments. SoRa's link-layer ACK delays alone reduce the attainable throughput to 28.1 Mbps, and our UDP measurements approach that figure.

If TCP/HACK encapsulated all TCP ACKs in LL ACKs, it would achieve almost the same throughput as UDP (though UDP's packet headers are smaller). In practice, TCP/HACK's single-client throughput of 25.0 Mbps (mean of C1 and C2) is very close to the UDP benchmark. TCP/802.11a only achieves 19.4 Mbps in this scenario. TCP/HACK improves performance by 29% and 32.2% in the one- and two-client cases respectively. Both TCP/HACK and TCP/802.11a are fair.

		UDP/ 802.11a	**TCP/ HACK**	**TCP/ 802.11a**
Client 1	no retries	99%	97%	87%
	1 or more	1%	3%	13%
Client 2	no retries	99%	98%	88%
	1 or more	1%	2%	12%
Both	no retries	99%	98%	86%
	1 or more	1%	2%	14%

Table 3.1: Percentage of frames successfully sent on the first attempt (no retries) and after one or more retries, when the AP is sending to Client 1 and Client 2 alone, and both clients at the same time, using UDP/802.11a, TCP/HACK, and TCP/802.11a.

Where do TCP/HACK's savings come from?

We note with interest that TCP/HACK improves throughput more than predicted analytically in Section 3.1.1. That prediction focused solely on saving medium acquisitions for TCP ACKs. In Table 3.1 we show the percentage of frames received after the first transmission, and the percentage that required one or more retransmissions. We see that TCP/802.11a experiences far more link-layer retransmissions than TCP/HACK or UDP/802.11a. These retransmissions occur because of collisions between TCP ACKs sent by clients and TCP data packets sent by the AP. TCP/HACK obviates most (but not all) of these TCP ACKs, and so significantly reduces the number of retransmissions needed. TCP/HACK not only eliminates costly channel acquisition overheads, but by encapsulating TCP ACKs in LL ACKs, also incurs fewer collisions.

3.1.5.3 Simulation Results

We now examine how TCP/HACK interacts with frame aggregation, with a larger number of clients than possible in our testbed. To this end, we implement A-MPDU support and TCP/HACK in ns-3. We evaluate both the opportunistic and MORE DATA variants of HACK described in Section 3.1.3 to verify that the the latter outperforms the former as hypothesized.

We simulate multiple WiFi clients scattered randomly within a circle of 10-meter radius centered on the AP. Our aim is to model the scenario where several clients connect via 802.11n WiFi to a server located nearby on a high-speed LAN. We present results modeling an 802.11n single-antenna setup using data packet and link-layer ACK bit-rates of 150 Mbps and 24 Mbps, respectively. The wired link between the server and the AP has a latency of one millisecond and a bit-rate of 500 Mbps.

To glean the benefits of the MORE DATA scheme, we would like AP's transmit queue to contain at least 126 packets per flow. We choose this number so that the AP may buffer of up to three batches of 42 packets per client, accounting for some variability in the A-MPDU size in the presence of TCP retransmissions. To avoid

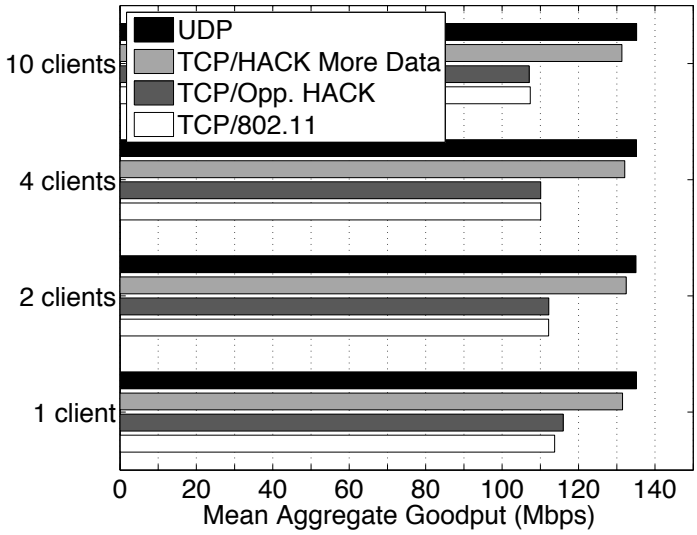


Figure 3.6: TCP goodput for different transmission schemes with 1–10 clients, and UDP for comparison.

adverse "buffer bloat" effects [13], the transmit queue should not be too large in the case of one flow, but rather grow as the number of flows increases. A large buffer in our system would cause an excessive loss of packets when slow start overflows the buffer, with or without TCP/HACK. With ten clients, the AP's transmit queue would be 1260, which is reasonable since Linux drivers usually use buffer sizes of 1000 packets.

TCP/HACK *vs.* TCP/802.11n To determine the benefit of TCP/HACK and its constituent parts, we compute the aggregate goodput for TCP flows sending 1460 byte packets, averaged across five simulated runs per experiment. To mitigate phase effects with multiple clients, we stagger the starts of clients' downloads. As such, we compute the aggregate goodput over the steady-state portion of the runs, once all the clients have more or less exited slow start.

Figure 3.6 shows that UDP maintains a roughly constant goodput as the number of downloading clients varies, as expected. As a unidirectional protocol, UDP's performance is minimally affected by the number of clients competing for the link. In contrast, the goodput of TCP/802.11n decreases slightly as the number of downloading clients increases. Although the AP elicits TCP ACK packets from clients in turn, there is still a chance that two or more clients' TCP ACKs can collide, or that a TCP ACK can collide with a data packet from the AP. These collisions account for the lower measured goodput than that predicted in Section 3.1.1.

We note with surprise that Opportunistic TCP/HACK does not significantly outperform TCP/802.11n: this most naïve implementation of HACK sends few compressed TCP ACKs in LL ACKs, and mostly regular TCP ACKs. It therefore does not achieve a TCP goodput closer to the physical rate.

Role of MORE DATA Bits We now turn our attention to the bars labeled "TCP/HACK More Data" in Figure 3.6. We observe that the MORE DATA variant of TCP/HACK achieves the most pronounced throughput gain over unmodified 802.11n. While simple, the MORE DATA mechanism is crucial to TCP/HACK's success in reducing medium acquisitions, and gives rise to goodput improvements between 15% for one client and 22% for ten clients at the physical rate of 150 Mbps.

Lossy Environment We next evaluate TCP/HACK under different SNR regimes. In addition to providing a wider spectrum of comparison between TCP/HACK and TCP/802.11n, these experiments will verify whether the HACK protocol can operate in a lossy environment and avoid stalls due to recurring TCP timeouts.
We begin with a setup similar to that described above, and then place a single client at varying distances from the AP in order to simulate a decreasing set of SNRs. In lieu of simulating bit rate adaptation explicitly, at each particular distance we simulate a download of a 100 MB file at a rate selected from a range of 802.11n high throughput rates. This range corresponds to rates which are achievable using a 40 MHz channel, 400 ns guard interval and one antenna. The corresponding LL ACK rates are chosen from the set of basic rates (6, 12 and 24 Mbps) according to the rules outlined in the 802.11n specification. To emulate a real system, we applied the 4 ms transmit opportunity limit to all transmissions, therefore limiting the size of A-MPDU packets for experiments using lower physical rates. At each distance/physical rate combination, we computed the average TCP goodput (including slow start) over five runs.
Figure 3.7 shows the average TCP goodput for TCP/HACK and TCP/802.11n. It plots a separate dashed curve per 802.11n physical rate for TCP/HACK. We use these curves to compute the envelope (in black), which indicates the best goodput achievable by an ideal bit rate adaptation algorithm. Similarly we plot the corresponding envelope for regular TCP/802.11n (the separate rate curves for TCP/802.11n are not shown).
Our simulations indicate that TCP/HACK functions correctly in a lossy environment and does not elicit any decompression CRC failures. Moreover, TCP/HACK improves TCP goodput by an average of 12.6% across the range of SNR values. Figure 3.7 shows that as the physical rate drops, the relative improvement increases slightly for the cases where the transmit opportunity limit reduces the number of packets a station can possibly transmit in an aggregate. Recall that 802.11n uses aggregation to amortize medium access costs, therefore we expect a better goodput gain for TCP/HACK over regular TCP at these rates. Similarly, as the physical rate increases past 90 Mbps, the overall improvement increases slightly to about 14%, because the 802.11n medium access delays now consume a larger portion of the transmission time relative to data.

3.2 MultiWifi

Traditional WiFi mobility techniques (as with all other L2 mobility mechanisms) are based on the concept of fast handover: when a mobile client exits the coverage area of one Access Point (AP), it should very quickly find another AP to connect to, and quickly associate to it. There is a great wealth of research into optimizing fast handover including scanning in advance, re-using IP addresses to avoid DHCP, synchronizing APs via a backplane protocol, even the using additional cards[4] to reduce the association delay. We think this is the wrong approach, for many reasons:

(i) To start the handover mechanism, a client has to lose connectivity to the AP, or *break-before-make*

(ii) There is no standard way to decide which of the many APs to associate with for best performance

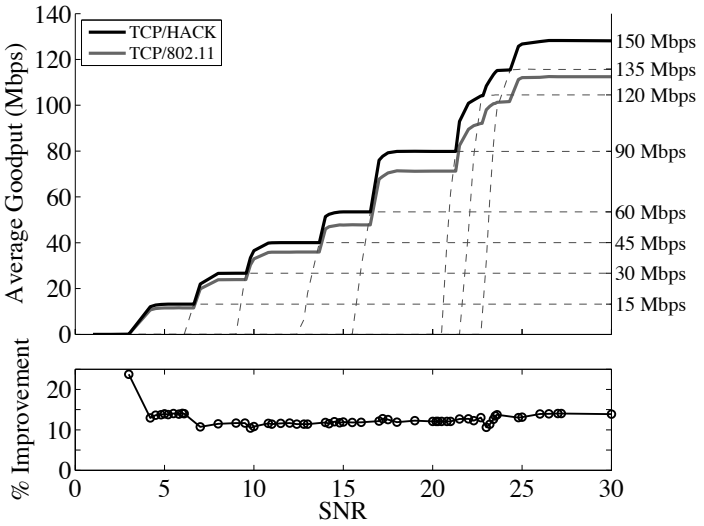


Figure 3.7: Envelope of average TCP goodput for TCP/HACK and TCP/802.11n under different SNR regimes and physical rates. The lower graph shows TCP/HACK's percent improvements over TCP/802.11n.

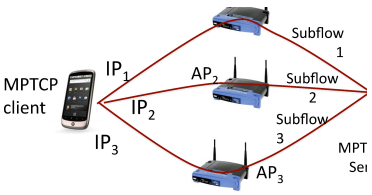


Figure 3.8: Instead of fast handovers, we propose that wireless clients associate to all the APs in range and use MPTCP to spread traffic across them.

(iii) Once a decision is made, there is no way to dynamically adjust to changes in signal strength or load

We conjecture that the emerging standard of Multipath TCP (MPTCP) enables radical changes in how we use WiFi: use of multiple APs becomes natural, whether on the same channel or different ones, and the perennial handoff problem at layer 2 gets handled at layer 4 allowing for a clean, technology independent, end-to-end solution for mobility. In this section we test the following hypothesis: *all WiFi clients should continuously connect to several access points in their vicinity for better throughput and connectivity.*

We carefully analyze the performance experienced by a mobile client locked into using a single WiFi channel and associating automatically to all the APs it sees, *without using any explicit layer 2 handover*. We run a mix of testbed experiments to test a few key usecases and simulations to analyze the generality of our testbed findings across a wide range of parameters and scenarios. We find that, surprisingly, the performance of this simple solution is very good out of the box for a wide range of scenarios and for many WiFi flavours (802.11a, b, g): a WiFi client connecting to all visible APs will get close to the maximum achievable throughput. We discuss in detail the reasons for this performance, namely the WiFi MAC behavior and its positive interaction with MPTCP. In particular, the *hidden terminal problem gets a constructive solution* with MPTCP, as subflows of a connection take turns on the medium instead of competing destructively.

3.2.1 Towards an optimal solution for Wifi Mobility

Consider a wireless client that can associate to three distinct APs, as shown in Figure 3.8. Which one should the client pick and associate to? Prior work has shown that using signal strength is not a good indicator of future performance, so the client may actively probe or passively measure [40] all three APs briefly before deciding on picking one of them. However, this initially optimal choice may quickly become suboptimal because of multiple reasons outside the client's control:

- The client may move.
- Other clients may use the medium, affecting this client's throughput and his choice.
- The wireless channel to the chosen AP may have temporary short-term fluctuations, affecting its capacity.

The combination of these factors is impossible to predict in practice, and the best AP for any given client changes not only in mobility scenarios, but even when the client is stationary. All existing solutions that connect to a single AP are forced to be conservative, because fluctuations (flopping back and forth between APs) can affect performance; thus they all tend to stick to their choice for some time.

We observe that the emergence of MPTCP enables a radically different approach to WiFi mobility: instead of using only one AP at a time and doing handovers, **mobile clients should connect to all APs at any given time**. The solution is conceptually very simple, and is shown in Figure 3.8: we have the client associate to multiple APs, obtaining one IP address from each, and then rely on MPTCP to spread data across all the APs, with one subflow per AP. As the mobile moves, new subflows are added for new APs, while old ones expire as the mobile loses association to remote APs.

How should traffic be distributed over the different APs? As the client has a single wireless interface, it can only receive packets from one AP at a time, even if it is associated to multiple APs. Should the client spend an equal amount of time receiving data via each AP? This policy is optimal only when all APs offer equal throughput. In practice, one AP will offer the best performance, thus it is preferable for the client to transfer most data via this access point. However, all other feasible APs should be used to send probe traffic to ensure that the client can detect when conditions change and adapt quickly. While simple in principle, the key to this solution is understanding the interactions between MPTCP and the WiFi MAC. There are two high-level cases that need to be covered:

APs are on the same wireless channel. If we disregard WiFi interference between APs, the theoretically optimal mobility solution is to always connect to every visible AP, and let MPTCP handle load balancing at the transport layer: if an AP has poor signal strength, its loss rate will be higher (because of lower bandwidth and similar RTTs) and the MPTCP congestion controller will simply migrate most of the traffic to the APs with better connectivity to the client. This way, handover delays are eliminated and the mobile enjoys continuous connectivity. Interference, of course, can be a major issue, and will be explored in depth in the next section.

APs are on different wireless channels. In this case the mobile client must dynamically switch channels while associated to multiple APs, giving each AP the impression it is sleeping when in fact it is going on a different channel. Channel switching has already been proposed as a technique to aggregate AP backhaul capacity by a number of works including FatVAP [19] and Juggler[25]. We discuss the interactions between MPTCP and channel switching in section 3.2.3.

3.2.2 Single Channel Mobility

We implemented a prototype client that is locked on a single channel and continuously listens for beacons of known APs; when a new AP is found, the client creates a new virtual wireless interface and associates to the AP, opening a new MPTCP subflow via the new AP. We ran this code on our 802.11a/b/g/n testbed without external interference, as well as in simulation to understand the interactions that can arise due to interference between different APs, and the extent to which this solution approximates the optimal one.

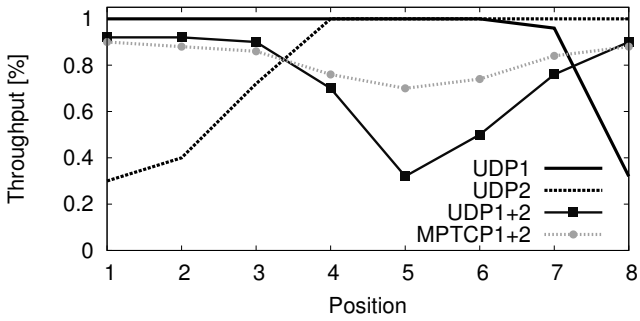
3.2.2.1 Hidden terminal experiments

The first case we test is a pathological one: consider two APs that are outside of carrier-sense range and the MPTCP client connects to both. Lack of carrier-sense means the CSMA mechanism does not function and the frames coming from the two APs will collide at the client. In fact, each AP is a hidden terminal for the other.

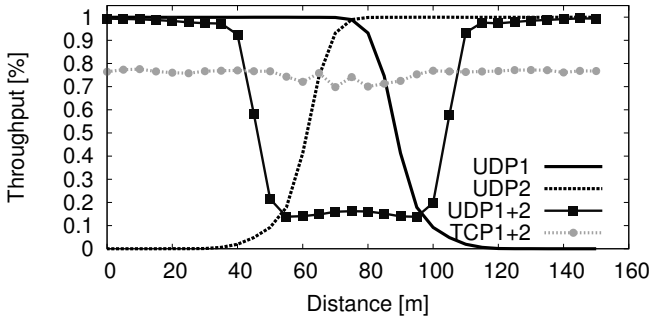
To run this experiment, we reduced the power of our two APs until they went out of Carrier Sense, with the client still able to achieve full throughput to at least one AP at all test locations. Then, we place the client close to one AP and move it towards the other AP in discrete steps and measure the throughput for UDP and TCP via either AP (the status quo) as well as MPTCP. As shown in figure 3.9(a), the graph exhibits three distinct areas. In the two areas close to either AP, neither UDP nor TCP throughput is affected: here the capture effect of WiFi predominates, as packets from the closer AP are much stronger, and the effect of a collision is very small—the client will just decode the stronger packet as if no collision took place, and the subflow via the furthest AP will reduce its rate to almost zero because of repeated packet losses.

The area in the middle is more interesting. As we expected, the combined UDP throughput of two simultaneous iperf sessions is greatly diminished by the hidden terminal situation. However, by running two simultaneous MPTCP subflows, the combined throughput is surprisingly good. Repeated runs showed this result is robust, and we also confirmed this via ns2 simulation (Figure 3.9(b)). MPTCP connection statistics show that the high-level reason for the high throughput is that traffic is flowing entirely over one of the two subflows, while the other one is starved, experiencing repeated timeouts. This would suggest that the starved subflow is experiencing much higher loss rates, which would explain why it never gets off the ground properly.

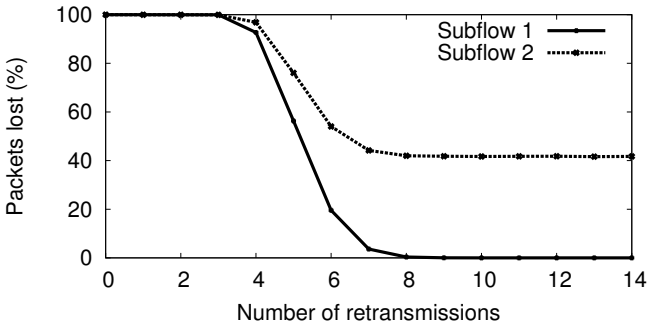
To understand the reason of this behavior, we used simulation to measure the loss probability of the two subflows when contending for the medium. When subflow 1 is sending at full rate, subflow 2 sends a single packet which collides with a packet of subflow 1. The WiFi MACs will then backoff repeatedly until the max retransmission count has been reached, or until one or both packets are delivered. We run the simulation for



(a) Experimental results with 802.11a, 6Mbps: UDP flows systematically collide, while MPTCP subflows take turns on the medium.



(b) ns2 simulation of the same situation in 3.9(a). In the middle region MPTCP exhibits higher variability because one subflow starves the other.



(c) The subflow sending packets infrequently experiences a much higher loss rate. This makes it hard for a (MP)TCP flow to escape its slowstart.

Figure 3.9: Hidden terminal (HT) experiments: using Multipath TCP results in very good throughput because one subflow monopolizes the air, while the other is starved.

a long time to repeat many such events, and show the percentage of events each subflow experiences a loss in Fig. 3.9(c) as a function of the retry count. When few retransmissions are allowed, both subflows lose a packet each when a collision happens, but the effect of the loss is dramatic for the second subflow pushing it to another timeout. As we allow more retransmissions, the loss probability is reduced dramatically: the second subflow loses around 40% of its packets if 6 or more retransmissions are allowed. The reason for the flattening of the line at 40% is the fact that the first sender always has packets to send, and when subflow 1 wins the contention for the first packet, its second packet will start fresh and again collide with the retransmissions from the second subflow, further reducing its delivery ratio. This also explains why subflow 1 experiences no losses after six retransmissions: either it wins the contention immediately, or it succeeds just after the second subflow delivers its packet. In effect, we are witnessing a capture effect at the MPTCP level triggered by the interaction with the WiFi MAC. This behavior is ideal for the MPTCP client.

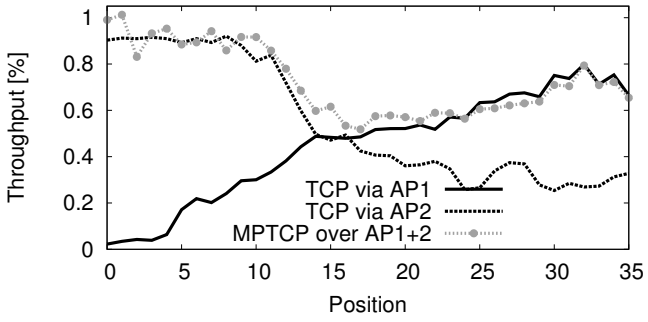
3.2.2.2 Carrier-sense experiments

The most common case is when a client connects to two APs on the same channel that are within carrier sense range of each other, so that the WiFi MAC will prevent both APs sending simultaneously. The mobile associates to both APs and again we move the client from one AP to the other in discrete steps. The performance of our MPTCP client in this case strongly depends on the rate control algorithm employed by the AP, and we explore a number of these to understand their effects.

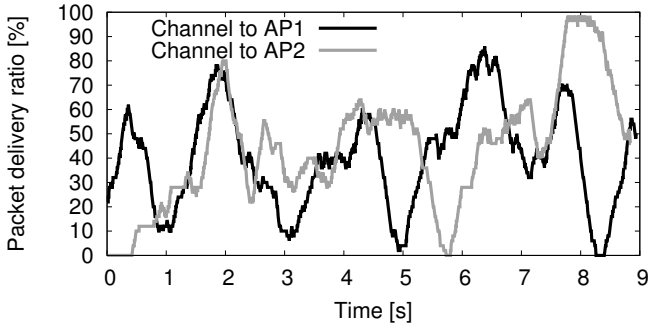
First, we have our Linux APs use 802.11a and run the default Minstrel rate selection algorithm. The results are given in Fig. 3.10(a), and they show that the throughput of the MPTCP client connected to both APs is as high as the maximum offered by any of the two APs. The reasons for this behavior are not obvious.

CASE I: In-between APs the client obtains slightly more throughput (10%) by using both APs than if we were using either AP in isolation. The fundamental reason lies at the physical layer: due to fading effects, individual channel quality varies quickly over time, despite both channels having a roughly equal longer-term capacity. To test this hypothesis, we simultaneously sent a low rate broadcast stream from each AP and measured their delivery ratio at the client. As broadcast packets are never retried, their delivery ratio captures the channel quality accurately; the low rate is used to ensure the two APs don't fight each other over the airtime, while still allowing us to probe both channels concurrently. The instantaneous packet delivery ratios computed with a moving window are shown in Figure 3.10(b), confirming that channels from the two APs are largely independent.

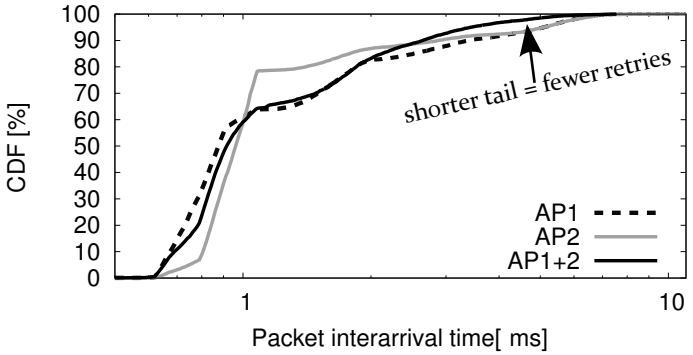
The 802.11 MAC naturally exploits physical channel diversity: the sender that sees a better channel will decrease its contention window, and will be advantaged even more over the sender with a weaker channel. This behavior is experimentally verified by previous work [8] with several clients and bidirectional traffic to/from the APs. For our client downloading from two APs, when one has a slightly worse channel, it will lose a frame and double its contention window before retrying, leaving the other AP to better utilize the channel.



(a) Throughput of a client moving between AP_1 and AP_2: the MAC favors the sender with the better channel. Max throughput is 22Mbps.



(b) A fixed node has channels with a raw PDR≈50% to each AP. The quality of the two channels varies independently in time.



(c) Packet interarrival rate exhibits a longer tail when a single AP is used. indicating periods where the channel is bad. When both APs are sending, the tail is much shorter.

Figure 3.10: Carrier sense experiments: the client using MPTCP gets the throughput of the best TCP connection when close to either AP, and better throughput when in-between.

To validate our hypothesis, we analyzed interarrival times between packets for the client using either AP or both at the same time, and plotted the CDF in Figure 3.10(c). The data shows that most packets arrive 1ms apart, and that AP1 prefers a higher PHY rate (24Mbps) while AP2 prefers a lower PHY rate (18Mbps) when used alone. Using both APs leads to interarrival times in between the two individual curves for most packets. The crucial difference is in the tail of the distribution, where using both APs results in fewer retries per packet. When one AP experiences repeated timeouts, the other AP will send a packet, thus reducing the tail.

In this setup, the optimal AP changes at timescales of a few seconds, and the only realistic way of harvesting this capacity is by connecting to multiple APs. Further experiments with 802.11n and simulations have shown this behavior is robust: when the APs offer similar long-term throughput, a client connected to multiple APs will manage to harvest 10-20% more throughput.

CASE II: One AP dominates. Consider now the case when the client is closer to one AP; in such cases the most efficient use of airtime is to use *only* the AP that's closest to the client. In this case, the throughput of a client connected to all APs strongly depends on the rate selection algorithms used.

In the experiment in Fig. 3.10(a) we use Minstrel that favours higher bitrates even at low signal strengths (with lower frame delivery rate), which leads to more retransmissions per packet for the far away AP. Each retransmission imposes an exponentially larger backoff time on the transmitter, which allows the AP with better signal strength to win the contention more often and thus send more packets; this explains the near-optimal throughput enjoyed by the MPTCP client near either AP. This behavior is strictly enforced by the L2 conditions, and we verified that the choice of TCP congestion control has no effect on the distribution of packets over the two paths; the same results were obtained with UDP.

We also verified in the simulator that when two senders use the same rate, the MAC preference for the better sender holds regardless of the maximum number of retransmissions allowed (0 - 10). This immediately raises the question: what happens when the AP farthest from the client sends using lower rates, thus reducing its frame loss rate? Simulations showed that the effect on total throughput can be drastic: the client connecting to both APs can receive less then half of the throughput it would get via the best AP. This is because lower rates give the farthest AP and the closest one similar loss rates and thus similar chances of winning the contention for the medium. However, packets sent at lower bitrate occupy more airtime, thus decreasing the throughput experienced by the client.

Is this case likely to appear in practice? We ran the same experiment on APs and clients running 802.11n in the 5GHz frequency band. When the client is close to one of the APs, the results differed from 802.11a/g: the throughput obtained with MPTCP was only approximately half the throughput obtainable via the closest AP.

Monitoring the PHY bitrates used by the transmitters shows that $minstrel\_ht$ (the rate control algorithm Linux uses for 802.11n) differs from $minstrel$ significantly: instead of using aggressive bitrates and many retransmissions, $minstrel\_ht$ chooses the rates to ensure maximum delivery probability of packets. The block ACK feature of 802.11n is likely the main factor in this decision, as the loss notification now arrives

at the sender after a whole block of frames has been transmitted (as much as 20): the sender can't afford to aggressively use high bitrates because feedback is scarce.

In fact, the block ACK mechanism together with the cautious choice of bitrates by the transmitters ensures block-level fairness between the two APs: the client will receive one block from AP1 at high bitrate, then one block from AP2 at lower bitrate, and so on.

This issue is not limited to 802.11n: any WiFi rate control algorithm that offers packet-level fairness between multiple senders in Carrier Sense range greatly harms the combined throughput achievable by MPTCP when utilizing multiple APs. Rate control is a modular element and is not regulated by the 802.11 standard, thus wireless NIC manufacturers can implement different algorithms in their drivers.

In summary, a client that associates to multiple APs and spreads traffic over them with MPTCP will receive close-to-optimal performance in a number of situations (hidden-terminal) but not in all. In particular, the performance achieved in carrier-sense environments is strongly dependent on the rate adaptation algorithms employed by the APs, and these are outside the client's control.

3.2.3 Channel-switching

To connect to APs on different WiFi channels, clients can use channel switching, a technique supported by all NICs for probing. This technique was proposed and shown to work in previous work [19, 15, 25, 38]; We implement a similar procedure, but with adaptation based on the actual bandwidth obtained on each channel. Say the client spends a slot c_i on channel i, such that the sum of all slots equals the global duty cycle $C = \sum_i c_i$. While on channel i, the client measured the bandwidth it receives on that channel, b_i, by counting the number of bytes received in a slot and dividing it by c_i. We consider the following family of algorithms for channel switching:

$$c_i = \frac{b_i^\alpha}{\sum_j b_j^\alpha} \cdot C \tag{3.1}$$

The equation prescribes how the value of c_i for the next interval is computed based on the throughput observed in the current interval, where the interval is a multiple of C. α dictates how aggressively we prefer the good channels over the bad ones: higher values lead to more time spent on the best channels. Choosing α strikes a tradeoff between throughput obtained and accurate probing that enables quick adaptation in channel conditions.

The discussion so far has assumed MPTCP is able to allow all APs on different channels to send at flat rate during their slot; in other words Multipath TCP manages to keep all the paths busy. Also note that there are no direct interactions between the MACs of the different APs during this time: enabling MPTCP to work over channel switching is a much easier task. All we need is to make sure the MPTCP subflows do not suffer frequent timeouts, which can occur due to:

- Widly varying round-trip times leading to inaccurate values of the smoothed RTT estimator.
- Bursts of losses suffered when congestion windows are small and fast retransmit is not triggered.

The first problem is quite likely to appear during channel switching, as the senders will see bimodal RTT values: path RTT for packets sent during the channel's slot, and C for packets sent while outside the slot. To avoid this problem, we impose that C is smaller than the smallest possible RTO at clients, which must be higher than the delayed ACK threshold (200ms). Hence, our first restriction is that $C \leq 200ms$.

To avoid the second problem, we lower bound the time spent on any channel to a minimum value that allows the AP to send at least one packet per slot; this implies that the smallest slot has to be at least 10ms.

We have implemented channel switching support in the Linux kernel, together with the family of algorithm discussed above. With this implementation, we ran a series of simple experiments to understand the basic performance of channel switching in our context. We have the client associate to two APs in (channels 40 and 44, 802.11a) and modify the transmit power of the APs while we observe the adaptation algorithms at work. The results are shown in the table below. The experiments with both APs set at max power show that the channel switching overheads (of around 5ms in our measurements) reduce the total available throughput by around 10% when switching between two channels with a duty cycle of $200ms$. If we decrease the power of AP2, $\alpha = 2$ does a good job of increasing the slot of AP1, and obtaining 87% of the optimal throughput. In contrast, the algorithm $\alpha = 0$ assigns equal slot to both APs and throughput is the average of both APs' throughput:

Power for AP1&AP2	TCP AP1	TCP AP2	MPTCP + switch $\alpha = 2$	MPTCP + switch $\alpha = 0$
Max & Max	20	20	18	18
Max & Low	20	14	17.5	16
Max & Low	20	5	17.5	12

Table 3.2: Throughput of both APs related to power

The experiments show that MPTCP and channel switching play nicely together. We note that the experiments work similarly regardless the WiFi standard used.

The key difference between the single channel and multi channel scenarios is the behavior when multiple users are connected to the same APs. When on the same channel, users tend to stick to the AP closest to them as our experiments showed in the previous section. When switching, the clients are not coordinated and will affect each other's throughput and channel estimates in quasi-random ways, depending on how their slots overlap. Here, simulations showed that higher values of $alpha$ also lead users away from bad equilibria, however the total throughput achieved is only 70% of the optimal.

Our driver independent channel switching procedure, through its adaptive slot, makes it possible for an MPTCP based mobile to access capacity on independent channels in a fluid manner.

3.3 PVTCP

Evolution of PVTCP APIs and Security

The sockets API really shows its age as it struggles to support the complex semantics demanded by libraries

that provide encrypted transport streams such as TLS. Applications need to directly deal with the complexities of certificate validation, connection re-keying, and keeping sensitive key material from prying eyes. They rarely do so consistently due to programming complexity, resulting in a regular stream of security breaches.

We present Shushi, a replacement architecture that uses trusted arbiters to separate the concerns of networked cross-layer encryption and key exchange from individual applications. Shushi integrates with OS sandboxing technologies such as Capsicum [41], supports high-performance interconnects via authenticated shared memory channels, and can reduce RTTs during connection establishment via out-of-band signalling. It is built with distributed and virtualized systems in mind, and shared memory channels can be reconfigured into distributed equivalents in the event of VM live migration or host failures.

3.3.1 Introduction

The increasing need for security in public networks (from wireless networks to multi-tenant datacenters) has seen a surge in applications shifting to using end-to-end encrypted connections. Across all of these networks, an efficient low-latency transport encryption is essential, since the standard interfaces are not resistant to adversarial snooping. Our target is to encrypt all traffic over untrusted links, but still do dynamic path selection to maintain performance and congestion control.

The common refrain 'just deploy SSL' is complicated by the heterogeneity of modern datacenters. Applications may be privilege separated (e.g. Wedge, Capsicum, Geode or SE Linux), run inside lightweight containers (e.g. LXC, Docker) or fully virtualized environments (e.g. Xen, KVM, VMWare), and also often employ non-TCP transports such as Unix sockets, shared memory or SCTP. Applications are currently forced to explicitly choose which layer of the network and OS stack they will deploy encryption on, which is brittle in the face of changing load and failure conditions.

Once the choice of transport encryption has been made, the next problem is secure key management. Every application needs to generate and manage keys at several layers of the network: from long-term identities to prevent MitM, to obtaining strong entropy through layers of visualization, to ephemeral key generation, to revocation checking. Applications are notoriously poor at fully implementing the complex requirements imposed by APIs such as OpenSSL , often leading to a false sense of security. At the same time, an application vulnerability could cause the corresponding private keys to be compromised (for instance, the recent Heartbleed vulnerability in the OpenSSL library).

We propose Shushi, a datacenter architecture that uses trusted arbiters to separate the concerns of transport selection and connection establishment (including the full key exchange procedure and authentication) from the applications themselves. Shushi integrates with lightweight sandboxing technology, such as Capsicum, and also supports high-performance interconnects via authenticated shared memory channels. It is built with distributed systems in mind, and so shared memory channels can be reconfigured into distributed equivalents in the event of live relocation or other events.

Equally important, Shushi enables the dynamic usage and synthesis of underlying communication mecha-

nisms, transparently to applications, in a fashion that satisfies the hints provided by the application developer. Innovation is encouraged since Shushi will always try to utilise new communication mechanisms, if they fit the applications' requirements, and fall back to basic mechanisms, such as TCP or Unix sockets, if that is the only alternative (e.g. under the pressure of middleboxes).

Encryption keys (usually rather small in size) need to be treated very separately from the bulk application data. Hence, Shushi creates a "control plane" for low-bandwidth secure coordination traffic that is isolated from application data via OS sandboxing and network isolation.

3.3.2 Architecture

Shushi defines a privileged arbiter to handle the establishment of I/O channels between peers, including key management, name resolution and transport suggestions. Arbiters in a network publish their public ephemeral keys and capabilities information in a directory service. Applications can run in unprivileged or sandboxed mode, and call the arbiter when they need to communicate to a remote or local peer.

Shushi delegates transport selection, key management and authority into a separate privilege domain from the main application, since these operations require a significant amount of complex code that is likely to be vulnerable to software exploits.

Shushi also defines strong security constraints for the arbiter implementation since it operates with all key material of applications. A compromised arbiter breaks the security of all data connections for the applications that it is mediating. The concerns broaden for the directory service, for which a set of active attacks, such as an attack, become available to an adversary. We propose solutions to reduce the risk of these threats (e.g. via type-safe programming languages), but cannot eliminate them completely. Shushi still represents a significant increase in security over the current practice of linking all data into one address space.

3.3.3 Key Properties

Dynamic selection of a suitable transport Datacenter traffic demands high throughput between peers, and optimal transport selection can involve choosing shared memory if on the same OS, inter-VM channels on the same hypervisor, and TCP only for remote communication. Shushi uses the directory to publish the hierarchy of nodes, and arbiters use that information to select the most appropriate transport for an I/O request.

Simplifying trust management There are two levels of trust in Shushi: (i) applications trust an arbiter over a trusted channel (e.g. a Unix domain socket to the kernel); (ii) arbiters use a directory service to authorize peer communications. A directory server in the network is used as a top-level source of trust, and we define methods to protect the integrity of authority communications.

Isolated key traffic channels Shushi's threat model depends on focusing trust into a small control plane for key traffic. We argue that it is much more realistic to have a tightly audited, version-controlled system for relatively small amounts of key traffic, rather than getting it mixed up with the large (gigabytes or

terabytes) of storage used by the applications themselves.

Isolated private key material Shushi never ever exposes any private keys to the application. Therefore, even if an application can be compromised, an attacker cannot obtain any private keys (for instance, the recent Heartbleed bug would not have revealed private keys with Shushi).

3.4 An Evolution of GRIN

Various full bisection designs have been proposed for datacenter networks. They are provisioned for the worst case in which every server sends flat out and there is no congestion anywhere in the network. However, these topologies are prone to considerable underutilisation in the average case encountered in practice. To utilise spare capacity we have designed GRIN, a simple, cheap and easily deployable solution that simply wires up any free ports datacenter servers may have. GRIN allows each server to use up to a maximum amount of bandwidth dependent on the number of available ports and the distribution of idle uplinks in the network. The original GRIN implementation was built with 1Gbps networks in mind, and we had to face the fact that higher speeds are becoming commonplace. We present the main GRIN implementation changes, the transition to 10Gbps networks, as well as new evaluation results, both in terms of performance benefits and possible interactions with other running applications. We describe how distributed application schedulers can be modified to become GRIN-aware, and also show the results of running GRIN on 1000 Amazon EC2 instances.

3.4.1 A GRIN primer

Datacenters heavily rely on the concept of *resource pooling*: different applications' workloads are multiplexed onto the hardware, and any application can in principle expand to utilize as many resources as it needs as long as there is capacity anywhere in the datacenter. In effect, the resources are pooled in time (when different users access the same machine at different times) and in space (where distributed applications can scale up and down as needed). Measurement studies show that datacenter networks are underutilized. Many links are running hot for certain periods of time, while even more links are idle, which results in an underutilized core. We set out to extend the resource pooling principle to datacenter networks.

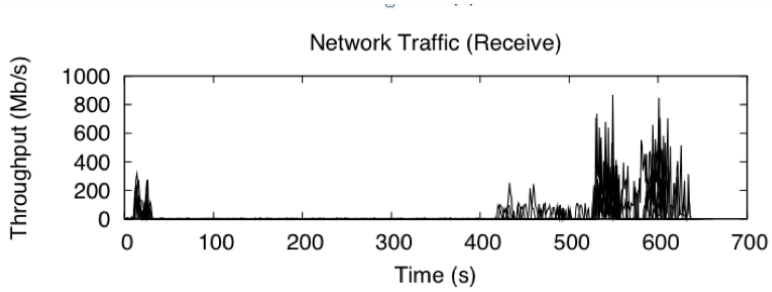


Figure 3.11: Network utilisation of a simple Map/Reduce job

To understand why networks are underutilised most of the time, we measure the network utilisation of a small cluster of ten servers connected to non-blocking switch and running a map-reduce job, a typical datacenter

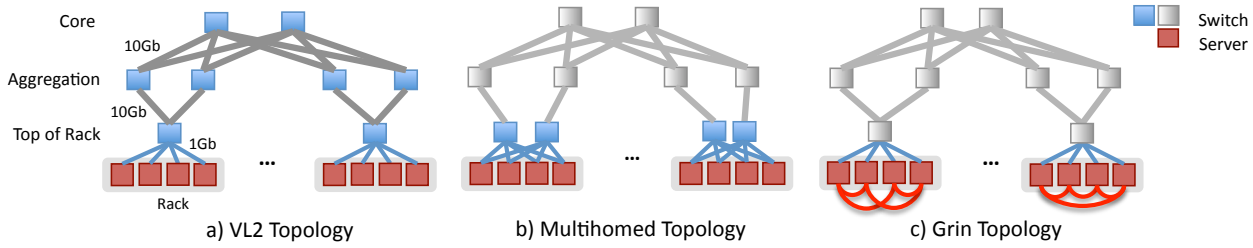


Figure 3.12: Enhancing a VL2 topology to improve network utilisation

application. The servers run Hadoop word-count over a collection of web pages (50GB) stored on the same servers at replication level 3, and we plot the individual network throughput measured for each server in Figure 3.11. In the map phase, there will be a small percentage of tasks whose data is non-local, thus requiring filesystem reads from other servers, which will be bottlenecked by the host NIC capacities, assuming appropriate storage provisioning. The shuffle phase will move the data generated by the mappers to the reducers. The shuffle phase is notoriously bandwidth hungry, but this depends on the number of reducers. In the worst case, all servers are reducers and download data at the same time, leading to an all-to-all traffic pattern that fully utilises the network core. In practice, the number of reducers is an order of magnitude smaller then the number of mappers, and the shuffle phase starts earlier for some servers, thus the core network utilisation will be a lot smaller; still, some reducers will be bottlenecked by their servers' NIC. Finally, the output of the reduce phase is written to disk leading to point-to-point transfers, again bottlenecked by the host NIC.

Barring extensive changes to the original topology, the most straightforward solution is to multihome servers by using additional TOR switches. We add a TOR switch for every additional server port (see Fig.3.12b), so that each server is connected to each of the multiple TOR switches from its rack. In order to keep the rest of the topology unchanged, we evenly divide the uplinks of the original TOR switch between all the local TOR switches. The resulting topology is oversubscribed, but now each server can potentially use much more bandwidth. Multihoming brings additional costs in terms of switching equipment, rack-space, energy usage and maintenance. As every additional server port could require an extra switch, this solution does not scale well with the number of server ports.

GRIN takes a different approach. The servers are directly interconnected using additional ports (some of which are already installed, and effectively free), while keeping the original topology unchanged. Each pair of servers that are directly connected in this manner become neighbours. Intuitively, when a server does not need to use its main network interface, it may allow one or more of its neighbours to "borrow" it, by forwarding packets received from them (or packets addressed to them) to their final destination. This is depicted in Figure 3.12c. When a server wishes to transmit, it can use both its uplink and the links leading to its neighbours. Conversely, the destination can be reached through both its uplink and via its neighbours. We call the links used to interconnect servers, horizontal (or GRIN) links, and reserve the term uplinks for those that connect servers to the switch in the original topology. The network interface where the uplink is connected becomes the primary interface of the server, while the others are considered to be secondary

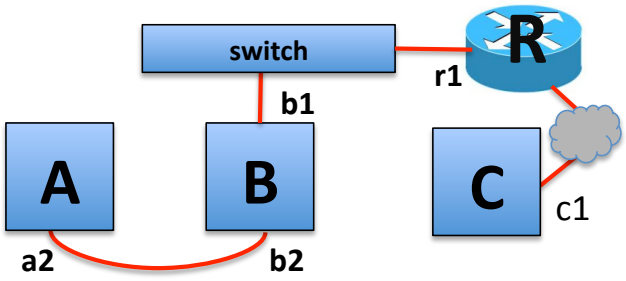


Figure 3.13: Simple GRIN1 Setup

interfaces. If every server has n GRIN neighbours, we say that the *degree* of the GRIN topology is equal to n. We use the term $horizontal_n$ routing (or h_n routing) to describe the fact that in a given GRIN topology we are only interested in paths which have horizontal segments of length at most n; by this definition, the original topology uses h_0 routing. The GRIN implementation employs h_1 routing, because it utilises most of the capacity, while being the cheapest from a forwarding point-of-view. We also rely on a custom addressing scheme. For any given address we are able to tell if it's associated with a primary or secondary interface, and it also possible to determine the primary address of the neighboring server in the latter case.

3.4.2 Implementation

To implement GRIN we reuse forwarding support provided by modern OSes. Linux, for instance, peaks at a rate of about 570Kpps in our tests. This is good enough for gigabit links, or even at 10 Gbps with jumbo frames, but cannot really keep up as NICs become faster.

Can we do better? When processing packets for its GRIN neighbours, a server is fulfilling three main functions: identification of packets intended for another server, rewriting some header fields and forwarding the result. For packet identification, we can leverage hardware filtering capabilities present in modern NICs which allow packets to be received on different queues based on various discriminants, such as destination address. With IP forwarding, each server must write at least the MAC source and destination addresses in the packet. As no hardware support exists for IP forwarding in commodity NICs, this operation must be performed in software.

In theory, GRIN allows us to avoid this extra work by using *bridging* instead of forwarding. In this context, bridging means simply passing a packet from one interface to another, without doing any kind of software processing on it. The origin server knows the MAC address of the next IP hop for a packet. In an L2 network this is the primary interface of the destination (if the packet is heading for a primary address) or the primary interface of one of the destination's neighbours. For a L3 network, the next hop is the designated first router. In both cases, we use ARP to find the proper MAC address, as the IP address is already known (directly from the packet for L2 and by configuration for L3).

This concept is presented in Figure 3.13: servers A and B are neighbours, and the default gateway for B's uplink traffic is router R. When A sends a packet to C, with basic forwarding the packet will have the destination IP address $D_{ip}(p) = c_1$ and destination MAC address $D_{mac}(p) = mac(b_2)$. The latter will

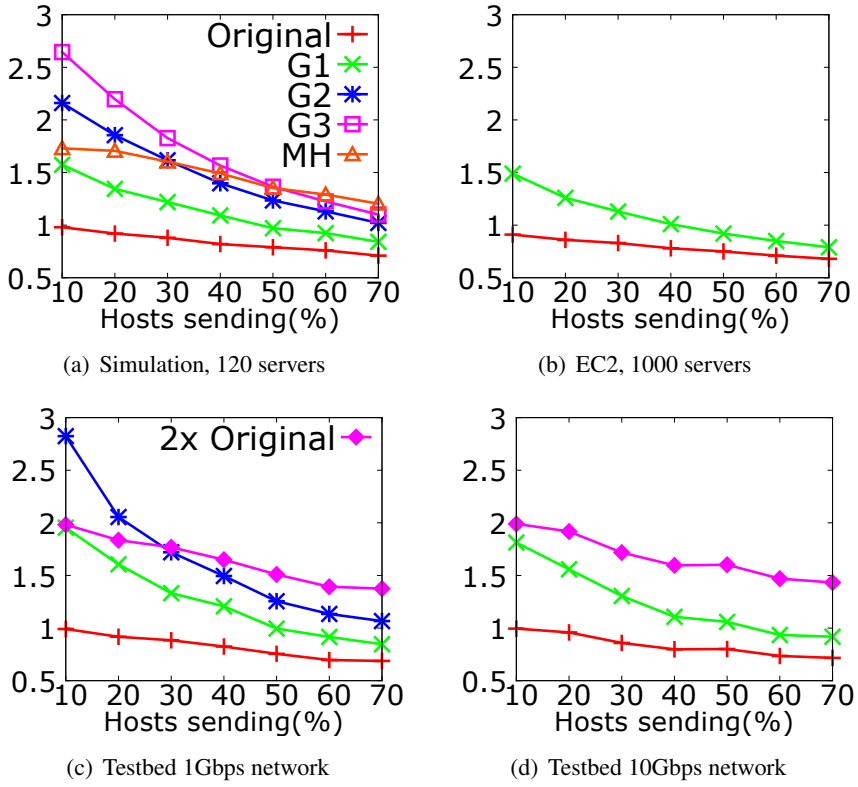
change over the following hops, first to $mac(r_1)$ and, eventually, $mac(c_1)$. With bridging, the packet leaves A with $D_{ip}(p) = c_1$ and $D_{mac}(p) = mac(r_1)$; it can find $mac(r_1)$ by sending an ARP request. B can simply pass the packet to the uplink upon reception; the original contents are enough to steer it to the destination.

In summary, h_1 routing allows GRIN to not only avoid source routing, but also IP forwarding: intermediate servers can just copy packets between interfaces without processing them. Modern commodity 10Gbps NICs (e.g. based on the popular Intel 82599 chipset) already have a simple hardware switch, but it can only move frames between different queues of a single NIC in the *tx*→*rx* scenario; GRIN needs to pass packets from one *rx* queue of an interface to a *tx* queue of another interface (the two interfaces will likely belong to the same multi-port card). With this in place, we could eliminate forwarding overhead altogether. Until hardware support is available, we continue to rely on Linux forwarding for our implementation. This section will conclude with the description of a prototype that shows the viability of the bridging solution.

The GRIN implementation works with a MPTCP-enabled Linux kernel and mainly deals with address assignment to secondary interfaces in user-space. The GRIN addressing scheme allows us to use any routing mechanism that was already in place in the original topology, as long as the original addresses can be adapted to the new structure. The assignment itself relies on pre-existing mechanisms, e.g. DHCP. To automatically configure secondary interfaces, we have implemented a simple server that runs on every computer. It only serves requests that arrive on GRIN interfaces, and its primary functionality is the dissemination of proper secondary addresses to neighbours. After the endpoints of a horizontal link exchange addresses in this manner, each server also adds the required information to the local routing tables. There are two such entries needed per neighbour: one to designate it as the default gateway for all traffic leaving that particular secondary interface, and another to state that the address of the remote endpoint is reachable via the same interface. Additionally, we use Proxy ARP to make servers reply to ARP requests for their neighbours' secondary interfaces, while also making them ignore queries for their own such interfaces.

Bridged implementation. We have also implemented the prototype of a bridged GRIN1 topology that uses netmap as a stand-in for the missing hardware functionality. Outgoing packets that are heading to a primary interface receive no special treatment. For all others, we make sure that the MAC destination address field contains the L2 address of the proper gateway. For both primary and secondary NICs, we use ethtool to enable ntuple filtering and add filters that make sure any packet destined for the local server arrives on *rx* queue 0, while all others are received on queue 1. Finally, the netmap bridge ensures that packets received on queue 0 of each NIC are sent to the host TCP stack and packets coming from the host stack are sent using *tx* queue 0. Also, packets received on *rx* queue 1 of any interface are simply sent to *tx* queue 1 of the other interface. This could easily be extended to work with higher GRIN degrees by using an additional *rx/tx* pair of queues for each secondary interface added.

For the setup in Figure 3.13, when a packet p going from a_2 to c_1 reaches B, it will be placed in *rx* queue 1, because $D_{ip}(p) \neq ip(b_2)$, the netmap bridge will transfer it to *tx* queue 1 of b_1. The switch will direct the



(a) Simulation, 120 servers

(b) EC2, 1000 servers

(c) Testbed 1Gbps network

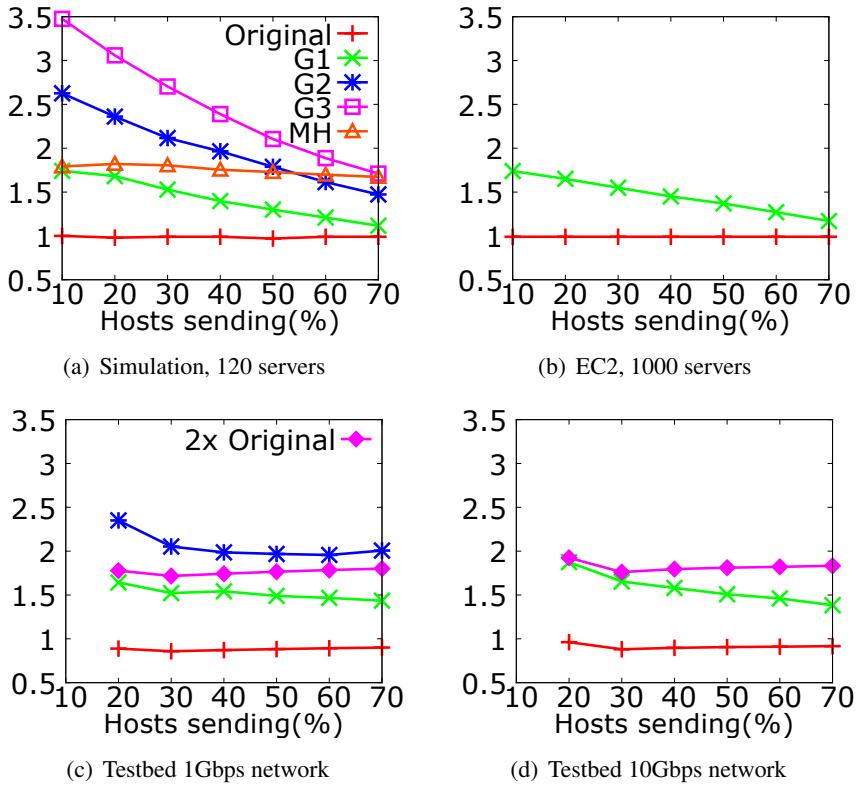
(d) Testbed 10Gbps network

Figure 3.14: GRIN improves performance by 50% to 150% for Random traffic.

packet towards r_1, based on the destination MAC address, and from R it will make its way to the destination. To understand the performance difference between forwarding and bridging we stressed our prototype with minimum sized packets, finding that it can bridge 14Mpps on a single core. If netmap is used to do forwarding, the rate drops to 9Mpps (30% less). We still used the main implementation in our evaluation because netmap does not support hardware offload when exchanging packets with the Linux TCP stack, and this affects Linux performance.

3.4.3 Evaluation

This section presents new experimental results and also compares some of them with related previous results. The GRIN implementation was deployed on a small local cluster of ten servers directly connected to a switch to examine real-world application performance. Each server has a Xeon E5645 processor, 16 GB of RAM, a quad-port gigabit NIC (one port is used for management) and a dual-port 10Gbps NIC. In our testbed, we can build 1Gbps GRIN1 and GRIN2 topologies and a 10Gbps GRIN1 topology. We use both gigabit and ten gigabit networks in our evaluation. Gigabit links to servers are still in wide use today, and GRIN can offer an immediate and much needed increase in performance for deployed networks assuming extra server ports are available. Our 10Gbps tests aim to establish is GRIN is also applicable to newer networks that use 10Gbps links to the servers. The small size of the testbed prevented us from building a useful multihomed setup; even if the bandwidth constraints could be enforced, we could only have at most two racks of five servers each which would allow communication at double speed with half of the servers in the testbed. An upper bound is



(a) Simulation, 120 servers

(b) EC2, 1000 servers

(c) Testbed 1Gbps network

(d) Testbed 10Gbps network

Figure 3.15: GRIN improves performance by 80% to 250% in the All-to-all traffic pattern.

provided instead by simply doubling the results obtained in the original setup.

We also deployed a larger GRIN1 topology on 1000 Amazon EC2 c3.large instances that allows us to test our solution's scalability in practice. Every instance has three ENIs (elastic network interfaces): the first is used for management purposes, the second is considered to be the uplink, and the last is the secondary interface. While it was not possible to directly connect neighbouring secondary interfaces, we managed to achieve a similar behaviour using policy routing in an Amazon VPC, that offers the illusion of an L2 network. We rely on dummynet to limit both primary and secondary interfaces to 100Mbps, to ensure the variable bandwidth provided by the EC2 network does not impact our experiments too much.

Finally, to understand the basic properties of GRIN across a wider range of parameters than feasible in practice, we used simulation in *htsim*, a scalable packet level simulator. This has the advantage of giving very precise results and allows us to study reasonably large networks, but doesn't account for factors outside of the transport protocol itself and cannot be used to evaluate applications. Our simulations were based on the same 120 server topology described in the previous section. Increasing network sizes up to tenfold provides qualitatively similar results, however it takes substantially longer to run.

3.4.3.1 Basic performance

To understand how GRIN works in practice, we begin our tests with synthetic traffic patterns that we can easily reason about. We use the same patterns described our hop-count evaluation, namely permutation, random, group and all-to-all, and run the experiments in both simulation, on Amazon EC2 on 1000 servers

and on our local testbed with gigabit and 10 gigabit networks. In all cases we run the baseline and GRIN1. In simulation we also run multihoming, and GRIN3, which we view as an upper bound of the number of ports that GRIN may use in practice.

In Figure 3.14 we present the results for random traffic as we vary the percent of active servers from 10% to 70%. One thing to keep in mind is that, due to the small size of our local testbed, connections between neighbours will happen more often, and the results will appear better overall. Finally, to allow easy comparison between graphs, we normalize the resulting throughput measurements to that of the original topology (941Mbps for the 1Gbps network, 9960Mbps for 10Gbps and 100Mbps for EC2).

The results show that, as expected, lower percentages of active servers lead to significantly better results for GRIN topologies. Performance improvements are smaller as more servers become active, GRIN2 and GRIN3 performance is better than multihoming until 40% of servers are active, and match it after that.

Note that the simulation results are matched very well by EC2 results and are qualitatively similar to the testbed results, giving us confidence in our evaluation. The EC2 results underline the scalability of our solution: a real-life deployment of GRIN1 can smoothly run on 1000 servers. Running GRIN at 10Gbps is also worthwhile, doubling the throughput when few servers are active. The results for permutation traffic are similar.

We next turn to all-to-all traffic, a pattern mimicking the shuffle phase of map-reduce. When running this experiment on EC2 we ensure that no server initiates or receives more than 20 concurrent connections to reduce the effects of incast. The results in Figure 3.15 show that every additional port used with GRIN brings close to 100% performance improvement when few hosts are active. Multihoming is almost always dominated by GRIN2 and GRIN3, however it outperforms GRIN1. As expected, the testbed results are better. The EC2 results accurately track those obtained in simulation.

Finally, the group connection matrix simulates scatter-gather communication. This is the most favorable situation for GRIN topologies, because the large number of sources will fill every link of the receiving server. The results are consistent across both simulation and actual implementation: we get close to the optimal throughput.

3.4.3.2 Opportunistic usage for 10Gbps networks

Regular applications don't scale nearly as well as the iperf experiments do at 10Gbps, even with jumbo frames and hardware offload functionality enabled. Thus, we focused on a few apps have high bandwidth requirements and are fast enough to take advantage from it: the NFS, HDFS and Spark.

With NFS, the server hosts a single 12 GB file which can be transfered by one client in around 10 seconds with GRIN disabled, and a little less than that with one secondary interface enabled. GRIN does not help because the client is CPU-bound. If two clients attempt to transfer the same file *simultaneously* over a GRIN1 network, they both finish in around 10.5 seconds, implying that the server was able to fully utilise both its 10Gbps interfaces.

	kernel	memcache	transcode
idle, 5 cores	137s	5.9 mreq/s	122s
fw, 5 cores	139s	5.9 mreq/s	124s
idle, 6 cores	118s	6m mreq/s	106s
fw, 6 cores	132s	5.5 mreq/s	120s

Figure 3.16: GRIN forwarding brings little overheads if resource-hungry apps run on separate cores from forwarding. In the worst case, the overhead is 10%

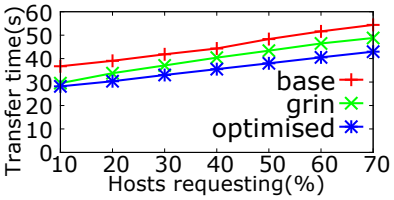


Figure 3.17: HDFS running on 1000 EC2 instances

We used a variable number of Spark workers, connected to a single-node DFS deployment, to count the occurrences of a string in the same 12GB file. At first, GRIN is disabled. A single worker completes the task in 14.5 seconds on average. We need two workers to reduce to time to 10.5 seconds, which is very close to the duration of the data transfer, so we can consider the computation to be finally network-bound. With GRIN1 enabled for both workers, the completion time drops to 7.5 seconds. By starting a third one we can improve this result to around 5.5 seconds, and Spark is now network-limited.

3.4.3.3 Perils of Opportunistic GRIN Usage

There are a number of issues that may be caused by the transition to a GRIN topology. The additional flows may increase buffer pressure and overall latency, while servers could find themselves competing with neighbours for their own uplinks. Our main goal in this regard is to do no worse than the original topology. To deal with these two issues we employ a simple priority scheme, based on DSCP. Each direct flow receives a high-priority codepoint (such as EF), while secondary flows retain the default low value. We rely on iperf to test the fair use of uplinks, and on a simple client-server program to measure the latency of transferring small amounts of data. GRIN specific contention may happen in two distinct situations: server-local when multiple flows use the uplink and at the switch, on the egress port leading to a particular server. In order to honor DSCP markings locally, servers use a priority aware queuing discipline, such as PRIO in Linux. Our experiments show that a single, high-priority flow is able to fully utilise the uplink, regardless of the presence of any other low priority flows.

An idle 10Gbps link takes 100μs on average to transfer 1KB, and 140μs for 100KB with TCP. When competing with several running iperf connections without priorities, the transfer time increases to around 1.8ms in both situations. With the PRIO qdisc, it decreases to 350μs for 1KB and 400μs for 100K. On the downlink, the transfer time grows in both cases to around 2.4ms without priority. If we add priority and configure the switch to discriminate based on DSCP marking, the latency drops to around 110μs and 160μs, respectively.

Does enabling GRIN slow down resource-intensive local applications? Our tests show that GRIN forwarding does not impact storage bound apps, but it is interesting to examine CPU bound and memory bound scenarios. We ran three resource-intensive applications, Linux kernel compilation, video transcoding and a memcache server, on one of our 6-core Xeon servers. Where relevant, we use a ram disk for persistent storage. Running time is the metric for kernel compilation and transcoding. For memcache, we preload 2^{20} keys with

corresponding 60 byte values, and then measure the number of requests that can be fulfilled during 20 second intervals. The requests are generated by 60 local threads that connect to the server using UNIX sockets.

The first two lines in the results in Table 3.16 show that running the app on five cores gives near-identical performance regardless of whether the sixth core is idle or forwarding traffic bidirectionally at 10Gbps: if we can spare a core for forwarding, there will be negligible impact on all other apps. The last two lines show the overhead when we run the app on all cores: here forwarding decreases application performance by 9%-13%.

We stress, however, that in many cases clusters of computers are dedicated to one distributed application (e.g. web-search) to avoid bad performance interactions. In such cases the side-effects of forwarding are irrelevant as long as the application as a whole runs faster. In the next section we describe results with GRIN-aware applications where the total completion time is reduced despite the negative effect of forwarding.

3.4.3.4 GRIN-aware applications

A downside of opportunistic usage is the probability than two neighbours will be using their uplinks at the same time; the busier the network is, the higher this probability. However, most datacenter applications have centralized schedulers that decide how to partition the work across the many workers in the system. We can gain performance comparable to multihoming solutions without the associated costs if we modify application schedulers to take into account GRIN links. We have implemented such optimisations for Hadoop, HDFS and Spark.

One simple and very effective optimisation is to schedule bandwidth intensive jobs onto servers that are not direct GRIN neighbors. We have optimised HDFS for reads by placing replicas of the same block in this manner. When a read request comes in, the scheduler replies with the least-recently accessed server that has a replica, and records that the server and his neighbor were “accessed”. We deployed the optimised version of HDFS on 1000 EC2 instances. In each experiment we used a fraction nodes to transfer a 400MB file from HDFS to local storage. We used a replication factor of three and a block size equal to 140MB. The results, found in Figure 3.17, compare the default implementation of HDFS, HDFS running over GRIN1 and, finally, HDFS optimised for GRIN. The results show that running HDFS opportunistically improves download time on average by 14%, and running optimised HDFS brings a 28% improvement. The results also show that, as expected, the optimised version is superior at higher loads, where the probability of GRIN neighbours to be active is much higher.

Next, we optimised Hadoop and Spark by changing the job placement algorithm to avoid scheduling mappers and reducers on neighbour nodes, whenever possible. We ran Spark to run the same string occurrence problem in a large 24GB file over the 1Gbps network. Two HDFS nodes store the data and two Spark nodes do the actual processing. Without GRIN, the total execution time is around 111 seconds. Even if all nodes are clumped up together, using GRIN1 brings the execution time down to 106 seconds. This is caused by a somewhat uneven distribution of the data (one node holds 11G and the other 13G). Since the application can process it pretty fast, GRIN allows one server to “help the other out” after it finishes sending all the local

data. With optimisations enabled, every node has an idle neighbour, and the execution time drops to around 54.5s. A similar optimisation for Hadoop reduces the overall shuffle duration by 20%.

4 Cross provider liquidity

The cross-liquidity tools are built on top of existing resource pools and manage the interactions between multiple pools. One of the three liquidity scopes presented in this deliverable is cross-provider liquidity, where resources owned by different providers are placed together and interaction between them is governed to create a single domain.

Two tools are presented in this section. The first one is the Federated Market. It introduces mechanisms for controlling virtual machine resources on a set of connected cloud domains allowing sellers to announce their spare capacity and buyers to acquire the resources they need. The Federated Market includes tools for exchanging information about resource availability and usage, tools for sharing the resources and tools for enforcing the resource usage. By placing multiple seller entities in the same pool, a market is created where monetisation is the incentive for sellers to offer their resources.

The second tool presented in this deliverable is MPTCP Path Management. The MPTCP protocol allows the bandwidth resources to be aggregated and efficiently allocated to application requirements. The basic implementation of the protocol allows little control on the subflow creation and termination. MPTCP Path Management introduces tools that enable applications to control the MPTCP paths and to react to network events by implementing their own path management. Since the network paths associated with different MPTCP subflows can be owned by different network providers, this solution offers flexibility in choosing network characteristics like delay and throughput to efficiently meet the application requirements.

4.1 Federated Market

4.1.1 Introduction

The ability to trade resources is central to the concept of a liquid network. The Federated Market (usually referred to as the Market) is a tool that allows the creation and management of virtual machine resources on a set of connected cloud domains. It enables liquidity by acting as the management system between multiple cloud end points. The Market allows “Sellers” with spare server capacity to trade this with “Buyers” that need extra capacity. Each individual cloud system is operated independently in an established manner with the Federated Market providing a web-based dashboard UI for the trading and management of remote resources.

4.1.1.1 History of the Market

The Federated Market resulted from early development work on the testbed platform that was established in Year 1 as part of D1.1. OnApp connectors for using the CompatibleOne[1] platform were developed and released as open-source components that can be used to control OnApp resources from the CompatibleOne ACCORDS[2] platform. The efforts associated with developing for this platform were then refocused to work

[1] http://compatibleone.org/

[2] https://github.com/compatibleone/accords-platform

on an OnApp Market instance that removed some of the limitations encountered and allowed for more direct control of the resources on the platform. The development of a focused platform also allows for more direct control and makes it easier to incorporate other outputs of Trilogy 2. The Federated Market was released as an Alpha as part of OnApp 3.2.0 (2014/01/28) and closed-Beta in OnApp 3.3.0 (2014/07/01). The plan is to make the platform live in Q1 2015 as part of the commercial exploitation of the results of Trilogy 2.

4.1.1.2 Overview of the Market

The Market is designed to allow spare server resources to be traded on the open market by providing the tools needed to safely and efficiently inter-connect disparate resources across multiple cloud providers. The Market knows about the individual resources that make up each VM including storage, compute and bandwidth. However it defers control of individual Virtual Machine (VM) resources to the specific Hypervisors (HV) on the associated IaaS (Infrastructure as a Service) platforms. The Market makes the information about available servers visible at the different end-points and allows resources that are not directly comparable to be traded. This allows buyers and sellers to set end-point prices that will ultimately determine the price of resources through normal market dynamics.

The Federated Market is a collection of tools and platforms designed to meet the following objectives:

- Tools for exchanging information about resource availability and usage. The Federated Market allows for the collation and communication of resources available in the platform. It also maintains a record of who acquires resources so can be used to show is the current owner of a resource.
- Tools for sharing the resources. The Federated Market allows for encapsulated network, compute and storage resources to be shared within a community and provided to buyers of the resource for a monetary cost. Once the resource lease has expired these resources are then made available for the other members of the community to use.
- Tools for enforcing the resource usage. The Federated Market can be used to limit the usage of resources.

Recent work has included adding new features to improve the management of the platform from a cloud provider perspective and the provision of an improved top-level management overview of the platform. There has also been effort into identifying and improving the business model associated with the Federated Market to promote its adoption by users.

4.1.2 Architecture

The architecture of the Market is straightforward and is based on some simple design goals. A market should clearly identify the resources or services that are on offer from buyers to sellers. These can exist in many varieties so it must be able to cope with highly heterogeneous resource sets. In the end there may be several markets that are competing to trade the same resources so it must handle this. At certain points the process behaviour needs to be transactional as the resources will be traded and may change hands multiple times. Thus

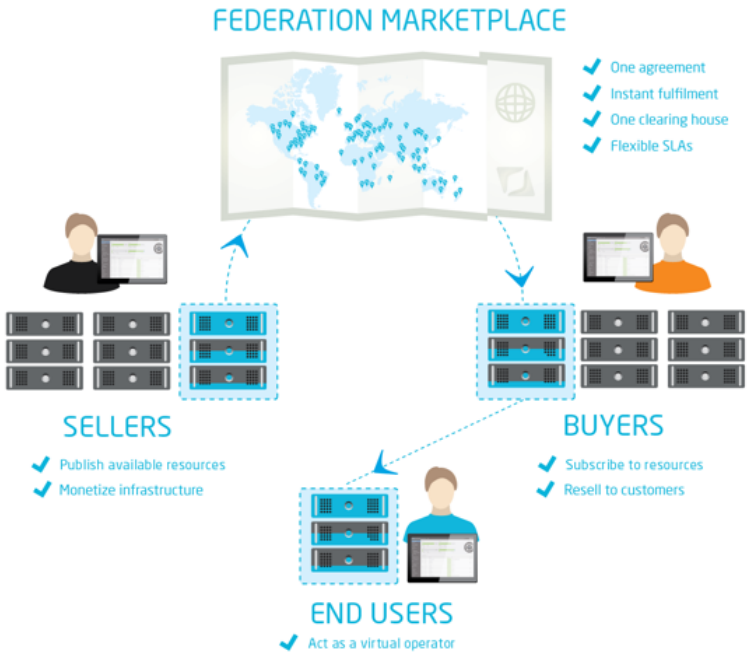
the system will need to be synchronised at that point or possibly suffer from over- or under-provisioning of resources.

There are three main roles within the market:

- Buyer. A buyer wishes to acquire additional resources for their cloud. They purchase resources from the available pool advertised on the Market and will then enter a contract with the Seller of those resources. They may on-sell the acquired resources but will then operate in the role of a (re-)seller.

- Seller. A seller is a resource owner with spare capacity they wish to monetise. They can promote this spare capacity to the Market which then assists them in establishing contracts with buyers who will acquire those resources.

- Market operator. The Market operator is tasked with maintaining an accurate representation of the resources available to trade based on the operating conditions that it states. They also provide the mechanisms needed for accountability and monitoring.



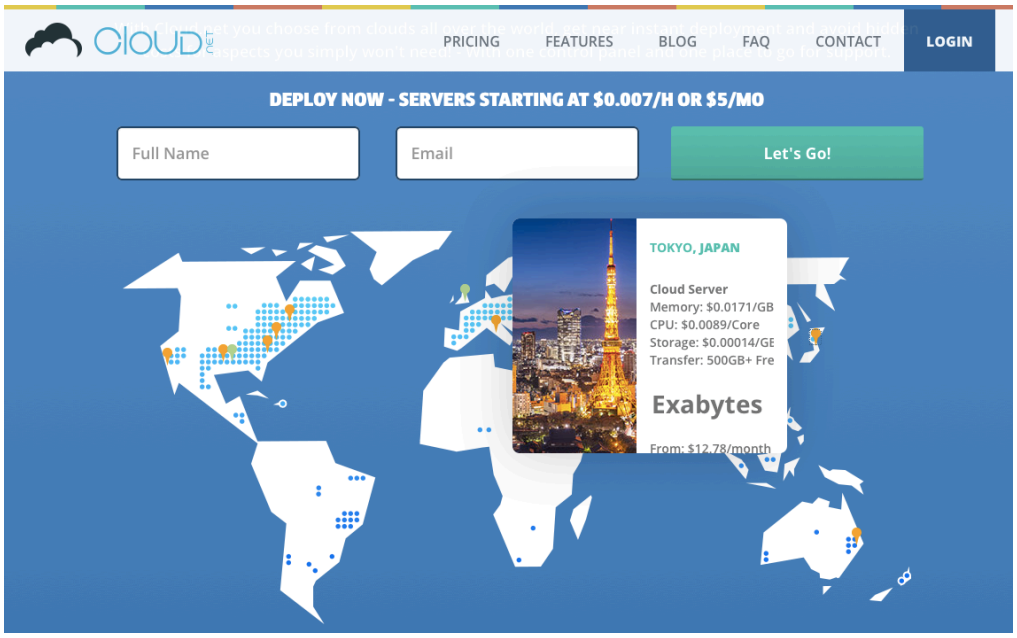
1.2cm 9.5cm 0cm,clip=true

Figure 4.1: Roles in the Market

In addition as shown in Figure 4.1 there is a fourth role of Customer. Customers don't own their own infrastructure, nor do they manage infrastructure. Instead they are clients of the larger buyers who trade their VMs via a system such as OnApp's Cloud.net (Figure 4.2).

4.1.2.1 Current Design

Currently the Market is a centralised system that runs as an appliance server. Central to the market is the idea of zones. A seller with spare servers they wish to monetise add that set of servers to a zone. Buyers are



9.5cm 0cm,clip=true

Figure 4.2: The Cloud.net Platform

then able to search for servers matching their required specification and then can subscribe to the appropriate zone(s).

Aside from marking the physical servers that will be part of the zone, the other elements that a seller needs to set up are the network configuration, the storage layer and the available template types. The network configuration details to the system how IP address handling should be performed and also to which network interfaces the networks are associated. The storage layer requires storage resources to be described in such a way that they can be properly utilised. Template types are VM templates that indicate what kernel to use, which ISO image to load the VM from and also the minimum resource settings required by those VMs.

Once the system has been described sufficiently to host VM resources on the local cloud, this zone can then be added to the Federation using the 'Add to Federation' command from the OnApp UI in a process also known as publishing. This action sends a description of the resource types to the Market system that contains all the necessary information including the IP address and the template types as well as the billing model. This then gets interpreted by the Market system and is added to a local database of the available resources and associated with the appropriate cloud license settings from the originator.

A buyer sees the available zones that are advertised on the Market and can then specify the zone to acquire resources. This operation will establish a subscription to the seller's resources via the Market. In turn the physical resources will then be displayed as a remote zone in the local UI. Creation of a VM on a remote system can then be performed as if it were on a local system with the only difference being the billing plan. VM actions can also be performed on the remote cloud as if it were local, with all commands being routed through the Market place to ensure consistency.

4.1.2.2 Security and Triple-A

One of the key requirements for the Market is the provision of Authentication, Authorisation and Accountability (triple-A) similar to the system from the Diameter Protocol[3]. Users of the system must be authorised to use resources and this requires that contracts are established between resource owners and buyers. API commands between the different clouds are transmitted via the Market. These API commands must be authenticated and secure as they cross administrative boundaries between buyers and sellers. To achieve this, a user with sufficient roles and permissions is set up on both clouds. In this iteration of the market there is a single virtual user that is set up at the time of promoting or subscribing to resources. This user once set up on a cloud is then shared between all remote cloud requests via the Market which might lead to issues of accountability unless the market can clearly delineate where the commands are destined for.

Each message that is sent from any cloud in the Federation must go through the Market, which therefore means there is a single point of failure in the system. Changes to make the system work in a distributed manner are ongoing but will require careful synchronisation steps to ensure that operations that should be carried out by a single end-point are handled correctly and that the other systems are warned of any failure to acquire or sell resources. Consistency throughout the market is a complicated design and architectural issue. Currently, worker threads update the resources periodically but any time that servers are added or removed from a zone this change must be handled in the queue and correctly resolved before proceeding.

4.1.3 Ensuring fairness

A key aim for the Market is to enable "fairness" across the platform for those participating. Fairness is a tenuous concept, but in this context we mean that resources should be as advertised, that no buyer should be able to game the system to get a greater share than they are paying for, that the performance a buyer sees should be isolated from other tenants and that billing should be accurate and transparent. Future monitoring systems can be used to index and rank performance. Those sellers/providers that fall below minimal levels for specified criteria can be removed from the Federated Market and/or highlighted as running a sub-performance system. Using schedulers that already exist in Hypervisor platforms, fairness of resource usage within the platform can be configured, although it is difficult to enforce completely fair sharing of resources unless the workloads are capped at a certain maximum usage level. Capping the usage level allows for providing some level of assurance of resource usage between all users in the system but comes at the cost of a lower overall utilisation of the platform in general as workloads can at most only take up their maximum share.

4.1.4 Adoption and Business Case

Provisioning VM resources in multiple locations requires the establishment of separate contracts between the buyer and the various sellers. Having a single portal that handles all of the management of the contracts should improve the platform. Currently the Federated Market is only available to some OnApp customers. Feedback and the research done in Trilogy 2 have driven improvements to the beta platform. Feedback from

[3] http://tools.ietf.org/html/rfc6733

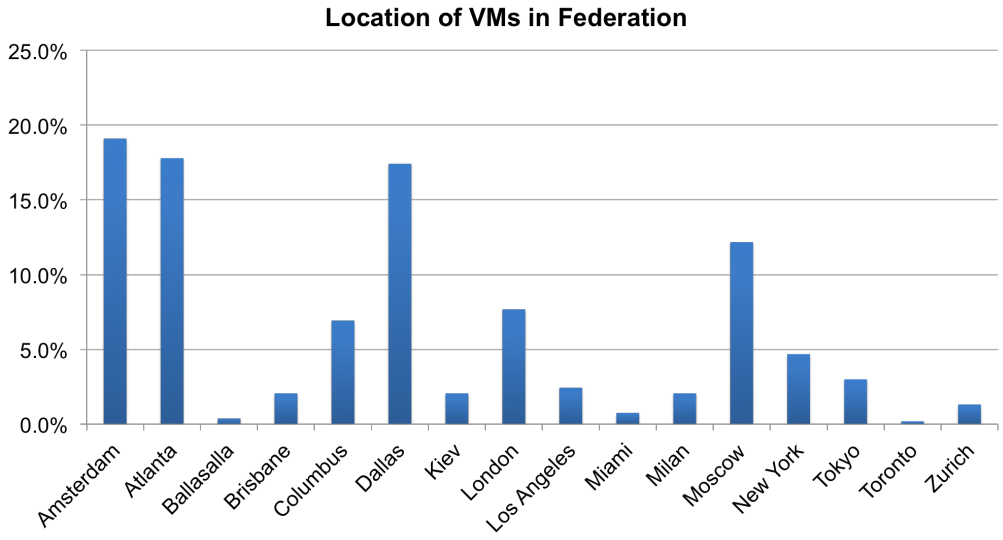


Figure 4.3: Cloud.net already has availability in many locations world-wide.

customers has been vital as it has helped to identify issues in the current platform that would otherwise not be visible from a pure research platform. Customers using the publicly available Cloud.net have chosen sites to provision VM resources based on location and price (see Figure 4.3). There have been requests to better filter the results available based on search queries but generally the main hurdle thus far has been to get people interested in and using the platform. Currently VMs that are provisioned in different clouds are managed separately as individual instances. Offering the ability to create connections between the VMs using publicly accessible network routes is seen as one way to promote adoption. In D3.2 we describe VM mobility across the wide-area as being a use-case that will encourage usage of the platform.

4.1.5 Future work

There are a number of limitations with the current system. The centralised model of the Market, leads to potential issues in accessibility and reliability of the Market. Downtime of the Market can lead to operations being blocked, which will count as a disruption of service. The centralised model does not scale well so there are changes to make this work in a distributed manner.

In its current state the Market will not work with other cloud resource types as there is no information model associated that allows the interpretation of resources to be performed. Having connector pieces is possible but for each cloud platform that is supported there will need to be another connector which ultimately leads to the same issue as was found with CompatibleOne in that effort would go into translation types that are non-standardised.

Using a freely available open Information Model that identifies resource types will lead to greater chance of adoption. Several options are being investigated in this space based on the work in Deliverable D2.2, the Information Model. This will feed into the later WP2 Deliverables. Integrating the Information Model with the Federated Market is an important next step to allow multi-cloud support to be performed.

Although initial work has been taken into incentives and enforcement, the associated business practices and

contracts that are needed from a business perspective are still in their infancy and require further development as there is further uptake of the Market from real customers.

4.2 MPTCP Path Management

The Multipath TCP protocol [11] allows communicating hosts to establish subflows at any time and for any reason. This flexibility is important given the wide range of environments where Multipath TCP can be used and the different needs of the applications. However, until now there was a gap between the theoretical capabilities of the protocol and the real capabilities of its implementation in the Linux kernel. The Multipath TCP implementation in the Linux kernel contains several hooks to enable the definition of different strategies to manage the subflows of each Multipath TCP connection. However, as of this writing, the current release (v0.89) of Multipath TCP in the Linux kernel only supports two strategies to manage subflows that are implemented as modules in the Linux kernel :

- The **full-mesh** path manager creates a full-mesh of subflows among all addresses of the client and all addresses advertised by the server. All these subflows are created by the client since it could be behind a NAT or a firewall that would block subflows initiated by the server.

- The **ndiffports** path manager has a different objective. It assumes that the client and the server are single-homed and the client creates n subflows.

These two path managers are static. The subflows are created by the client as soon as the Multipath TCP connection is established and subflows are only modified based on the reception of `ADD_ADDR` or `RM_ADDR` options. Many applications have specific requirements and Multipath TCP-aware applications will need to better control the establishment of the subflows. For example, a VPN application running over TLS would probably want to be notified of the reception of RST segments over a subflow and react by using another interface. An interactive application would prefer to use the subflows with the lowest delay while a streaming application could be satisfied with a single subflow provided that its bandwidth is above some threshold. Implementing a kernel module that can cope with all these various requirements seems very difficult. Another more flexible solution is required. In this section, we extend the Netlink library proposed in [9] that allows to develop Multipath TCP path managers in user space. Thanks to this library, each Multipath TCP-aware application can define its own path manager and react to various events intelligently.
Section 4.2.3 explains several motivating use cases. Section 4.2.2 describes the architecture of the proposer solution and section 4.2.4 analyses the first results obtained with this prototype implementation.

4.2.1 Use cases

This section discusses several of the motivating use cases for the development of a flexible Multipath TCP path manager.

4.2.1.1 Selection of the best performing subflow

A Multipath TCP connection aggregates several subflows that may have very different performance. Several authors have shown that a low performing subflow may affect the performance of the Multipath TCP connection. For example, [31] shows that when a Multipath TCP connection has a limited window and is used over a 3G and a WiFi with different delays, then the higher delay on the 3G subflow can severely affect the overall performance. This problem is solved in [31] by reinjecting packets from the 3G subflows over the WiFi subflow and penalising the congestion window on the low performing subflow. However, this subflow continues to forward some packets that regularly need to be reinjected over the other subflow, which reduces the performance of the Multipath TCP connection. A Multipath TCP path manager should be able to detect that the 3G subflow under performs and disable it.

Another example is described in [6]. A Multipath TCP host is transferring over both WiFi and 3G. In the middle of the transfer, the WiFi interface becomes very lossy and many packets are reinjected over the 3G interface. Unfortunately, the WiFi interface remains up and Multipath TCP continues to retransmit the data that was sent initially on the WiFi interface (and already acknowledged at the Multipath TCP level over the 3G interface). This behaviour of Multipath TCP is necessary to cope with middleboxes that expect segments in sequence over the WiFi interface. However, this retransmitted data consumes window space and lowers performance. A Multipath TCP path manager should detect that reinjected data has already been acknowledged at the Multipath TCP level and terminate the under performing subflow by sending a RST. This RST would remove all middlebox state on the under performing path and a new subflow could be initiated over this path.

A third example is the impact of bufferbloat on interactive applications. Bufferbloat occurs when there is some congestion and too much buffering on a given path. Measurements have indicated that WiFi home routers and various cellular networks can suffer from bufferbloat that can delay packets for up to a few seconds. Multipath TCP can also be affected by this bufferbloat problem [16]. Even if the default RTT-based scheduler prefers paths with the lowest round-trip-time, it may send packets over paths than have a much longer delay that the delay that could be tolerated by interactive applications. Consider for example an `ssh` session where some of the data suffers from a two second delay. For interactive applications, a Multipath TCP path manager should measure the round-trip-time of the established subflows and disable the subflows that have too long a delay.

A fourth example is streaming applications running on mobile devices. These applications are frequently used on smartphones and tablets to listen to radio or view video clips. These applications require a minimum bandwidth to operate correctly. Usually, they do not need to aggregate two wireless interfaces to work and from an energy consumption viewpoint, it would be useful to disable the unnecessary interface. A Multipath TCP path manager should detect that an application is correctly served with a single interface and disable the other one to minimise the energy consumption.

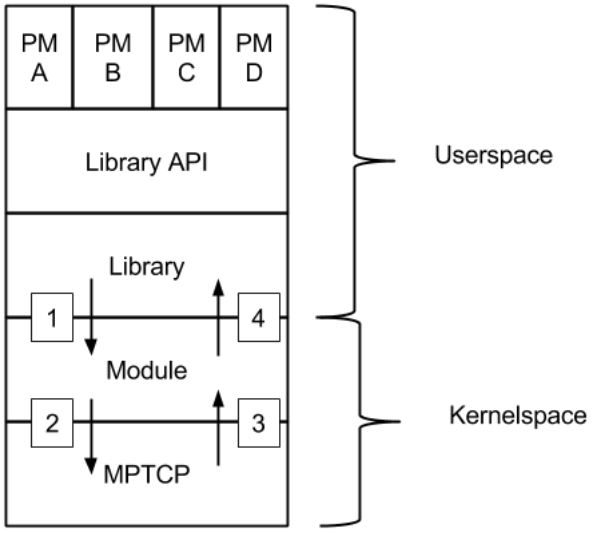


Figure 4.4: MPTCP path management architecture

A fifth example is the middleboxes that are used in some cellular networks and some countries to slowdown the long-lived TCP connections. Such middleboxes detect those long-lived connections and they place them in a low priority queue after some time. This gives a benefit to short connections although the user sometimes needs to transfer large files over a long period. A Multipath TCP path manager could cope with such interference by terminating the subflows every few minutes or once their performance decreases. This path manager would also be useful on single-homed hosts.

A sixth example is the middleboxes that generate RST segments because they have detected a "prohibited" payload or because an unknown application is using a standard port [17]. Such middleboxes can be a nuisance in some networks that want to restrict the traffic to a small number of applications. Although the IAB considered this behaviour to be harmful [10], there are still such middleboxes in some countries or cellular networks. A Multipath TCP path manager should react to the reception of such RST segments by re-establishing a subflow, possibly over another interface, to preserve the ongoing Multipath TCP connection.

4.2.2 Architecture

The Multipath TCP path manager is composed of two complementary parts. The first part is implemented inside the Linux kernel and the second part is implemented in userspace. The two parts communicate through a new Netlink family [9]. The global architecture is depicted in figure 4.4 and discussed in more detail below. The kernel part of the Multipath TCP path manager is implemented as a kernel module and is mainly composed of two subparts. The first subpart generates path manager events for the userspace library(4). Because it is attached to the kernel MPTCP path manager interface defined by `struct mptcp_pm_ops` (3), the module knows when to send these events. This kernel facility delegates all choices linked to the path manager. In our case, the module also delegates the choice by sending an event to the userspace library. The module generates the following events :

MPTCP_EVENT_CREATED : This event indicates that a new Multipath TCP connection has been created.

MPTCP_EVENT_ESTABLISHED : This event indicates that a new Multipath TCP connection has been established. It now possible to establish new subflows.

MPTCP_EVENT_CLOSED : This event indicates that a Multipath TCP has been terminated.

MPTCP_EVENT_ANNOUNCED : A new address has been announced by the peer.

MPTCP_EVENT_REMOVED : An address has been removed by the peer.

MPTCP_SUB_CREATED : This event indicates that a new subflow has been created.

MPTCP_SUB_ESTABLISHED : This event indicates that a new subflow has been established.

MPTCP_SUB_CLOSED : This event indicates that one of the subflows has been closed.

MPTCP_EVENT_SUB_PRIORITY : This event indicates that the priority of one of the subflows has changed.

This allows the userspace path manager to react to events generated by the Multipath TCP kernel.
The second subpart of the modules handles commands sent through the Netlink interface from the userspace to the kernel. This allows the userspace part to react to events received. The available commands are :

MPTCP_CMD_ANNOUNCE : Announces a new address to the peer.

MPTCP_CMD_REMOVE : Announce that an address has been lost to the peer.

MPTCP_CMD_SUB_CREATE : Request the creation of a new subflow.

MPTCP_CMD_SUB_DESTROY : Request the termination of a subflow.

MPTCP_CMD_SUB_PRIORITY : Request a modification of the priority of a subflow.

The module handles these commands by directly interacting with the MPTCP kernel. To sum up, the kernel part defines a new Netlink family that is used to both send events from the kernelspace to the userspace and also to receive commands from the userspace to the kernelspace.
Complementary to the kernel part, the userspace part listens to the events described above and binds them to the userspace API offered by the library [9]. Concretely, a new userspace path manager can register itself through the userspace library that will relay the events by calling specific functions when events are received from the Netlink socket. On the other hand, the userspace library exposes functions that generate command messages over the Netlink socket. To summarise, the userspace part is a glue that translates events/commands received/sent over the Netlink socket into an easy to use API to implement an MPTCP userspace path manager.

4.2.3 Support for the different use cases

In this section, we explain how the library [9] has been enhanced to support the use cases described earlier.

4.2.3.1 Selection of the best performing subflow

This path manager monitors every subflow of every MPTCP connection that it manages. In order to monitor the subflows, this path manager uses a modified version of `Idiag`[4], which is a library that works on top of `libnl`[5] which itself implements the userspace part of Netlink protocol. The path manager regularly polls the kernel to retrieve information about the performance of the different subflows. The poll period is configurable. In the prototype, we have used a polling interval of 300 msec. `Idiag` provides access to a wide range of performance statistics about the different subflows, e.g. : `struct tcp_info`, `RCVBUF`, `SNDBUF`,... In practice, for each new MPTCP connection, the path manager maintains some state that it updates after each new completed request. The path manager sends requests over Netlink via `Idiag` and the responses come back via an asynchronous call to a special function in the path manager.

Note that this path manager can be coupled with other path managers such as a userspace implementation of the kernel fullmesh path manager to react to new local addresses etc. Various information about the state of the subflows can be retrieved via `Idiag` and we can further divide this path manager into more specific path managers. In the rest of this section we describe two specific use cases for this particular path manager. Our first example is a path manager that reacts to `rtt` changes and the second reacts to various of `snd_cwnd`.

4.2.3.1.1 Reaction to rtt changes

This path manager uses `Idiag` to monitor the evolution of the `rtt` over each subflow. The application configures a maximum round-trip-time for the subflows and the path manager disables the subflows whose `rtt` becomes too long compared to the others. This reaction complements the default scheduler that already prefers the subflows having the smallest round-trip-time.

4.2.3.1.2 Reactions to snd_cwnd changes

For streaming applications, the path manager monitors the transmission rate on the Multipath TCP connection and the average rate on the different subflows. It tries to predict the available bandwidth available in the near future by monitoring the congestion window of the sender. To evaluate the bandwidth, we simply use the following formula :

$$bw = cnwd * mss/rtt$$

where each term represents the following:

- bw - bandwidth

[4] `http://www.infradead.org/~tgr/libnl/doc/api/group__idiag.html`
[5] `http://www.infradead.org/~tgr/libnl/`

- cnwd - congestion window size
- mss - maximum segment size
- rtt - round-trip time

These three variables can be regularly accessed by using `Idiag` from the userspace Multipath TCP path manager.

4.2.3.2 Reset reactions

Since the path manager receives an event when a subflow is reset, it can react by re-establishing a new subflow immediately, over the same interface or over another interface. It should be noted that the next version of Multipath TCP [12] will include MPTCP specific options to indicate a reason for a RST. If the remote peer sends a RST to terminate a subflow, it will include this option in the RST segment. The absence of this option would be an indication that the RST has been generated by a middlebox or an attacker. In this case, the host could react by establishing a new subflow over another interface.

4.2.3.3 Refreshing subflows

The path manager can be configured to always have n active subflows and refresh them every m seconds to mitigate the effects of middleboxes slowing down a long connection. The path manager simply refreshes the subflows by opening a new one and closing the corresponding one when the new ones are fully established.

4.2.4 Early results

For the path manager that reacts to changes in the congestion window state, we made a special version of the path manager that is able to output the evolution of the monitored variable. The results are described below for a particular use case.

The laboratory setup that we consider is depicted in figure 4.5. We have one host (A) with two interfaces. The default interface to be used by the host is connected to a shared bottleneck (shared with host B). The other interface should not be used unless we can not reach 5Mbit/s with the default interface.

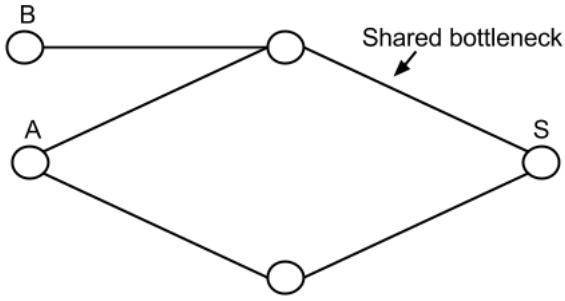


Figure 4.5: Laboratory setup

Furthermore, we consider two scenarios, in the first scenario A sends a 30MB file to S and B is inactive, in other words no cross traffic occurs on the default interface. In the second scenario, we also consider the exchange of a 30MB file from A to S but B generates cross traffic after 10sec.

This setup corresponds to the smart phones use case described in 4.2.3.1.2 where we have a cheap interface with a variable quality (bandwidth) that we want to use as much as possible and an other interface which cost more to the user that is only enabled if it is needed.

We use the path manager to poll the congestion window every 300ms and calculate the available bandwidth based on the formula described in section 4.2.3.1.2.The results are shown on figure 4.6. Note that we used a smoothed version of the formula presented in section 4.2.3.1.2 to avoid too much oscillation. The figure shows the evolution of the available bandwidth on the default interfaces in the two cases. The first case, with no cross traffic corresponds to the red line, while the second case with cross traffic after 10 seconds corresponds to the green line. We observe that the red line stays close to 5 Mbit/s during the whole duration of the connection while the green line drops after 10sec. In this case, it is therefore possible to quickly detect the degradation of the primary link and the path manager can decide to open another subflow on the other interface.

Note that the curves are high at the beginning of the connections. This is due to two effects. Firstly, the slow start of the TCP connection generates high value for the congestion windows. We may mitigate this effect by only considering the size of the congestion window when we are in congestion avoidance phase. Secondly, the curve does not stabilize immediately because we used a smoothed value of the bandwidth to avoid fast oscillations. We used the same factor as defined in [30] for the calculation of the smoothed RTT (0.125). This factor may be tuned to react faster to congestion window changes. However we should be careful when we chose this factor because it may lead to a lot of oscillations.

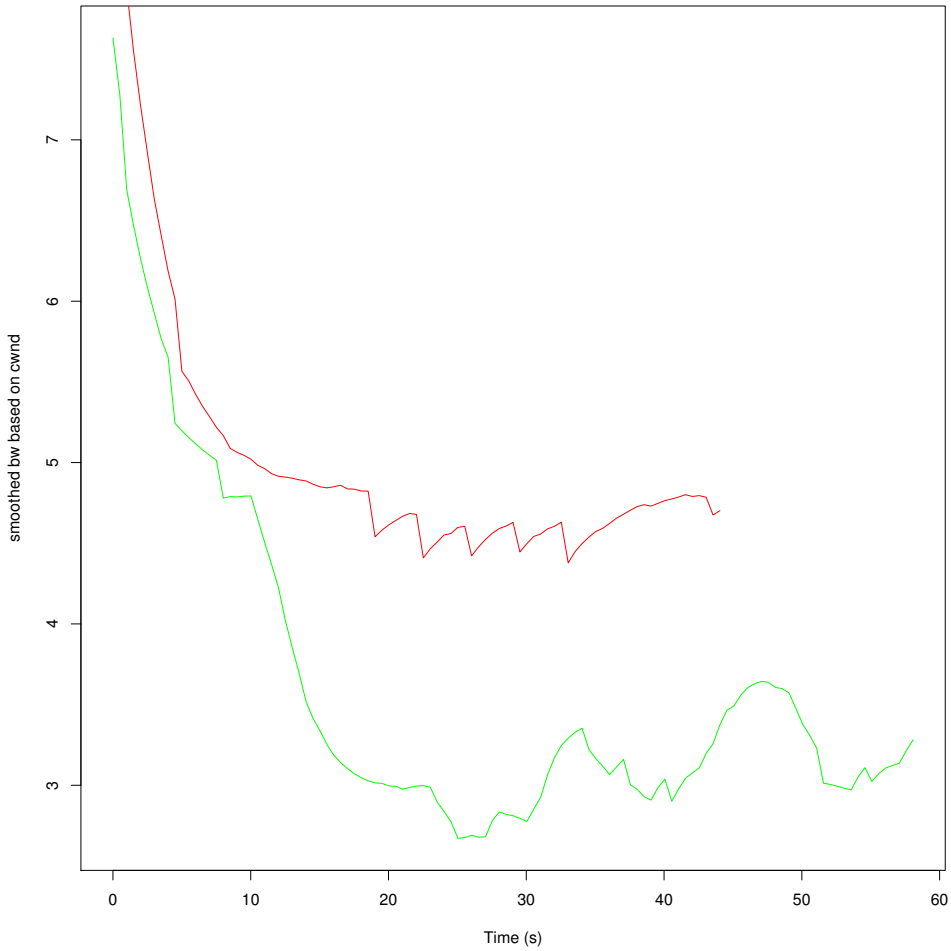


Figure 4.6: Cross traffic after 10sec in the second can be detected based on the evolution of the congestion window

5 Conclusion

This Deliverable focused on presenting liquidity pools and how resources can be shared by grouping them together and facilitating a flexible access to them for the applications. Creating homogeneous pools is only the first step towards having liquidity for resources and the tools presented in this Deliverable build advanced interaction scenarios between resources.

Based on work started in D1.2, the mechanisms presented here improve the combination of resource pools by bridging islands of heterogeneous resources. The advanced cross-liquidity tools create three dimensions in resource liquidity, based on the nature of the bridges: cross-resource, cross-layer and cross-provider. These dimensions are presented by trading between CPU, bandwidth and storage to create liquidity pools that efficiently meet the application demands.

Current work was focused on all three areas, with part of the tools covering multiple liquidity dimensions (for example the Federated Market extends both cross-resource sharing and cross-provider sharing). Since bandwidth is a common denominator that allows CPU and storage resources to be accessed, a part of the presented tools are mode bandwidth-centric.

The work performed in this Deliverable is used by Work Package 2 to create a better control over the liquidity and also by Work Package 3 to use liquidity pools in set of usecases. Also, the results from this Deliverable will be used to build the Liquid Net in D1.4 by bridging across different resource types, layers and providers in the Internet.

Bibliography

[1] Alexandru Agache and Costin Raiciu. Grin: Utilizing the empty half of full bisection networks. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[4] Vladimir Brik, Arunesh Mishra, and Suman Banerjee. Eliminating handoff latencies in 802.11 wlans using multiple radios: Applications, experience, and evaluation. In *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, IMC '05, pages 27–27, Berkeley, CA, USA, 2005. USENIX Association.

[5] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 67–86, Berlin, Heidelberg, 2012. Springer-Verlag.

[6] Yung-Chih Chen, Yeon-sup Lim, Richard J. Gibbens, Erich M. Nahum, Ramin Khalili, and Don Towsley. A measurement-based study of multipath tcp performance over wireless networks. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 455–468, New York, NY, USA, 2013. ACM.

[7] S. Cheshire and M. Krochmal. Multicast DNS. Technical Report 6762, IETF Secretariat, February 2013.

[8] Sunwoong Choi, Kihong Park, and Chong-kwon Kim. On the performance characteristics of wlans: Revisited. *SIGMETRICS Perform. Eval. Rev.*, 33(1):97–108, June 2005.

[9] G. Detal and S. Barre. A software library for multipath tcp path managers. Unpublished software, 2014 October.

[10] S. Floyd. Inappropriate TCP Resets Considered Harmful. Technical Report 3360, IETF Secretariat, August 2002.

[11] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. Technical Report 6824, IETF Secretariat, January 2013.

[12] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. Internet-Draft draft-ietf-mptcp-rfc6824bis-03, IETF Secretariat, October 2014. I-D Exists.

[13] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark buffers in the Internet. *CACM*, 55(1), 2012.

[14] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[15] Domenico Giustiniano, Alberto Lopez Toledo, Eduard Goma, and Pablo Rodriguez. Wiswitcher: An efficient client for managing multiple aps, 2009.

[16] David Hayes, Simone Ferlin, and Michael Welzl. Shared Bottleneck Detection for Coupled Congestion Control for RTP Media. Internet-Draft draft-hayes-rmcat-sbd-01, IETF Secretariat, October 2014. I-D Exists.

[17] Andy Heffernan. Protection of BGP Sessions via the TCP MD5 Signature Option. Internet-Draft draft-ietf-idr-rfc2385bis-01, IETF Secretariat, March 2002.

[18] IEEE. IEEE Standard 802.11-2012, March 2012.

[19] Srikanth Kandula, Kate Ching-Ju Lin, Tural Badirkhanli, and Dina Katabi. FatVAP: aggregating AP backhaul capacity to maximize throughput. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 89–104, Berkeley, CA, USA, 2008. USENIX Association.

[20] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[21] Anil Madhavapeddy, Alex Ho, Tim Deegan, David Scott, and Ripduman Sohan. Melange: Creating a "functional" internet. *SIGOPS Oper. Syst. Rev.*, 41(3):101–114, March 2007.

[22] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[23] Anil Madhavapeddy, Richard Mortier, Ripduman Sohan, Thomas Gazagnaire, Steven Hand, Tim Deegan, Derek McAuley, and Jon Crowcroft. Turning down the lamp: Software specialisation for the cloud. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[24] E. Magistretti, K. Chintalapudi, B. Radunovic, and R. Ramjee. WiFi-Nano: Reclaiming WiFi efficiency through 800 ns slots. In *MobiCom*, 2011.

[25] Anthony J. Nicholson, Scott Wolchok, and Brian D. Noble. Juggler: Virtual networks for fun and profit. *IEEE Transactions on Mobile Computing*, 9(1):31–43, January 2010.

[26] NS3. ns-3 simulator. `http://www.nsnam.org/`, 2013.

[27] Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, Pittsburgh, PA, USA, 1996. AAI9813847.

[28] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[29] 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.

[30] Vern Paxson, Mark Allman, Jerry Chu, and Matt Sargent. Computing tcp's retransmission timer. Technical report, RFc 2988, November, 2000.

[31] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[32] R. Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321, April 1992.

[33] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[34] Lynne Salameh, Astrit Zhushi, Mark Handley, Kyle Jamieson, and Brad Karp. HACK heirarchical ACKS for efficient wireless medium utilization. In *USENIX ATC*, 2014.

[35] Kristofer Sandlund, Mark West, and Ghyslain Pelletier. RObust Header Compression (ROHC): A Profile for TCP/IP (ROHC-TCP). Internet-Draft draft-ietf-rohc-tcp-16, IETF Secretariat, February 2007.

[36] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[37] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[38] Hamed Soroush, Peter Gilbert, Nilanjan Banerjee, Brian Neil Levine, Mark Corner, and Landon Cox. Concurrent wi-fi for mobile users: Analysis and measurements. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 4:1–4:12, New York, NY, USA, 2011. ACM.

[39] Kun Tan, Jiansong Zhang, Ji Fang, He Liu, Yusheng Ye, Shen Wang, Yongguang Zhang, Haitao Wu, Wei Wang, and Geoffrey Voelker. Sora: High performance software radio using general purpose multi-core processors. In *Proc. of the* NSDI *Conf.*, April 2009.

[40] Sudarshan Vasudevan, Konstantina Papagiannaki, Christophe Diot, Jim Kurose, and Don Towsley. Facilitating access point selection in IEEE 802.11 wireless networks. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurements*, pages 26–26. USENIX Association, 2005.

[41] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.