Distributed File Systems in Cloudlets

Sang Jin Han

CMU-CS-16-126
July 2016

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Mahadev Satyanarayanan, Chair
Padmanabhan Pillai

Submitted in partial fulfillment of the requirements
for the degree of Master's of Science.

Keywords: Cloudlet, Distributed File System

Abstract

We explore the implementation details and consequences of running a distributed file system on cloudlets. Using a centralized Samba server VM that exports a Coda mount point, we allow client VMs within a cloudlet to have a common shared file system. We use the tenancy system in the cloudlet for access control, meaning that only VMs owned by the same user or organization share the file system mount.

Along with the discussion of the implementation of the system, we investigate the effects of our architecture on the cloudlet ecosystem. Particularly, we focus on its integration with the cloudlet's VM migration feature. Running a centralized distributed file system per tenancy allows the file system servers to communicate with each other across cloudlets for prefetching frequently used files in parallel with VM migration. Furthermore, we argue that our architecture integrates well with existing software and we show we can run Hadoop, a widely used distributed data analysis framework, on top of it.

Acknowledgments

Thank you Professor Satya, for giving me the opportunity to write this thesis. Thank you Wolf, for giving me the chance to work as a research assistant, which ultimately lead me here. Thank you Babu, for your support as a second reader despite the time restraints. Thank you Kiryong, for your help on Openstack++ and various other tips on the cloudlet ecosystem. Thank you Jan, for helping me throughout this journey, guiding me out of the pitfall whenever I was stuck. Thank you so much for revising my drafts and giving me awesome feedback everytime.

Thanks to the 702 crew, especially Eric, for providing me with the friendly banter to destress everyday. Thanks to JY and WY for the friendship since freshman year. Finally, I want to thank my loving family for their unconditional support. Love you all, more than anything.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Cloudlets

The cloudlet architecture attempts to bring the cloud closer by providing a cloudlet server closer to the client that executes minor computations and caches data from the faraway cloud. Within the current architecture of the cloud computing landscape, there exists no middle ground between the compute and data intensive cloud and the end user device. Hence, the connection between the user device and the cloud remains to be limited with low bandwidth and high latency. The cloudlet architecture, pioneered by professor Satyanarayanan, aims to alleviate this problem by suggesting a cache-like structure called the cloudlet.

The cloudlet contains more compute power and has better connectivity than the end device. It resides much closer to the end device than the cloud, thus maintains logical proximity (low latency, high bandwidth). The cloudlet aims to keep only soft state, leading to less complexity in the system. Designers can use the cloudlet to cache the working set of the end user to improve the overall experience of the cloud.

Figure 1.1 shows a diagram from the original cloudlet paper[8]. Several mobile devices connect to the cloudlet, which in turn is connected to the distant cloud. These end devices benefit from the strong connection between the cloudlet and themselves while still being able to communicate with the remote cloud.

Cloudlets become crucial as more and more latency-sensitive mobile applications surface in the already mobile-dependent computing scene. The mobile ecosystem already depends heavily on the cloud to provide servers, and adding another layer to the architecture to improve latency and computing power gives application writers more options. For example, new cognitive assistance applications that help the visually impaired capture information from a built-in camera in the user's glasses and offers verbal help can take advantage of the low latency and the computing power cloudlets provide.

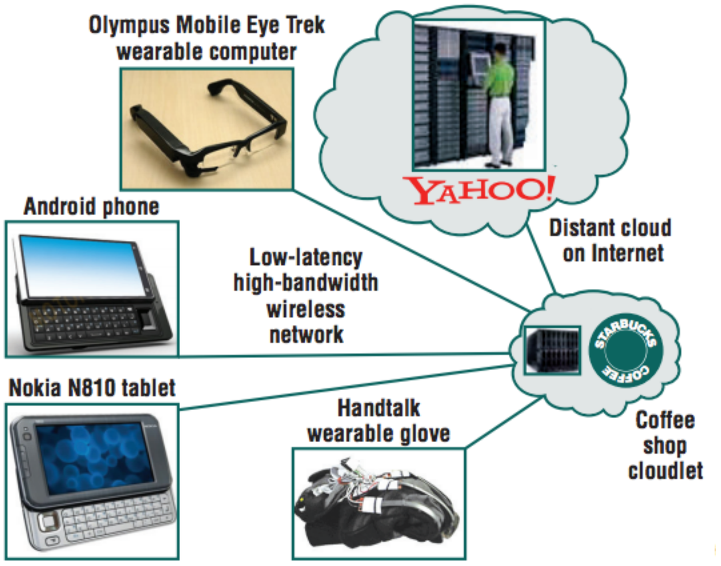

Olympus Mobile Eye Trek wearable computer
YAHOO!
Android phone
Low-latency high-bandwidth wireless network
Distant cloud on Internet
Nokia N810 tablet
Handtalk wearable glove
Coffee shop cloudlet

Figure 1.1: Example setup of the cloudlet architecture

1.2 Distributed File Systems

Distributed file systems provide a way for multiple clients to share files easily. These file systems generally contain carefully constructed concurrency and consistency logic to provide simple and familiar interfaces that resemble conventional file systems to its users. Furthermore, most modern distributed file systems contain performance optimizations such as caches that decrease the amount of communication between the client and the remote server and batch propagation of updates which fully utilize the bandwidth of the connection to the server.

For example, AFS caches the entire file from the server to the local client upon open, and keeps consistency by using a callback system[5]. Coda uses the same mechanism, but also supports disconnected operations where the user can still use the file system without a network connection[6].

1.3 Motivation

The current cloud computing landscape frequently involves several VMs that work coherently with each other on similar working sets. For example, the visual assistant mentioned in chapter 1.1. might have a VM that analyzes the user's use patterns along with a VM that provides the actual service. In this case, the two VMs must work together on a similar set of files (the user's data) and need to share data between each other naturally. In the current architecture, VMs in the cloudlet are connected only by the internal network, and there exists no easy way to share files between VMs both within a single cloudlet and across cloudlets. Hence, we wanted to ease the sharing of files between VMs both within one cloudlet and across cloudlets.

Running a distributed file system allows VMs across cloudlets to share files through the distributed file system without directly contacting each other across WAN. We also wanted to ease the sharing of files between virtual machines within a single cloudlet that belong to the same organizational group, such as a group of developers working on the same program or a team of students working on the same school project. By installing a distributed file system within the cloudlet for these VMs, we look to increase cohesiveness of the VMs and to broaden the scope of tasks the VMs in the cloudlet can execute. We also intend to enforce access control mechanisms to aid these groups organize their development and to protect data from outsiders who may operate on the same cloudlet.

1.4 Thesis

We explore the benefits of running a distributed file system within an organizational group in the cloudlet. The virtual machines within the same organizational boundaries will all be connected to a single distributed file system, and hence will be free to share files with each other.

By running a distributed file system in the cloudlet, we state that we can deliver cohesiveness, increase the spectrum of tasks that VMs can handle together, and optimize the performance of file system activities. We demonstrate this with an experiment that shows the benefits of prefetching upon migration, and by running the widely-used Apache Hadoop on top of our architecture.

1.4.1 Scope of Thesis

The thesis covers the design and implementation of our architecture in the cloudlet and the experiments we performed to validate the benefits of running a distributed file system in the cloudlet. Additional focus is given to the prefetching feature when migrating VMs across cloudlets and running Hadoop on our distributed file system. We also cover past related work to our research and any future work that remains beyond our work.

1.4.2 Approach

We work on top of Openstack++, also known as elijah-openstack, an enhanced version of Openstack developed by Kiryong Ha that offers cloudlet support[1]. Furthermore, we use Coda, a descendant of the Andrew File System, as our distributed file system implementation to integrate into the cloudlet.

1.4.3 Validation

In order to validate our system, we perform experiments that test the implementation. We first validate the functionality of our architecture upon migration by ensuring that client VMs can access the mount before and after the migration. Furthermore, we test the performance of our prefetching feature by running an experiment to compare the re-compilation time of the Coda source with and without prefetching. Lastly, we run Hadoop on our architecture to verify that our architecture is compatible with existing well-known software such as Hadoop.

1.5 Document Roadmap

Chapter 2 offers commentary on the background of our thesis, introducing basic elements that act as foundations of our implementation. Chapter 3 dives into our implementation covering everything starting from the design to the reasoning behind the resulting product. Chapter 4 covers the experiments performed to verify the functionality of the implementation when VMs migrate across cloudlets. Chapter 5 extensively covers the prefetching feature, from its implementation details to experiment results. Chapter 6 deals with running Hadoop in our architecture, and chapter 7 offers related and future work with the conclusion for the thesis.

Chapter 2

Background

We will be building on top of the following components for the distributed file system in cloudlets. The general architecture involves running a Coda client on a single VM and having that VM export the Coda mount via Samba to other VMs. We also limit this share to an administrative entity, so that one user's VM cannot access another user's Coda mount.

2.1 Openstack

Openstack provides an open-source solution to deploying a public or private cluster. The framework gives administrators fine-grained control of the cluster, including starting and suspending virtual machines, assigning floating IP addresses, configuring network connections between the VMs, and managing the VM images. Ever since its beginning as a joint project between Rackspace and NASA in 2010, Openstack emerged as a major cloud infrastructure software with over 500 companies supporting its cause.

Openstack is organized into several microservices that provide a specific service. These microservices interact with each other to deliver the full Openstack feature set. Some of the key microservices include: Keystone, which handles authentication of all services, Nova, which runs on the compute node to manage virtual machines within the node, Cinder, which provides block storage, and Glance, which supervises the virtual machine images in Openstack.

Due to its widespread use in academia and industry, Openstack was chosen as the framework in which to implement the cloudlet architecture.

2.1.1 Openstack Networking

Openstack supports two types of networking. Neutron, the most recent version of its networking suite, and Nova-networking, its predecessor. Neutron and Nova-networking both provide the administrator with the ability to configure the cloud/cloudlets network, including assigning floating

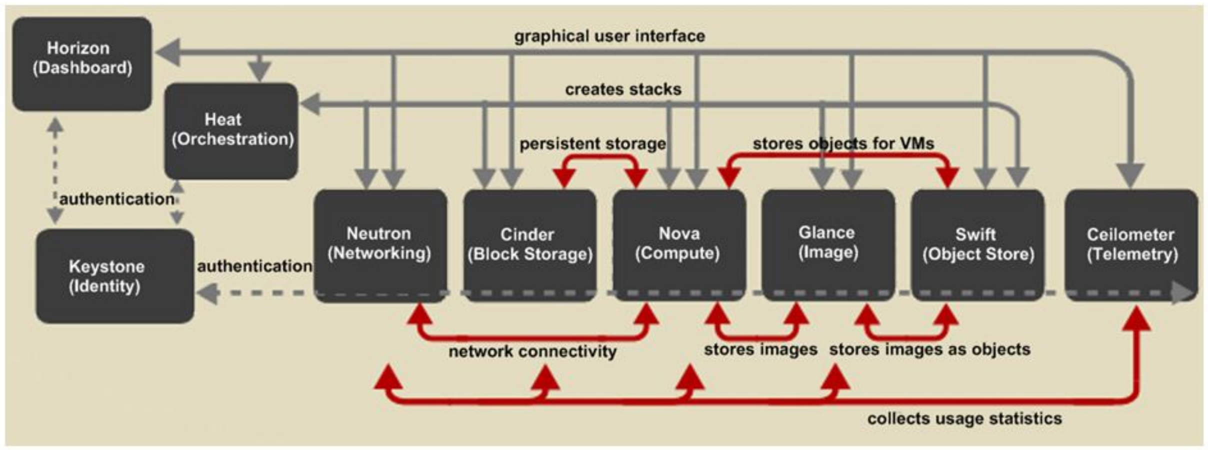


Figure 2.1: Organization of Openstack

IPs, creating new subnets, and assigning subnets to tenants. Tenants each represent one administrative entity within the cloud environment. Openstack uses tenants to isolate separate user groups. For example, one tenant might be a CMU student running analysis jobs for his school project while another might be a group of developers from a startup using virtual machines as scalable servers for their platform. A VM in one tenant cannot reach a VM in another tenant via the internal network.

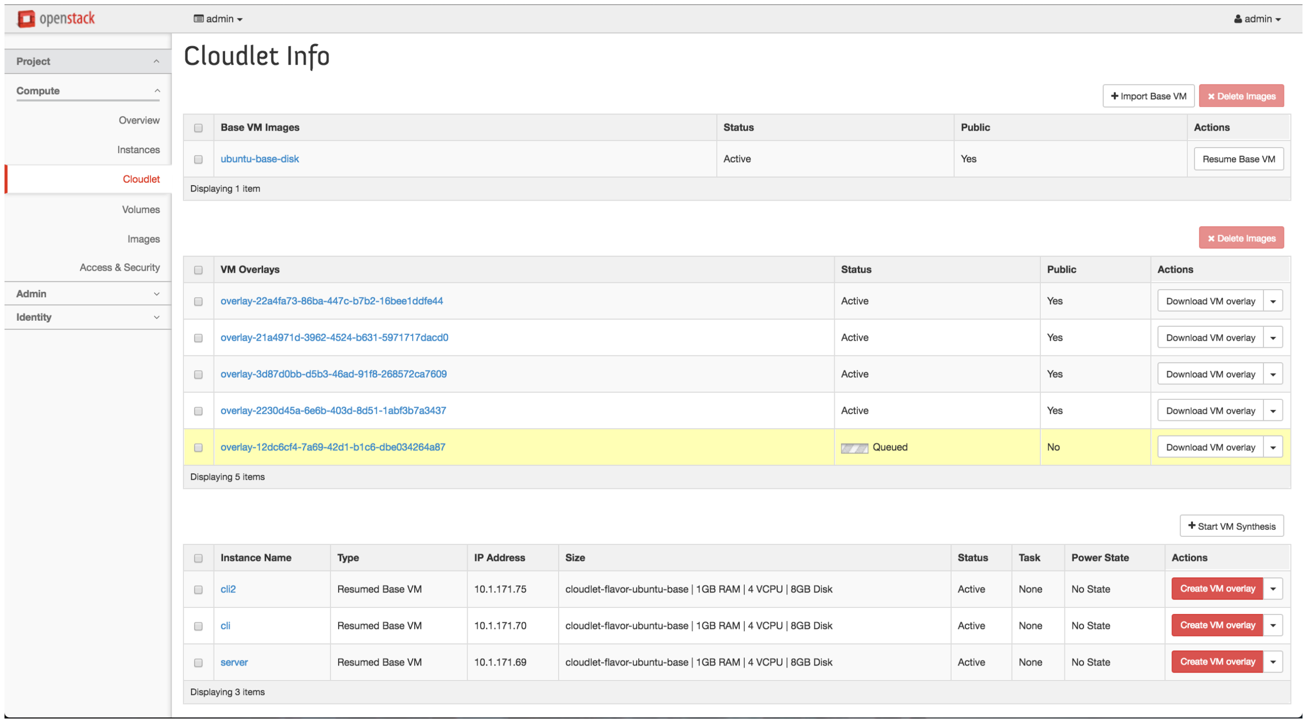
While Openstack deprecated Nova in favor of Neutron, Neutrons stability remains in question by the community. Hence, we decided to use Nova-networking for our implementation.

Nova-networking has three networking modes available: Flat, Flat DHCP, and VLan. Flat networking mode assigns private IP addresses to VMs on startup from a pre-allocated subnet. Furthermore, the network administrator must manually create bridges himself on the compute node. Flat DHCP networking mode addresses this problem by starting a DHCP server (dnsmasq). This server automatically configures IPs upon VMs startup, removing the need for a manual setup. However, these two modes do not support any type of isolation between entities, as they do not differentiate traffic originating from a VM in one tenant from a VM in another.

VLAN networking mode isolates VMs based on tenants. In the compute node, each tenant will receive its own VLAN and a bridge. Then, Openstack achieves isolation through VLAN tagging, where any traffic originating from a VM will be tagged according to its tenancy. VMs that do not live within the same tenant will not be able to communicate.

2.1.2 Devstack

Because of myriad of functionalities and options Openstack provides, it takes much time and effort to configure and run Openstack its bare state. Devstack is a series of scripts that enables quick setup of the Openstack environment. While it is not an official installer for Openstack, it is widely used to deploy Openstack for development purposes. The deployment is ephemeral, but using utilities such as GNU screen allows the sessions to last.



Cloudlet Info

+ Import Base VM | × Delete Images

Base VM Images	Status	Public	Actions
ubuntu-base-disk	Active	Yes	Resume Base VM

Displaying 1 item

× Delete Images

VM Overlays	Status	Public	Actions
overlay-22a4fa73-86ba-447c-b7b2-16bee1ddfe44	Active	Yes	Download VM overlay
overlay-21a4971d-3962-4524-b631-5971717dacd0	Active	Yes	Download VM overlay
overlay-3d87d0bb-d5b3-46ad-91f8-268572ca7609	Active	Yes	Download VM overlay
overlay-2230d45a-6e6b-403d-8d51-1abf3b7a3437	Active	Yes	Download VM overlay
overlay-12dc6cf4-7a69-42d1-b1c6-dbe034264a87	Queued	No	Download VM overlay

Displaying 5 items

+ Start VM Synthesis

Instance Name	Type	IP Address	Size	Status	Task	Power State	Actions
cli2	Resumed Base VM	10.1.171.75	cloudlet-flavor-ubuntu-base | 1GB RAM | 4 VCPU | 8GB Disk	Active	None	No State	Create VM overlay
cli	Resumed Base VM	10.1.171.70	cloudlet-flavor-ubuntu-base | 1GB RAM | 4 VCPU | 8GB Disk	Active	None	No State	Create VM overlay
server	Resumed Base VM	10.1.171.69	cloudlet-flavor-ubuntu-base | 1GB RAM | 4 VCPU | 8GB Disk	Active	None	No State	Create VM overlay

Displaying 3 items

Figure 2.2: Openstack++ interface

2.2 Openstack++

Openstack++ extends the functionality of Openstack to support cloudlets. VM synthesis allows a VM that originated from one cloudlet to freeze its state and resume in any cloudlet, including the origin cloudlet. VM handoff allows VMs to migrate between cloudlets. For example, distance between the user's device and nearby cloudlets can serve as one such metric. As the cloudlet user moves away from Carnegie Mellons campus to his home an hour away from the school, the device will disassociate from CMUs cloudlet and connect to a cloudlet closer to the user's home. Another metric could be the load of a cloudlet where regardless of the distance to nearby cloudlets, the user's device will connect to the cloudlet with the least load.

2.3 Coda

Coda is a distributed file system based on AFS2. Coda supports disconnected operations, client side caching, and continued operation in cases of temporary network failures. It provides an open-close semantics, meaning that Coda will fetch the entire file to the local cache on open and updates to a file will not be propagated until all local writers have closed the file[6].

Coda implements client operations by running a user-process daemon called Venus. Venus handles all the client side logic, including controlling the file system operations, managing the cache, and keeping a persistent log of operations. Venus replays these logs to propagate changes to the servers[6].

Coda manages replication in units of volumes, a subtree of the Coda namespace. Servers replicate these volumes to ensure availability [7]. For better performance, Coda assumes a low amount of write sharing within the system and uses optimistic concurrency control [6]. Coda also allows disconnected operations in case the user lacks network connectivity for a certain period of time [4]. When network connection is restored, Venus replays operations to achieve consistency. For access control, Coda requires that the user logs in with credentials (username and password) using the clog command [7]. Also, a single Coda client can hold multiple tokens to multiple users as long as they authenticate through clog[7].

We chose to run Coda mostly for its familiarity and its capability to operate in WAN environments (versus say, NFS). Coda also provides considerable security measures including encryption, which better suits the cloud.

2.4 Samba / CIFS

Samba is an open-source software used to share files, folders, and printers across SMB clients. SMB, which stands for Server Message Block, is a protocol that operates in the application layer to allow sharing of resources stated above. SMB protocol was initially used in Windows platforms.

Common Internet File System, or CIFS, is an enhanced implementation of the SMB protocol. Unix clients can use the CIFS client in the kernel to mount SMB/CIFS shares located across the network.

SMB2 and SMB3 are more modern implementations of the SMB protocols that offer significant advantages over the CIFS protocol. However, for our purposes, we use the CIFS protocol which is offered as a default in Linux kernels.

Needless to say, although the SMB protocol originated from Windows, Samba ports the protocol to work in Unix-like systems. Hence, the use of SMB guarantees compatibility with most widely-used operating systems families.

Chapter 3

Implementation

3.1 Design Goals

3.1.1 Cohesiveness

We wanted to ease sharing of files between a cohesive group of virtual machines within a single cloudlet and across cloudlets. The ability to share files easily increases productivity of these VMs as the distributed file system handles complex issues such as update propagation and conflict resolution. Integrating a distributed file system with the cloudlet ecosystem also removes the need for the user to manually setup such an architecture to handle simple sharing.

A set of VMs in a single cloudlet launched by the same user can benefit immensely from a shared file system, because they naturally tend to have overlapping working sets. For example, group of developers working on a common source repository on their mobile devices via VMs in the same cloudlet can use the distributed file system to store their repository.

A set of VMs across cloudlets but launched by a same organizational entity can also benefit from a distributed file system. Sticking with the group of developers example, a team can setup a continuous integration server on the cloudlet with their source repository on the distributed file system to allow developers to work together remotely without requiring them to be on the same cloudlet.

Also, multiple modern data workloads run on a distributed file system, such as the popular Mapreduce framework Apache Hadoop. We wanted to run Hadoop using Coda in the cloudlet to emphasize these benefits.

3.1.2 Small VM Overhead

Running a distributed file system in the cloudlet reduces the number of files that need to be kept in each of the VMs. Each VM only needs to keep its working set in its storage, not the entire dataset. Furthermore, reduction of the storage space for each of the VMs implies a smaller overlay image. This in turn leads to a faster migration time of the VM between cloudlets.

3.1.3 Fast File Access

Fast access to the working set for a task is essential in the cloudlet environment. By running a Samba server as a separate VM inside the compute node, other VMs communicate with the nearby Samba server to obtain the necessary files. The speed of access greatly improves as the working set is cached on the Samba server's Coda client which caches files from the remote Coda servers.

3.1.4 File Prefetch Upon Migration

The ability to prefetch working sets and hot files when migration starts allows these files to be ready to be used by the virtual machines when they arrive at the destination cloudlet. By coordinating the Samba servers across cloudlets, we concurrently start the transfer of these recently used files with the migration itself to mitigate the performance hit associated with moving VMs to a cloudlet with a cold cache.

3.1.5 Security

We strived to ensure that the distributed file system does not compromise security in the cloudlet. Not only should running the distributed file system attain the level of security already existing in the cloudlet, a user's data in the file system should only be visible by VMs associated with the user and nobody else.

3.1.6 Familiarity

We wanted to provide a familiar interface to the client VMs with the distributed file system in the cloudlet. This is to offer a familiar interface to both software running on top of our architecture and developers working on new applications within our architecture.

3.2 Design

We present our setup for running Coda in the cloudlet. The cloudlet runs in VLAN networking mode to support tenants. Each tenant has a VM that runs on a well-known IP address. This VM, which we call the File System Master VM (FSMVM), runs the Coda client and exports the mount point to other VMs within the tenancy using Samba. The FSMVM is part of the cloudlet infrastructure, and should be installed by the cloudlet administrator, not the tenant.

The Coda client should be authorized according the tenant user's Coda credentials. Hence, the FSMVM will only have access to files and directories that the tenant user has access to Coda, precisely bound to Coda's ACL. This implies that all VMs within the tenancy will also be bound to the tenant user's credentials, since Samba preserves directory and file permissions when sharing.

Furthermore, the FSMVM runs an OpenVPN server for other client VMs to connect to. All other client VMs within the tenancy connect to the Samba Server on the FSMVM via OpenVPN. Once connected, client VMs can access their folder on Coda using the Samba mount point in their file system tree.

It is crucial for the FSMVMs to run at a well-known address, at least within the tenancy across cloudlets. For example, Tenant 1's FSMVM may run at 128.0.1.5 and Tenant 2's at 128.0.1.6 in cloudlet 1, but in cloudlet 2, Tenant 1's FSMVM must run at 128.0.1.5 and Tenant 2's at 128.0.1.6. This is to ensure that when client VMs migrate across cloudlets (but within the same tenancy), they can discover and reconnect to the FSMVM over the destination cloudlet's internal network. After reconnecting, the VM can resume file system activities by communicating with the destination cloudlet's FSMVM.

3.2.1 Example Setup

Figure 3.1 shows this setup in an arbitrary cloudlet. Two tenants are shown, T1 and T2, with each one running its own FSMVM and a number of worker VMs that respectively connect to their tenancy's FSMVMs. The FSMVM runs a Coda client which is connected to a Coda server (or servers) running elsewhere.

The FSMVMs across tenants run at the same well-known private IP address ()10.1.171.5). They also have the same OpenVPN address so that clients can access them. Furthermore, notice how VM_1 in Tenant 1 has the same private IP address as VM_2 in Tenant 2. This is possible since subnets are bounded by tenancy.

As mentioned before, VMs in one tenancy cannot access VMs in another tenancy, and thus T1_VM_X cannot communicate with any of T2_VM_Ys. The same applies for FSMVMs in each of the tenants. Even though they may be connected to the same Coda server, Coda ACL ensures that files of one tenancy are safe from VMs in another tenancy as long as the two clients have different credentials.

3.2.2 Design Decisions regarding running Coda

Notice we decided to run Coda in a single VM and export the mount to all other VMs within the tenancy. There were two other approaches we considered before coming to this decision: running Coda inside every single VM, and running one FSMVM per cloudlet that would export the Coda share to the entire cloudlet. The first design, running Coda inside every single VM, removes the delay involved with transferring files from the FSMVM to the client VM, since

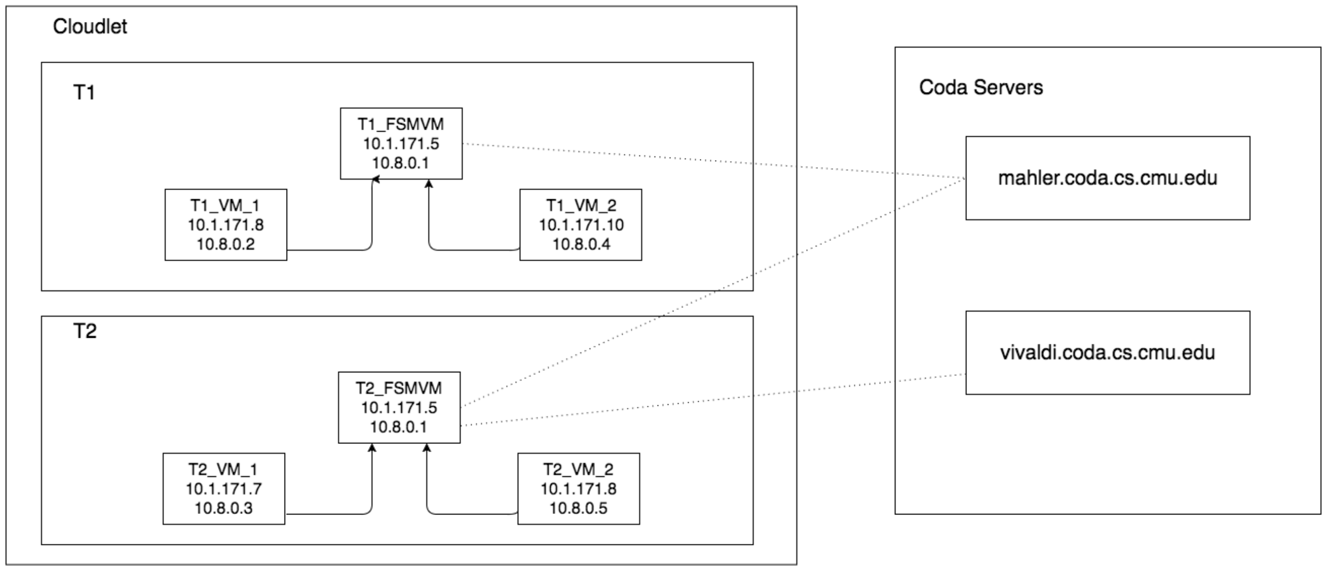


Figure 3.1: Example cloudlet setup

every client VM runs a Coda client that directly communicates with the Coda server and has its own cache. However, the overlay size of these VMs would increase since every client has its own Coda cache, and coherent set of VMs within the same cloudlet performing tasks on similar files will be subject to open-close semantics.

The second design, running one FSMVM per cloudlet that exports the Coda share to all the VMs in the cloudlet introduces access control problems. Notice that Coda enforces access control by having users authenticate and receive tokens, and that a single Coda client can hold multiple tokens for multiple users. Hence, if we ran one FSMVM that held tokens for all users within a single cloudlet and exported the mount via Samba, any VM in the cloudlet would be able to access files of any other user residing in the same cloudlet.

Our design uses the tenancy mechanism in Openstack's VLan networking mode to enforce access control. Furthermore, users can use Samba's share access control features to further fine-tune access control within the tenancy. For example, a software company might not want interns to access sensitive user data. Hence, while granting interns access to VMs within the same cloudlet as full-time employees, the company may create a new Samba user that grants interns access only to non-sensitive data in the Coda mount.

3.3 Architecture

3.3.1 Migration

Since we run OpenVPN in VLAN networking mode, we have the ability to run each tenant with the same IP subnets. For example, both the CMU student who runs his project on the cloudlet

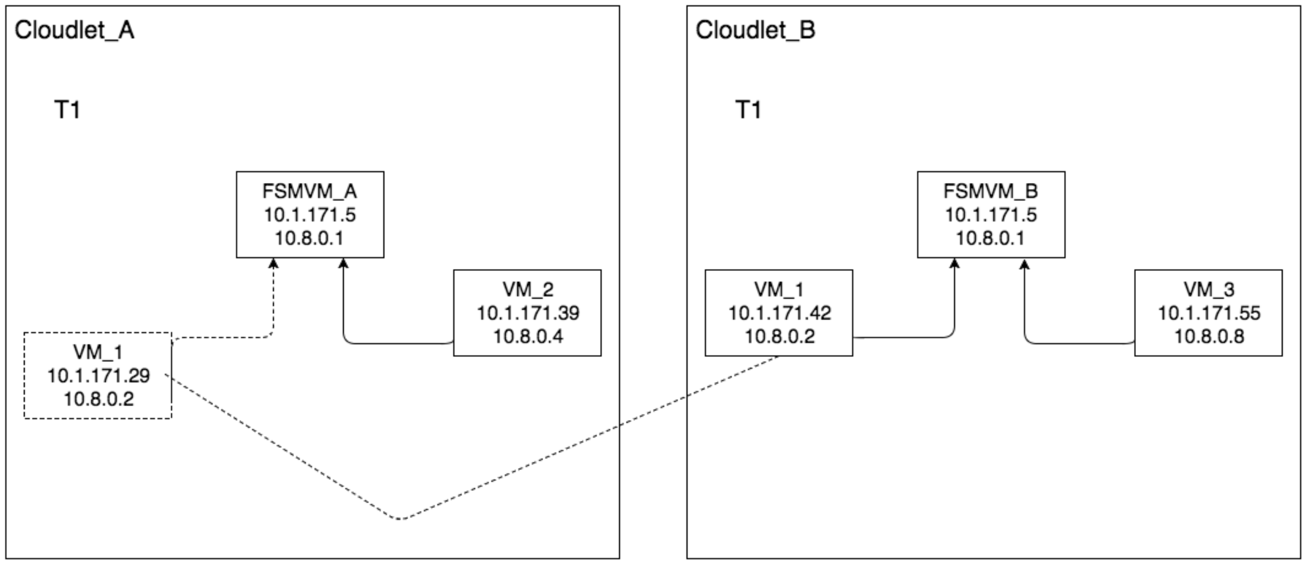


Figure 3.2: Example cloudlet migration

and the group of developers from Intel deploying their service on the same cloudlet can have a subnet of 10.1.171.0/24. This implies that the FSMVM for both tenants will run at a well-known address of say, 10.1.171.5.

Users have the ability to move their VMs from one cloudlet to another, whether they choose to do so manually or through some discovery service mentioned in the Goals section. FSMVMs do not migrate between cloudlets, but rather are present in each cloudlet per tenant as part of the cloudlet infrastructure. When these VMs migrate, they disconnect from the source cloudlets FSMVM and reconnect to the destination cloudlets FSMVM upon arrival. Notice that private IP address of the migrating VM changes when it wakes up in the destination cloudlet. However, since the FSMVMs use a well-known address, the OpenVPN client in the migrating VM is able to identify the OpenVPN server running on the destination FSMVM and reconnect without significant delay.

Figure 3.2 shows an example of the migration procedure. VM_1 migrates from Cloudlet_A to Cloudlet_B. Notice that FSMVM_A in Cloudlet_A does not migrate along with VM_1, and that FSMVM_A and FSMVM_B run at the same private IP address. Once VM_1 arrives at Cloudlet_B, its private IP changes from 10.1.171.29 to 10.1.171.42. However, VM_1's OpenVPN client can still find the OpenVPN server on FSMVM_B since FSMVM_A and FSMVM_B have the same private IP address. Hence, VM_1's CIFS mount can still locate FSMVM_B's Samba export, and VM_1 can continue using the file system immediately after its migration.

3.3.2 OpenVPN

OpenVPN is a open-source solution to running a private VPN network. It uses SSL for key exchange and also for encryption of packets between two endpoints. It can also run on top of

UDP or TCP.

OpenVPN requires a public key infrastructure to run. For the server and each client, both public and private keys are needed. A master Certificate Authority certificate and key are used to sign both the servers and clients certificates. These keys must be pre-generated since they must be verified by the CA. The administrator can generate a limitless number of keys and using OpenVPN does not limit the number of client VMs that can connect to the FSMVM. The cloudlet administrator can automate the generation of keys and configuration files associated with OpenVPN as part of the infrastructure, by either storing them in the VM image or some sort of file transfer mechanism such as scp.

The FSMVM runs an OpenVPN server and clients connect to the FSMVM via OpenVPN. Since the FSMVM runs on a well-known IP, client VMs can configure their OpenVPN clients to connect to the FSMVM.

3.3.3 Why OpenVPN?

The reason that we do not connect clients to the FSMVM directly via the Openstack's network is the delay the client VMs suffer when after migration. When we first tested migration on top of the bare Openstack network, we noticed that access to the local mount point of Samba in the migrating VM freezes for about 300 seconds when it wakes up on the destination cloudlet.

Upon investigation, we realized that the Samba client code inside the Linux kernel waits at least 300 seconds before reestablishing the connection to the Samba server. The issue is a well known one, and several online reports mention the frustration of users with Samba freezing approximately 5 minutes when switching between wired and wireless connections.

Using OpenVPN provides a simple and elegant solution to this problem. OpenVPN server has the ability to setup a directory called the client-config-dir where client configuration files will be kept which are bound to the client key upon creation of the key. The `ifconfig-push` option allows the administrator to allocate a range of OpenVPN IPs to assign to the client with a specific key. By narrowing this range to one IP address, the administrator can effectively assign a single static address to a client. Now, we set client-config-dir to be a folder in Coda, so that when the VM boots in one cloudlet with a specific commonName and a specific IP address, the VM will be bound to that address across all cloudlets.

Furthermore, the keepalive parameter in the server's tun0.conf configuration file allows administrators to control two things: the period of keepalive messages the remote peer sends, and the time to wait until the server/client assumes the remote peer is down. By keeping these periods short, we can effectively remove the delay associated with reestablishing the Samba mount when the VM arrives at the destination cloudlet. Furthermore, both the server and the client check to see if the other side is down, the source FSMVM can also pickup on the VM's departure and free any resources associated with the old connection.

Using OpenVPN also creates another layer of indirection in the stack, thus adding flexibility without relying on a specific component in the system. For example, if we modify the Samba client code inside the Linux kernel to remove this delay, we create a dependency on Samba development. However, with this approach, we actually have the option to deploy another overlay network instead of OpenVPN or to even write our own if a better alternative becomes available or the need arises. We also have the option to disable encryption completely on OpenVPN, removing the overhead associated with encryption when using OpenVPN.

3.3.4 Design Decisions Regarding the Delay

We considered several alternatives other than running OpenVPN to solve the migration delay problem. However, none of them provided a workable solution for different reasons.

Changing the kernel code

The first solution consisted of changing the Samba code inside the kernel. The Samba client code resides inside the Linux kernel as an implementation of the VFS interface. It can be found in the `/fs/cifs` directory within the source code tree. In the server_unresponsive function in `connect.c`, the CIFS code reconnects to the server after `2 * SMB_ECHO_INTERVAL` seconds have passed without any interaction with the server. The value of the `SMB_ECHO_INTERVAL` constant varies by the Linux distribution of the kernel.

Browsing online, we discovered that the long reconnection delay is due to CIFS clients being careful not to overload the Samba server with reconnection requests. One client's frequent reconnect request may not slow the Samba server down, but several of these requests may. Furthermore, Samba servers hold quite a bit of information associated with each client's session, so it may be wise to delay resetting the connection in hopes of the server becoming responsive in an enterprise environment. In previous versions of the Linux kernel, a way to change `SMB_ECHO_INTERVAL` existed as a command line option. However, developers of CIFS removed that option as they felt 300 seconds to be an adequate standard. Hence, changing the constant value inside the kernel source was the only option if we wanted to lower the delay at the destination cloudlet.

We avoided the approach due to complications involved with directly changing the kernel source. If we modify `SMB_ECHO_INTERVAL` inside the kernel, we would need to modify the constant in every VM image the users wanted to boot in the cloudlet. Furthermore, this creates a dependency to the Samba development as well, since `SMB_ECHO_INTERVAL` is an internal constant not exposed to users and developers are free to tinker with it.

Identical private IP upon arrival

The second solution comprised of forcing the migrating VM to have the same IP address on the destination cloudlet as it did on the source cloudlet. There exists an API to assign a specific private IP address to a VM via creating a port in the Neutron networking mode. After creation of the port, the administrator has the option to boot a VM using the port. However, elijah-cloudlet uses the Nova networking mode, and in Nova-networking, there is no API to assign a specific private IP to a VM upon its boot. Furthermore, even with the Neutron networking mode, we would have to incorporate the port mechanism with the migration feature. This would involve dealing with a pre-existing VM in the destination cloudlet that may have the same private IP as the migrating VM, which entails additional complexities.

Agent daemon inside the VM

The last solution consisted of having an agent daemon program inside the VM which would unmount the Samba mount on start of the migration process and remount it upon arrival. First, some authority program in the cloudlet gives a signal to the daemon that the VM is about to migrate. Then, the daemon observes the processes who are currently using the mount point via the lsof command. If there are no processes in the VM using the mount point, the daemon goes ahead and unmounts the Samba mount to begin the migration. Otherwise, the daemon performs a lazy unmount on the mountpoint. In either case, the daemon remounts the Samba mount using the FSMVM from the destination cloudlet.

The complexities involved with this approach stems from unmounting the mount point. Unmounting when no process uses the Samba mountpoint is simple. However, after performing a lazy unmount and migrating to the destination cloudlet, the mount point still hangs for 300 seconds before reconnecting to the destination FSMVM.

3.4 Security and Access Control

As mentioned before, VMs cannot communicate with other VMs across tenancy. This ensures isolation between tenants and thus adds a measurable level of security to the system. Furthermore, we ensure that no tenant can access other tenants data through Coda by running one FSMVM per tenant. Client VMs within the tenancy can only access folders authorized to the account logged in on the FSMVMs Coda client. Also, Coda uses authentication secret-keys to ensure secure connection between the client and the server, which means that any communication between the FSMVM and the Coda server benefits from Coda's cryptosystem. On top of access control provided by Openstack's tenancy system, users can further enforce fine-grained access control within the tenancy by using Samba's access control system.

Chapter 4

Validation

4.1 Live Migration

We tested our architecture within the Openstack++ environment to ensure that it functions as expected with the live migration feature. We wanted to test three things. Firstly, we wanted to test that once the migrating VM arrives at the destination cloudlet, it stops its connection to the source FSMVM. Secondly, we wanted to test that the migrating VM starts using the destination FSMVM upon arrival. Lastly, we wanted to test that the Samba mount on the migrating VM becomes available without significant delay upon arrival.

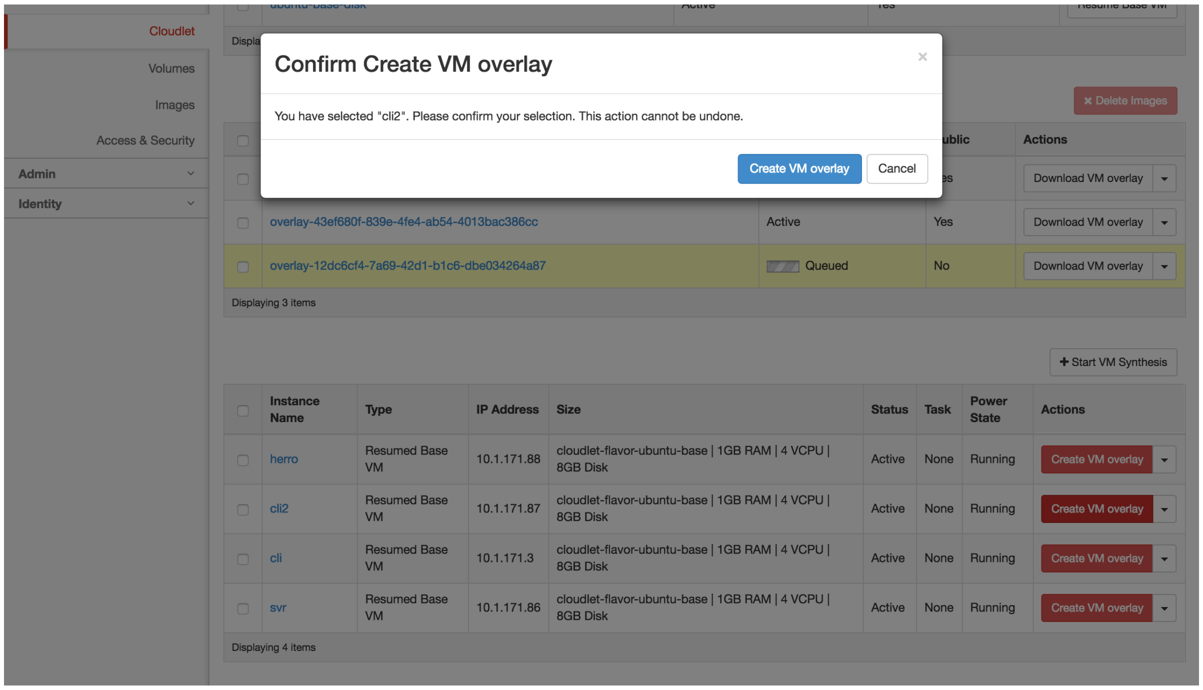
4.1.1 VM Overlays

VM overlays contain the delta between the client VM and the base image VM. They contain all the changes needed to rebuild the state of the client VM at the moment of migration including memory and disk state. Openstack++ deduplicates the data within the overlay file for compression. Upon migration, Openstack++ transfers the VM overlay over the network from the source cloudlet to the destination cloudlet. Then, the destination cloudlet synthesizes the client VM by applying the deltas over the base VM image, which the destination cloudlet already has a copy of.

4.1.2 Setup

We used two machines, squall and fog, located in our lab's machine room to serve as two cloudlets.

We conducted three experiments. The first one consists of manually creating an overlay of a client VM in squall, and synthesizing the overlay in squall again. Although synthesizing a VM in the source cloudlet rarely occurs in natural settings, it serves as a quick proof of concept since the private IP of the VM changes upon synthesis. In the next experiment, we also manually create



Confirm Create VM overlay

You have selected "cli2". Please confirm your selection. This action cannot be undone.

			ublic	Actions
			es	Download VM overlay
	overlay-43ef680f-839e-4fe4-ab54-4013bac386cc	Active	Yes	Download VM overlay
	overlay-12dc6cf4-7a69-42d1-b1c6-dbe034264a87	Queued	No	Download VM overlay

Displaying 3 items

	Instance Name	Type	IP Address	Size	Status	Task	Power State	Actions
	herro	Resumed Base VM	10.1.171.88	cloudlet-flavor-ubuntu-base | 1GB RAM | 4 VCPU | 8GB Disk	Active	None	Running	Create VM overlay
	cli2	Resumed Base VM	10.1.171.87	cloudlet-flavor-ubuntu-base | 1GB RAM | 4 VCPU | 8GB Disk	Active	None	Running	Create VM overlay
	cli	Resumed Base VM	10.1.171.3	cloudlet-flavor-ubuntu-base | 1GB RAM | 4 VCPU | 8GB Disk	Active	None	Running	Create VM overlay
	svr	Resumed Base VM	10.1.171.86	cloudlet-flavor-ubuntu-base | 1GB RAM | 4 VCPU | 8GB Disk	Active	None	Running	Create VM overlay

Displaying 4 items

Figure 4.1: Creating a VM overlay

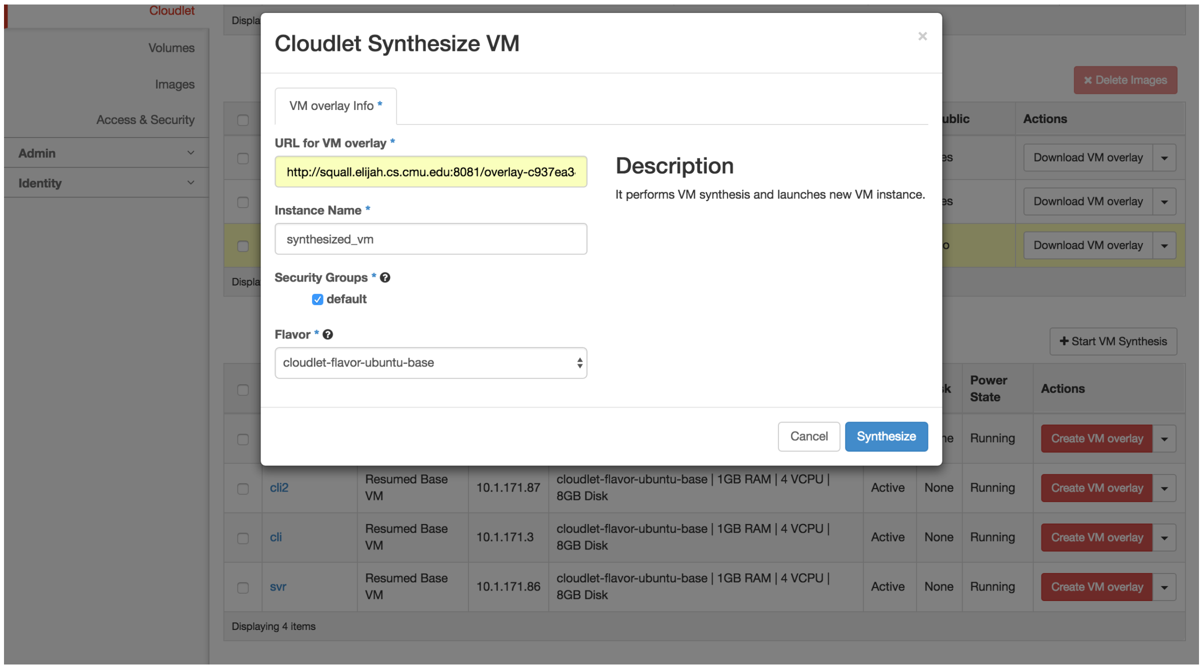
an overlay of a client VM in squall, but synthesize the overlay in fog. This accurately depicts the scenario where the VM migrates to a different cloudlet, but with the user manually creating the overlay. Lastly, we migrate the client VM from squall to fog using the live VM handoff feature in Openstack++.

4.1.3 Manual Synthesis on a Single Cloudlet

We first create a VM overlay of a client VM in squall via Openstack++'s overlay creation as depicted in figure 4.1. Then, we upload the VM overlay to nginx running in squall since the Openstack++ requires that an overlay image be served by a web server. As portrayed in figure 4.2, we enter the address of the VM overlay image and we synthesize the VM back in squall. We confirm that the delay for the client VM to reconnect to the FSMVM and reestablish the Samba mount is approximately 10 seconds.

4.1.4 Manual Synthesis Across Cloudlets

We create an overlay of a client VM in squall and upload the image to nginx in squall similarly to the previous experiment. However, this time we synthesize the VM in fog. This simulates synthesis of a client VM where the source and the destinations cloudlets are different. With this experiment, we not only observe the reconnection delay but also that the client VM disconnects from the source FSMVM (in squall) and starts using the destination FSMVM (in fog) upon



Cloudlet Synthesize VM

VM overlay Info *

URL for VM overlay *
http://squall.elijah.cs.cmu.edu:8081/overlay-c937ea3

Instance Name *
synthesized_vm

Security Groups *
☑ default

Flavor *
cloudlet-flavor-ubuntu-base

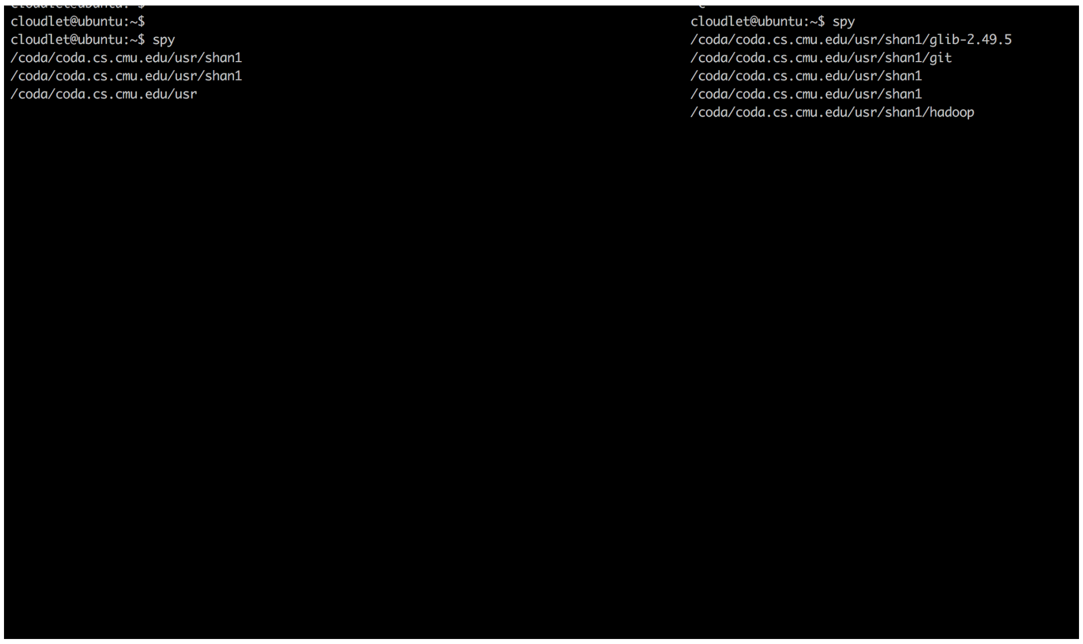
Description

It performs VM synthesis and launches new VM instance.

Cancel | Synthesize

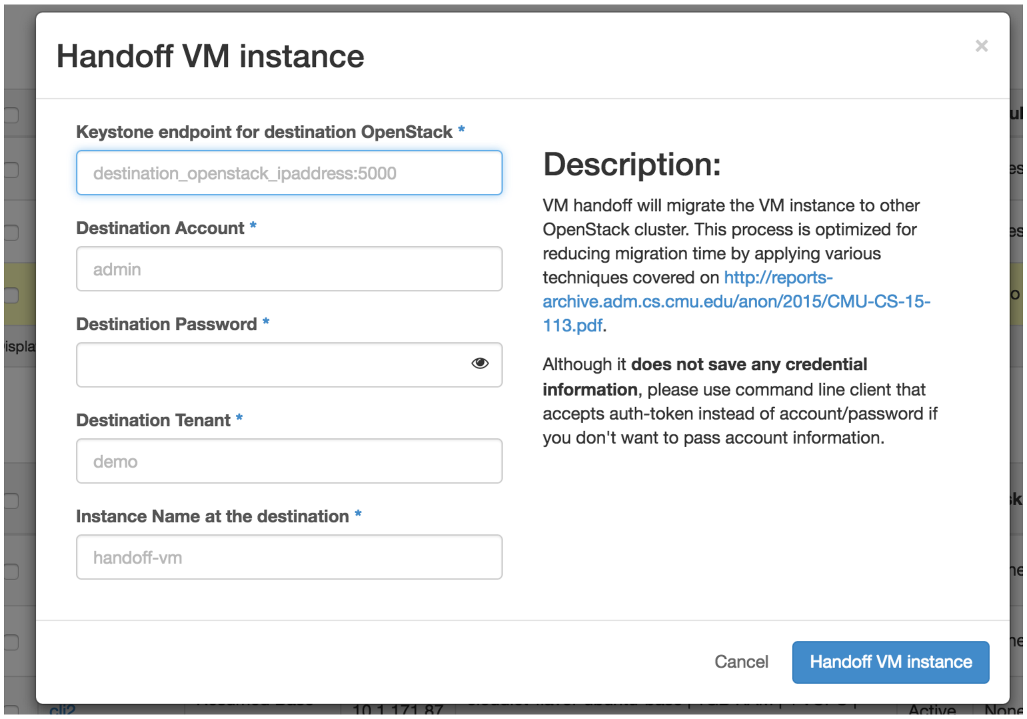
							Power State	Actions
cli2	Resumed Base VM	10.1.171.87	cloudlet-flavor-ubuntu-base | 1GB RAM | 4 VCPU | 8GB Disk	Active	None	Running	Create VM overlay
cli	Resumed Base VM	10.1.171.3	cloudlet-flavor-ubuntu-base | 1GB RAM | 4 VCPU | 8GB Disk	Active	None	Running	Create VM overlay
svr	Resumed Base VM	10.1.171.86	cloudlet-flavor-ubuntu-base | 1GB RAM | 4 VCPU | 8GB Disk	Active	None	Running	Create VM overlay

Figure 4.2: Beginning VM synthesis



```
cloudlet@ubuntu:~$                                                        cloudlet@ubuntu:~$ spy
cloudlet@ubuntu:~$ spy                                                    /coda/coda.cs.cmu.edu/usr/shan1/glib-2.49.5
/coda/coda.cs.cmu.edu/usr/shan1                                           /coda/coda.cs.cmu.edu/usr/shan1/git
/coda/coda.cs.cmu.edu/usr/shan1                                           /coda/coda.cs.cmu.edu/usr/shan1
/coda/coda.cs.cmu.edu/usr                                                 /coda/coda.cs.cmu.edu/usr/shan1
                                                                          /coda/coda.cs.cmu.edu/usr/shan1/hadoop
```

Figure 4.3: Spy command from both squall and fog FSMVMs


Handoff VM instance

Keystone endpoint for destination OpenStack *

destination_openstack_ipaddress:5000

Destination Account *

admin

Destination Password *

Destination Tenant *

demo

Instance Name at the destination *

handoff-vm

Description:

VM handoff will migrate the VM instance to other OpenStack cluster. This process is optimized for reducing migration time by applying various techniques covered on http://reports-archive.adm.cs.cmu.edu/anon/2015/CMU-CS-15-113.pdf.

Although it **does not save any credential information**, please use command line client that accepts auth-token instead of account/password if you don't want to pass account information.

Figure 4.4: Handing off a VM to another cloudlet

arrival to the destination cloudlet. In order to verify this, we rely on an administrative tool in Coda called spy. Spy is a commandline tool which creates a trace of recently touched files. We run Spy on both the squall FSMVM and fog FSMVM to observe both Coda clients' activities before and after migration.

We first notice that the delay for the client VM to reestablish the mount is also approximately 10 seconds. Figure 4.3 shows the results of this experiment. The left tmux screen runs spy on squall's FSMVM, and the right tmux screen runs spy on fog's FSMVM. Here, we see that once the VM migrates, the folders being accessed by the migrated VM on fog does not appear on squall's spy command.

4.1.5 VM Handoff Across Cloudlets

We use the VM handoff feature in Openstack++ to perform live migration of a client VM from squall to fog. Figure 4.4 depicts the VM handoff interface. We can only handoff VMs that have been synthesized, so we synthesize the client we created in the previous section in squall by using the existing overlay file. We enter the destination cloudlet's address with the Keystone port (fog.elijah.cs.cmu.edu:5000) along with the account and password on the destination cloudlet. Furthermore, we enter the destination tenant and name of the VM to be used once it arrives at the destination.

We again notice that the delay for the client VM to reestablish the mount is 10 seconds, and that Spy running on both FSMVMs behave as expected similar to figure 4.3

Chapter 5

Prefetching

5.1 Background

Prefetching allows the destination cloudlet to simultaneously transfer the working set from the source cloudlet with the VM migration. Once the migrating VMs arrive and wake up at the destination cloudlet, frequently and recently used files will be pre-loaded locally in the destination FSMVM. Without prefetching, the FSMVM's Coda client will see a cache miss for the requested files, and hence will have to first fetch the files from the Coda server. Instead, the destination FSMVM VMs will have the working set locally cached ready to be served out to the client VMs upon their arrival.

The weaker the connection between the FSMVM and the Coda servers is, the more prefetching affects the performance of our architecture. While the client VMs and the FSMVM are connected by the stable and fast local network within the cloudlet, the FSMVM and the Coda servers are likely to be connected via WAN. Hence, file transfer between the FSMVM and the Coda servers is the main bottleneck in serving the necessary files to the client VMs from the file system.

With prefetching, we look to mitigate this bottleneck by fetching files likely to be used by the migrating VMs upon arrival as the VMs migrate to the destination cloudlet. Working sets can be inferred from observing the file system usage by the VMs in the source cloudlet. The files frequently and recently accessed by the VMs in the source cloudlet are very likely to be accessed soon by the VMs in the destination cloudlet. Hence, fetching these files before the VMs wake up in the destination cloudlet in the FSMVM helps increase file access speed upon arrival.

5.2 Implementation

We implement the prefetching feature in two parts. First, we modify Venus, the user-space Coda client, to dump the list of files in the cache along with the priority for each of the files. Then, we write a script that takes the dump from Venus and derives a list of files that should be transferred to the destination cloudlet.

5.2.1 Priority

Coda uses priority as a metric to measure importance of a file. Coda calculates priority as a function of two things: short-term priority and medium-term priority. The two priorities are multiplied by their respective weights and added together to calculate the overall priority.

Medium-term priority is just the hoard priority. Hoarding is a way for users to inform Venus of important files that should be kept in the cache most of the time. Users can assign relative priorities to the files and Venus keeps a database of these hoard values. Hoarding priority also affects how Coda handles disconnected operations, as files with high hoard priority are more likely to be available offline. However, in our context, hoard priority does not affect the overall priority as no hoarding is done by the FSMVM.

The short-term priority conveys the recency of the file's usage. It is calculated by scaling the difference between the file's most recent reference and the most recent reference to any file in the Coda file system and subtracting it from the maximum possible short priority value. Since no hoarding is done, the overall priority becomes a function of how recently a file was used.

5.2.2 Modification to Venus source

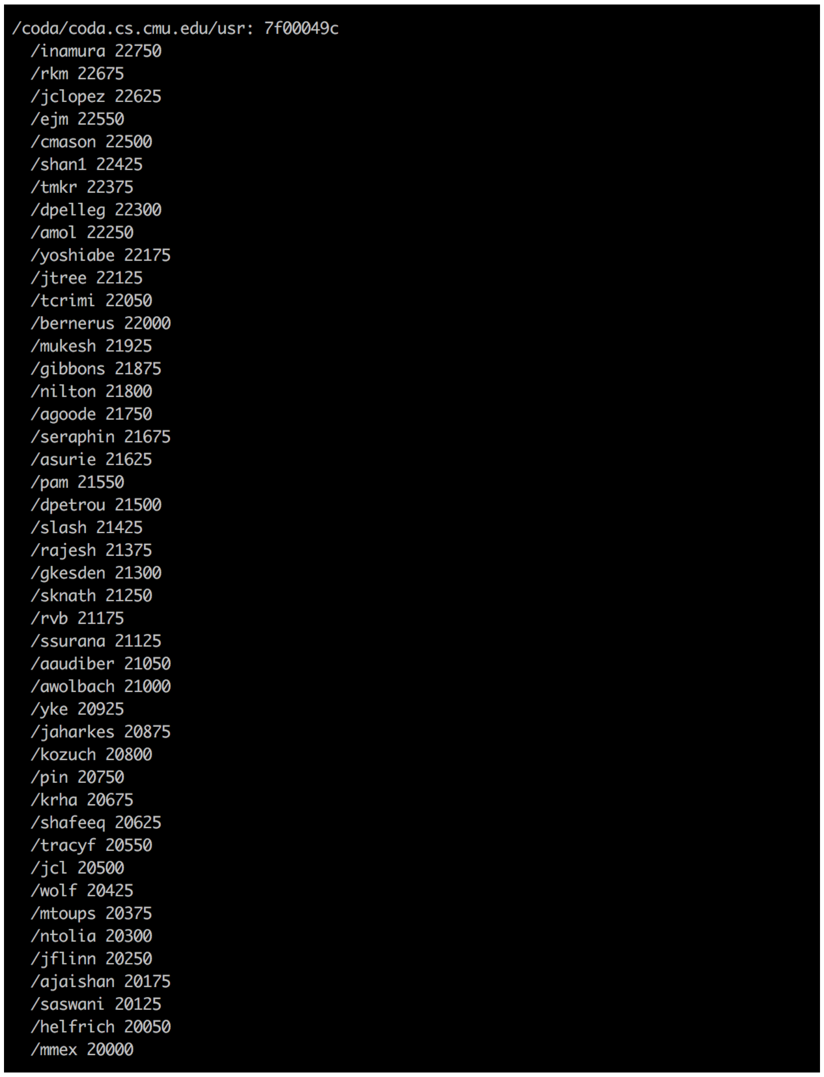
Venus uses a class called `fsobj` as a representation of files and directories within the file system. It is declared in `/venus/fso.h` header file. The `fsobj` can be either a file or a directory, with parent and child relationships defined as expected. The `fsobj` already has a priority value in its class declaration, which is used by Venus to manage cache eviction and hoarding.

Venus also has a function to list the fsobjs in the cache currently by printing a dump of the cache to stdout. The function is aptly called `listCache`, and the user can invoke it by running `cfs lc` on the command line. Venus walks the cache from the root directory and dumps the validity and the path of the `fsobj`. An `fsobj` is deemed valid when its data is valid, status is valid, and it is not dirty.

We modified the `ListCache` function for an `fsobj` in the `/venus/fso1.cc` file to dump the priority of the `fsobj` as well as the path and the validity. Figure 5.1 demonstrates the modified `ListCache` function. Next to the path of each user directory in the `/coda/coda.cs.cmu.edu/usr` directory, the `ListCache` function displays the calculated priority of the child directories.

5.2.3 Coda Migrator

We wrote a Python script named the Coda Migrator to implement the actual transfer of high priority files from the source cloudlet to the destination cloudlet. The Coda Migrator is a daemon that runs in the FSMVM of each tenant in the cloudlet. Once the Coda Migrator is notified of the



```
/coda/coda.cs.cmu.edu/usr: 7f00049c
  /inamura 22750
  /rkm 22675
  /jclopez 22625
  /ejm 22550
  /cmason 22500
  /shan1 22425
  /tmkr 22375
  /dpelleg 22300
  /amol 22250
  /yoshiabe 22175
  /jtree 22125
  /tcrimi 22050
  /bernerus 22000
  /mukesh 21925
  /gibbons 21875
  /nilton 21800
  /agoode 21750
  /seraphin 21675
  /asurie 21625
  /pam 21550
  /dpetrou 21500
  /slash 21425
  /rajesh 21375
  /gkesden 21300
  /sknath 21250
  /rvb 21175
  /ssurana 21125
  /aaudiber 21050
  /awolbach 21000
  /yke 20925
  /jaharkes 20875
  /kozuch 20800
  /pin 20750
  /krha 20675
  /shafeeq 20625
  /tracyf 20550
  /jcl 20500
  /wolf 20425
  /mtoups 20375
  /ntolia 20300
  /jflinn 20250
  /ajaishan 20175
  /saswani 20125
  /helfrich 20050
  /mmex 20000
```

Figure 5.1: Example dump of Coda cache from modified Venus

migration of VMs, it takes the dump of the cfs lc command and parses it. From the parsed list of fsobjs, the Coda Migrator determines at most `MAX_MIGRATION_NUM`, a preset constant, many files with the highest priority. Then, it contacts the Coda Migrator in the destination cloudlet, and sends the list of high priority files that should be prefetched from the Coda server to the destination cloudlet's FSMVM. Since Coda has open-close semantics, the Coda Migrator uses the open system call to fetch the file from the Coda server. Once the open call returns, it immediately closes the file, and moves to open the next file in the list until there are no more files to prefetch.

The timing of the notification of migration to the Coda Migrator is crucial. If the daemon is notified of the migration before the VM itself is suspended, then the Coda Migrator's dump of the cache (i.e. the output from `cfs lc`) can be stale as the VM can access files between `cfs lc` and the creation of the overlay file. On the other hand, as the time elapsed between the start of the creation of the overlay file and the notification to the Coda Migrator increases, we have less time to transfer files to the destination FSMVM hence missing the opportunity to prefetch more files before the migrating VM wakes up on the destination cloudlet.

5.2.4 Notification of Migration

As of now, there exists no mechanism in the cloudlet ecosystem to notify the migration of VMs to the Coda Migrator. The users must manually inform the daemon of a VM migration and the address of its destination FSMVM. This also implies that FSMVMs must have their own public IPs so that the Coda Migrator in the source cloudlet can transfer the list of files that must be prefetched over the network to the destination FSMVM. In the future, we hope to integrate the cloudlet discovery feature with prefetching to remove these constraints.

To remove the need for the FSMVMs to have public IPs, we can devise an architecture where the cloudlet discovery component contacts the source cloudlet, which in turn contacts the corresponding FSMVM. The FSMVM directly transfers the list of files to be prefetched to the cloudlet discovery component. The same applies for the destination cloudlet, in that the cloudlet discovery component contacts the destination cloudlet which in turn contacts the destination FSMVM with the list of files to prefetch.

Another approach we can take involves sharing the list of files to prefetch on Coda itself. This involves a naming scheme in Coda for each cloudlet (i.e. its IP) so that the source cloudlet can identify where to place the list of files, and so that the destination cloudlet can identify which files to fetch.

5.3 Example: Pause and resume analysis

Figure 5.2 shows an example of the prefetching feature. Tenant 1 has one VM running on Cloudlet A and is running a Python program that analyzes a series of ratings on movies made by one person for future recommendations. The Python program writes the results out in the results.data file. In the midst of running the program, tenant 1 migrates VM_1 to Cloudlet B.

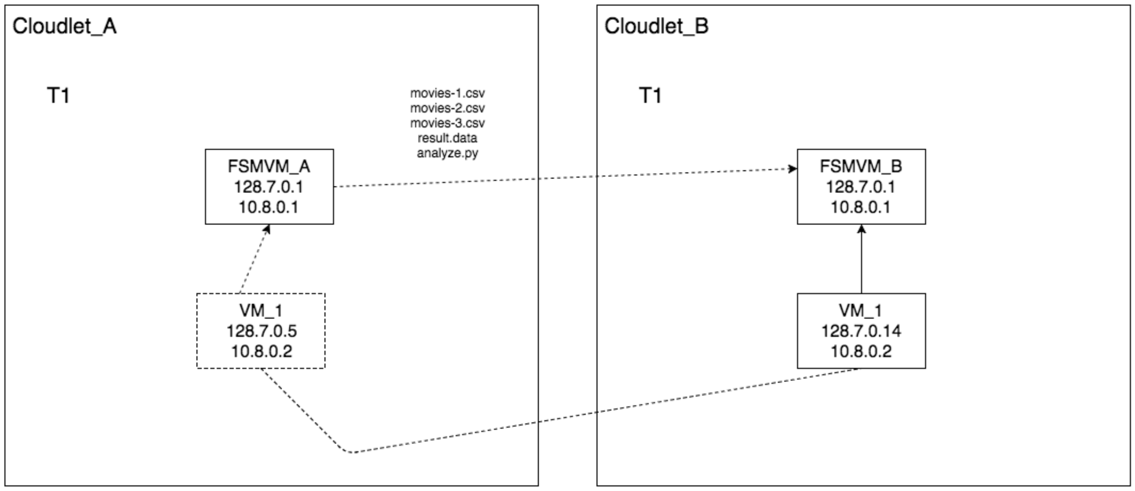


Figure 5.2: Example of prefetching important files

The priorities of the csv files and the result file are very high in the Coda client running in FSMVM_A, since the Python program used them until the point of disassociation from Cloudlet A. Furthermore, the Python program itself has high priority in FSMVM_A.

Now, as described in the Notification of Migration section, FSMVM_A receives a signal from a migration control authority that VM_1 is moving to Cloudlet B. Noticing that the Python program, the csv files, and the results file all have high priorities, the Coda Migrator in FSMVM_A sends the list of files to FSMVM_B. The prefetching of these files occurs simultaneously with the migration of VM_1. More specifically, as cloudlet A creates the VM overlay file and transfers the overlay file to the destination cloudlet, cloudlet B resumes the VM using the overlay file, these high-priority files are transferred in parallel.

Now, upon waking up, VM_1 continues running the analyze.py Python program. FSMVM_B also has the csv datasets and the result.data files locally, and VM_1 can fetch them through the LAN again which is a significant speedup over fetching these files on demand from the Coda servers potentially located far away from cloudlet B.

5.3.1 Migration with an Open File

Notice that VMs can migrate without suspending a program. If a client VM has a program that is holding a reference to a file in the distributed file system and migrates to another cloudlet without suspending the program, our distributed file system exhibits a defined behavior, albeit a confusing one. Since Coda provides open-close semantics, when the lease on the files that the migrating VM is holding expires, the source FSMVM will close these files. Upon closure, the updates on the FSMVM will propagate globally. However, it is not guaranteed that these updates will be propagated before the VM wakes up in the destination cloudlet. If not, the file will have conflicts that needs to be solved if the VM continues to update the file after it arrives on the destination.

5.4 Prefetching Experiment

5.4.1 Separate Compilation

Continuous Integration

Continuous integration looks to frequently unify developers' working copies in a mainline build, trying to maintain a single stable source build at all times. With CI, the development team keeps a single source that reflects the most recent functional version of its code. As a developer looks to modify the program, he checks out the source from this mainline repository to his local development machine. He makes the necessary changes, including writing new tests, and builds and tests his changed source on his local machine. When the developer is ready to commit his code, he first checks the main repository for changes. If there are no changes, he is free to commit his code to the mainline. If there are changes made by others, the developer first pulls those changes in locally and rebuilds to check for conflicts. After solving any possible conflicts, the developer finally pushes his changes into th e mainline repository. However, with CI, the mainline code on the committed repository is built again on an integration machine. This is done to check for any missed changes on the developer's last local build and test. Furthermore, many modern tech companies run a large number of tests that include unit tests, feature tests, and end-to-end tests, that the developer's local machine might not be able to handle with its spec. These tests can be scheduled to run on the more powerful integration machine at a frequency of the team's desire (for example, once a day).

CI in the Cloudlet

Running CI in the cloudlet allows a team of developers to share the FSMVM cache and to reduce compilation time because of common object files. We imagine a scenario where a developer writes code on his mobile device, and has a CI server running in one of the VMs in the cloudlet. He finishes making changes in the source repository, and builds the updated source in one cloudlet. Afterwards, he moves locations and his VM migrates to another cloudlet. Now, notice that the FSMVM in the cloudlet he just moved to has none of the files needed for compilation locally without prefetching. However, with prefetching, the destination FSMVM would have fetched the files needed for building simultaneously with the migration of the client VM, and hence would have the necessary files when the client VM requests another build in the new cloudlet.

Separate Compilation for Development

Lots of modern IDEs perform partial compiling to aid developers. For example, Eclipse displays the available methods for an object in Java when requested by the developer. The same applies for other languages such as C++ and Python. However, most of the time, the machine the developer uses to develop code is not the machine the code will run upon deployment. Hence, partial compilation may not work in the developer's local machine due to many reasons, including missing modules and incompatible operating systems.

Separate Compilation in the Cloudlet

We suggest a scenario where the developer merely edits code on his mobile device, and leaves compilation to a more compatible and powerful VM in the cloudlet. The developer could be using his Surface tablet to develop a Linux program, or even on an iOS mobile phone. Using a cloudlet for separate compilation greatly aids the developer in that he will be able to receive feedback from the IDE quickly because of the low-latency communication between his device and the cloudlet. Furthermore, the developing platform can be OS-agnostic, in that one can use separate compilation to compile code on one operating system while editing code on another.

5.4.2 Experiment

Setup

The setup for our CI experiment exactly the same as the one mentioned in the Evaluation section. Squall and Fog remain as our two cloudlets, with Squall being the source and Fog being the destination cloudlets regarding migration. Also notice that the two Coda servers, mahler and vivaldi, are within CMU as well.

We use the Coda File System source to simulate continuous integration in a cloudlet environment. The source repository is located in squall's FSMVM, so that it is shared through Samba to all the client VMs. We run configure in one of the client VMs, so that we create the files needed for compilation, including makefiles, in the client VM's local file system. Then, we run make in the client VM, so that it fetches the source from the FSMVM and builds the Coda binary on the client VM. Afterwards, we migrate the client VM to fog once with prefetching and once without it, and rebuild. We compare the resulting two times against each other.

Results

Tables 5.1 illustrate the results from our experiments. Notice that running make on the cached FSMVM runs about 10% faster than on the uncached FSMVM. The results clearly highlight the advantages of the prefetching feature. Furthermore, as mentioned in the setup section of this experiment, two Coda servers mahler and vivaldi are located within CMU and the Coda source code tree is about 25M. As the DFS' servers move away from the cloudlets and as the source tree increases in size, we expect to see more speedup with prefetching.

We also ran the experiment on a freshly booted VM without any prior runs of the compilation and obtained similar results. This shows that the results are not affected by the page cache.

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
Prefetch enabled	2m29.660s	2m30.108s	2m28.906s	2m29.853s	2m30.017s	2m29.710s
Prefetch disabled	2m44.982s	2m45.091s	2m47.827s	2m42.127s	2m49.819s	2m45.969s

Table 5.1: Results from 5 runs of make for migration with and without prefetching enabled

Chapter 6

Hadoop

In this section, we run Hadoop on top of our architecture to demonstrate that pre-existing and widely used applications can run within our system without modifications to their source.

6.1 Hadoop

Hadoop is an open-source framework which aids developers in processing large data with distributed computing. It usually runs on a cluster of commodity hardware. Hadoop is modeled on Google's MapReduce[2], and runs on Hadoop Distributed File System, which in turn is modeled on Google's GFS[3].

Hadoop's processing model derives from functional languages, where the user applies operations such as maps and reduces on data to generate a new set of data. The transformations are applied in separate chunks, where a single process, called the worker process, handles one chunk of the operation. One node in the cluster can have several worker processes. The original data is stored in HDFS, and worker processes fetch their respective chunks out of HDFS at the start of the map phase.

First, the map phase applies a user defined map transformation to the original data, usually to a list of key value pairs. Then, Hadoop relocates all the key value pairs with the same key into the same reduce process. Notice that one reduce process can handle key value pairs for more than one key. Then, each reduce worker reduces the values with the same keys to (usually) a single value, and outputs the list of keys back to the user.

6.2 Experiment

As mentioned in the previous section, Hadoop normally runs on HDFS. However, we run Hadoop on top of our distributed file system on the cloudlet to demonstrate that our architecture can handle a preexisting and widely-used application without modifications to our architecture.

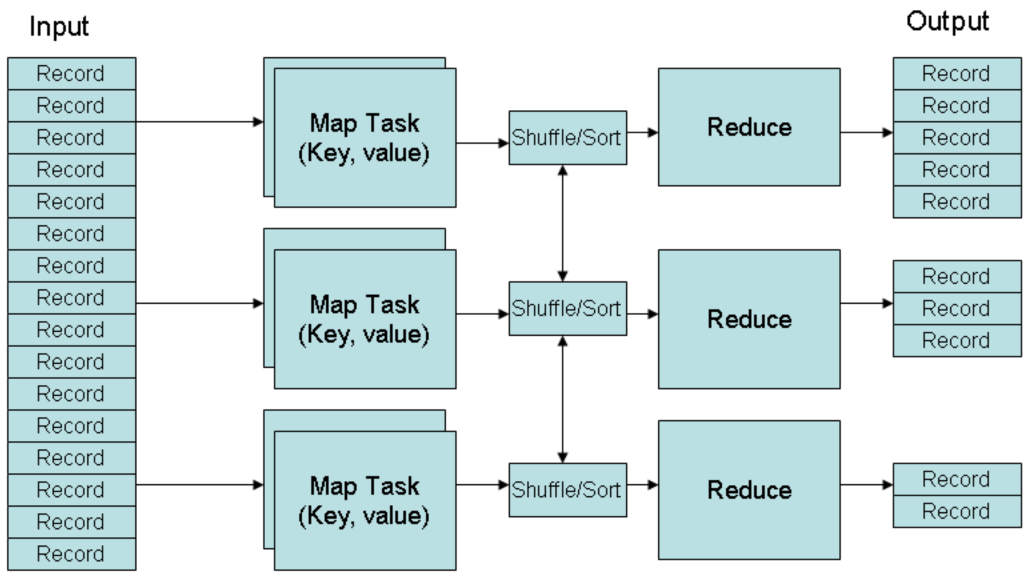


Figure 6.1: Illustration of Hadoop's mechanism

6.2.1 Setup

We use Hadoop 1.1.2 for our experiment. This version provides the functionality necessary to change the default file system from HDFS to a distributed file system. First, we boot up two VMs. One VM will act as a hadoop-master and one will act as hadoop-slave. We set up our VMs and mount the distributed file system by connecting them to the FSMVM. Then we install Hadoop 1.1.2 on both VM's local file system, and configure the basic setup involving establishing passwordless ssh and setting the right environment variables such as JAVA_HOME. Then, we make the necessary changes in Hadoop's config files to use our distributed file system instead of HDFS.

In core-site.xml on both master and slave, we add

```
<configuration>
<property>
  <name>fs.default.name</name>
  <value>file:///</value>
</property>

<property>
  <name>hadoop.tmp.dir</name>
  <value>/home/cloudlet/coda/coda.cs.cmu.edu/usr/shan1/hadoop</value>
</property>
</configuration>
```

Changing the fs.default.name configuration allows us to notify Hadoop that we wish to use our own file system rather than HDFS. The hadoop.tmp.dir option sets the root folder for all tmp directories that Hadoop uses. This is usually the folder that HDFS uses, but since we use our own DFS here, we enter a folder within our Samba mountpoint for the tenant's user. We also add

```
<property>
  <name>mapred.job.tracker</name>
  <value>master:8021</value>
</property>

<property>
  <name>mapred.local.dir</name>
  <value>/tmp/mapred-local</value>
</property>
```

We point the mapred.local.dir to the local tmp directory for the outputs of our intermediate files after the map phase. Since the Hadoop community recommends that we use a local directory for the option, we point it to the local /tmp directory.

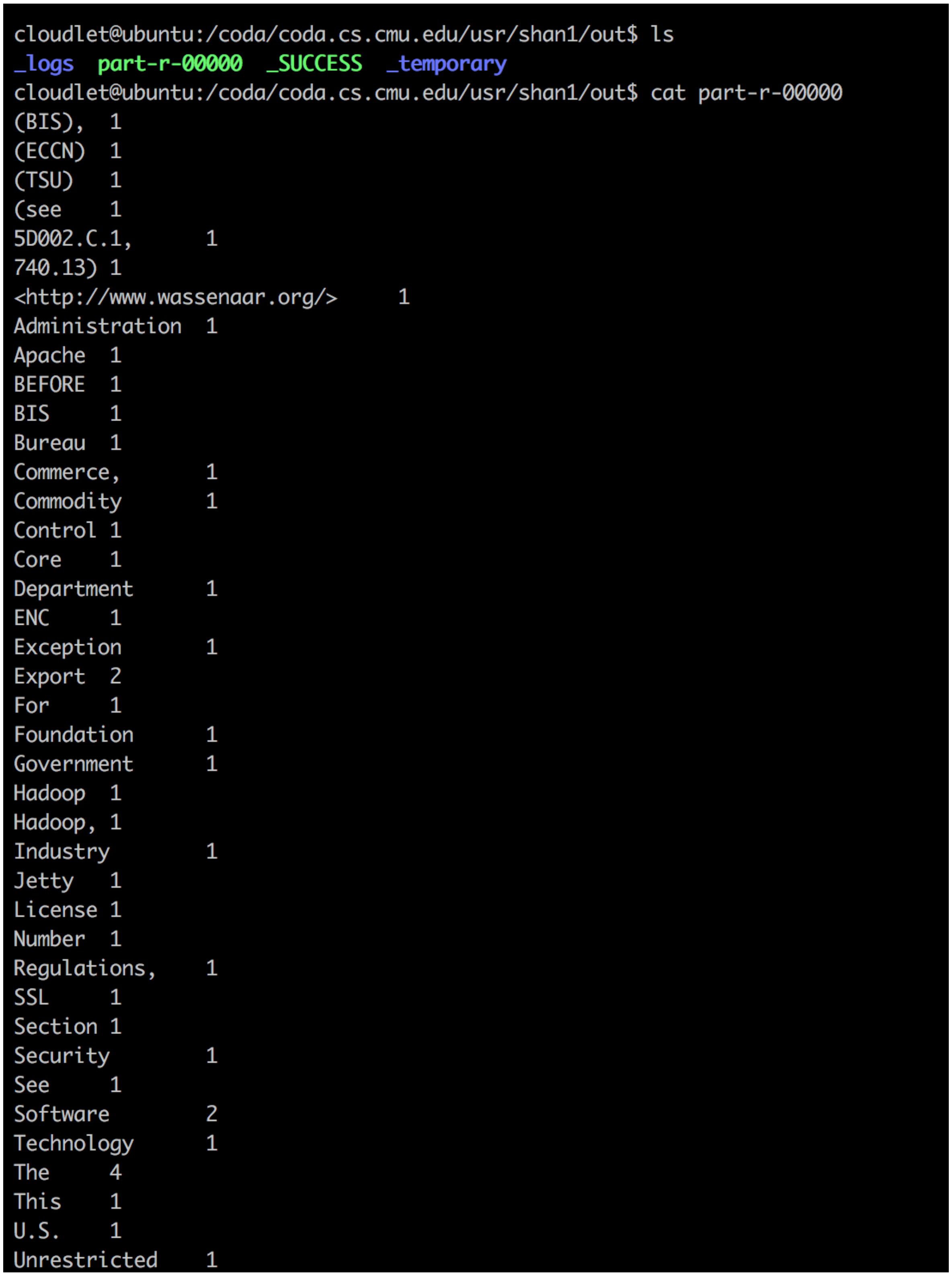
6.2.2 Results

To run Hadoop, we first start the jobtracker and the tasktracker processes on the master and the tasktracker process on the slave by running ./bin/start-mapred.sh. We run a word count mapreduce application on the cloudlet to demonstrate the functionality of the Hadoop setup. We enter the command

```
bin/hadoop jar hadoop-examples-1.1.2.jar wordcount
../coda/coda.cs.cmu.edu/usr/shan/input
../coda/coda.cs.cmu.edu/usr/shan1/hadoop/res
```

where both the input and the output files are located in our Samba mount in the cloudlet.

We observe the Hadoop logs to confirm that the wordcount job completes. Furthermore, we notice the result file in our DFS directory consists of word number pairs that inform how often a word has appeared in the input file as evident in figure 6.2.



```
cloudlet@ubuntu:/coda/coda.cs.cmu.edu/usr/shan1/out$ ls
_logs  part-r-00000  _SUCCESS  _temporary
cloudlet@ubuntu:/coda/coda.cs.cmu.edu/usr/shan1/out$ cat part-r-00000
(BIS),  1
(ECCN)  1
(TSU)   1
(see    1
5D002.C.1,      1
740.13) 1
<http://www.wassenaar.org/>     1
Administration  1
Apache  1
BEFORE  1
BIS     1
Bureau  1
Commerce,       1
Commodity       1
Control 1
Core    1
Department      1
ENC     1
Exception       1
Export  2
For     1
Foundation      1
Government      1
Hadoop  1
Hadoop, 1
Industry        1
Jetty   1
License 1
Number  1
Regulations,    1
SSL     1
Section 1
Security        1
See     1
Software        2
Technology      1
The     4
This    1
U.S.    1
Unrestricted    1
```

Figure 6.2: Result file of our Hadoop run

Chapter 7

Future Improvements

In this chapter, we discuss the future improvements towards the architecture that we could not implement due to time constraints. For each improvement, we state the benefits it brings and suggest a possible implementation roadmap.

7.1 Launching the FSMVM with a specific IP

As mentioned before, all the FSMVMs need to have a well-known private IP address that remains static across cloudlets. However, nova-networking does not provide the functionality to assign a specific private IP address to a VM. Currently, we launch the FSMVM first to ensure that it gets assigned the first private IP address available in the IP pool. For example, for the pool 10.1.171.0/24, the FSMVM will have the IP address 10.1.171.3 since it is the first IP from the pool.

Since the FSMVM is part of the cloudlet infrastructure, we can give it the first IP in the pool when a user requests a tenancy in the cloudlet. However, a problem occurs when the FSMVM fails and has to be restarted. Since the administrator cannot specifically assign the same IP the FSMVM held before the VM's failure (ie 10.1.171.3) when he restarts the FSMVM, the distributed file system within the cloudlet collapses as a result. Hence, the FSMVM is a single point of failure within the architecture, and even worse, one that cannot be recovered on failure.

We suggest two ways to overcoming this problem. The first way consists of having an agent within the guest to deal with the failure of an FSMVM. When the FSMVM restarts, either by the administrator or automatically by some daemon program that tracks failures and respawns FSMVMs, these agents are informed through the local network by a central figure. Then, these agents will stop the OpenVPN client in their client VMs, modify the OpenVPN tun0.conf configuration file to point to the FSMVM's new private IP address, and restart the OpenVPN client. The Samba client will reconnect to the new FSMVM once the OpenVPN connection resets, leading to a smooth transition.

However, the above strategy exposes the client VM to security vulnerabilities. Installing an agent inside a client VM itself leads to potential security flaws. Furthermore, the agent requires privileges since it needs to modify the tun0.conf file located in /etc/openvpn, worsening the situation.

The second way consists of changing the cloudlet's network setup to neutron instead of nova-networking. Neutron allows the administrator to assign a specific IP address to an instance in the means of Neutron ports. A port is a network abstraction in neutron which contains fields such as MAC address, fixed IPs and VM identifier. Through this abstraction, the administrator can assign the previous private IP to the rebooted FSMVM. The client VMs can reconnect to this FSMVM without any modifications to their configuration files, since the FSMVM has the same IP as before.

7.2 Automating the infrastructure setup

Currently, setting up our distributed file system in the cloudlet requires 3 steps.

1. Booting up the FSMVM with a well-known private IP address
2. Running the FSMVM setup shell script on the FSMVM
3. Running the client VM setup shell script on the client VM

Booting up the FSMVM is covered in the previous section. The FSMVM setup shell script does the following things:

1. Install the Coda client, Samba, and OpenVPN on the FSMVM
2. Authenticate the Coda client using clog
3. Configure Samba to export the Coda client folder
4. Configure OpenVPN and launch the OpenVPN server

The client VM setup shell script does the following things:

1. Install the Samba client and OpenVPN on the client VM
2. Copying the pre-generated OpenVPN keys to the /etc/openvpn directory
3. Starting OpenVPN and connecting to the FSMVM
4. Mounting the Coda mountpoint via Samba

Since the FSMVM is part of the infrastructure, the cloudlet administrator can initialize it when a new tenant is created. However, every time the user launches a new VM, the client VM setup shell script needs to run in order to connect the newly booted VM to the FSMVM. We can use Openstack's cloud-init feature to automate this process. Cloud-init allows the cloudlet administrator to register a script that will run upon a VM's startup. With cloud-init, the VM user does not have to run the script manually to connect to the FSMVM, and the VM will be ready to use the distributed file system upon startup.

7.3 Reducing the VM Overlay Size

One of our original goals was to reduce the VM overlay file size by using a distributed file system. It is true that since the entire library of the user's files is stored on the Coda server and only the working set of the VM is stored locally, the VM overlay file size decreases.

We conducted an experiment where we compared the VM overlay sizes of one VM in our architecture and one without a distributed file system. These two VMs used the same 10mb text file, so that the VM without the distributed file system opened the file using vim from the local file system, and the VM with the distributed file system opened the file using vim via Samba and the FSMVM. We noticed that the overlay file of the VM with the distributed file system was the same size as the overlay file of the Vm without the distributed file system. In the VM using the distributed file system, we conjecture that the file is stored twice in the memory once on the network buffer, and again on the allocated memory of vim. This mitigates the benefits of not having the file stored in local disk, and hence the same size of the two overlay files. Since Openstack++ does not use rolling window deduplication with the VM's memory, if two identical stream of bytes are not aligned, they will not be deduplicated.

In the future, we should look to investigate if this is indeed the cause of the bigger overlay file size, and to implement rolling-window deduplication to reduce the overlay size.

Chapter 8

Conclusion

We implemented a distributed file system in the cloudlet to allow easy sharing of files between VMs both within and across cloudlets. Integrating the distributed file system with the cloudlet ecosystem, we implemented prefetching to fetch frequently used files from the source FSMVM to the destination FSMVM. This allows migrating VMs to avoid cache misses upon arrival at the destination cloudlet. We also explored the semantics offered by our distributed file system, both within and across cloudlets. Within a single cloudlet, we ran Hadoop to demonstrate that our architecture supports pre-existing and widely-used software without involving core modifications to the program.

8.1 Contributions

The main contribution of our work is the implementation of our distributed file system architecture on the cloudlet that features prefetching and offers a familiar interface to users and developers. VMs in the cloudlet can now share files easily across and within cloudlets. Furthermore, because of the familiar interface we offer, pre-existing software can run on top of our distributed file system without modifications.

We also offer technical details and analysis of our architecture in this thesis. We outline the implementation and analyze key decisions we made to make our system more transparent and approachable to administrators. We also suggest improvements that can be made in the future and provide a roadmap for each one to guide development.

8.2 Final Thoughts

Our architecture eases sharing of files between VMs in the cloudlet significantly. This provides great advantages to the users of the cloudlet architecture, especially as computation becomes more and more distributed. The fact that our architecture offers sharing both within and across cloudlets greatly increases the utility of VMs in the cloudlet, as any VM can now access any file in the distributed file system (within the same tenancy) and share updates made to the file without any setup or intervention by the user. The current setup also has the potential to improve as distributed file systems advance, as our layered implementation allows swapping and replacing components without a complete overhaul of the architecture.

Bibliography

[1] Elijah-openstack. URL `https://github.com/cmusatyalab/elijah-openstack/`.

[2] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251254.1251264`.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, Oct. 2003. ISSN 0163-5980. doi: 10.1145/1165389.945450. URL `http://doi.acm.org/10.1145/1165389.945450`.

[4] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. *SIGOPS Oper. Syst. Rev.*, 29(5):143–155, Dec. 1995. ISSN 0163-5980. doi: 10.1145/224057.224068. URL `http://doi.acm.org/10.1145/224057.224068`.

[5] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–18, 20–21, May 1990. ISSN 0018-9162. doi: 10.1109/2.53351. URL `http://dx.doi.org/10.1109/2.53351`.

[6] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE TRANSACTIONS ON COMPUTERS,*, 39, 1990. URL `https://dl.acm.org/citation.cfm?id=79820`.

[7] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, David, and C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39:447–459, 1990.

[8] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8, 2009.