Project Acronym:	**TRESCCA**
Project Title:	Trustworthy Embedded systems for Secure Cloud Computing
Project number:	European Commission – 318036
Call identifier	FP7-ICT-2011-8
Start date of project:	01 Oct. 2012 **Duration:** 36 months

Document reference number:	**D3.3**
Document title:	Initial prototypes of Emulation and VM Serialization
Version:	1.0
Due date of document:	30st of September 2014
Actual submission date:	14th of October 2014
Lead beneficiary:	OFFIS
Participants:	Bernhard Katzmarski(OFFIS) Alvise Rigo (VOSYS)

Project co-founded by the European Commission within the 7th Framework Programme		
DISSEMINATION LEVEL		
PU	Public	**X**
PCA	Public with confidential annex	
CO	Confidential, only for members of the consortium (including Commission Services)	

EXECUTIVE SUMMARY

This deliverable captures and describes the intermediate state of the prototype implementation for Task 3.3. It is intermediate in a sense that, according the description of work, final results of this task will be shipped as D3.4.
In chapter 2, this deliverable presents results of an intensive related work study to identify similar approaches and technologies. Chapter 3 identifies the most significant technical requirements that need to be considered for a VM-based solution of Task 3.3. The current prototype and its implementation decisions are described in Chapter 4. Chapter 5 concludes this deliverable and gives an outlook on future work in this task.

The most significant result in the second period of Task 3.3 is a first prototype called tresccad. Its implementation allows VM migration between a Cloud (OpenStack) and local clients. A first version of this prototype was already published on GitHub as Open Source. Since the beginning of Task 3.3, it was clear that VM-based migration will introduce a large overhead compared to lightweight approaches based on Java, for example. We decided to stick with VMs because of its compatibility and to study its feasibility. Furthermore, we can expect bandwidths and device resources to increase and a VM-based approach is generally more compatible to existing IaaS solutions. Also, lightweight operating systems like OSv, which was discovered and evaluated during the second period, could reduce the overhead a lot. OSv for example is especially designed to be run at clouds and avoids some components that are not needed for virtual systems.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 Purpose of the Document

This document is deliverable D3.3 of the TRESCCA project. Task 3.3 mainly aims to develop a prototype that will be delivered as D3.3 and D3.4 respectively and serves as a basis for WP4 and the implementation and evaluation of the described use cases in D1.3. This document describes the results achieved in Task 3.3 and documents the current state of the developed prototype.

1.2 Document Versions Sheet

Version	Date	Description, modifications, authors
0.1	31.07.14	Created initial version, B.Katzmarski(OFFIS)
0.2	28.08.14	Added Evaluation and Conclusion, B.Katzmarski(OFFIS)
1.0	13.10.14	Prepared final version for Review, B.Katzmarski(OFFIS)

2 Related Work Study

The main goal of Task 3.3 is to connect the cloud-side to the TRESCCA platform by enabling the possibility to migrate virtual machines between cloud and client devices. Numerous projects already addressed the idea to move computing tasks between platforms. It can be described as mobile code which can be found in several real world examples. Most of the time, this means code is send to a foreign location to be executed, like for example database queries or custom postscripts for printers. Most prominent example for mobile code is JavaScript or Java Applets that are transferred to the client-side on demand and executed locally. Obviously, the creator of mobile code cannot trust the client's execution platform. Hence, they are mainly used to increase the user experience and validation or critical processing of data still has to take place at server-side.

Mobile agents represent a stronger form of mobile code, which offer the migration of code, data and execution state. Mobile agents were found to be very similar to the objectives of Task 3.3 because the migration of virtual machines also offers the possibility to transfer data and all execution state to the local device and back.

This section gives an overview about technologies and evaluated projects that were considered as an alternative to achieve the objectives of Task 3.3.

2.1 Mobile Agents

Mobile Agents describes a concept where software agents can move between different computing platforms. It was originally inspired by the idea to reduce network traffic in communication intensive applications. As only agents have to be transferred, network dependency is reduced, which leads to lower bandwidth demand and overcomes latency problems. Thereby, it is offering a reasonable alternative to the classical client-server model.

A Mobile Agent can *decide actively* when and where it wants to be transferred. Security concerns prevented Mobile Agent Systems to be adopted into real world applications. Many existing MAS use Java as a platform due to its platform independence. A bytecode virtual machine introduces an additional software layer for application execution. Application partitioning on that level usually has a much smaller footprint compared to VMs. It introduces more granularity for migration as for example whole applications, threads or even single methods are becoming candidates. It's even possible to take device dependent functionalities like cameras into account and only transfer independent application parts. The possibly greatest benefit of this method is its analyzability. With this approach, it becomes possible to attest runtime behavior and check for security compliance with static analysis.

Java offers the possibility to have multiple threads within an application. These threads are reasonable candidates for migration between platforms. However, the runtime environment does not offer any method to suspend or resume Java threads. Existing methods are deprecated due to possible deadlock problems. This functionality has to be modelled by the programmer himself, caring manually for inconsistent states and race conditions. Same goes for serialization: indeed, Java offers object serialization but this mechanism is incompatible with threads. Threads depend on local state which is not in the scope of the serialization process. In another computing environment a deserialized object will never be the same object: It will always be a new object with same properties. Furthermore, within pure Java, it is not possible to access the instruction pointer to store at which point execution should be continued. Instead, it is only feasible to model a state machine: Transitions can mark fix points where migration can be done. The current state stored in a custom attribute is serializable and execution can continue in the next state.

These shortcomings limit the applicability of a pure Java approach. Certainly, it would be possible to extend the existing JVM to support serialization of threads. Previous work already addressed this [Suezawa2000, Quitadamo2008, Bouchenak2003], but unfortunately, existing extensions are rather old and most likely not compatible with current versions of the JVM.

However, all thread serialization approaches are facing the same problems: related local state, like e.g. file descriptor, open database or network connections are living outside of the JVM and are usually hard to serialize.

2.1.1 Threats and Countermeasures

Security concerns prevented Mobile Agent Systems to be adopted into real world applications. Vigna et al.[Vigna2004] stated 10 reasons against Mobile Agent Systems and 4 of them are related to security issues. Without a reasonable level of trust between platforms and agents, they will not be considered to operate on sensitive data. To protect agents against malicious platforms is still a challenging topic. The underlying problem is that agents are always executed within the environment of the foreign platform. There is no way for the agent to determine if the platform is acting correctly. From a software vendor view, there is no point in trusting processing results of remote client devices. For instance, most web applications often fail because they blindly trust their clients input or some client-side processing results, without further validity checks.

2.2 Linux Containers

Linux Containers, also called lxc, are available since Linux Kernel 2.6.24. They offer a lightweight alternative to virtual machines with less overhead. They provide a virtual file system and own process and network space which are isolated directly by the kernel. Containers are a promising technology especially for Platform as a Service Clouds, as they reduce the overhead for installation and setting up environments. Applications are simply bundled together with all dependencies and configuration into a transferable package. For the migration framework in WP 3, containers would have been a realistic technology. Unfortunately, containers do not have *suspend and resume* functionality built-in inherently which would be the basis for migrating containers between platforms. It would be possible to achieve this by using a third party component like Checkpoint/Restore in Userspace (CRIU[1]), which allows generic serialization of processes, but this hasn't been further evaluated. Additionally, the drawback introduced by containers is the fact that they need to stick with the same version of kernel as the host platform and would not be compatible for virtualization and hybrid migration developed by VOSYS.

2.3 JADE

JADE [2](Java Agent DEvelopment Framework) is a mobile agent framework completely developed in Java. The project already started in 1998 and is still under active development (current version of JADE relased 28/03/2014). Besides other MAS frameworks like IBM Aglets[3] or D'Agents, JADE is the most promising one, not only because it's still under active development.

JADE uses a weak migration approach and is overcoming the local state problems with the serializable interface in Java. It is only possible to transfer serializable classes. Open files or sockets are not

[1] http://criu.org
[2] http://jade.tilab.com/
[3] http://aglets.sourceforge.net/

serializable and will throw NotSerializableException. Furthermore, due to the nature of the JVM as explained earlier, no execution state is serialized: after migration, a callback function is used as an entry point to continue execution on the foreign platform.

In this sense, Jade was very interesting because it offered a straight forward way to implement and execute agents. For developers familiar with Java, there is no major issue to set up the framework. Furthermore, the approach is somewhat lightweight because only JAR files of the application need to be transferred.

Using JADE for agent migration is in no way secure. Agents are packed into jar files which are encoded via base64 and transmitted to an http server in clear text. In the JADE security guide[4] only authentication and message integrity/confidentiality between agents are covered. This is by no means a protection for migrating agents or maliciously modified platforms.

2.4 JavaFlow

JavaFlow is a library that offers thread suspend/resume functionality within pure Java. It is based on previous research project called Brakes [Truyen2000]. In JavaFlow, this feature is called Continuation and it uses bytecode instrumentation, which means compiled class files need to be enhanced before they can offer continuation. Code that should be resumable must implement the Runnable interface which is usually meant for threads. Continuations can be started (Continuation.startWith) or continued (Continuation.continueWith).

A benefit of this approach is that it works out of the box with any JVM. No extensions need to be installed. It could create the impression of strong migration, as the execution can just continue. On the other hand, it introduces some overhead for enhancing necessary class files.
Additionally, *Continuation.suspend* calls have to be manually inserted.

Unfortunately, there is no way to suspend a thread from the outside if control flow is not exited via an active suspension call. However, a control mechanism which reduces the need of actively spreading out suspend statements can be easily build on top of JavaFlow.

[4] http://jade.tilab.com/doc/tutorials/JADE_Security.pdf

3 Technical Requirements

During the beginning of work package 3, some technical requirements were identified which heavily influence the development of a prototype. Compared to state of the art live-migration used in data centers, a migration framework has some special requirements that were identified and are presented in the following.

3.1 Active Migration

Active migration allows VMs to change their physical location dynamically at runtime. This is a very unique feature allowing application parts to decide actively to migrate to other locations, depending on the use case. In this aspect, the concept is very similar to that of Mobile Agent Systems: It's imaginable to have separated information and functionality and an actor part as a visitor moving around to fulfil the process. Existing approaches use migration to automatically balance applications according to performance or energy consumptions. They migrate transparently without applications' notice. The unique feature of our method is that an application can actively request the migration to a remote location. However, we need to study the impact on Infrastructure as a Service providers, as they are currently not prepared for VMs dynamically joining and leaving their environment. The generality introduced by VMs could make active migration also applicable in context of Big Data where it is obvious that moving the application towards data is much cheaper.

3.2 Shared Storage

In a cloud environment, shared storage like SAN is used to avoid the need to transfer disk images between hosts. This reduces the migration time for VMs significantly. However, when it comes to migration between cloud and local clients, we cannot rely on the availability of shared storage. Therefore, we need to consider transferring the disk image of the VM, as well. Obviously, this will introduce an overhead, as VMs usually contain a full operating system which can require several gigabytes of hard disk space. Although bandwidths are still expected to increase, it's not arguable to waste this amount for non-significant information. We address this issue partially by the use of shared base images but see potential improvements in using small operating systems with smaller footprint or bare metal applications.

3.3 Location-Awareness

A hypervisor is responsible for execution and migration of virtual machines. It provides a full virtualized hardware interface, where VMs shouldn't even notice they are not running on physical hardware; they are location-agnostic. In our model, we have to soften the location-agnostic property of virtual machines and provide them with certain location information for decision-making. This is important, because our concept focuses security in a sense that a program only gets access to sensitive data or services if it resides in the respective environment.

3.4 Off-line Migration

WAN environments will most likely not be practicable due to latency problems. Additionally, the classical Mobile Agent Model doesn't even require the agents to be responsive while they are being transferred. In that sense, it will be enough to use a stop-and-copy migration strategy.

3.5 WAN Migration

Another challenge is introduced by low bandwidths, latencies and availability in WANs. For live migration this has already been addressed [Riteau2011, Hirofuchi2009]. Still due to the dynamics of WAN connections, this is usually a very hard task. Therefore, our approach will not require the use of live migration. Instead, for our purpose, it will be sufficient to use an off-line migration approach that uses a stop-and-copy strategy.

3.6 x86 vs. ARM

A hypervisor provides fully virtualized hardware interfaces and is responsible for execution of VMs. While there are different hypervisors on the market, we consider QEMU/KVM as a basis. VM execution can benefit from hardware assisted virtualization that dramatically improves the VM performance. Both x86 and ARM offer hardware assisted virtualization through proprietary virtualization extensions integrated in the processors. These capabilities are exploited by the Linux kernel through the KVM special device that can assist the QEMU system emulator which allows running guest code directly in the host CPU.

Most mobile devices are using ARM while at server-side the most used architecture is x86. This introduces challenges when it comes to migration, as a full virtualized machine on an ARM platform is not transparently executable on x86. The lowest common approach regarding interoperability would be to avoid the use of hardware acceleration (hardware assisted virtualization) in favor of emulation, which leads to significant higher execution times.

In addition, the migration requires all the CPU features and devices exploited by the virtual machine to be present in both the source and the destination. This means that the definition of a virtual machine model that can be migrated between different architectures is needed to successfully deploy heterogeneous clouds. This is an interesting challenge particularly in heterogeneous clouds.

4 Current Prototype

Since the beginning of Task 3.3 it was clear that VM-based migration will introduce a large overhead compared to lightweight approaches based on Java, for example. We decided to stick with VMs because of its compatibility and to study its feasibility. Furthermore, we can expect bandwidths and device resources to increase and a VM-based approach is generally more compatible to existing IaaS solutions.

The current prototype consists of multiple components. For the cloud-side we chose OpenStack as an IaaS platform and extended its functionality by a plugin. The overall goal is to find a practicable way to import and export virtual machines in OpenStack. The current prototype already offers some of the basic required features like package creation and transferring VMs between platforms. To understand the architecture and the interaction, we need to first explain the basics of OpenStack.

4.1 OpenStack

Multiple open-source cloud platforms have evolved over recent years including OpenStack, Eucalyptus, CloudStack, OpenNebula. We choose OpenStack as a Cloud Platform due to its modular and extendible architecture and because it is already widely adopted and has a strong community. It should be mentioned that OpenStack is still in its infancy compared to commercial cloud products. However, for small or medium private clouds, open-source solutions are already viable choices.

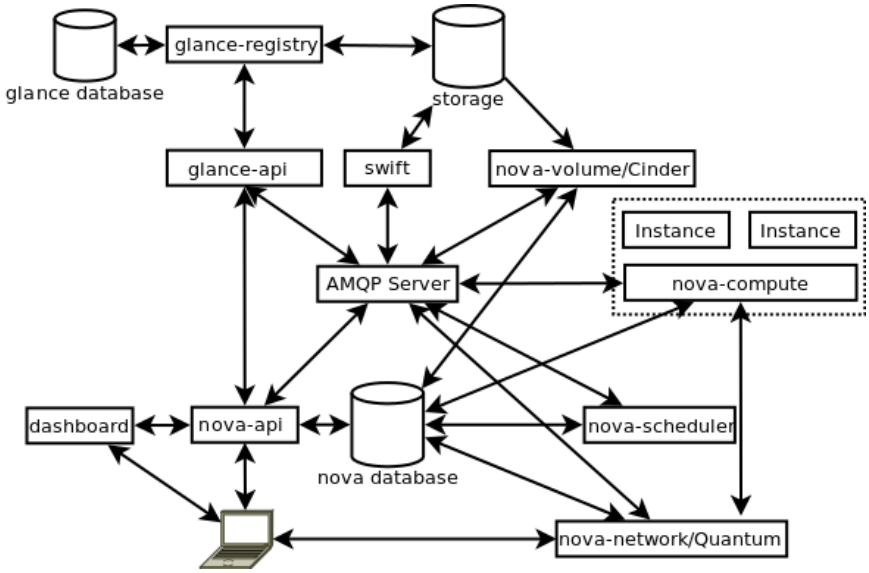

Figure 1 OpenStack Architecture Overview

Historically, OpenStack is a joint venture project of NASA and Rackspace, merging two important components which are inherent parts of today's OpenStack releases. First, block storage service *Swift*, originally developed by NASA to store and operate on their huge amount of data. Second, compute service *Nova* developed by Rackspace for managing virtual machines. Today, an OpenStack release consists of many small components and services that can be deployed freely among physical infrastructures. OpenStack makes strong use of existing open-source technology, system libraries and services like iptables, libvirt or rabbitmq-server(AMQP). Additionally, it is possible to use different hypervisors such as KVM, XEN, HyperV and others. Libvirt is hereby a commonly used abstraction interface to manage virtual machines. All components are communicating via a message queue protocol AMQP. An overview of existing core services and their interactions is given in Figure 1 and

will be shown using the creation of a new instance as an example. A user can request a new virtual instance via API or dashboard. Nova-scheduler receives the request and selects an appropriate host by looking into nova-database. Afterwards, nova-scheduler informs nova-compute on the selected host about the new instance creation request. Nova-compute creates the new instance and informs nova-network to setup networking functionality and accessing rules for the instance.

4.2 Architectural Overview

TRESCCA is relying on QEMU/KVM as a hypervisor. Therefore, it is reasonable to use libvirt as a common abstraction layer to manage VMs. Libvirt is running as a daemon on the respective compute host and bindings for different languages are available as well as a command line interface called virsh. Hence, the idea is to create a small TRESCCA daemon (tresccad) that is able to take control over libvirt and can be run all compute hosts. It makes no difference if this host is at cloud or at client-side as the setup requires the presence of QEMU/KVM and libvirt on all involved parts, anyway. However, the architecture of OpenStack prohibits the use of a second service managing libvirt.

If we compare the required features of tresccad with nova-compute it turns out that nova-compute is basically the cloud-side counter-part to tresccad, as it also takes care of virtual machines by using libvirt. Hence, we develop a twofold approach where tresccad is meant to be deployed to the local client side and required features at cloud side are deployed via plugins. This is the most non-intrusive approach and basically only affects OpenStack services nova-api and nova-compute which functionality is extended. At client-side this approach is favourable to reduce dependencies, as nova-compute requires multiple other services to be executed like nova-network, nova-scheduler or Glance. Hence, only the required features of TRESCCA are implemented in tresccad and their necessary counterparts at cloud-side are provided via plugins to the services nova-api and nova-compute.

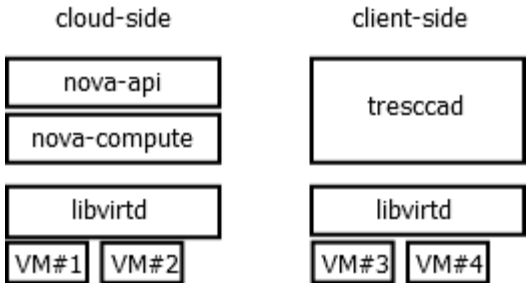

Figure 2 Tresccad Overview

4.2.1 Cloud-Side

The cloud-side already offers everything needed for a virtualization platform. Hypervisor, libvirt and control services are installed. The current prototype consists of a rudimentary python script that serves as a plugin for nova-api and offers a restful api-extension to save and restore instances. This is currently done by issuing shell-commands to virsh. The obvious drawback of this approach is that vm-packaging and restoring is currently running in the context of nova-api and local file access to the instance files is necessary. This approach works for a single-node setup, where all OpenStack services are deployed on one host. But as soon as the installation consists of more than one host, the current implementation will not work anymore. In the long run, required packaging tasks need to be delegated by nova-api to the respective nova-compute services which will issue the packaging for residing VMs. However, this trade-off was made to focus development on the client-side and to fasten integration.

4.2.2 Client-Side - tresccad

In order to be able to manage the migration of virtual machines, we create a small daemon called tresccad. On each local host system an instance of tresccad is required.

The client-side development is currently done on Ubuntu 14.04, but it can be expected that other linux-based distributions will be compatible as well. Tresccad at client-side requires certain packages for virtualization like libvirt, qemu-kvm.

Tresccad provides a small web server that offers the same API as OpenStack. This applies for the extension to save and restore VMs, as well as the necessary API of Glance to exchange disk images. In this way, a local device gains the ability to react to the same requests for up- and downloading images and control of VMs. The service is also responsible for managing the internal network to provide connectivity to residing VMs. Moreover, its API is reachable for VMs under the link local address[5] 169.254.169.254. The concept of using link local address is also used by Amazon's EC2 API or Openstack's compatible API. Running instances can reach their metadata-service under this address to get additional configuration information. This is usually done by spawning instances that need to configure their name and login credentials according to input of the user. Internally, these requests are routed via iptables to a nova-api-metadata web service. Using http connections for this case is a simple and lightweight approach which can be used by rich applications as well as tiny command line scripts. This approach can be used to offer the migration API to residing VMs to gather environment information and request migrations. In the long run, this can be used for active migration where VMs can request migration on their own. In order to enable communication between VMs and the host system, tresccad can be deployed inside the VM as a library to ease the development of applications.

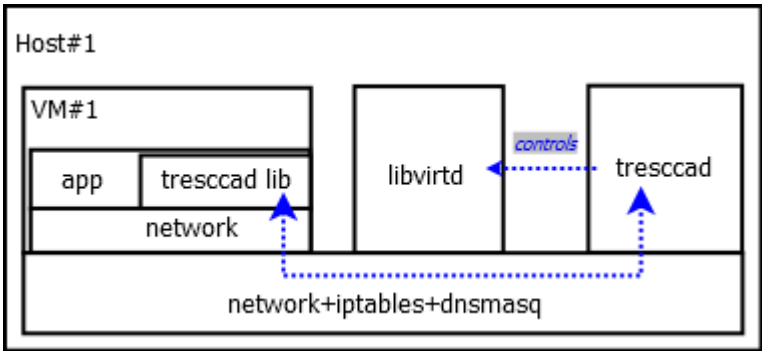

Figure 3 Tresccad VM Communication

4.2.3 Off-line Migration and VM Packaging

Libvirt already offers the feature to save and restore virtual machines by dumping CPU and memory state into a file. This functionality is used as starting point for off-line migration. Unfortunately, this feature is only meant to be used on the same host as it also dumps configuration data that is specific to the compute node. We are relying on a setup where VM configuration and virtual hard disks are stored to local compute nodes file system. All related files are given in Table 2 and they are all created and managed by OpenStack except the snapshot file, which is used by TRESCCA for off-line migration.

[5] RFC 5735

Table 1 VM Package Content

File name	**Description**
disk	Virtual hard disk file
kernel	Kernel used
ramdisk	Ramdisk
snapshot	Snapshot containing CPU and memory state created by libvirt
libvirt.xml	XML configuration file for libvirt

As already mentioned, we assume the presence of VM-related files on the local hard disk of the respective compute node. This makes it easy to collect them and bundle them into a package (currently an uncompressed zip-file) which can be transferred as a simple binary file over the available network channel. Again, the cloud architecture introduces some constraints regarding the handling of VM packages. In most setups of OpenStack, the related compute nodes are most likely not directly accessible by the clients. Hence, a created VM package on a certain compute node could not be easily fetched by the respective client. As Glance is the service of OpenStack dedicated to manage images for virtual machines, we decided to use this service as an exchange platform. The package containing all related files of a VM will be directly pushed to Glance and the remaining original files are destroyed to purge it completely from OpenStack. The respective VM will only remain as a serialized package in Glance. In this way, we preserve compatibility of related Glance API and can use it for transferring the packages. The process of saving a VM to Glance is illustrated in Figure 4. First, a user sends a request to save a VM which ends up on a compute node that uses libvirtd to snapshot the respective VM. Thereafter, all related instance files are packed together, the package is uploaded to Glance and the remaining VM is removed by deleting all related configuration and files.

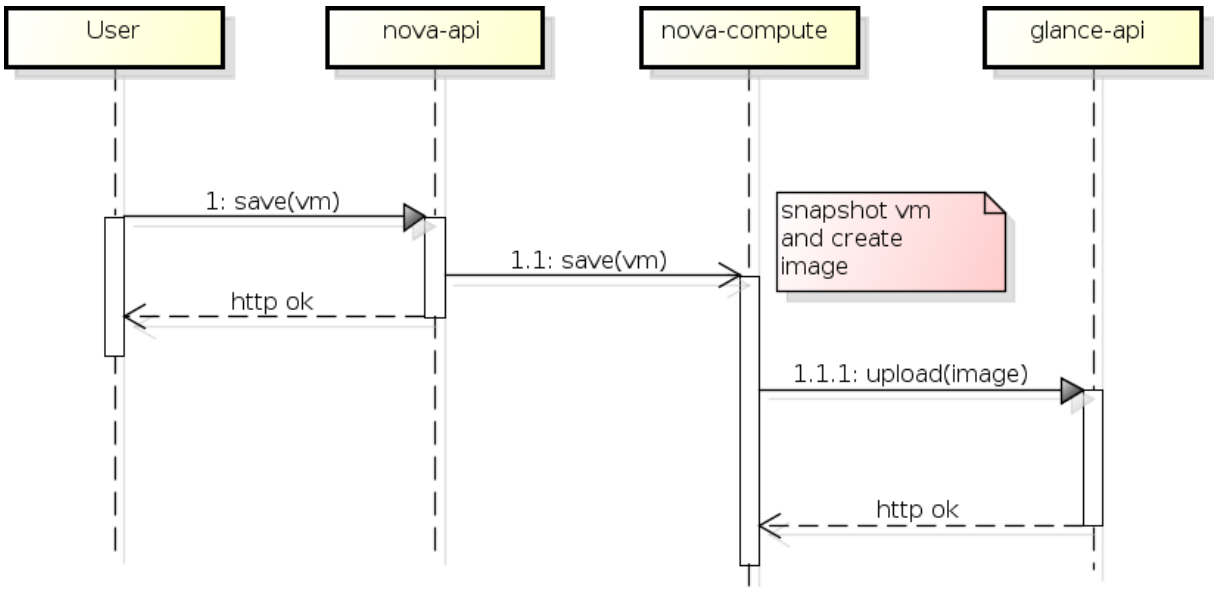

Figure 4 Process of Saving a VM

Unfortunately, when saving an instance, libvirt saves a copy of a dumped libvirt xml configuration file directly in the snapshot. This introduces problems because absolute paths, names and other configuration like addresses need to be adjusted when restoring the snapshot on a foreign platform. This lies basically in the responsibility of tresccad and (nova-api, nova-compute).

OpenStack can make use of QCOW base images out of the box (refer to section 4.5). Although this reduces the amount of data that needs to be transferred the main problem is, that the derived images use absolute paths to find their base image. Obviously, these paths will not be present on the target platform; therefore they need to be adjusted to relative ones before packaging and transferring the VM. On the other side, to restore a VM from a package, it's necessary to link the disk image to the corresponding base image. This is currently done by making a symlink in the respective instance folder. However, the prototype currently only supports one dedicated base image. As development continues it is planned to support multiple base image and ease the management of them.

As tresccad is relying on libvirt for saving and restoring virtual machines this also means known limitations are applying. In theory, restoring an image from given snapshot only works once and in case of any error, it cannot be repeated. The respective instance could make changes to its disk immediately after restore. If the instance didn't resume properly and restore would need to be executed a second time it could operate on inconsistent data. This is a known limitation by libvirts save and resume functionality and one should have backup of the respective disk file. This limitation applies for tresccad as well but due to the chosen concept of having VM packages containing snapshot and disk overlay, this risk can be reduced to a minimum because the needed backup is implicitly available in the VM package.

We need to remark the importance of the configuration option called *force_raw_image* within OpenStack. When spawning a new instance in OpenStack, this option indicates whether the base image should be converted to the raw format. This can be useful in order to achieve higher IO throughput on virtual machines but also means the amount of needed disk space on the host system will increase dramatically. If the configuration option *force_raw_image* would be set to true and a respective instance should be migrated to a local device, this would mean the base image of several giga bytes would need to be transferred as well. By setting the option to false this can be avoided and the benefit of smaller disk sizes can be used.

4.3 Communication

Communication between multiple trescca daemons will exclusively take place over the network stack. Although this introduces an additional overhead, the advantages outweigh this. Using network communication makes this approach platform and operation system independent and more flexible when it comes to integration into cloud platforms. Most cloud providers already make use of network based communication for most of their API's and work internally with http connections as well. As for example the Amazon EC2 API is offered as a web service specified in WSDL. This service oriented approach therefore is a good way to follow. It offers compatibility to many other libraries and programs like web browsers or even command line tools.

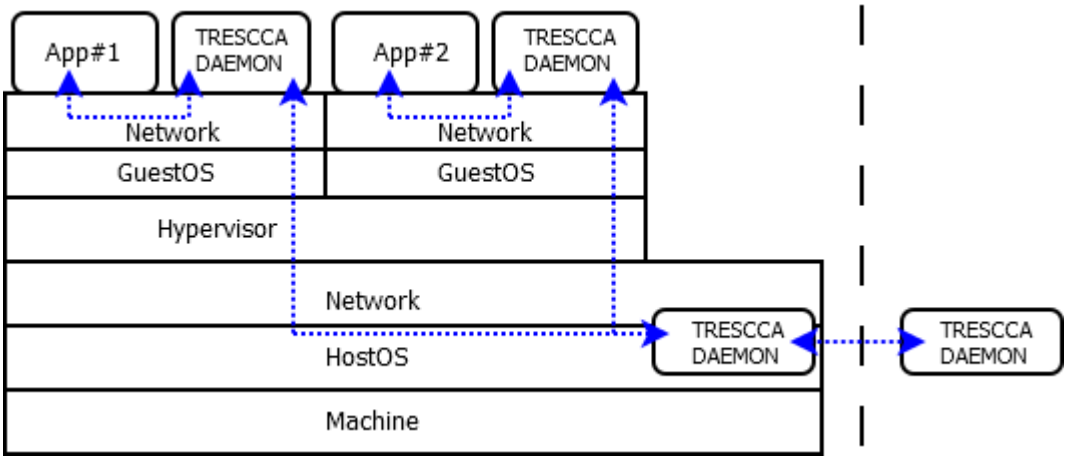

Figure 5 Communication Overview

VMs will be connected to a virtual network devices managed by libvirt. A DHCP server is listening of the respective bridge that serves as a gateway for VMs is responsible to assign dynamic addresses to the instances. For the prototype the open source DHCP server dnsmasq is used, which will also be needed on the local device and is controlled by tresccad. There are certain pitfalls, when tresccad at client-side is responsible for managing libvirt, dnsmasq and iptables to provide connectivity to the guests. It always needs to be aware of all VMs running on the host and needs to take care of a conflict-free assignment of ip addresses to mac addresses, as this mapping needs to be provided to dnsmasq and also to the nwfilters of libvirt. A single inconsistency in this set up can cause the VM not having any network connectivity.

4.3.1 tresccac

In order to ease development and testing, a CLI-Interface has been implemented. It can be used to issue commands like save/restore or up and download VM packages and images. In this way, commands can be triggered easily and no API-Requests have to be built manually. With tresccac, tresccad and the OpenStack counterpart plugin can be controlled remotely. By specifying the target for a command it is possible to manage multiple clouds or devices. The terminology has been adapted by OpenStack. An instance is a VM and an image is either a disk image or a VM package created by tresccad. Currently, tresccac features following commands:

tresccac delete ID

This command deletes an image or instance by a given id.

tresccac list

This command returns a list of available images and instances on the target.

tresccac make_config

This command creates a configuration file containing the different endpoints for hosts and cloud environments that should be used. Tresccac assumes the presence of a configuration file called config.yml in the working directory and one can specifiy on which target a given command should be issued by using the argument *–cloud=X*.

tresccac migrate INSTANCE_ID --dest=DEST --source=SOURCE

This command issues a migration of INSTANCE_ID from SOURCE to DEST. It is basically a convenience function which uses the other basic methods like save, restore and pull to stop execution on the source system, transfer the package and restore execution on the target system, as can be seen in Figure 6.

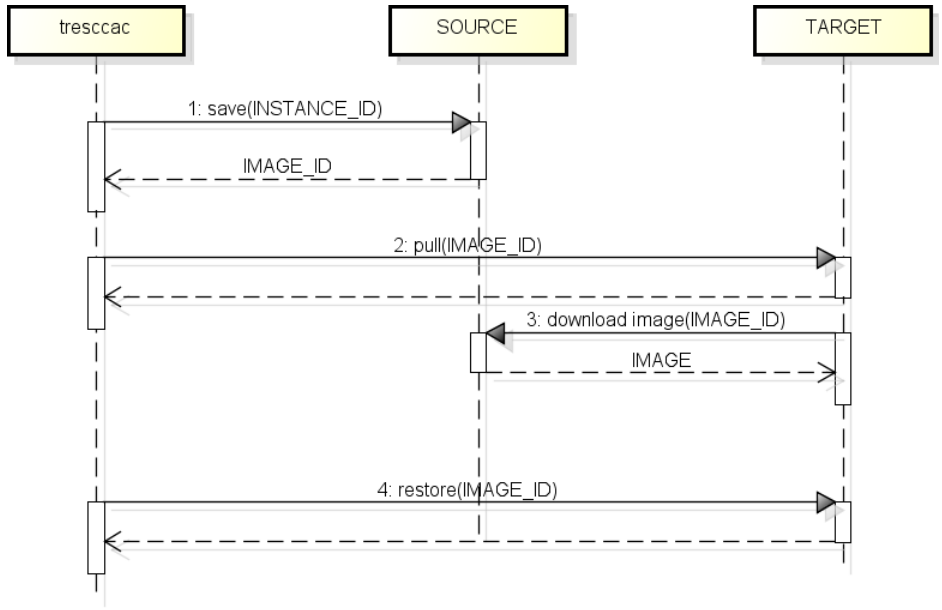

Figure 6 Tresccac Migrate Sequence Diagram

tresccac pull IMAGE_ID --dest=DEST --source=SOURCE

This command instructs tresccad at DESTINATION to download the image with the id IMAGE_ID from SOURCE.

tresccac push FILE

This command is useful to upload an image on the local file system to tresccad.

tresccac restore IMAGE_ID

This command restores a VM package by a given IMAGE_ID.

tresccac save INSTANCE_ID

This command saves a running VM with id INSTANCE_ID and returns the image id that can be used for transferring or restoring the VM from the package.

tresccac spawn IMAGE_PATH

This is another convenience method that is useful to create a new VM on tresccad. One can specify an IMAGE_PATH that refers to a disk image on the local file system and tresccad will take care of creating the instance.

4.3.2 API Extensions

The plugin for OpenStack adds two additional RESTful routes to the compute API for saving and restoring VMs in OpenStack.

/:project_id/servers/:server_id/save

Given a respective project(project id) and VM(server id), the introduced command is able to stop and serialize a VM. Internally, this request is forwarded to nova-compute which is responsible for stopping and snapshotting the machine. It uses libvirtd's save functionality to stop execution and to store CPU and RAM state to a snapshot file. Afterwards, it creates a package of all related files in the instance directory and pushes it to Glance. The VM and all related files are destroyed to purge it completely from OpenStack. It will only remain as a serialized package in Glance. We consider this two-step approach reasonable because a compute node might not be reachable by the client for streaming the package directly. Unfortunately, this creates certain overhead, but in this way, we preserve compatibility of related Glance API and can use it for transferring the packages.

/:project_id/servers/restore, {"image_id":":image_id"}

The above command is used for restoring a VM from a given package(image_id). As a precondition for the process, a serialized package needs to be available in Glance. This can either be created by saving the VM or by uploading a previously saved VM from somewhere else. The responsible compute node downloads the package and creates a new instance from its content. After the instance is properly registered in OpenStack, the execution can be continued by resuming the snapshot with libvirtd's resume functionality. Unfortunately, resume functionality will not work out of the box because some configuration values need to be adjusted. This includes uuid, name and paths to related files that are stored in libvirt.xml. Moreover, OpenStack is used to assign IP and MAC addresses to its newly created instances on its own. It would not be possible to force OpenStack to use the same IP for a restored instance that it had before, as it would result in potential conflicts if this address is already in use or the corresponding virtual network does not even exist.

4.4 Hybrid Migration

Since not every host platform can benefit from hardware assisted virtualization (KVM in the context of TRESCCA), QEMU is also capable of running guests in pure emulation mode, meaning that the guest code is firstly translated to an intermediate language to be then translated to the host architecture assembly code, even if guest and host share the same architecture. This is the concept of the QEMU tiny code generator (TCG). Even if slower than KVM, TCG is still currently used and maintained because it allows to run virtual machines compiled for an architecture different from the host one (for example it allows to run an ARM Android image on x86).

Generally, QEMU supports both live and offline migration. When used with the KVM hypervisor, its migration depends on the type of processor used to run the guests. When migrating machines from KVM enabled guest to QEMU instances that can't support KVM (like when an ARM virtual machine is migrated from an ARM host to a x86 host), some incompatibility issues have to be solved since QEMU doesn't describe the two processors inner state in the same manner, and the migration of virtual machines is nothing but a migration of saved states of all the guest components (devices, CPU, memory, ...). In addition to that, both the outgoing and the destination QEMU instances need to be of the same version, otherwise differences in the platform model emulation could lead to a migration failure. This achieves better performance, but since the guest instructions are directly executed on the host's processor, it requires the host and guest CPU architectures to be homogeneous. On the other hand, the latter completely emulates in software the target CPU, thus translating the guest into host instructions. This introduces certain overhead, but enables VMs to be executed on different architectures. Except for performance, these two solutions should be totally equivalent. However, they are currently some differences that prevent migration from KVM and TCG using the mainline QEMU.

To combine the performance and the flexibility of both solutions, Virtual Open Systems provides a patch for KVM to TCG hybrid migration for ARM VMs. It enables the guests to achieve best performance while running on ARM platforms, and additionally gives the possibility to migrate ARM virtual machines to cloud systems using different architectures. All architectures able to run the QEMU emulator (i.e. x86, s390, MIPS, etc.) are thus possible targets for migration. The main challenge faced in this work is the different handling of coprocessor registers by KVM and TCG. In fact, a one to one correspondence between these registers must be found, in order to allow the migration code to correctly restore the state of the CPU in the destination.

The hybrid migration supports both live and off-line KVM TCG migration. In case of live-migration, QEMU relies on a separated thread to perform all the operations needed for carrying out the migration of the VM. This thread and the execution of the virtual machine run side by side, allowing the downtime period of the VM to be reduced to a minimum. As for the off-line migration, all the operations for saving and transferring the state are made when the virtual machine is no longer running. This is a much more simple scenario which is however not indicated when the downtime factor is crucial.

The integration of VOSYS's Hybrid Migration solution has not been integrated into the prototype yet. The current prototype only runs on x86, but integration of hybrid migration will be focused in the remaining time of Task 3.3.

4.5 Reducing the Overhead

Virtual machines usually need disk images and memory of several giga bytes to operate. The additional overhead of full operating systems limits the possible granularity of application partitioning. However, this approach is more general and has no programming language restrictions, a benefit for legacy applications. Although we are aware of the introduced overhead, we are taking this approach to study its feasibility. Moreover, a VM-based approach still offers the highest possible isolation. Clearly, with this approach we are addressing Infrastructure as a Service (IaaS) providers and trying to develop a compatible approach to existing open-source IaaS solutions like for example OpenStack. We can also expect that bandwidth is still increasing which makes our approach increasingly feasible in the future for home environments. Nevertheless, we try to keep VMs tiny and therefore the introduced overhead as low as possible.

A lot of the overhead can already be reduced by existing standard technology. Modern virtual disk formats like qcow2 support overlays in terms of a base image. The base image always remains static and the used disk image only contains change sets. Multiple VMs can even share the same base image. This approach reduces the needed disk space enormously. To transfer only overlays is much more efficient than it would be the case with normal migration. This approach can be called dynamic VM synthesis like described by Satyanarayanan et al. [Satyanarayanan2009]. The downside of this approach is that a possibly huge base image has to be present on the target device. The best way would be to distribute this base image beforehand as the benefit is only available for migrations that can refer to this base image.

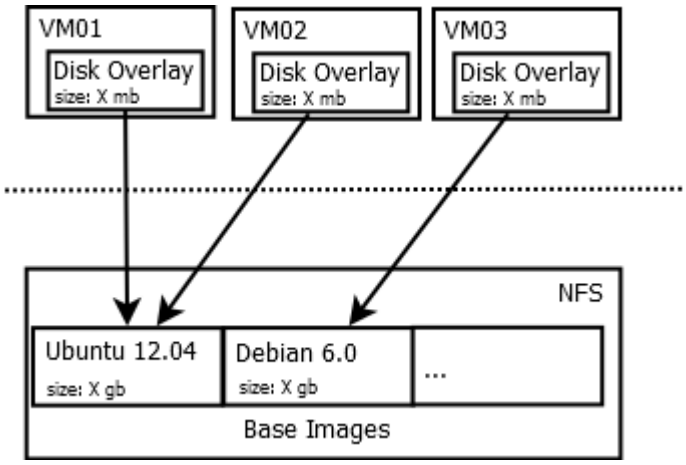

Figure 7 Disk Image Overlay Concept

Not to take a full linux distribution as a guest system, but instead use a more lightweight system, like for example:

- SliTaz
- Damn Small Linux
- tinycore linux
- ttylinux
- PuppyLinux

4.5.1 OSv

The most promising system for a guest system is the recently introduced OSv[6]. This is an operating system especially designed to run on a hypervisor. Many OS features aren't necessary for a cloud operating systems, like for example multi-user support. Furthermore, some features are even redundant, as the hypervisor already serves it. For instance, process-isolation and scheduling are already done by the hypervisor and as the guest system itself also takes care of processes and isolation, this introduces redundancy and certain overhead. Therefore it makes sense to develop a special OS for clouds where all unnecessary features are removed.

In the context of TRESCCA, OSv could be the basis for the migratable VMs that need to be lightweight and only serve one application. The project also has ARM-support on its agenda, but unfortunately this hasn't reached a usable state.

As the evaluation section 4.6 shows, package sizes achievable by using OSv are already quite reasonable and create confidence that the overhead can be handled. It demonstrates that using VM migration actively to fulfil a use case could be of practical.

4.6 Evaluation and Laboratory Testing

For development and integration testing, in-house infrastructure at OFFIS is used. It consists of several dedicated servers running OpenStack. During development, this is continuously used to test progress against real infrastructure. Unfortunately, all servers are only connected to a 100Mbit Ethernet switch, which is obviously a limitation but gathered performance results are already quite reasonable. Furthermore, integration of WT's cloud infrastructure has advanced by using their cloud as an evaluation platform as well.

[6] http://osv.io

Some performance measurements were already conducted to evaluate the current implementation. Table 3 shows the different devices that were used for evaluation. The domain type indicates whether hardware acceleration (KVM) was available and the column implementation shows if the client daemon(*tresccad*) or the OpenStack *plugin* counterpart was used. One virtual machine provided by WT was used for evaluation in order to push forward the integration of WT's cloud infrastructure.

Table 2 Devices used for Evaluation

Device	Location	Implementation	Domain Type	CPU
eee	OFFIS	tresccad	QEMU	900GHz
OpenStack	OFFIS	plugin	KVM	2270GHz
VM	OFFIS	tresccad	KVM	2400GHz
local	OFFIS	tresccad	KVM	2500GHz
VM	WT	tresccad	QEMU	2530GHz

We also tried to activate a Samsung Chromebook and a Raspberry PI for evaluation. Unfortunately, OSv wasn't able to boot properly on the Chromebook due a *tcg fatal error*. A similar problem was encountered on the Raspberry PI, where the guest stopped booting because of a failed assertion.

Evaluation is meant to get an impression of the runtime overhead introduced by serializing VMs. The time it needs to transfer the VM package from one host to another will only depend on its size and the available bandwidth. However, more laboratory testing needs to be done; especially tests in WAN environments have to take bandwidth limitation and latency-issues into account. But for this evaluation, we only focus on the time it needs to save and resume a VM. Save includes the whole process until the serialized package is available. All measures were taken by the linux utility *time* which can measure the overall execution time of a command until it finished. We consider this approach reasonable, because it indicates how long a user would effectively have to wait.

Table 4 shows all guest systems that were used. The package size refers to the used disk space for the respective serialized VM package, which was zip -compressed with-level 9. The sizes of these packages are almost identical on all platforms. Fedora and Ubuntu are just the standard popular linux distributions in their respective server edition (no gui) and serve as a reference to demonstrate overhead introduced by using traditional operating systems as guests. The remaining ones are different customized versions of OSv. JDK is a bare OSv image with Java Runtime on top. Redis is a popular key value database belonging to the family of NoSQL databases. MRuby and Native are custom OSv versions containing small hello world programs that print to STDOUT every 5 seconds. They were implemented using C and Ruby respectively. Agent is the only guest system containing some meaningful application logic. It is developed for evaluation in work package 4 and shall be used within scenario 3.

We need to remark at this point, that the guest systems MRuby and Native result in a package of only 13MB, which means only 13MB need to be transferred in order to instantiate the VM on the target platform and be able to resume its execution. Another observation can be made by just looking at the package sizes: using java for developing the application seems to introduce a lot of overhead in terms of serialized package size. Compared to the bare JDK image, the size demand has nearly doubled just by implementing some application logic. We need to further evaluate, if more lightweight solutions can be implemented by using native or MRuby guest systems.

The content of a VM package is described in section 4.2.3. The snapshot containing memory and CPU state and the overlay base image are having the highest influence on the package size. As no guest system is making heavy use of its virtual disk this contribution is nearly negligible. In all cases the overlay image is just around a few megabytes in size. The main contribution is coming from the

serialized memory snapshot. Although zip compression of the package is helping a lot to reduce the overall size, it becomes clear that using traditional operating systems as guests will most likely not be acceptable. Besides package size, the base image of OSv instances is also much smaller. The base image needs to be available on all systems that should be able to continue execution.

Table 3 Guest Operating Systems for Evaluation

Guest	Package	Snapshot	Disk Overlay	Base Image
Fedora	58.0MB	175.0MB	6.3MB	205.0MB
Ubuntu	53.0MB	174.0MB	5.9MB	243.0MB
Agent	36.0MB	98.0MB	1.4MB	82.0MB
JDK	15.0MB	39.0MB	1.2MB	80.0MB
Redis	15.0MB	40.0MB	1.2MB	12.0MB
MRuby	13.0MB	43.0MB	1.2MB	14.0MB
Native	13.0MB	33.0MB	1.2MB	17.0MB

Table 5 and Table 6 show the time needed to save and restore instances measured with the *time* utility. These results are also visualized in Figure 7 and Figure 8. Several observations can be made:

- More Computing power (CPU clock rate) results in lower execution times. This is most likely coming from the computational overhead of the zip-compression.
- Bigger snapshots result in higher execution times.
- Restore tend to be faster than save, which is also traceable to the zip-compression overhead.
- Save and Restore times in OpenStack are notably higher which is coming from the additional overhead introduced by OpenStack. Some optimization might be possible here to achieve similar numbers like the local tresccad version.

Let Δs be the time needed to save a VM on the source host and Δr to resume the respective VM on the target host. If Δt indicates the time it needs to transfer to vm package between source and destination the overall time needed to migrate the vm is given by: $\Delta m = \Delta s+\Delta r+\Delta t$.

For example transferring an Ubuntu VM between the local notebook and the local OpenStack environment will at least need $\Delta s+\Delta r=43.6s+29.5s=73.1s$ and this does not include the time needed to transfer the VM package of about 53MB. Given a 100Mbit/s connection, the transfer will take about another 5 seconds. Transferring VM packages in WAN environments with much lower bandwidths will most likely not be feasible.

Transferring a native application on OSv between the local notebook and a VM@OFFIS achieves the best results. It only takes $\Delta s+\Delta r=7.2s+3.3s=10.5s$. The additional time it needs to transfer the package of about 13MB seems also be acceptable even with lower bandwidths available.

Table 4 Measurements of time needed to save a VM

Save

	Fedora	Ubuntu	Agent	JDK	Redis	MRuby	Native
eee(QEMU)	-	-	75,1s	43,0s	46,4s	46,0s	42,0s
OpenStack@OFFIS	63,7s	70,9s	29,2s	19,3s	19,8s	16,5s	16,4s
VM@OFFIS	48,8s	47,6s	21,8s	9,2s	9,7s	9,8s	8,5s
local	38,6s	43,6s	17,5s	7,2s	8,0s	7,9s	7,2s
VM@WT(QEMU)	83,1s	55,9s	26,5s	8,2s	8,7s	10,4s	7,8s

Table 5 Measurements of time needed to restore a VM

Restore

	Fedora	Ubuntu	Agent	JDK	Redis	MRuby	Native
eee(QEMU)	-	-	53,3s	31,6s	28,9s	29,3s	24,1s
OpenStack@OFFIS	27,6s	29,5s	22,4s	15,9s	16,2s	15,3s	14,9s
VM@OFFIS	13,1s	13,5s	10,5s	3,9s	3,9s	3,8s	3,3s
local	13,8s	14,7s	8,0s	3,9s	3,5s	4,7s	3,0s
VM@WT(QEMU)	172,1s	155,6s	9,2s	21,6s	6,9s	8,8s	6,0s

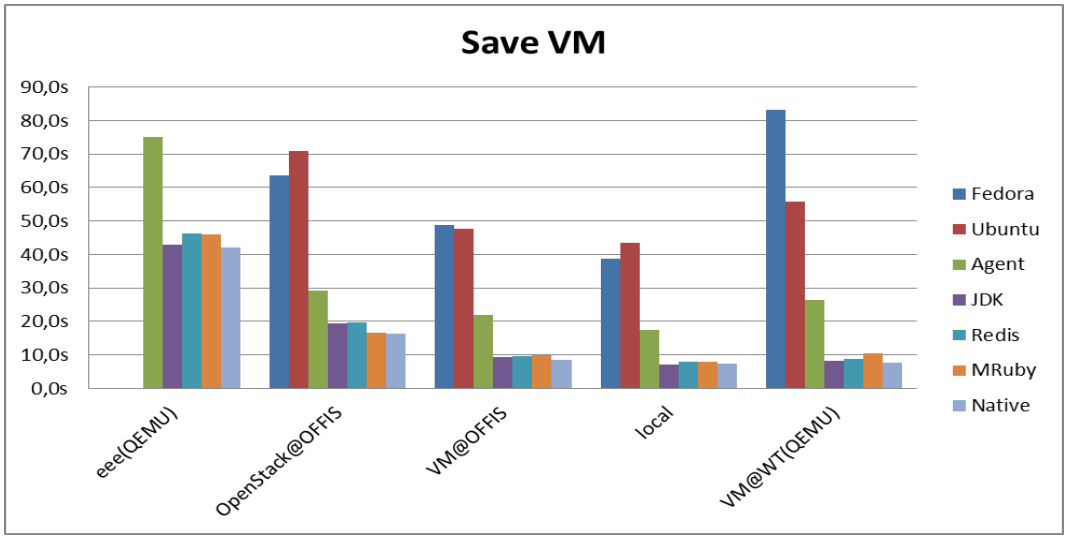

Figure 8 Measurements for saving a VM

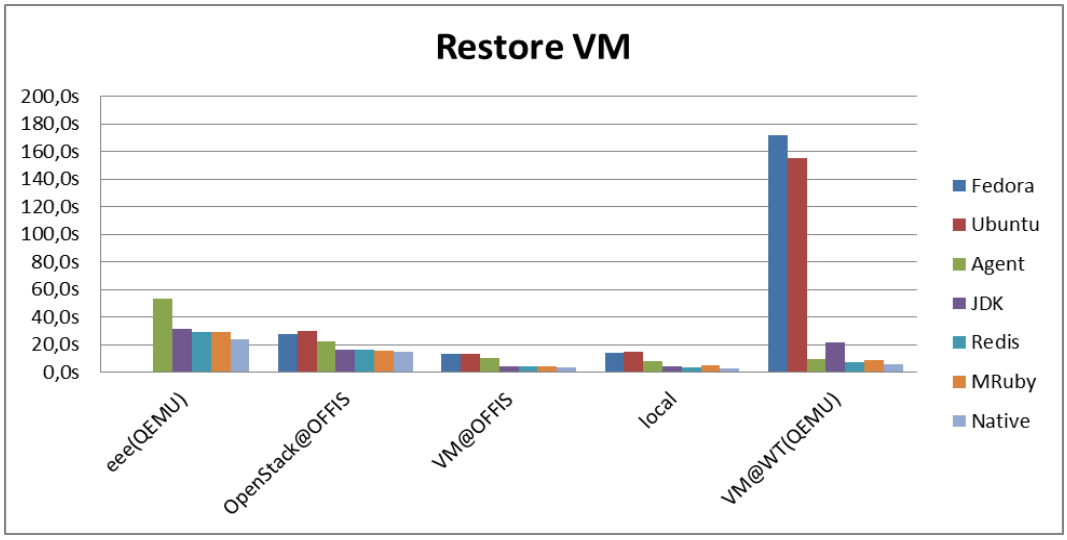

Figure 9 Measurements for restoring a VM

5 CONCLUSION

This document describes the initial prototype for emulation and VM serialization. It makes sophisticated progress towards the goal of Task 3.3 to securely off-load functionality to the cloud. This is done by utilizing the ability of strong off-line migration of virtual machines. Functionality can be encapsulated in tiny virtual machines that are able to be transferred between clouds and local devices by using the prototype called tresccad. This component serves as a manager for running virtual machines and is able to serialize them into packages and send them off to remote locations. OpenStack or any other tresccad instance at the remote location can then unpack and continue the execution of the respective VM. Furthermore, the developed API is compatible to existing OpenStack Endpoints and an appropriate plugin for OpenStack has been developed as well.

The evaluation shows that the feared overhead introduces by full virtual machines can be reduced by using QCOW base images and small and optimized guest operating systems like OSv. Although, OSv is first of all meant as a lightweight operating system for the cloud, it is also a promising direction within the TRESCCA project and can serve as a lightweight system for the evaluation scenarios. The achievable reduction of overhead in terms of package size is already quite reasonable when using OSv. Other optimizations could be explored in this context, as well. For example, it would be possible to omit the need to transfer a complete VM package but only sent differences that will be applied to a base package. Such a concept was introduced by [Satyanarayanan2009] and called dynamic vm synthesis and could be applied for tresccad as well.

The overall approach offers strong migration, meaning execution can be stopped at any point in time and all related CPU and memory state can be serialized. Previous work on code off-loading based on Java always struggled to achieve this feature because the JVM does not provide any mechanism to serialize instruction pointer and related native state. Most often, weak migration is used as a work-around which requires the programmer to be aware and explicitly mark points where migration can be done or she needs to serialize relevant execution state on her own. However, it’s not yet clear what benefit strong migration will have on application development. Some previous work claim that strong migration would be easier to understand for programmers [White1998] but more sophisticated empirical evidence would be beneficial. Moreover, for scenarios in which VMs would need to quickly switch their locations back and forth it would mean, although the overhead is reduced, a lot of meaningless information coming from the guest operating system will be spread around all the time. In these cases it would be more efficient to rely on classical protocol based communication. Undoubtedly, it’s necessary to find proper applications where strong migration of VMs can clearly demonstrate its benefits.

5.1 Outlook

5.1.1 ARM at Cloud Side

The current approach for hybrid migration assumes that hardware accelerated VMs on a local device can continue their execution at server-side by using QEMU. But we also expect that ARM will find its way into the server market resulting in a higher diversity in terms of architectures at cloud-side. This will introduce the possibility to make use of ARM hardware acceleration in the cloud as well.

5.1.2 Active Migration

The point of active migration is that VMs could decide actively when and where they want to be migrated. The term is coming from mobile agents and could be adapted for virtual machines as well. Residing virtual machines can reach their respective tresccad by using the link local address. And the

available functionality to migrate VMs could also be called actively by themselves. This introduces several potentials right up to new alternative development paradigms to classical client-server architectures and need to be explored further.

5.1.3 CloudStack

The integration into other cloud environments is planned but it would also possible to focus on tresccad as this can be run inside any given VM regardless on which cloud infrastructure it is running. This implies nested virtualization, but as the integration of hybrid migration implies the usage of QEMU at cloud-side anyway, this could be reasonable. Furthermore, nested virtualization doesn't necessarily mean a huge performance decrease because a guest system can share the virtualization extensions of its host to its own nested guests.

6 BIBLIOGRAPHY

[Bouchenak2003] Efficient Java thread serialization, Sara Bouchenak, Daniel Hagimont and Noël De Palma, in: Proceedings of the 2nd international conference on Principles and practice of programming in Java, Kilkenny City, Ireland, pages 35--39, Computer Science Press, Inc., 2003

[Hirofuchi2009] A Live Storage Migration Mechanism over WAN for Relocatable Virtual Machine Services on Clouds, Takahiro Hirofuchi, H. Ogawa, H. Nakada, S. Itoh and S. Sekiguchi, in: Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on, pages 460-465, 2009

[Quitadamo2008] Mobile JikesRVM: A framework to support transparent Java thread migration, Raffaele Quitadamo, Giacomo Cabri and Letizia Leonardi (2008), in: Science of Computer Programming, 70:2–3(221 - 240)

[Riteau2011] Shrinker: Improving Live Migration of Virtual Clusters over WANs with Distributed Data Deduplication and Content-Based Addressing, Pierre Riteau, Christine Morin and Thierry Priol, in: {17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011)}, 2011

[Satyanarayanan2009] The Case for VM-Based Cloudlets in Mobile Computing, Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres and Nigel Davies (2009), in: IEEE Pervasive Computing, 8:4(14--23)

[Suezawa2000] Persistent execution state of a Java virtual machine, Takashi Suezawa, in: Proceedings of the ACM 2000 conference on Java Grande, San Francisco, California, USA, pages 160--167, ACM, 2000

[Truyen2000] Portable Support for Transparent Thread Migration in Java, Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen and Pierre Verbaeten, in: IN ASA/MA, pages 29--43, Springer-Verlag, 2000

[Vigna2004] Mobile agents: Ten reasons for failure, Giovanni Vigna, in: Mobile Data Management, 2004. Proceedings. 2004 IEEE International Conference on, IEEE, pages 298--299, 2004

[White1998] Eric White. A comparison of mobile agent migration mechanisms. Senior Honors Thesis, Dartmouth College, June 1998