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:	**D3.4**
Document title:	Emulation and VM Serialization
Version:	1.1
Due date of document:	30st of March 2015
Actual submission date:	30th of April 2015 (V1.0), 23rd of January 2015 (V1.1)
Lead beneficiary:	OFFIS
Participants:	Bernhard Katzmarski (OFFIS), Sven Rosinger (OFFIS)
Reviewers:	Michele Paolino (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 document is Deliverable 3.4 and describes the results coming from Tasks 3.3 of the TRESCCA project. It builds on previous work already documented in D3.3 within the first period but can be considered as a stand-alone document. It thus repeats the related work study and technical requirements in Chapters 2 and 3. Chapter 4 has completely been updated to incorporate changes in the VM migration prototype architecture and implementation as well as to describe the final feature set. Further this report includes a step-by-step how-to for installation of the prototype to make use of it by the public domain.
Chapter 1.3 gives a more detailed overview of changes and updates provided in this new release D3.4 of the Emulation and VM Serialization report.

The main purpose of Task 3.3 is to come up with a solution that allows applications to change their execution location dynamically at runtime. This movement can also be called migration and within TRESCCA, it can allow applications to either make use of processing power in the cloud or security properties on local devices depending on the current task. Hence, the most significant result in the second phase of Task 3.3 is the prototype called tresccad. Its implementation allows off-line migration of VMs between clouds and local clients while respecting heterogeneous architectures (x86, ARM). Applications will be embedded in these VMs in order to make use of isolation properties coming from the hypervisor. The most recent version of this prototype is available on GitHub[1] as Open Source.

Conceptually, the goal is very similar to Mobile Agent Systems (MAS) and since the beginning of Task 3.3, it was clear that VM-based migration will introduce a large overhead compared to MAS based on Java. We decided to stick with VMs because of its compatibility and to study its feasibility. A VM-based approach is generally more compatible to existing IaaS solutions and a lot of overhead coming from the VM itself can be reduced by using more lightweight VMs like Unikernels. One of the main advantages of following the VM-based approach is that it does not restrict applications to a fixed programming language and that also legacy applications can be included.

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 addressed for a VM-based solution of Task 3.3. The current prototype and its implementation decisions are described in Chapter 4. Evaluation is done in Chapter 5, whereas Chapter 6 concludes this deliverable and gives an outlook.

[1] https://github.com/TRESCCA

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 Purpose of the Document

This document is deliverable D3.4 of the TRESCCA project. It describes the work done in Task 3.3 and is the successor of D3.3 which described the intermediate state. Task 3.3 mainly aims to develop a prototype for dealing with virtual machines as containers to support secure data processing between cloud and client devices. The prototype itself serves as a basis for evaluation in 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
1.0	30.04.15	Submitted to European Commission
1.1	30.11.15	Revised document. Changes done: • Section 1.3 added describing updates with respect to intermediate report D3.3 • TRESCCA deamon *How-To* added to Section 4.5 • Added Section 2.5 “Conclusion and distinction of presented technique to related work” • General revision of document including references and figures

1.3 Developments in the last period and update to intermediate report D3.3

In D3.3 the status of the prototype has been described as it has been developed until September 2014. Since then, the prototype has been improved in many directions and updated versions have been published regularly on GitHub as Open Source. These updates include performance improvements, feature updates as well as refactoring updates and are listed below:

- Support for ARM instances
- Compute node tracking and key management
- Native libvirt binding for better performance
- Native socket protocol with protobuf instead of http for better performance
- SSL socket communication
- Merged tresccad deamon and tresccac client
- Active migration support

These changes implied several changes in comparison to the initial report D3.3 that are summarized in the following:

Section 4.1 gives an updated architectural overview in Figure 1 including the interfaces and used APIs. Further it is described why the initially planned direct integration into OpenStack has been skipped and how the current tresccad daemon implementation benefits from this decision.
Section 4.2 focuses on changes regarding the communication between multiple tresccad instances as a result of skipping the direct OpenStack integration (Sections 4.2.1 and 4.2.2), due to performance increasing measures such as using a native socket protocol with protobuf instead of using http (see

Section 4.2.3) as well as refactoring measures such as the merge of tresccad and tresccac (see Section 4.2.4).
While in D3.3 only the concept for hybrid migration as well as existing limitations at that time have been described, Section 4.3 now describes the recent implementation and feature set. In addition, Section 4.3 also describes new security features that have been integrated in the past month such as SSL socket communication as well as VM encryption.
The final evaluation of the prototype in Section 5 has been updated to cover the changes described above. In addition, the evaluation now covers a lightweight Linux Kernel with an embedded ramdisk containing only the required minimum of applications and libraries that has finally been used in the demonstrator to limit the overhead for the VM migration.

As a latter aspect, a conclusion and a distinction of the presented technique and related work has been added to Section 2.5.

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 clouds and client devices. Numerous projects already addressed the idea to move computing tasks between platforms. This idea is often described as mobile code and 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 examples for mobile code are 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 but 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.

Section 2.5 gives a conclusion and summarizes the relation of the proposed VM-based technique developed in TRESCCA and related work.

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. Especially Docker[2] created a huge hype in this context as it makes containers available for the masses by providing easy tools and interfaces in user-space. Applications are simply bundled together with all dependencies and configuration into a 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[3]), 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 [4](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[5] 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 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.

[2] http://docker.io
[3] http://criu.org
[4] http://jade.tilab.com/
[5] http://aglets.sourceforge.net/

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. Base64 is an encoding scheme meant to represent binary data as ASCII strings and is by no means a replacement for any encryption scheme: it can be encoded and decoded on any platform. In the JADE security guide[6] only authentication and message integrity/confidentiality between agents are covered. However, this does not protect agents against other maliciously agents or 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.

2.5 Conclusion and Distinction of Presented Technique to Related Work

In our work that is presented in the following and that has initially been published in [Katzmarski2014], we combine the concept of Mobile Agents presented in Section 2.1 with the benefits of virtual machines. Virtualization offers a generic execution layer that provides isolation and strong migration. Hence, we consider the virtual machine itself being the agent, able to move between platforms.

As a result and unlike previous work in this area, the Tresccad solution present a full operating system level virtualization based approach where a VM will be able to trigger migration processes actively to meet security requirements. A VM-based approach is also more general and has no programming language restrictions, which makes it possible to even use legacy systems.

A Mobile Agent System on top of virtual machines can create new possibilities, as well as introduce distinct challenges. In this setup the virtual machine is becoming the agent, with the ability to move actively between systems. Due to its strong isolation property, it overcomes known issues in existing MAS as described in Section 2.1. Within a partly trusted domain, this approach can create some security advantages, as data does not have to leave anymore, but software is coming as a visitor. On the other hand, this approach allows the integration into Infrastructure as a Service (IaaS) platforms that are based on VMs as well. As it is the most generic solution in today’s cloud layer model, this allows the development of a flexible architecture and no programming language restrictions with maximum synergy.

[6] 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 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.

3.2 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.

3.3 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.

3.4 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.

3.5 Active Migration and Location Awareness

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 possible impact on IaaS 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.

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.

4 Tresccad

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 with other platforms as it only requires the presence of QEMU and Libvirt.
For this purpose, a prototype called tresccad has been developed. Its name is simply a shortcut for trescca daemon. Tresccad offers required features like package creation of residing VMs and is able to move them to other platforms where an instance of tresccad is present.
The development was done on Ubuntu 14.04, but it can be expected that other Linux-based distributions will be compatible as well. Tresccad requires certain packages for virtualization like libvirt, qemu-kvm.

4.1 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 that tresccad is able to take control over libvirt and can be run on 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.

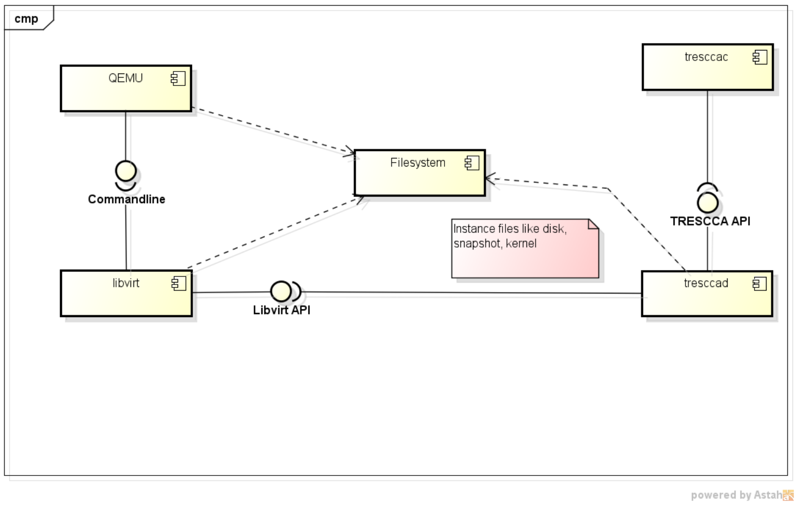


Figure 1 Tresccad architecture overview

4.1.1 OpenStack Integration

Multiple open-source cloud platforms have evolved over recent years including OpenStack, Eucalyptus, CloudStack, OpenNebula. At the beginning, we chose 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.
While the implementation of tresccad is inspired by concepts of OpenStack and other solutions, direct integration into OpenStack has been sacrificed during the development. The concept of pulling and pushing VM packages (see Section 4.3.2) into the image repository of OpenStack introduced a lot of overhead that can be saved. Also, it turned out that integrating other solutions like CloudStack into tresccad would introduce unnecessary overhead to implement required plugins. Hence, tresccad can be installed directly on instances running in a cloud without direct integration into the management layer

which offers higher flexibility and lower runtime requirements. Furthermore, running instances can make use of existing KVM extensions on the host which leads to nested virtualization but the performance impact is very low. Currently, this would only be possible for x86 servers because nested virtualization for KVM on ARM is not supported yet.

4.2 Communication

Communication between multiple tresccad instances exclusively takes 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 and third party applications. Most cloud providers already make use of network based communication for most of their API's and work internally with HTTP among others. 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.
Tresccad listens on a configurable port and accepts SSL-secured connections from other tresccad instances or the command line client (see Section 0).

4.2.1 Host-level

Tresccad is also responsible for managing an internal network on the host system to provide connectivity to residing VMs. They will be connected to a virtual network device managed by libvirt. A DHCP server is listening of the respective bridge that serves as a gateway for VMs and is responsible to assign dynamic addresses to the instances. For the prototype the open source DHCP server dnsmasq is used, which is controlled by tresccad. There are certain pitfalls, when tresccad 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 need 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 at all.

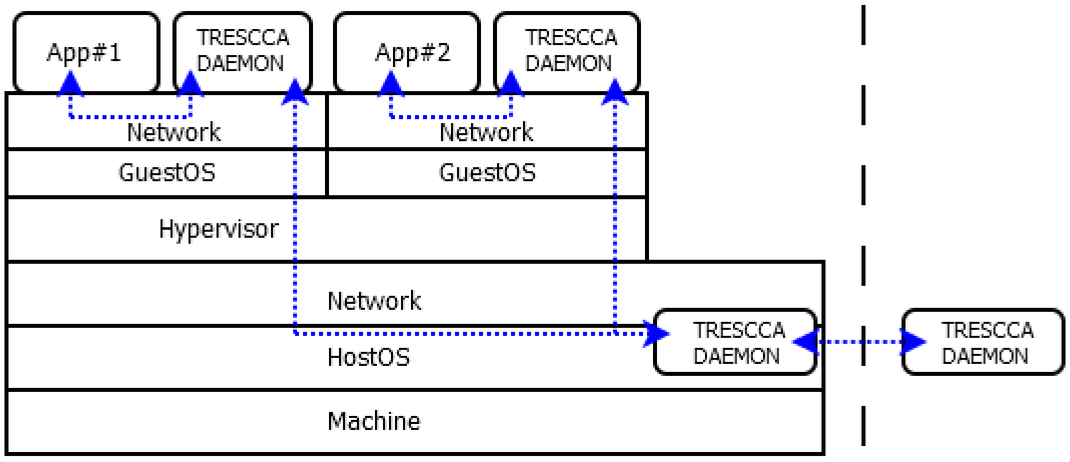


Figure 2 Communication Overview

4.2.2 VM-level

Tresccad can allow communication between VMs via iptables rules. All VMs are assigned to a private subnet to communicate with each other and reach external services by using the hosts gateway. Moreover, tresccad's API is reachable for VMs under the link local address[7] 169.254.169.254. The

[7] RFC 5735

concept of using link local address is also used by Amazon's EC2 API or Openstack's compatible API, where 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 could be deployed inside the VM as a library to ease the development of applications.

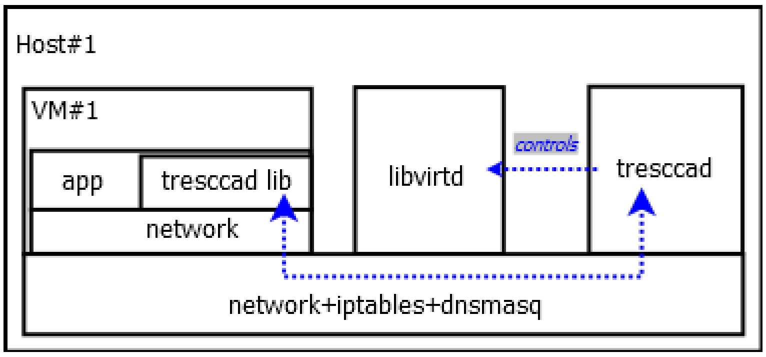


Figure 3 Tresccad VM Communication

4.2.3 Protocol

The HTTP-based communication protocol has also been replaced by Procol Buffers[8] over a native socket connection. This reduces the overhead coming from HTTP and leads to a more lightweight and readable protocol. Communication between tresccad and tresccac or other tresccad instances then simply follows the request-response pattern by exchanging Protobuf messages like depicted in Figure 4.

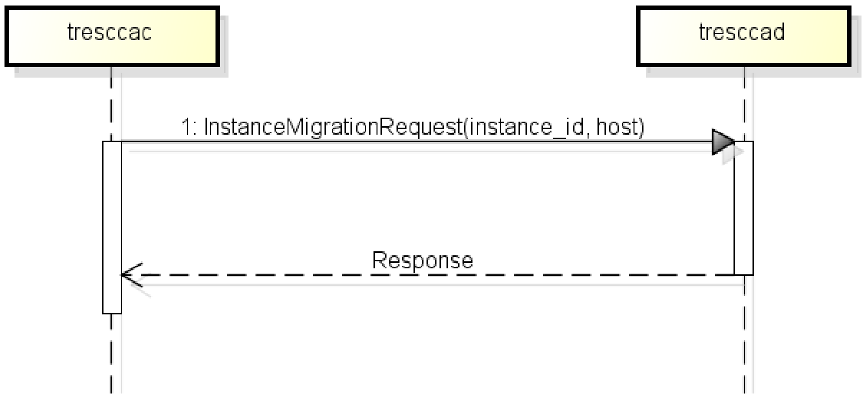


Figure 4 Request-response communication based on exchanging Protobuf messages

4.2.4 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. The terminology has been inspired by OpenStack. An instance is a VM and an image is either a disk image or a VM package created by tresccad. At the beginning, tresccac and tresccad had been developed as separate projects. They have been merged together because it doesn't offer a specific benefit the keep them

[8] https://code.google.com/p/protobuf/

separate. It eases management of the code base and introduces optimization potential. Currently, tresccac features following commands:

```
$ tresccac
Tasks:
 tresccac delete_all          # delete all images and/or instances
 tresccac help [TASK]         # Describe available tasks or one specific task
 tresccac images <command>    # images tasks
 tresccac instances <command> # instances tasks
 tresccac list                # receives a list of available instances/images
 tresccac nodes <command>     # node tasks

$ tresccac images
Tasks:
 tresccac images add IMAGE_PATH                  # adds an image by a given path
 tresccac images delete ID                       # delete an image
 tresccac images help [COMMAND]                  # Describe subcommands or one
                                                 # specific subcommand
 tresccac images pull IMAGE_ID NODE              # asks NODE(by name or
                                                 # fingerprint) to pull the
                                                 # residing image
 tresccac images spawn IMAGE_ID or IMAGE_PATH    # instructs tresccad to spawn a
                                                 # vm from a given image

$ tresccac instances
Tasks:
 tresccac instances delete ID                 # delete an instance
 tresccac instances help [COMMAND]            # Describe subcommands or one
                                              # specific subcommand
 tresccac instances migrate INSTANCE_ID NODE # migrates an instance to another
                                              # NODE
                                              # given by name o...
 tresccac instances save INSTANCE_ID          # saves a running instance as a
                                              # zipped package

$ tresccac nodes
Tasks:
  tresccac nodes add HOST             # adds a node to the trusted pool
  tresccac nodes delete FINGERPRINT   # deletes a node from the trusted pool
  tresccac nodes help [COMMAND]       # Describe subcommands or one specific
                                      # subcommand
  tresccac nodes list                 # list all known node
```

The "*migrate"* 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 5.

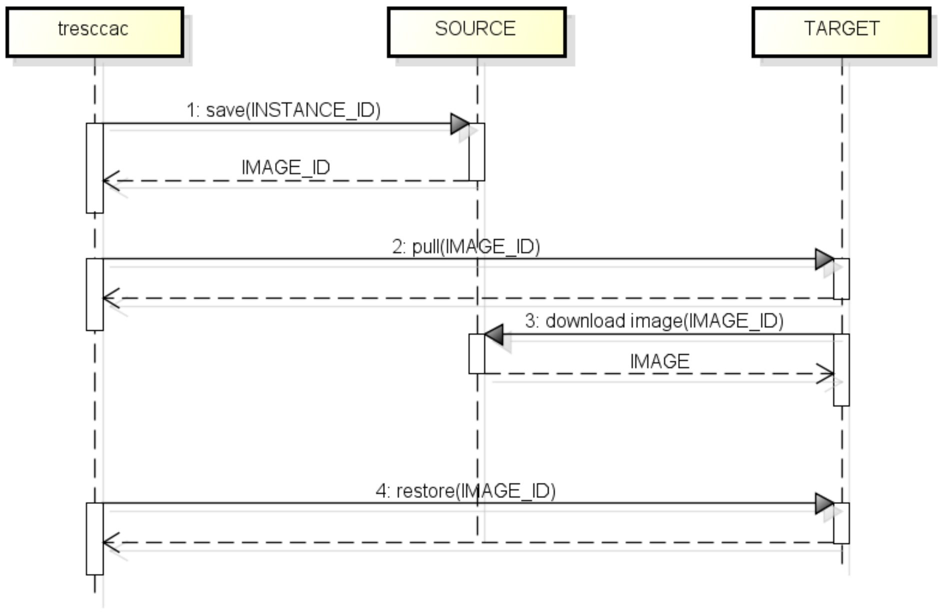


Figure 5 Tresccac Migrate Sequence Diagram

4.3 Migration

The following sections describe the technical solution for the migration functionality implemented in tresccad in more detail.

4.3.1 Off-line Migration: Save and Restore

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.
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 is basically in the responsibility of tresccad.
As tresccad is relying on libvirt for saving and restoring virtual machines this also means known limitations apply. 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 libvirt's 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 (see Section 4.3.2) containing snapshot and disk overlay, this risk can be reduced to a minimum because the needed backup is implicitly available in the VM package.

4.3.2 VM Packaging and Encryption

As already mentioned, we assume the presence of VM-related files (given in Table 1) 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. Compression can be turned on and off depending on the use case and the corresponding compute node.

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

QEMU can make use of QCOW base images out of the box (refer to section 4.4). 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.

Protecting VM packages during transfers from eavesdropping can be achieved in many ways and should be applied in general because without package encryption, sensitive data can easily be extracted or modified from the VM snapshot. On the one hand TLS is used to protect the communication channel between two tresccad instances. Furthermore, a basic Public-Private Key Scheme has been implemented to protect VM packages. In this way, they can be encrypted using the public Key PK_Host#x of the respective destination host. These keys need to be exchanged before moving VM packages as shown in Figure 6. Private keys need further runtime protection that may come from other TRESCCA components.

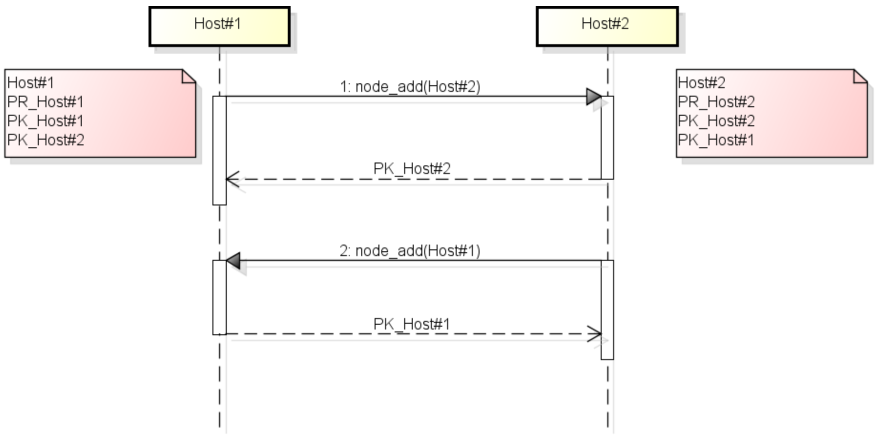


Figure 6 Two tresccad instances exchanging their Public Keys

Further visualizes the implemented virtual machine package encryption. The VM package is compressed and symmetrically encrypted with a random password first. This password is again encrypted using the pubkey of the destination host and transmitted together with the package.

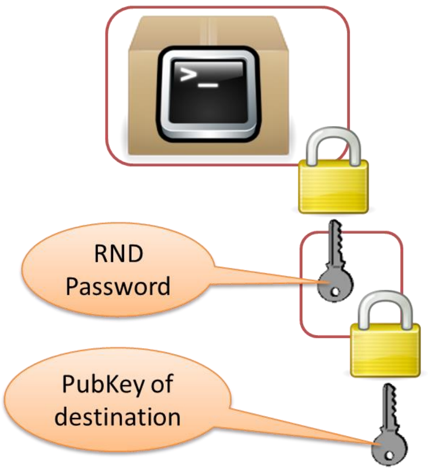


Figure 7 Virtual machine package encryption

4.3.3 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 QEMU/KVM, 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.

KVM 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 functionally 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 hybrid migration and the porting of tresccad to ARM platforms were integral parts of the final phase in Task 3.3. Tresccad is now deployable on both target platforms and can handle x86 as well as ARM VMs, while the latter are more reasonable choices for evaluation and demonstration. X86-based VMs should not be used because they are not stable when executing them on ARM.
Furthermore, hybrid migration developed by VOSYS has been fully integrated and allows hardware-accelerated guests on ARM platforms to continue their execution on x86 hosts by making use of emulation. Hybrid migration code has already been sent to the QEMU developer's mailing list[9].

4.3.4 Active Migration

Tresccad also features an early proof of concept implementation for active migration. Usually, tresccad is responsible for managing the host's internal network and to assign IP addresses to running instances (refer to Section 4.2.1). A running VM on the other hand needs to set up its virtual network device properly with DHCP in order to obtain an address. Under these circumstances, a VM gains the ability to reach tresccad running at host-level, by connecting to its gateway, which is in fact a virtual network bridge on the host. It can issue valid requests to tresccad in order to get migrated to another host platform. In this way, tresccad is even able perform some validation because it can associate requests to their respective issuers by looking at their source addresses.
Furthermore, other host-level services like for example the X11 server can be made available to residing VMs as well. In this way, an application inside a VM can display some graphical user interface within the current display on the host. All of this is illustrated in **Figure 8** and has been prototypically implemented. Sources can be found on Github[10].

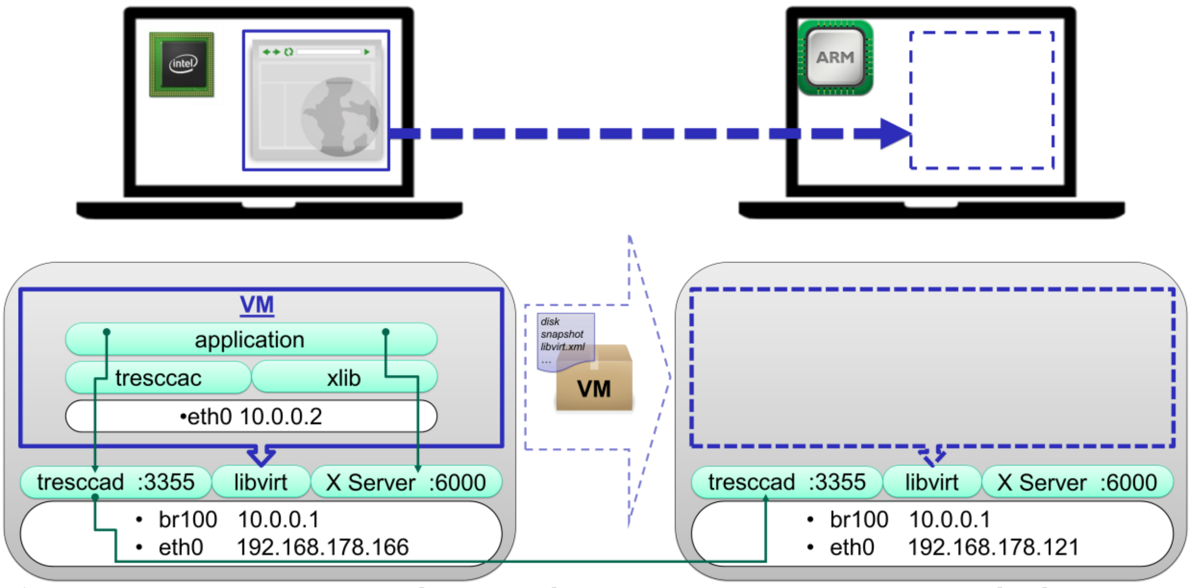


Figure 8 Active Migration: An application within a VM requests a migration to another host

4.3.5 Integration of Security Features

Thus far, the presented approach does not offer any direct security functionality and is by no means more secure. However, by integrating other components developed in TRESCCA the foreseen approach can demonstrate its benefits. Most promising will be the integration of HSM-Mem to protect external memory of moving VMs. Application developers can then decide to move to a target device in order to benefit from memory protection for sensitive operations. During the integration phase some

[9] https://lists.gnu.org/archive/html/qemu-devel/2014-03/msg03660.html
[10] https://github.com/TRESCCA

technical problems will need to be solved. It will be necessary to define how security policies of HSM-Mem can be migrated. The simplest solution will be to protect the whole address space used by a VM but this overhead will most likely not be practicable. As Linux drivers for the HSM-Mem are becoming available they could be made available for VMs as well and offer residing applications to use it from inside a VM. Configuration and management for HSM-Mem would then be in the responsibility of the application developer and would offer the most flexible solution.
For implementing a real use case, some PKI environment would be needed as well. Before executing foreign VMs, their origin would need to be attested to assure they were created by trustworthy parties. Required keys could either be stored in TPMs or the TEE. In this way, a key infrastructure would also allow signing VM packages on each host, so that their entire execution trace can be validated. This certification can be part of tresccad and may be implemented during integration phase and scenario development.

4.4 Reducing the Overhead

Virtual machines usually need disk images and memory of several gigabytes 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.

4.4.1 Disk Overlays

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.

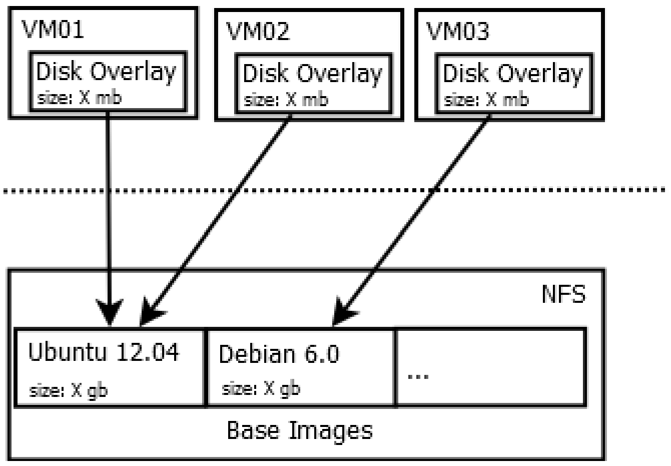


Figure 9 Disk Image Overlay Concept

4.4.2 Suitable Guest Operating Systems

A lot of overhead can be reduced by not taking a full Linux distribution as a guest system, but instead a more lightweight system, like for example:

- SliTaz
- Damn Small Linux
- Tinycore Linux
- ttylinux
- PuppyLinux

Another promising Operating System for a guest is the recently introduced OSv[11], which is especially designed to run on a hypervisor. OSv belongs to a recent development that can be called Unikernels or Library OS. The main idea is to have an all-in-one image for the respective application and keep the operating system stack as light as possible. 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. While OSv is mostly POSIX compatible, more radical approaches in this field like MirageOS[12] implement all necessary operating system components and libraries from scratch and thereby scarify all legacy parts. Because of this, MirageOS is language-dependent to OCaml but is also offers the most lightweight solution, as the application is compiled only with libraries and operating system features that are really used by the application. Unfortunatly, MirageOS cannot be used within TRESCCA, because it is currently only available for XEN Hypervisors. A port for KVM could be done but is out of the scope for this project. Same applies for OSv; it could be a basis for the migratable VMs that need to be lightweight and only serve one application. In fact, the project also has ARM-support on its agenda, but unfortunately this hasn’t reached a usable state. This is why OSv cannot be properly evaluated especially in the context of hybrid migration between ARM and x86 platforms.
As the evaluation section 5 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.

Another lightweight solution is to build a system from scratch by using only the Linux Kernel with an embedded ramdisk containing only the required minimum of applications and libraries. Such a Kernel only is about ~5MB in size and a created package results in ~24MB (refer to Section 5). The final evaluation of the prototype and the development for the upcoming demonstrators is based on this solution as it offers a maximum of compatibility and flexibility.

4.4.3 Performance Improvements

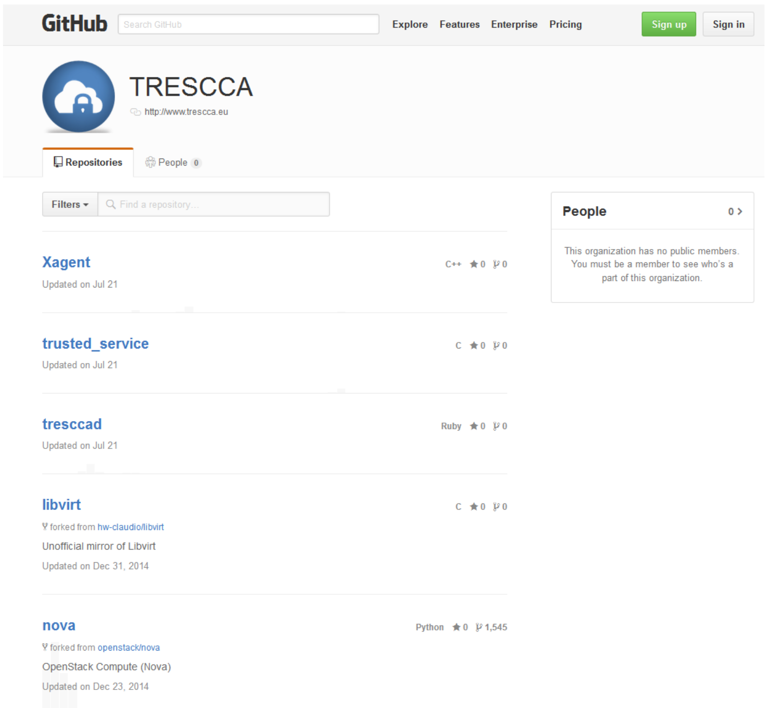
Since the release of the first working prototype of tresccad some effort has been spent to improve the overall performance. These improvements shall be reported in this section shortly. As previously stated, a lot of overhead could be saved by scarifying direct integration into OpenStack. Most overhead in terms of migration duration was coming from the image repository Glance, where VM packages were pulled and pushed into. By omitting the usage of Glance this additional step could be avoided and VM packages now are sent directly between tresccad instances.
The HTTP-based communication protocol has also been replaced by Procol Buffers[13] over a native socket connection. This reduces the overhead coming from HTTP and leads to a more lightweight and readable protocol.
Furthermore, communication between tresccad and libvirt has been improved. Tresccad now uses Libvirt’s native API for communication, instead of command-line parsing of virsh. This does not only reduce overhead coming from calling external programs and parsing their result but also enables to

[11] http://osv.io
[12] http://www.openmirage.org/
[13] https://code.google.com/p/protobuf/

benefit from custom modifications made in libvirt. For instance, this has been done to achieve Abiless restore, which is a short term for a problem encountered very early in the development: Libvirt saves the XML configuration of saved instances inside the resulting snapshot (compare section 4.3.1). The drawback of this approach is that some values, like paths or unique names need to be modified before an instance can properly be resumed on a foreign platform. The previous approach was to modify the snapshot on the target platform directly to change required values. Depending on the size of the snapshot this caused some overhead. Indeed, libvirt offers the possibility to provide an additional XML configuration for overwriting values when resuming a snapshot, but the content is compared to the residing one and any required change fails the ABI compatibility checks within libvirt. Therefore, a patch[14] for libvirt has been created that allows skipping this kind of validation in order to maximize freedom when providing custom XML configurations, which improves restore performance of machines. Due to the integration of native Libvirt API, tresccad can check for the presence of this feature and use it transparently on platforms where it is available.
Some performance overhead is surely coming from using an interpreted language namely Ruby[15] for implementing tresccad. However, this offered a fast and agile development cycle for the prototype and does not prevent a native implementation of tresccad in the future. Some findings and concepts of tresccad may also be integrated into libvirt directly. But currently it is beneficial to keep it as separate service because requirements of TRESCCA are somewhat special compared to uses cases covered by libvirt.

4.5 Installation of tresccad

TRESCCAD is available on GitHub as Open Source as shown in Figure 10 via the following link: https://github.com/TRESCCA.



TRESCCA
http://www.trescca.eu

People 0
This organization has no public members. You must be a member to see who's a part of this organization.

Xagent — C++ ★0 0
Updated on Jul 21

trusted_service — C ★0 0
Updated on Jul 21

tresccad — Ruby ★0 0
Updated on Jul 21

libvirt — C ★0 0
forked from hw-claudio/libvirt
Unofficial mirror of Libvirt
Updated on Dec 31, 2014

nova — Python ★0 1,545
forked from openstack/nova
OpenStack Compute (Nova)
Updated on Dec 23, 2014

Figure 10 TRESCCA repository on GitHub

Development and testing have been done against Ubuntu 14.04 LTS.

[14] https://github.com/TRESCCA/libvirt
[15] https://www.ruby-lang.org

4.5.1 Prerequisites

Following packages need to be installed on the target system.

```
$ sudo apt-get update
$ sudo apt-get install ruby1.9.1 ruby1.9.1-dev rubygems1.9.1 build-essential
libsqlite3-dev zip unzip qemu-kvm libvirt-bin libvirt-dev bsdiff
```

Make sure you have rake and bundler installed:

```
$ gem install rake
$ gem install bundler
```

4.5.2 Install tresccad

```
$ git clone https://github.com/TRESCCA/tresccad.git
$ cd tresccad
$ bundle install
```

Run the tests with:

```
$ rake test
```

Install with:

```
$ rake install
```

Create initial configuration and set up virtualization:

```
$ sudo tresccad-virt-install
```

Start the daemon:

```
$ sudo tresccad start
```

4.5.3 Usage

```
$ tresccac
  Tasks:
    tresccac delete_all                    # delete all images
        and/or instances
    tresccac help [TASK]             # Describe available tasks or
              one specific task
    tresccac images <command>        # images tasks
    tresccac instances <command>     # instances tasks
    tresccac list                    # receives a list of
                                           currently available
                                           instances/images
    tresccac nodes <command>         # node tasks
```

4.5.4 Migrate an ARM-based instance

Add the foreign host to the pool:

```
    $ tresccac nodes add 192.168.178.121
    Node chromebook added
```

Spawn an instance from an existing kernel image:

```
    $ tresccac images spawn zImage --domain_type=arm
    Image 650d0e3b-e237-ada0-a5ec-25dcd710a247 created
    Instance 3e7f0524-dde6-3e0d-7587-b7b60edea3cd spawned with IP 10.0.0.2
```

Migrate the instance to the remote host:

```
    $ tresccac instances migrate
3e7f0524-dde6-3e0d-7587-b7b60edea3cd chromebook
    Instance 73a95d67-7f9e-d9bc-e70f-e7a2b08dc5b6 migrated to chromebook
```

5 Evaluation and Laboratory Testing

This section describes the evaluation and laboratory testing that was conducted with tresccad. The main goal here is to evaluate the runtime overhead introduced by using VM migration to fulfil some use case. The question to be answered is namely how long does it take to migrate an instance between two hosts. Keeping runtime overhead as low as possible is an important requirement in this sense because users will need to wait for migrations to be finished and this duration should be within acceptable time frames.
As already indicated in Section 4.4.1, a lot of overhead can be reduced if base images are distributed to the target hosts beforehand. When a migration shall take place this data doesn't need to be transferred anymore. Figure 11 shows the disk space demand for base images of different operating systems. 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 a standard builds of OSv, one containing the JRE and the other Redis which is a popular key value database belonging to the family of NoSQL databases. The last one is a custom built Linux Kernel for ARM (refer the Section 20). It becomes clear that there is not much sense in using full operating systems like Ubuntu or Fedora compared to Unikernels or custom Linux Kernels specially built for the application.

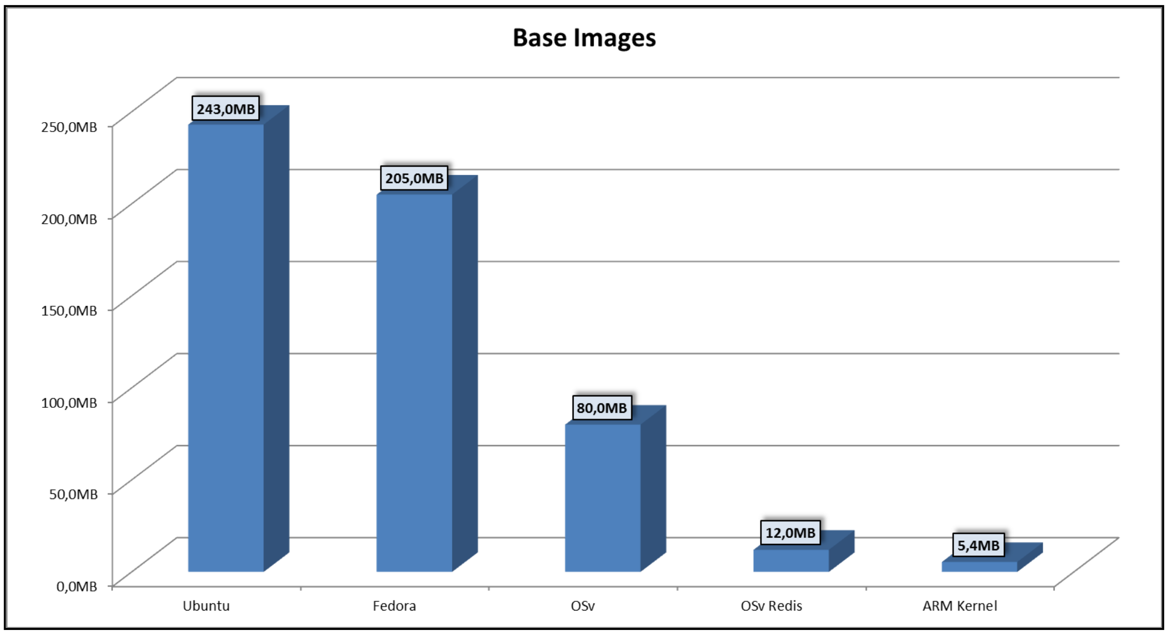


Figure 11 Required disk space for different operating system images

- Transfer VM package to destination host
- Restore instance from VM package at destination host

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: $\Delta m = \Delta s + \Delta r + \Delta t$.
Obviously, larger VM packages in terms of required disk space will result in longer transfer times to the destination host depending on the available bandwidth. This evaluation does not cover measurements for this dynamic behavior but it focuses on the disk space demand for serialized VM packages as this is directly responsible for higher transfer times. As previously described, tiny Kernels also outperform when looking at VM packages as it can be seen in Figure 12. The package size refers to the used disk space for the respective serialized VM package or a zip-compressed version respectively. The content of a VM package is described in section 4.3.2. 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 not be acceptable. Although OSv as a Unikernel comes off well compared to standard Linux distributions, the lowest overhead can be achieved when using a custom built Linux Kernel as guest system. Furthermore, as already mentioned OSv and other Unikernels currently cannot be used on ARM platforms. Because of these two facts the following evaluation will focus only on the custom built ARM Kernel. In other words, an ARM based virtual instance serialized into a VM packages requires between 14MB and 26MB depending on the chosen compression-level. This is the amount of data that finally needs to be transferred to a remote system in order to continue the execution. Regarding only package sizes, compared to disk space demand of high resolution images, these dimensions already seem to be very feasible for certain use cases.

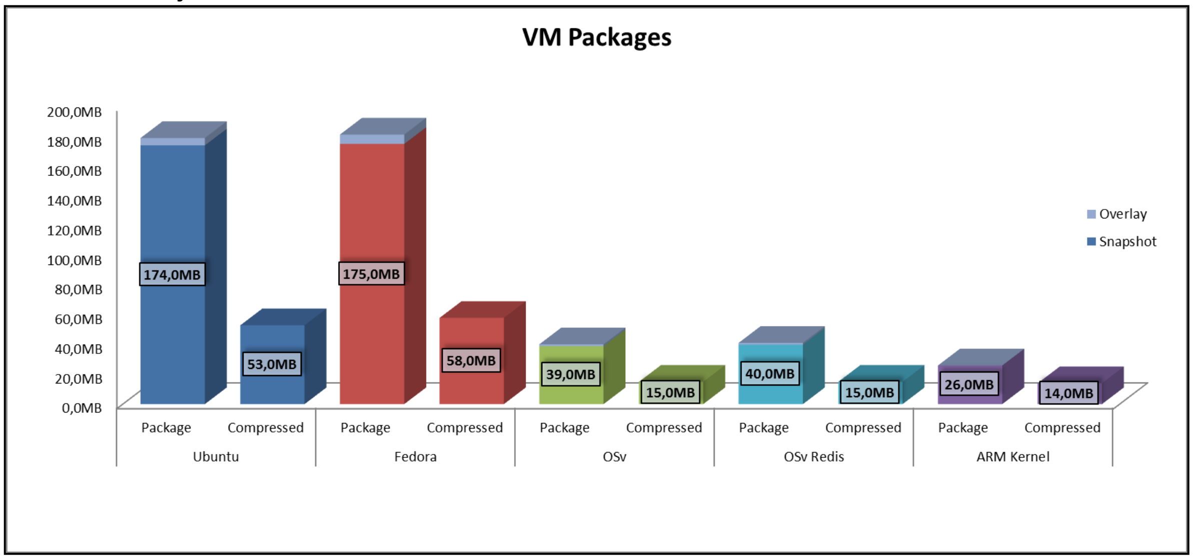


Figure 12 Required disk space for different VM packages

The following evaluation part is meant to get an impression of the runtime overhead introduced by serialization process itself. As previously mentioned, we only focus on the time it needs to save and resume a given VM. Save includes the whole process until the serialized VM 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.

Measurements have been conducted using the ARM Kernel image on four different host systems:

- x86 Notebook (zeus), 2500MHz
- Samsung Chromebook, 1700MHz
- Raspberry PI, 700MHz
- Server at Wellness Telecom Cloud (D-FDI-TRS-APL01), 2530MHz

Figure 12 illustrates the time needed to save and restore the instance measured with the *time* utility. Several observations can be made: More Computing power (CPU clock rate) and higher I/O speeds lead to lower execution times. This is most likely coming from the computational overhead of the zip-compression. This can especially be seen when looking at values from Raspberry PI, which were only included for reference purpose.

For example, transferring the VM as an uncompressed VM package between zeus and chromebook will need approximately $\Delta m = \Delta s + \Delta r + \Delta t = 1{,}7s + 2{,}8s + \Delta t = 4{,}5s + \Delta t$. Given a 100Mbit/s connection, the transfer of 26MB will take about 3 seconds but is generally hard to predict due to dynamic and unpredictable traffic within the network. However, it can be concluded that migration duration with about 10 seconds in this scenario is acceptable.

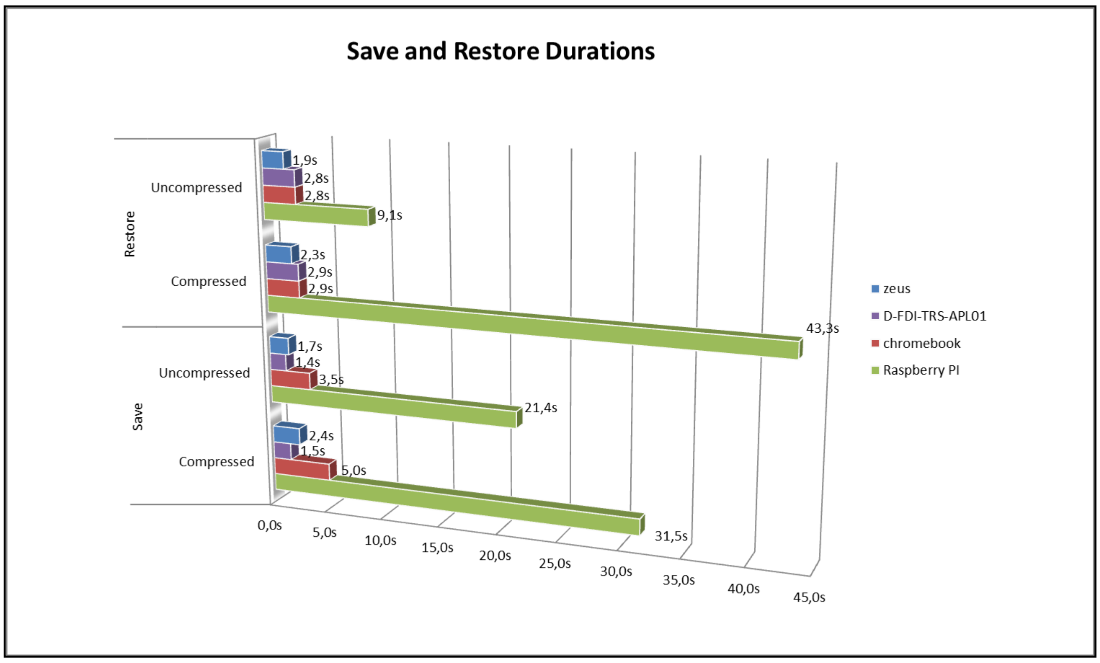


Figure 13 Save and restore durations for different hosts

6 CONCLUSION

This document describes the prototype for emulation and VM serialization. It made 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. Other tresccad instances at remote locations can then unpack and continue the execution of respective VMs.

The evaluation shows that overhead introduced by full virtual machines can be reduced by using QCOW base images and small and optimized guest operating systems. These lightweight systems are a promising direction within the TRESCCA project and can also serve as systems for evaluation scenarios. The achievable reduction of overhead in terms of package size is quite reasonable. Still 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.

7 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

[Katzmarski2014] Mobile Agents based on Virtual Machines to Protect Sensitive Information, Bernhard Katzmarski, Gunnar Schomaker and Wolfgang Nebel (2014), in: Cyber Security and Privacy, Third Cyber Security and Privacy EU Forum, CSP Forum 2014, Athen, Greece, pages 97-106

[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