Secure Domain Transition of Calvin Actors

José María Roldán Gil
jose_maria.roldan_gil.551@student.lu.se

Department of Electrical and Information Technology
Lund University

Advisor: Ben Smeets
ben.smeets@eit.lth.se

June 13, 2016

Printed in Sweden
E-huset, Lund, 2016

Abstract

Calvin, an Ericsson application framework for Internet of Things (IoT), simplifies the development and deployment of applications for IoT systems where many heterogeneous devices are deployed and connected through the Internet. The application engineer just needs to design and codify the application applying Calvin syntax. The Calvin environment is formed by runtimes and applications. The runtime provides the engine to execute applications abstracting the device to the application itself. These applications follow the actor pattern such that actors and the connections among them complete an application design. When an application is deployed in a runtime, the actors are instantiated and subsequently they can be migrated to another runtime that will host the actors and give them its resources.

Calvin is still being developed and several parts are lacking or are immature. One of these issues has been analysed in the case that many application domains form the environment. Currently the assumption is, from a given domain point of view, that the runtimes, applications and users are trusted. However, this simple view changes when one starts to consider interactions with others domains which are expected to be untrusted. As long as actor migrations are possible, domain crossing is feasible and actors can move among different domain runtimes. This exposes the domain to risks and non authorized access to resources of actors working on behalf of untrusted users.

This thesis considers a solution and its implementation in Calvin code that tries to enhance security and reduce risks with respect to domain crossing. Roughly, by identifying every actor in the domain and applying a policy, we realize a managed access to the domain resources. In addition, a translation policy is introduced that will allow identities from another domain to be translated into identities in a receiving domain. Hence all actors have identities that are valid in the domain's namespace. The translated identities are still considered untrusted, but their actions are limited in the domain due to the policy applied. The translation is stateless and involves only the target receiver domain. We also discuss the limitations of this approach and discuss ways to extend this work.

Popular Science Summary

The Internet of Things technology (IoT) will become a reality in the following years. IoT refers to the connection of many devices via the Internet. For instance, a lamp, a fridge, a temperature sensor, a car, etc. Applications such as alarm reception because the fridge is open, or traffic control thanks to car sensors will be usual. What is characteristic here is that devices will be heterogeneous: different type of hardware and software characterize them. This makes application development a tricky task. Fortunately, application frameworks are available and they abstract the device and other issues for the application developer, i.e.,the framework hides the complexity so the developer just can focus on designing the application functionality. Calvin, developed by Ericsson, is one of such frameworks. Calvin manages connections, operative systems, resources, etc. such that an application can be executed.

Applications in Calvin are formed by actors, that are small programs that can be connected to form the whole application function. For instance, one actor can be a calculator and another a display, such that the result of the calculator can be printed in the display actor. The applications (and thereby actors) are executed on runtimes, a software installed in the device, which, as said, abstracts the device to the application and executes it. Actors can be migrated and executed in other runtimes beyond the first runtime on which the actor was created. Hence the device that executes the actor may change and the actor is not fixed to a specific device or runtime. This is another reason why the framework is important. The conditions change but the application needs to be executed in the same way irrespective of the actual device or runtime. Of course, all what is said about migration supposes that the rutime one migrates to is able to host the actor.

It is expected that different domains can be established. A domain will be a defined group of runtimes, users, and a namespace for identities. For instance, a domain could be a company which has some hardware and some employees running applications. As mentioned before, it is assumed that these elements are trusted within the domain and others domain are untrusted since one can not control them. Foreign domains could have malicious users, malicious software, etc. Hence, migrations out of or in to the domain are risky operations. Currently, Calvin lacks any useful countermeasure beyond disabling migration among different domains. This thesis has accomplished a solution to handle domain crossing and enhance the security. All users in a domain will have an identity and actors executed by these users will handle the respective identifier. A policy will authorized to use the resources requested by the actor. But the problem of migrations is still there. If an actor identified from other domain arrives to the domain, the identifier is not valid and a translation will be performed to adapt it to the new domain. One can imagine that each domain speaks a different language, or gives names to its users independently of each other. Hence it may well be that different domains cannot "understand" each other in the sense that names used in one domain are either not understood or could actually map to other users. The use of a translation table will give an incoming actor a new identifier and through a correct combination with the runtime policy the migrated actor will not be deployed if it attempts to use resources that are not possible for it.

The work performed goes from the study of Calvin and the study of the domain crossing problem, to the design of a solution and the implementation in Calvin. Several discussions related to the solution conclude this work.

Acknowledgements

It has been a really nice time here in Lund, where, apart from other things, I have carried out this Master Thesis. First, I want to thank my professor and thesis supervisor Ben Smeets for his guidance and help, and also some of his colleagues in Ericsson for their opinions about the work. Finally, I thank my family and people from Spain, of which I have been separate and with I could not enjoy this year, although I have made many new friends here which deserve the thanks and some of them have shared with me the study time every week.

Table of Contents

List of Figures

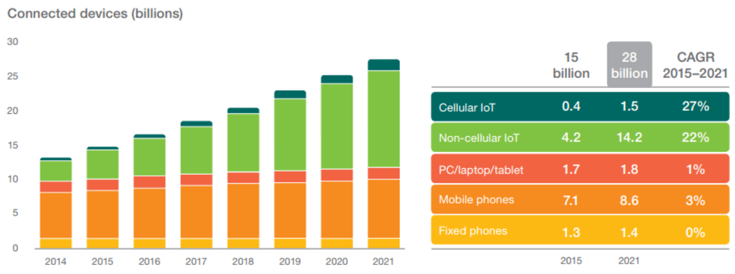
List of Tables

Chapter 1

Introduction

1.1 Background: Internet Of Things

Companies like Cisco and Ericsson have published estimates that by 2020 there will be over 25 billion connected devices, e.g., [1], [2]. Although these estimates are likely optimistic as we are only four years from 2020 we see a steady increase not only for traditional devices but also a real growth for newer type of devices such as machine-to-machine (M2M) products. These connected devices make up for and are seen as the forerunner of the devices that form what is called the Internet of Things (IoT). The notion of IoT is attributed to Kevin Ashton [3] who mentioned it in 1999. As an area IoT has really developed during the last years and is becoming a field of new technologies, products, and services. As is described in [4], ongoing developments and trends like the deployment of Cloud technology, mobile networks driving ubiquitous connectivity, the smart phones as a powerful tool for managing our things, and the Internet as an accessibility enabler have been supporting IoT's strong development.



	15 billion	28 billion	CAGR 2015–2021
Cellular IoT	0.4	1.5	27%
Non-cellular IoT	4.2	14.2	22%
PC/laptop/tablet	1.7	1.8	1%
Mobile phones	7.1	8.6	3%
Fixed phones	1.3	1.4	0%
	2015	2021	

Figure 1.1: Connected devices predictions. Source: Ericsson Mobility Report June 2016. Compound Annual Growth Rate (CAGR) indicates the mean anual growth rate of each device type.

Some devices like smart phones are not considered to be IoT devices. What is IoT then and which devices are IoT devices? There is no clear answer to this question. One can find many definitions of IoT. In [5] one has been to gather some of the definitions in the the absence of a single definition. For instance:

- "A network of items -each embedded with sensors- which are connected to the Internet", by IEEE.
- "Machine-to-machine communications (M2M) is the communication between two or more entities that not necessarily need any direct human intervention. M2M services intend to automate decision and communication processes", by ETSI (note that M2M is another term used to refer to device communications).
- "A global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies", ITU-T study group 13.
- "The basic idea is that IoT will connect objects around us to provide seamless communication and contextual services provided by them. Development of RFID tags, sensors, actuators, mobile phones make it possible to materialize IoT which interact an co-operate each other to make the service better and accessible any time, from anywhere", by IETF.

Thinking about what IoT can bring us, we see that IoT includes a new communication paradigm. While for many years the aim of communication technologies have been to connect humans to humans, or even humans to machines, nowadays we can talk about machine to machine communications where humans do not interact, as is expressed in [6].

The data and information that is created by and around the IoT devices is a key factor for the interest in the IoT technology. IoT devices can be viewed as a link between the physical and the virtual world. Data gathering and data analysis allow the creation of information that not only is interesting for its immediate purpose, e.g. temperature sensors to control the heating in a building, but through processing and correlation with other data one can create information that can be utilized to act more intelligent, e.g. detecting malfunctioning in the heating system or energy planning. Through collecting and interacting with heterogeneous devices we can create novel services and ways to control our environment. Huge quantities of data will be created, processed and shared to be offered as service or function that packages the data and ability to interact with devices in a smart manner. Here engineers will find work to deliver the applications and algorithms that can get the most out of the data. Since the application area of IoT technology is so diverse the application scope for the engineering skills is quite broad too and there will be ample opportunities for specializing solutions for specific use cases and also for reuse of solutions. A few important applications scenarios have been described in [7], for instance sensor networks, smart grid, urban planning, optimization, time saving, navigation...

According to [8], IoT can be classified in the following categories: Service, Computation, Communication and Sensors. Many challenges have been identified in each category. For instance, easy deployment of sensors and low power consumption in Sensors; connection and management of growing number of devices, high volume of data management or security related to Communication; real time, reliable computation and distributed computation (server and devices) to reduce the load (Computation); standardization and more independent machines in Service. Security challenges and power management challenges are studied more in depth in [9]. In addition, [10] or [11] focus only on security challenges. Finally, in [12], the authors discuss some requirements that IoT should fulfil. Apart from these requirements of security and energy efficiency, scalability, quality of service, mobility and heterogeneity are discussed. Since IoT devices are geographically spread and networked and become often useful only when attached to a processing facility, the use of networks and that of computing will increase as IoT devices will be deployed. In order to keep up with the growth in the deployment of devices, the supporting technologies need to be scalable. Furthermore, the data could be treated through many heterogeneous types of devices that request different levels of quality of service as well.

The IoT area has attracted a lot of interest leading not only to commercial products but also leading to or inspiring research and reflections on how IoT will affect our lives. Since we are still at the beginning of the IoT area it is useful to reflect on future directions of IoT. This is, for example, done in [13] which enumerates some questions about the future focusing on the use of Big Data technology and its impact on individuals, organizations, industry and society. Remember that the collective of IoT devices will generate a huge data stream where Big Data technologies can be utilized to process it. Questions on privacy and data ownership, that is who is the legal owner of the data, and how we can secure that ownership can be maintained are important questions. Furthermore, it is interesting to understand how other promising technologies such as 5G networks can cooperate with IoT, this is for example, studied in [14].

1.2 Background: Calvin

The development and deployment of applications for IoT will be a substantial cost factor when creating products and services based on IoT devices. Beside the legal and social aspects there are many requirements to be met and use cases may pose specific challenges to the developer. Today the task to build an IoT based service would be a tricky assignment due to the expected requirements e.g. of mobility, heterogeneity or security. We will in this study focus on one specific aspect of such an assignment which is the technical development, deployment, and operation of IoT applications using a specific application framework.

There exists software like frameworks or middlewares which offer a basis to develop applications (apps) and which simplify greatly the development task. One novel framework for apps that also includes a concept for deploying is the Calvin [15] framework from Ericsson. Calvin is defined as "an application environment that lets things talk to things, including a development framework for application developers and a runtime environment for handling the running application". It can be said that Calvin takes care of many aspects of steps needed to field an app and just lets the developer to build its application using the provided syntax. It provides interoperability and scalability. The current implementation is written in Python and can be downloaded from [16].

In Calvin, the applications follow the Actor Pattern and one uses Dataflow Oriented Programming. The Actor model has been around since the middle eighties [17] and has been used in other others areas. In Calvin an application will be a group of actors and their inputs and outputs that can be connected so they are ready to share the information generated. Runtimes are the engines of the system. An application will be deployed on a runtime and consequently the actors described by the application will be deployed as well in the runtime. One of the outstanding points of Calvin is the migration of actors. An application, and therefore its actors, are not tied to a runtime. They can be hosted by other runtimes that fulfil the requirements the actor/application has on the runtime. For example, the actor requires to have access to a temperature sensor. Because actors can move around among several runtimes one can consider the set of runtimes that can be used as a Cloud scheme [18], since the hardware is decoupled from the software. Figure 1.2 shows an example. There are three runtimes deployed and accessible by Internet. Two of them have installed a temperature sensor, meanwhile the third one has an alarm light. An application has been deployed, consisting of two actors: a sensor actor, which takes samples from the temperature sensor; and an alarm actor, which checks the value of the sample and turns on the alarm light in case it is above a given threshold. The sensor actor can be migrated to the runtime3 to check that temperature as well.

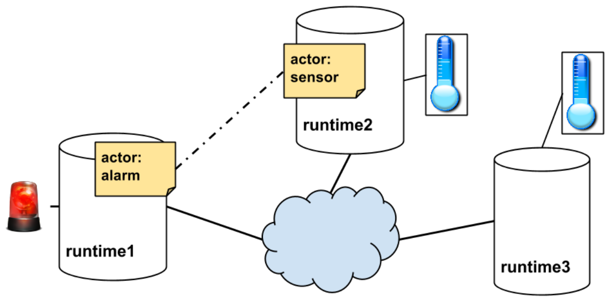


Figure 1.2: Calvin use case example.

Calvin is still under development, mainly driven from Ericsson Research in Lund. Besides the fact that many planned features are still in line to be implemented and that functional issues have been identified it is in the security field where Calvin needs many improvements. Although there are some rudimentary security features implemented Calvin is lacking a comprehensive security framework that covers the intended use space. Recently Calvin has been augmented with concepts around identities and domains which allow better access control and trustworthy communication between applications or actors.

In this context, we are interested in the case of an environment composed of several domains. Users being member of a domain trust the actors and runtimes of the domain they belong to. But what happens when a deployed actor is migrated outside its starting domain? Or on the other hand, another domain's actor is received in my domain? These questions, at its outset simple, will as it turns out and described in this report lead to many security aspects and some of them are hard to find good solutions for. This explains also, in part, why the current Calvin framework has no solution for such domain transitions.

1.2.1 Other Application Frameworks for IoT

Calvin will be the framework for this study. What other frameworks exist that could be of interest? Below we give a short list of other available software:

- NoFlo [19]. Follows as well the Dataflow Oriented Programming pattern. The application architecture is like Calvin, changing the nomenclature with nodes instead of actors and edges for the connections. The language used is Javascript. As one can see, it is a similar software compared to Calvin.

- Orleans [20], a Microsoft open project. The Actor Pattern is key here again. With this software applications are easy to be established and abstract the developer from some common issues in distributed systems. Moreover there is a runtime designed to achieve high performance, reliability and scalability.

- Netbeast [21]. This software allows to easily develop and deploy IoT applications, written in Javascript. The app developer is abstracted from the communications issues and heterogeneous hardware. The software is formed by an operating system for embedded devices, a digital dashboard for hosting and running IoT applications and an API to abstracts underlying technology.

1.3 Goals

We already mentioned that the security aspects of the domain crossing by actors is the main topic of this project. Specifically our goal is to:

- Identify the problem of domain crossing when a migration is performed, and provide a security solution to that issue. The actual security architecture of Calvin is simple and there is no solution to handle the security issues that appear when the crossing is done. A security analysis of different approaches to solve this must be carried out with respect to the achievable security and the required complexity in the actors and the runtime environment. As a result of this objective, a solution should be proposed.

- Implementation of a solution: the project should study the impact on actors and their runtime by a reference implementation, implementing a solution that covers the problems obtained above.

1.4 Outline of the report

- **Chapter 2** describes the Calvin software and the most important features relevant to this project. The chapter concludes with the issues that should be the focus of the work.
- **Chapter 3** presents an examination of solutions to the issues discovered in the previous Chapter.
- **Chapter 4** contains the implementation realized following the concepts of the solution designed in the previous chapter.
- **Chapter 5** discuss some topics and possible future work.
- **Chapter 6** concludes the work realized.
- **Apendix A** summarizes some basic security concepts that are referred in this thesis.
- **Appendix B** presents the modifications implemented in Calvin code.

Chapter 2

Calvin: an application environment for IoT

2.1 Significant concepts about Calvin

As said before, the Calvin framework simplifies the development of IoT applications, abstracting the distributed, heterogeneous, scalable and concurrent system to the engineer. In this section we discuss the main concepts of Calvin that we make use of or that are relevant for our work. We briefly explain the concept of applications, runtime, and attributes as well as policies. Other information about how to install Calvin, configuration, applications, etc. can be found in [22].

2.1.1 The Calvin Application

We start to explain some basics of Calvin applications by using the simple example shown below.

```
# File 1st.calvin
# A small calvin example
source :  std.Counter()
output :  io.Print()

source.integer > output.token
```

The purpose of this program is to provide a counter and display the sequence of its count values. Applications are formed by actors, in this case we have two of them: the `source` and `output` actor, which correspond to `std.Counter` (a counter) and `io.Print` (displays the inputs) type of actors, respectively. The actors are instantiated as a result of the syntax written down in the first two lines, one per each actor. After the required actors have been created one has to specify their connections. Actors can have inports and/or outports. Looking in the actor's documentation, it is revealed that the actor `std.Counter` has an outport `integer` and that `io.Print` has an inport `token`. `Source.integer > output.token` just connects the outport of the counter with the inport of the `io.Print` such that the counter sends the data, called "token" in Calvin (in general, not only the `io.Print` actor uses this nomenclature), to the printer. By looking more closely (even) at this small example, we can conclude the following facts. An actor executes a programmed function that can consume values from the inports and submits results on the outports which can be connected to others actors. We also see that an actor can have a state. In our example the state of the counter. Calvin offers some built-in actors for common operations (the actors used in the example are two such actors that are given by Calvin) and of course new actors can be developed and added to the system (inheriting from the `Actor` class of Calvin). Also it is important to understand that we make sometimes a distinction between the actor and the actor instance. The same actor can be instantiated multiple times.

Furthermore, the state of an actor is the variable part of it and when migrating an actor the state has to be migrated. Inputs can trigger new actions in the actor's program which in turn may change the state. The state will be important for actor migrations when they move to another execution environment to run there. In fact in Calvin, the actor itself program is not migrated, instead data that is related to the actor instance, its state and attributes, are migrated. We come back to the migration later. Furthermore, attributes can be attached to the actor instance, describing properties and requirements in terms of resources to be able to function correctly. However, they are not part of the application design and will be explained later. Figure 2.1 shows an actor description.

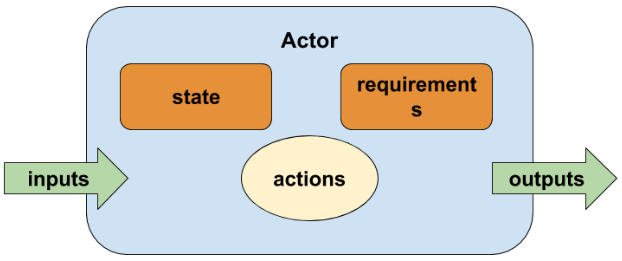


Figure 2.1: Actor description.

Instantiating and connecting different actors is the way to complete the development of an application. But how are applications deployed? A support environment is needed that interprets the code. In Calvin, it is called a runtime, and it is another important component of Calvin.

2.1.2 The Runtime

After an application has been developed we want to run it. To make this happen the application needs an execution environment, and Calvin calls this as a runtime. The runtime is the engine that manages and executes the deployed actors and provides services related to things like communication or access to resources, e.g. a button, a temperature sensor, a microfon, or video camera. The runtimes

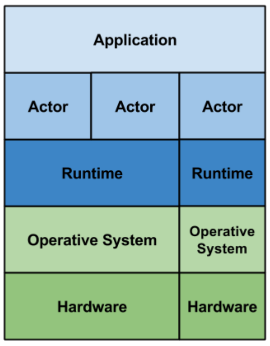


Figure 2.2: Layer model for Calvin.

can form a network of runtimes and therefore facilitate for the possible migration of actors. A migration of an actor is typically the process where an actor instance is relocated from a source to a target runtime (sort of Cloud pattern where code is not linked to a fixed platform and hardware can change during its lifecycle). As shown in Figure 2.2, the runtime is installed above the operative system of a platform. Please note that an application can be executed on one runtime, or that its actors execute on different runtimes.

In [23] it is explained that a runtime has a platform dependent sublayer and a platform independent one. The platform dependent sublayer handles the communication among runtimes and uses a plug-in for the transport protocol. In addition it abstracts other platform functionalities (for instance I/O). The platform independent sublayer provides "the interface towards the actors" and provide some basic services. Runtimes can be of very different nature. Some runtimes may be very limited, e.g. supporting only very simple resources like a button or temperature sensor, while other runtimes may be very powerful with ample memory and computational resources. Since runtimes have different resources it is may well be the case that an actor requires resources which a runtime does not provide. Beside the fact that this means that we cannot instantiate the actor on such a runtime it indicates the need to have a system to easily declare what a runtime supports. How this is addressed in Calvin is the subject of the next section.

Finally, let us return to the initial question in this section. Our ready developed application is deployed on a starting runtime. The actors described by the application will be instantiated there, and maybe they will be migrated to other runtime if this is required or better in some sense, e.g. using some rules the runtime(s) are equipped with.

2.1.3 Attributes and requirements

An important feature of Calvin is the possibility (it is an optional feature) to attach requirements to actors and deploy runtimes with attributes that provide information about the runtime, e.g. its capabilities. The attributes or requirements are provided by the user when deploying a runtime or an application, issuing the commands `csruntime` or `cscontrol`, respectively.

Definition: We say that the requirements of an actor meet the runtime attributes when the actor's attributes are such that the runtime has the required resources or attributes, e.g. location, a camera, etc.

Please note that this definition means that when the requirements are met it does not imply that the actor is allowed by the runtime to execute. This aspect, the permission to use the runtime and its resources, is a separate issue. Parallel with this thesis work there was a thesis work which had the access control as its subject, we refer the interested reader to the report of that work [24]. Summarizing an actor is able to be instantiated or to migrate to another runtime if its requirements are met by the target runtime and that the target runtime grants the use of the requested resources.

If the attribute and requirements scheme is used, a runtime should be started with a given set of attributes, specified in a JSON format. The current specification supports three data types that are available for the attributes, the so called `public`, `private` and `indexed_public`. The types `public` and `private` are still under development, so we will only study the `indexed_public`. This attribute type allows one to tag the runtime with the `owner`, `address` and `node_name`. In addition, it is possible for the owner of the runtime to write his own attributes (`user_extra` field). The data is indexed, i.e., it will have a key field for searching the attribute item and the respective value. For instance, a valid attribute declaration could be

```
{"indexed_public":
{"owner":  {"organization":  "com.ericsson"},
     "address": {"country": "SE"},
     "node_name": {"organization": "com.ericsson",
                    "name": "testNode"}
    }
}
```

Those attributes that are attached to a runtime at the moment it is launched are important for an application. As mentioned before, the application may have some deployment requirements that must be matched by the runtime attributes. Each actor that is part of an application identifies which requirements it needs. If the match between the attributes provided by the runtime and the actor's requirements is successful, the actor can be deployed on the runtime. The requirements matching allow automatic migrations, since an application will be started in a runtime, and then the actors will be moved to the node where the requirements are met.

An example of a requirement declaration could be like below. It shows a requirement for a concrete node, called `testNode2`

```
{
"requirements":  {
"actor1":  [{"op":  "node_attr_match",
      "kwargs":  {"index":  ["node_name", {"organization":  "org.testexample",
"name":  "testNode2"}]},
       "type":  "+" }]}
}
```

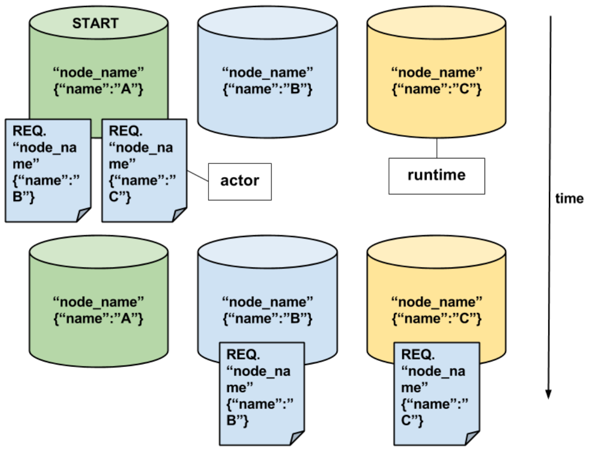


Figure 2.3: Example of attributes and requirements and migration.

Consider Figure 2.3. It is an example where both actors fulfil the requirements so they are migrated. In case one of them does not fulfil it, the actor would stay in the starting runtime until there is a match.

The approach taken by the Calvin designers is similar to that of Attribute Based Access Control (ABAC) policy [25], although the main different here is that while in ABAC the actors have attributes and the runtimes apply the policy to allow them or not, in Calvin these roles are the opposite.

2.1.4 Linking runtimes

Runtimes are expected to build a network to communicate. By network it is understood that a runtime establishes individual links with other peer runtimes, the so called runtime links. The communication makes it possible actors to communicate but supports also the actor migration. In the case we would have migrated an actor α that was connected to another actor β, the connection remains but the communication has now to be moved to a runtime link on the target runtime. In Figure 2.4, one can see the arrangement in a layer model of the communication between two actors in separate runtimes. On the right side, it is shown the logical link between two actors and the runtime link. It is important to understand that the logical actor link is implemented on top of the link(s) of the runtime.

In the Calvin implementation we have two classes that play an important role in the communication: `CalvinLink` and `CalvinNetwork`. While `CalvinLink` man-

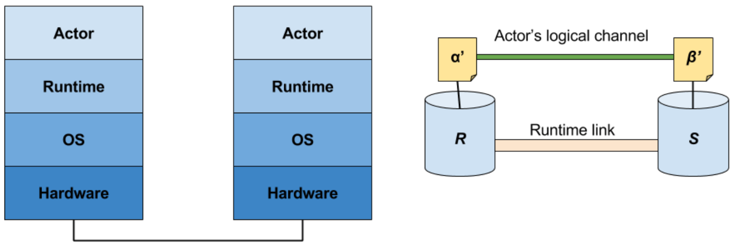


Figure 2.4: Runtimes and actors connections.

ages one specific link between two runtimes, `CalvinNetwork` manages all established links from a runtime, register the transport plugins and start the listeners of incoming join request (from other runtimes).

The protocol that supports the token shipment is handled by a transport plug-in. If we seek what is the default plug-in implemented in Calvin, the answer is Twisted. Twisted implements the network protocol for TCP servers that send messages in an asynchronous way. It is expected that the communication between two runtimes, both play the TCP server role and client role. In reality there are two channels, one for each communication direction. The Twisted documentation can be checked in [26]. The transport uses the so-called calvinip address. This address (IP and port) is specified in the runtime deployment and is different from the control port that manages the node. The connection made is just a TCP link. Currently TLS is not supported yet.

2.1.5 The migration

We have already discussed migration to some extend but we need to look into it in some more detail as migration is an important ingredient in the problem we address in this thesis work. Therefore this section contains a more detailed analysis of the actor's migration process. Remember that migrations is one of the features for which Calvin is considered a Cloud system: the software can be executed on different hardware during its execution life cycle when actors are migrated from a source to a target runtime.

The migrations can be manually ordered by the user or the runtime does it itself, e.g. when the requirements are fulfilled. Although Calvin allows migration to happen in a smart way that would optimize some desired parameters such response time it has still not the control mechanisms and heuristics to have smart migration. In our examples we migrate actors manually using command and web interfaces.

Thus, the migration process itself consists in short by the destruction of the actor instance in the source runtime and the creation of a new one in the target runtime. The code is not sent anywhere, and is located in the runtime's actor store (explained later). What is sent over the network is the state, type and ongoing connections of the actor. All this data is sent by first serializing (marshalling) all items and then sending the serialized version. The target runtime will deserialize the data and with this information, the target runtime can create the new instance and recreate the running actor.

We can say that three requirements need to be fulfilled to perform a migration:

1. First of all, the source runtime and target runtime must have established a communication channel, otherwise communication is not possible, i.e. they have to know how to reach each other previously.
2. Then, if the actor has requirements, the target runtime must match all of them to become a host runtime.
3. The target runtime must have the implementation of the actor type being requested by the source runtime. It should be pointed out that both runtimes do not need to have exactly the same implementation as long as the interfaces of the actor type are the same (for instance, the state received can be used to instantiate the actor).

2.1.6 The Security Module

A security module, *security.py*, was included to Calvin while this project was in progress. The security module will simplify the final Calvin implementation since it encapsulates new features to be implemented downstream. Basically this module is responsible for the authentication and authorization mechanisms. The Calvin configuration file indicates which security procedures will be used (they are optional, maybe they are not activated) and some parameters, like the users allowed or the path for the policy files.

With respect to the authentication procedure, a user needs to provide its credential(s) for deploying an application. By credentials it is understood a user identifier and a password. Actually it is possible to use a local authentication procedure or a Radius server. We have tested the local procedure, which consist in a list of feasible actors and their passwords that are allowed in the system. If the user introduces the correct credentials pair, the authentication succeeds. The credentials are stored in the actor instances so it can be known on behalf of what user the actor works. Moreover they are part of the actor state and sent in the migration (will be important for the project implementation).

After the authentication, the authorization policies, in case they are available, will be checked through an authorization request. In this thesis when referring to policies, it is understood a group of rules that determine the decision resulting of a process that can have different outcomes. In this case, the authorization policy gives or not the deployment authorization to a user according to the rules established in the policy. As it happened with authentication, in authorization the procedure can be local or external (where another runtime can act as authorization server). Again we have tested the local procedure. The policies can be applied to application and actors. They will allow actors deployment in the case the authorization request succeeds. A policy is formed by three important fields: `rule_combining`, `target` and `rules`. A policy example is shown below.

```
{
```

```
"id": "policy1",
"description": "Policy for actor signers. All possible signers (or
unsigned) permitted for user1",
"rule_combining": "permit_overrides",
"target": {
      "subject": {
            "user":["user1@test",".*@test"],
            "actor_signer": ".*"},
      "action": {
            "requires": ["runtime", "calvinsys.*"]},
      "resource": {
            "owner.organization": "com.ericsson"}
},
"rules": [
      {
      "id": "policy1_rule1",
      "description": "Permit if policy target matches",
      "effect": "permit"}
      ]
}
```

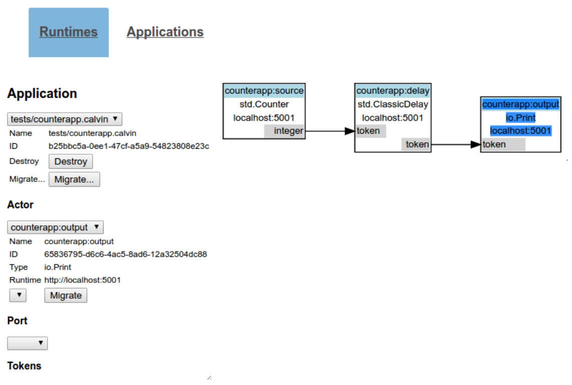
The `rule_combining` value sets whether the rules can be combined to allow or deny an authorization decision if just one of them is accomplished. This happens when many policies stored in the policy path indicated in the configuration file match the target of the authorization request. The target field describes to whom the policy will be applied. If the target is empty, it signify the policy is applied to everyone. The target can be formed by the subject (users and actor or application signers), the resource attributes, i.e., the runtime attributes and, just for actors, the action field which indicates the resources required by the actor. For instance, `runtime` value means runtime execution access, or can `calvinsys` to request a resource.

When a policy matches with the target, there is a rule part that gives the result or effect: deny or permit. Many rules can be implemented and make policies more or less complex regarding the system needs. The rules can have conditions to limit the effect of the rule and if they are accomplished the rule will take effect. The conditions can be formed with attributes and functions (for example, `equal, less_than_or_equal, greater_than_or_equal`...). Many conditions can be combined as well, regarding the requirements and complexity that one wants to achieve.

Furthermore, the file *certificate.py* complements the previously explained security module *security.py*. Some of the functions available in this module are generation and signature of certificates, file signature, certificate verification or cryptography key procurement. These functions are related to Public Key Infrastructure [27] and this is important for the future of Calvin. In Appendix A.2 we give a brief description of PKIs.

Finally, we comment that the actor and applications can be signed with `csmanage` tool provided by Calvin, it is not needed to use external tools. Signatures can be used in policies as one of the conditions.

2.1.7 Other components of Calvin



Application

tests/counterapp.calvin ▼

Name tests/counterapp.calvin
ID b25bbc5a-0ee1-47cf-a5a9-54823808e23c
Destroy Destroy
Migrate... Migrate...

Actor

counterapp:output ▼

Name counterapp:output
ID 65836795-d6c6-4ac5-8ad6-12a32504dc88
Type io.Print
Runtime http://localhost:5001
▼ Migrate

Port

▼

Tokens

Figure 2.5: Calvin's web interface.

In this section a few other components like the web interface, the actor store and the storage will be described. This just to know a little bit more about Calvin and because we sometimes mention them in the descriptions. They have no special security meaning in our work but, like storage, may have their own security issues.

A web interface is available to manage runtimes after they are deployed. A Calvin server must be executed and it is ready to use the web through a web browser, as shown in Figure 2.5. The interface is built around the HTTP protocol. Basically with this interface a user can deploy applications on an specific runtime, destroy it, migrate an actor to the runtimes available in the network or get the port's FIFO.

Another component is the storage. The runtimes can search in the storage to obtain information about other runtimes, actors, ports, etc. By default it is implemented using a Distributed Hash Table plug-in called Kademlia. The storage information is obtained by an API. There is a recent master thesis where the interested read can find out more details about how it works, [28].

Finally we will mention what is called in Calvin the Actor Store. In the Actor Store one can find the actors available in a node (many runtimes can use the same actor store) and is where the code is stored. In the current Calvin implementation, two classes are relevant in this context: the `ActorStore` and `GlobalStore`. However there is still not too much information about these two classes. It is reasonable to think that the first one is the local store associated with a runtime and the other is a distributed global actor store shared by many runtimes. But as of today the last one is not used.

2.2 The concept of domain

When reasoning about actors and their deployment one arrives fairly quickly to a point that one realizes that actors perform tasks on behalf of someone. But this someone is in most cases not a single entity. Now different entities may trust each other or may not. The same type of reasoning can be applied to runtimes and the links through which runtimes communicate. Using the trust relation one basically can partition the runtimes in groups and we call these groups for domains.

A domain can be characterized as a group of users, runtimes and a namespace. The domain is closed in the sense that its boundaries, that is who belongs to the domain and who not, is well-defined. In practice we also assume that there exists a proper authority by which we can verify if a runtime is indeed a member of the domain or not. Towards that end, the authority must have a means to perform the verification. In Calvin one goes a step further and not only checks if a runtime belongs to the domain but one assigns a unique name (identifier) taken from the namespace to each runtime. The implementation of all this is based on public-key cryptography and the namespace is realized by having a Certificate Authority (CA), who can issue certificates for the public keys. We see that the PKI of the CA and the domain are in essence the same.

A certificate provides information about the subject it has been issued to, e.g. the domain it belongs to, and binds the subject identifier to a public key. Anyone can with knowledge of the CA root public key verify a certificate and can use the information in the certificate to identify, for example, a runtime. The information in the certificate is interpreted on the basis of the namespace that has been defined. Hence, another domain with possibly a different namespace, and maybe another Certificate Authority, cannot use certificates from another domain. As a consequence any inference by using certificates such as identifications of runtimes performed in a domain, is just valid in the domain the certificates are issued for.

The runtimes of a domain and the actors running on them are trusted, e.g. the users that receive an identity are known or the runtimes have correctly established their identity. Users are supposed to execute applications and use the resources in the best way satisfying the policies of that domain. On the other hand, the components of another domain are considered untrusted, mainly because the initiating users are not known and not identifiable. The runtimes' software will not be verified nor will the applications. As long as runtimes are reachable through a network, migrations are available and domain crossing migrations can be realized. This implies risk on both sides. Actors would leave their trusted environments (the domain) to be deployed in a untrusted one.

In the remainder we assume that Calvin is using the aforementioned domain concept using a PKI for each domain. However, the current Calvin implementation is not fully supporting the concept.

When an actors migrates from one domain to another one the receiving domain cannot trust an incoming actor right away. This observation lies at the heart of the problem of this thesis work, the domain crossing of actors.

2.2.1 A first take on the domain crossing problem

After discovering the problem of domain crossing in the last section, it would be interesting to get an initial understanding of the problem.

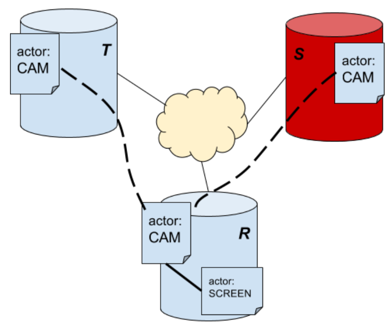


Figure 2.6: Example of interdomain migration (red) or interdomain (blue).

A runtime *R* could receive and host actors being migrated from a runtime *S* in another domain. Such actors, working on behalf of unknown users, will use the available resources in *R*, that could be memory or computation or even cameras or sensors. Moreover these actors may gather information in *R* and communicate to other actors hosted in *S*. *R* does not know the actor's functions in *S*, and this added to the fact that the actor's initiators are untrusted, implies security risks to the domain where *R* belongs to.

The users that migrate actors to *R* are unknown because their certificate is not available in *R*'s domain, where another Certificate Authority is governing.

If *R* migrates actors to *S*, it should be aware that, since *S* is an untrusted runtime, which could establish a compromised communication and will deploy actors, which could have different features than expected (for instance, the operativing system is not updated and have security bugs).

This example explains why from a security point of view, the Calvin domain concept is immature as it does not provide a scheme to handle domain crossing. As we see later there are many problems associated with the domain crossing and it turns out to be a real tricky step to handle when considering all its aspects. We come back to this later.

Chapter 3

Towards a solution

To address the security issues that were identified when an actor crosses from one domain to another we have to develop a methodology to describe Calvin in a more precise manner. Part of this section is devoted to the development of such a methodology. It turns out that one important notion that we need is that of an identity for the actors and runtimes. Basically this implies an identification scheme for actors and runtimes such that the actor's behaviour can be managed with the use of policies, and other measures necessary to handle the domain crossing.

3.1 Definitions

First we define a way to denote domains and actors. Consider

$$\mathcal{D} \overset{\triangle}{=} \text{the set of all domains}$$

We will use the first capital letters A, B, C, D, to denote specific domains. Each domain has a number of runtimes and we define

$$\begin{aligned} \mathcal{R}(A) &\overset{\triangle}{=} \text{the set of all runtimes of the domain} A && (3.1) \\ &= \{R_i; R_i \text{ is a runtime of domain } A\} && (3.2) \end{aligned}$$

We adopt the convention that capital letters R, S, and T denote runtimes unless explicitly stated otherwise. Sometimes we want to explicitly indicate that a runtime R belongs to a specific domain. In such a case we write

$$R^D \overset{\triangle}{=} \text{runtime belongs to domain } D.$$

Actors are denoted by Greek lowercase symbols e.g. α, β (denotes actor type) and actor instances are denoted by $\hat{\alpha}$ and $\hat{\beta}$. In case we have two instances of the same actor, α say, we use sub-indices or other attributes to $\hat{\alpha}$ to denote the different instances; e.g. $\hat{\alpha}_1$ and $\hat{\alpha}_2$. It may be necessary to identify the runtime R or the specific domain D where an actor instance is being executed. Then we adopt the notation

$$\hat{\alpha}^R, \text{ and } \hat{\alpha}^D \text{ and } \hat{\alpha}^{R^D}$$

where R^D is a runtime of domain D.

Each runtime has attributes and policies associated with it. The runtime itself has intrinsic resources (e.g. memory, execution capabilities) and can beside these have other resources associated with it, e.g., a temperature sensor, a button, a camera, etc. The policies and attributes characterize the properties of the resources of the runtime and the conditions for use of these resources. The attributes are used to describe which resources or properties of resources apply to a runtime. In Calvin there are a set of predefined attributes, as it was explained before, that can be used but in principle runtimes can add additional attribute types. Note that the attributes are not necessarily static but we adopt here the convention that they are.

Policies express how actors can or may utilize the resources available in a runtime of the domain. We will use small letters u, v, w to denote resources of a runtime and use p, q to denote policies. Hence $p(u)$ denotes the policy of resource u. Together these policies can be called deployment policy since the actor deployment will be allowed or denied after the policy has been applied.

It is convenient to use the notation

$$R \rightarrow_a \stackrel{\triangle}{=} (\ldots, a, b, c, \ldots)$$

to denote the attributes of the runtime. Also, let

$$R \rightarrow_r \stackrel{\triangle}{=} (\emptyset, \ldots, u, v, w, \ldots)$$

denote the resources of the runtime, where $\emptyset$ denotes the runtime itself and

$$R \rightarrow_p \stackrel{\triangle}{=} (\ldots, p(u), p(v), p(w), \ldots)$$

the policies of the runtime R.

A runtime can be connected with other runtimes using links. The collection of links of the runtimes in a domain characterizes the network available in the domain.

$$\begin{aligned} \mathcal{L}(R) &\stackrel{\triangle}{=} \text{the set of all links of the runtime } R && (3.3)\\ &= \{L_i; L_i \text{ is a link of runtime } R\} && (3.4) \end{aligned}$$

Every link is going to be categorized as *interdomain* or *intradomain* by a respective method. The purpose of this measure is to know if that link leaves the domain or not and apply the appropriate translation for incoming actors to guarantee the system integrity in case there is domain crossing.

Now let

$$L_i \leftarrow c$$

denote the assigned of category c to link L_i. Note that we do not indicate which method is used for the assignment, nor do we describe the conditions that influence this assignment.

An actor instance is part of (or is) an application that is being executed. Each actor instance has also requirements associated with it which are a combination of the actor attributes and application attributes that are attached to the actor when it is instantiated. We will not formalize this process and satisfy ourselves with the notation

$$\hat{\alpha} \rightarrow_r \stackrel{\triangle}{=} (\ldots, a, b, c, \ldots)$$

where a, b, c are requirements of the actor instance $\hat{\alpha}$ and ignore the details of the requirements and how they are linked to the actor instance.

An actor instance can exist in a runtime R if and only if

1. The runtime meets the demands of the actor instance (in case attributes and requirements are being used).
2. The actor instance meets the (policy) conditions for using the resources on the runtime.

If one of these conditions is not met the runtime will refuse to create or host the actor instance. Note that $\tilde{\alpha}(\mu)$ may, but not necessarily denotes a running actor. We use it here to denote the actor instance recovered from deserializing a (received) state of the actor instance. It should be remembered that if the actor is created by migration, the conditions expressed in the section 2.1.5 must be satisfied as well so the actor instantiation is accomplished. But the previous stated two conditions are always applicable regardless if there is a migration or not. We use the notation

$$\tilde{\alpha}(\mu) \subset_a R$$

to denote that the actor α with actor state μ *CAN BE HOSTED* by runtime R and let

$$\hat{\alpha} \subset R$$

denote that the actor instance $\hat{\alpha}$ *IS HOSTED* by runtime R.

3.1.1 Namespaces and identities

There is one attribute of an actor instance that deserves special considerations namely the attribute that characterizes for whom the actor instance is providing its actions. We will call this the owner of the actor and the corresponding attribute links the actor instance to an owner identity which in fact is a user in a domain.

Towards this end we have for each domain the set of the identities that are defined

$$\mathcal{I}(D) \stackrel{\triangle}{=} \text{the set of defined identities of domain } D.$$

Here an identity is an abstract notion that allows us to identify members of set. Each member is associated with an identifier taken from a namespace that, for example, is used in the domain D, and for each identifier there is an identification scheme by which a runtime in domain D can assign an identifier value to an actor instance. For example, during the actor instance creation the runtime requires a PIN or password entry from the user that initiates the creation. We see that strictly speaking an identity is a tuple consisting of an identifier and a protocol used for identification. For convenience we often will refer to identities by the identifier.

The identifiers link to users that initiate and have applications running for them. But beside the identifiers for such users there are a number of other identities, let's say "special cases". We introduce the following two identities

$$\Lambda \triangleq \text{the null identifier in a domain with no relation to any user} \quad (3.5)$$

$$G(uest) \triangleq \text{identifier of a foreign user.} \quad (3.6)$$

Λ can be seen as something similar to the notion of `this` in object oriented programming syntax and represents the domain itself.

It is hereafter understood that the identifier attribute of an actor instance is only set through the runtime and that the runtimes in a domain trust the identifier of an actor instance to be correct. However, a runtime or an actor may trigger a runtime to perform the identification scheme to confirm the link between the identifier and ownership of the actor instance inside the domain. Of course, it should not be applied to the Λ and $G(uest)$ identifiers due to the lack of final user in the domain.

Now, we let

$$\hat{\alpha} \rightarrow_I \triangleq \text{identifier of the actor instance}$$

and let

$$\hat{\alpha} \leftarrow_I I$$

denote the operation by which we set the attribute of the instance to the identifier value I.

Finally, since the identifiers indicate the real user the actor is working on behalf of, these identifiers may be used by policies, to be discussed later. By doing so one can see to it that an actor will be given the proper permissions. This is one of the reasons why identifiers are important. The mechanism by which we link users to identifiers can be considered as the gate between the real and the virtual (software) world.

3.2 Defining the domain through the TLS protocol

We mentioned earlier that a domain is characterized by private keys and public-key certificates issued by a CA that is associated with the domain. How the keys and certificates are provisioned is something that we consider to happen out-of-band, that is we rely on procedures we do not specify or develop in this thesis work. As such the PKI provides the namespace, identifiers and identification scheme for runtimes in the domain. Yet, although we can directly use these tools to define and identify domains, we will use an indirect approach that identifies a domain by the link one makes to it.

One observation that we already made is that the transport layer among runtimes need to be improved over the present plain TCP protocol. By harnessing Transport Layer Security [29] features and implementing this protocol, the security of the channel is enhanced and will provide secure communications protecting the data sent and authenticating the two runtimes. Some brief information about TLS is available in A.1.

But we are not only interested in enhancing the security level in communications. If we perform TLS in a mutual authentication mode and let the certificates used in the protocol to be tied to (or the same as) the runtime certificate, we can realize an indirect way to assess the domain one is connecting to. Because the two connecting runtimes have performed the TLS handshake they have a proof that the link is established to a runtime of a given domain. The runtime can be one that belong to the same domain or one that belongs to a foreign domain. Hence, we see that TLS helps to define the domain since the collection of TLS links with certificates of the domain's PKI establishes the contours of the domain, i.e. which runtimes are inside the domain and which are outside and which runtimes are connected to foreign runtimes.

In the reasoning above the runtimes and the TLS protocol for the links use identities that belong to the same domain PKI. Maybe for security reasons we want to use two identification schemes to be used here, one for TLS and one for identifying the runtimes. However, for the sake for simplicity of our exposition we say that the keys and certificates for TLS are also those used for the runtime. Thus through the authentication method of TLS we identify the runtime and the domain it belongs to. In the same way as the actor instances we can thus define:

$$R \rightarrow_I \triangleq \text{identifier of the runtime}$$

and

$$R \leftarrow_I I$$

The authentication provided by TLS is really important. With the proper implementation, the runtimes can know if the link goes to another domain or not, and apply the translation procedure described in the next section. This TLS handshake process can set the value of the link category, $L_i \leftarrow c$, to the expected *interdomain* or *intradomain*, as showed in Figure 3.1.

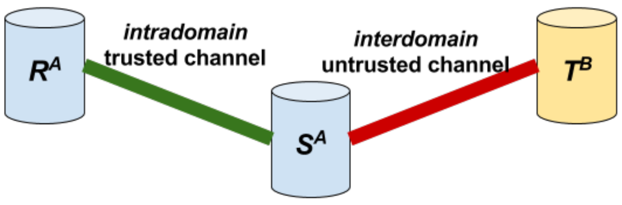


Figure 3.1: Link category example

For accomplishing this in practice, the runtimes need to handle an identity in the form of certificate and associated private key, and we have to specify how the notion of namespace and identifiers map on the specifics of the certificate profile. The certificate profile defines the fields, extensions, and attributes that are used in the given Public Key Infrastructure. In Appendix A.2 we give a suggestion for a possible certificate profile for our purpose. We only mention here that identifiers for runtimes can be the subject names given in the certificate, and that the subject name of the root certificate is the same as the issuer that appears in all the certificates, identifying the domain.

3.3 The translation process

With the definitions provided above, at least the behaviour of an actor instance in the runtime can be limited regarding the proper policies inside the now well defined domains. However we have a problem when dealing with migrations that perform domain crossing, that is when an actor identified by $\mathcal{I}(C)$ arrives in D that uses the namespace $\mathcal{I}(D)$. The core of the problem is that the identifier from domain C will no have meaning in D. In fact, even if the identifier of an arriving actor can syntactically be interpreted in domain D there is no sound argument why the interpretation is correct. It is easy to imagine that in domain C and D we have two different entities that syntactically have the same identifier. Thus any identifier information that an incoming actor carries cannot be trusted and needs special treatment.

A translation process would be thus a natural solution. Through a translation the identifier of the incoming actor from C is translated into an identifier that is valid in the namespace of D. In that way every actor is identified and the policies will manage the resource consumption regarding the identities requesting them. The trust of runtime D in the translation is a result that D trusts, at least to some degree, to connect (i.e. have a TLS link) to C and implicitly trusts the runtimes in C to assign meaningful owner/user identifiers to the actors. In practice such trust can be substantiated by legal agreements or other non-technical arguments for D to have a certain amount of trust the runtimes in C.

Let's considered the instance $\hat{\alpha}$ to be migrated from R^C to S^D, such that

$$L_d \leftarrow (c = interdomain)$$

Let $I = \hat{\alpha} \rightarrow_I$ be the identifier of the actor instance. The runtime S will execute the translation function described below for domain D.

$$\phi_D(I) = \begin{cases} T(I) & \text{if } I \text{ occurs as index in the table } T \\ & \text{and translation policy met,} \\ \text{fails} & \text{otherwise} \end{cases} \tag{3.7}$$

where the translation table $T(I)$ is defined in the target domain D and consists entries of (foreign) identifiers, the defined corresponding identifier in domain D, and an optional policy that describes additional conditions for the translation to take place. The translation table $T(I)$ is, itself, a translation policy.

RULE: a migration fails if the identifier translation in target runtime S^D fails.

We can define now that a domain D must have at least one translation table $T(I)$ to be implemented in $R(D)$ (maybe it can be considered to have the same $T(I)$ in every runtime in the domain D or it can be configured specifically for the runtime)

$$D \rightarrow_t \triangleq (\ldots, T_1(I), T_2(I), \ldots)$$

Translation Table $T(A)$	
Input ID	Output ID
.*@B	guestB@A
.*@C	guestC@A
(any)	guest@A

Table 3.1: Translation Table example

After the new identifier is provided, the runtime will check its policies whether to deploy or not the new instance, now that the identifier is meaningful. Figure 3.2 shows an example.

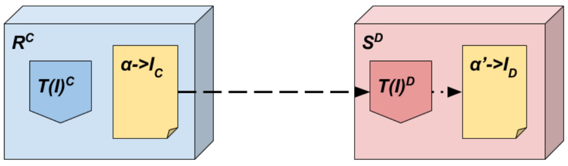


Figure 3.2: Example of a translated migration.

Since all runtimes in D trust each other it does not matter if runtime S did the translation or any other runtime in D. It could be the case that D has one or several runtimes that are specially assigned for carrying out a translation. We regard this as in implementation and deployment consideration which we regard as not being a topic for this thesis work.

We close this section by giving a high level description of an example of how a translation table may look like, Table 3.1. This table is used in domain A to translate identifiers used in domain B and C. The table is created in A after these domains have established a relationship with A. For the rest of domains, they will be granted with a general guest identifier. The deployment policies should be devised to make sense for these translated identifiers.

3.4 Benefits of the solution

We can summary up some benefits that support the solution we designed.

- The identification of actors combined with the use of deployment policies restrict the actor deployment and use of the runtime's capabilities, such that only the granted resources will be available for an user.
- The domain is well defined with the use of TLS for runtime's communication channels. Therefore a domain crossing migration can be detected. In

addition, this communication is more protected with respect to just implementing TCP protocol.

- Foreign identifiers will be valid in a domain after the translation is performed. With a proper translation (fitting the deployment policy in the desired way) the performance of external actors is limited. The translation is supported by a translation table on the target side, hence the source side is not involved if there is no intention to it, regarding how the policy is implemented. We make each domain independent and the migration is stateless since it can be considered a black box whose entry is the old identifier and the output the new one.
- The implementation is feasible and can be merged with Calvin code.

3.5 New issues and considerations

The translation scheme is however not solving all problems that exist around the domain crossing. There are aspects that are not covered and which deserve mentioning.

Once an actor has been migrated from domain C to domain D it is supposed to receive an new identifier. Hence from domain C's perspective the actor instance lost its identity and it is not obvious that when returning to C it could have the original back again, i.e., the identifier of the user that deployed this actor in the first place. Let's consider an example, Figure 3.3, that shows this use case.

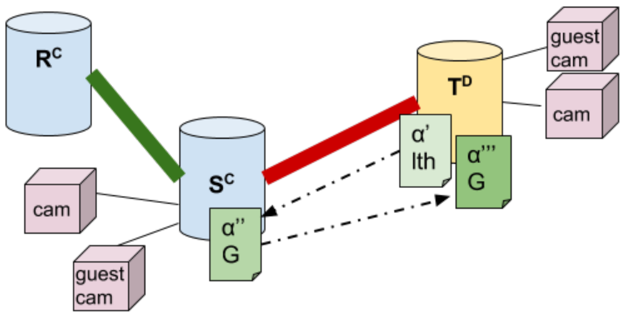


Figure 3.3: Issue: migration back after translation.

The runtimes S^C and T^D have been deployed and the channel has been established. Both of them have its proper translation policies, $T_C(I)$ and $T_D(I)$, that migrates any foreign identifier to a G(uest) user. Imagine as well that both are implementing a policy such that

$$p(camera) = \begin{cases} allow \rightarrow user \\ deny \rightarrow guest, other \end{cases} \tag{3.8}$$

$$p(guestcamera) = \begin{cases} allow \rightarrow user, guest \\ deny \rightarrow other \end{cases} \tag{3.9}$$

In addition, imagine that a user whose identifier is I_D="lth@domainD", initiates in T^D an application that involves an actor α, and $\hat{\alpha}$ is instantiated. This instance is allowed by the runtime's policy to use the resources "guest camera" and "camera", by the policies in the table. Then this instance is migrated to S^C, and regarding $T_C(I)$ we have

$$\hat{\hat{\alpha}} \leftarrow I_C, I_C = \phi_E(I_D) = guest@domainC$$

because this is the policy implemented for actors coming from D. This new identifier in C has access just to the "guest camera". After a time in C, the instance is migrated back and translated regarding the policy for actors in domain D.

$$\hat{\hat{\hat{\alpha}}} \leftarrow I_D, I_D = \phi_E(I_C) = guest@domainD$$

Note that with this new identifier, $\hat{\hat{\hat{\alpha}}}$ can just use the resource "guest camera" and the consequence is a privilege loss.

This approach is just handled by the translation table in the target domain, which can be described as simple to maintain, stateless and one side migration (since just one domain is involved).

One approach is that we only want a simple solution with low complexity and accept the limitations. This leads to a conclusion that we cannot let crucial actors do domain crossing, a fact one may specify in the policy of the application the actor is part of. Alternatively we could only send clones of these actors to the other domain, but that raises questions about the state of the original actor if that should go dormant or should remain active and what implications those choices have on the application as such.

Another alternative would be if the two implicated domains can synchronize their migrations such that the identifier is not translated, just given an equivalent in the other domain (user1@domainD and user1@domainC, for instance) and set the deployment policy to adjust the resource access properly so the same purpose can be achieved. But such an approach will cause an increase in the complexity of the effort to implement domain crossing.

On the other hand, there could be another translation solution that does not have this "coming back" problem. A few meetings were spent on this problem. It is a tricky problem because the actor instance is created an destroyed a few times in its life cycle and it is not possible to carry a permanent identifier. The possible path refers to involve the source domain in the translation. On this source side, the invariable part of the state, actual connections and actor type are signed by a Hash function and the value is kept associated to the identifier until some actor being migrated claims to have an old identifier there. We have this as a directions for future work. It is clear that more resources are required to handle the complexity such as that the source runtime has to keep track of all own actors migrated. This topic is later discussed in Chapter 5.2.

Finally, there is one aspect we assumed that can be questioned. Migrations are feasible if two runtimes establish a communication network. Actually the procedure is to issue the following command in one of the runtimes `cscontrol IP:port nodes add calvinip:address`. This could be easier done within a domain than when being outside of it, because users that are part of the domain might known the address of the runtimes, or at least have access to this information. On the other hand, if a user of another domain wants to migrate to this domain, it has to manually set the network. Hence, to guarantee the system feasibility a runtime's address should be made public. Of course just in case network is required and the disclosure level of address should depends on the purpose of the application.

Chapter 4

Implementation in Calvin code

In this chapter we detail an implementation of the domain crossing solution described in Chapter 3. The implementation is realized within the Calvin Code, using the developer's code version released on March 2016. The implementation we describe here is actually the second implementation we made. A concept implementation was done earlier to explore only some characteristics of the solution. This first implementation helped us also to get better acquainted with the existing code for the later work. In the second implementation features like the translation have been completely re-factored. The subsequent sections of this chapter will describe separate parts of the implementation one by one.

Please note that our transport category design is implemented using a fake transport procedure because the TLS transport has not been implemented yet in the Calvin code, and the development of such an implementation has not been part of the thesis objective. Due to its complexity we refrained from even trying to integrate TLS into Calvin. The fake transport procedure will be explained later. This transport realization is also the only thing that not has been implemented with respect to the previous chapter 3.

The implementation code is available in Appendix B.

4.1 Already being implemented in Calvin

The development of the Security module in Calvin has simplified the implementation of the domain crossing solution described in the previous chapter. The Security Module provides us with credentials for the identities and with support for runtime policies.

The actor's identity that is a key point of the domain crossing solution will be provided by the credentials used to authenticate an user in the system, formed by an identifier and a password. Since these credentials are part of an actor instance with the provided Calvin implementation, it fits the solution designed in the previous chapter. The credentials have been implemented as an attribute of the `Actor` class, and will be also part of the actor's state when migrating. In this way they can be received and accessed in the target runtime and make the proper translation if it is needed.

Building on the existing work, the runtime authorization policies have been incorporated into the Calvin stable version and fulfil the needs of our solution requirement for this type of policies. Applications and actors will be deployed and instantiated respectively only if the policies are met, otherwise a deployment will not be possible. Users will have associated some resources and their performance is limited by these policies, as expected by the approach. So we could include this in the implementation and not build a new one.

Finally, although TLS has not been still implemented, the certificates needed to identify a runtime can be created with the command `csmanage ca create -domain` *`domain`* `-name` *`nameCA`*. This uses the already commented functions in certificate module.

4.2 Translation policy

Before detailing the code that implements the domain translation, the method for describing a translation policy should be presented. An example of such policy is shown below:

```
{
"id": "translation_policy",
"description": "Policy managing the translation done to cross a domain.",
"rules": [
{
      "id": "policy4_rule1",
      "description": "Permit if policy target matches",
      "translation_category":"domain",
      "source":["ericsson", "google"],
      "result": "friendguest@test"
}, {
      "id": "policy4_rule2",
      "description": "Permit if policy target matches",
      "translation_category":"identifier",
      "source":["ben@lth"],
      "result": "ben@test"
}, {
      "id": "policy4_rule3",
      "description": "Permit if policy target matches",
      "translation_category":"default",
      "result": "guest@test"
} ] }
```

Basically the policy consists of a group of rules grouped within the field `rules`. The most important data field in the translation rules is the `translation_category`. There are three defined categories for this first version: `domain`, `identifier` and `default`.

- `Domain`: translates an identifier from one of the domains indicated to a specific new identifier. This can be used when we know a determined domain and we want to allow the users of this domain to do something.

- `Identifier`: translates a concrete identifier to another new identifier. For instance, it can be expected to have a specific user from a foreign domain that needs to use some resources in our domain, so it is given a new identifier to use the resources meeting the actual deployment policy.
- `Default`: translates any identifier to a default identifier value. For example, it is supposed a default value that gives a identifier with low rights, so every identifier not matter its origin domain can be deployed in the runtime and not compromise the worth resources.

The order in which rules are written is important: the implementation will stop with the first matching rule and the identifier that is the result is then committed. Hence it is expected that the `default` rule should be the last one after the specific rules, otherwise rules located below the `default` will never be checked.

Finally, the field `source` in domain and identifier categories refer to the incoming value that the policy will be applied to. Of course in *default*, since this category does not care about the incoming value, does not have this field. Then the `result` field sets the new value.

How the policy is actually created is something we have not addressed. Please note that we have considered in this implementation that an identifier should have the format `name@`*domain*. This syntax is understood in the domains. The use of the `default` rule in the translation policy makes this approach rather worthless because it will translate any (format) identifier to the result one. But it is indeed so that if we want to make rules for a specific domain, it is easier if the identifier's domain is incorporated in it. So the conclusion is that there are some cases that lead to risks if the identifier does not follow a standardized format. The receiving domain that has to implement the policy types `domain` and `identifier` needs to interact with the source domain to a) understand the syntax of the identifiers found in the serialized actor instance and b) specify the translation rules so the actor can be deployed in a useful manner.

4.3 Translation

A new Calvin module, realized in the file *translation.py*, has been implemented in the utilities source code directory. It contains the implementation of the class `TranslationPolicy` that is the skeleton for the translation. A short description of this class follows now.

`TranslationPolicy`

- `__init__(policy_file)`. This is the constructor. The argument `policy_file` is the JSON file which contains the translation policy (last section). The constructor opens the policy field and extracts the data.
- `get_policy_id()`. Returns the policy id, a policy's data field.
- `get_policy_description()`. Returns the policy description, another data field.
- `translate(identifier)`. Returns the translated identifier regarding the policy loaded. Could be a new identifier or the same in case no translation

is performed. `Identifier`: the incoming identifier. The value is obtained reading each translation policy rule, and if there is a match between the identifier and the rule, the

- `no_cheating(identifier_domain, domain, link_category)`. Checks the incoming identifier to find cheating. It is based in the statement that if the channel link is *interdomain*, the runtime can not receive an identifier from its own domain (example in Fig 4.1). Returns `True` if there is no cheating or `False` if there is. `Identifier_domain`: the domain to whom the identifier belongs. `Domain`: the host runtime domain. `Link_category`: the link category set up between the two runtimes. The method just check that if the `link_category` is *interdomain*, the coming identifier's domain can not be the same as the hosting runtime domain.

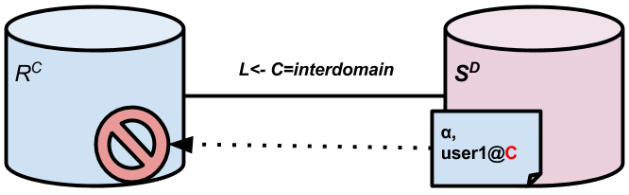


Figure 4.1: Cheating example when migrating.

4.4 Modifications committed in Calvin

After implementing the translation module, it was merged together with other modifications with the Calvin code to reach the complete desired functionality.

4.4.1 The configuration file

The configuration file, *calvin.conf*, was used to provision some fundamental values. Due to the requirement by the Calvin design team not to add more parameters to the Calvin commands, any data that parametrizes a running instance should be described in the config file.

Like in the Translation Policy section, let's consider an example:

```
{
"security": {
"security_conf": {
     "comment": "Security settings with access control enabled",
     "authentication": {
     "procedure": "local",
     "local_users": {"user1@test": "pass1", "user2@test": "pass2",
"user3@test": "pass3"}
},
      "authorization": {
      "procedure": "local",
      "policy_storage_path": "/home/pepe/.calvin/security/policies" },
      "translation": {
      "procedure": "local",
      "policy_storage_file": "/home/pepe/calvinExjobb/calvin-base/calvin/
translation_policy.json"
} },
"certificate_conf": "/home/pepe/.calvin/security/test/openssl.conf",
"certificate_domain": "test",
"peers_in_domain": [],
"fake_transport":"interdomain",
"domain":"ericsson"
} }
```

Here we added `translation` to `security_conf`, to indicate the path to the translation policy file. `Peers_in_domain` and `fake_transport` are used to the above referred fake transport category. `Peers_in_domain` should contain a list of calvinip addresses that constitute a domain. For instance, ["calvinip://10.10.10.10", "calvinip://10.10.10.20, "calvinip:"10.10.10.30"]. For the sake of testing and simplicity, the runtimes were deployed on the same machine so this approach was not worthy. `Fake_transport` can be used instead to force the value but one must be aware that `peers_in_domain` value needs to be empty. Finally, `domain` is used to specify to which domain the runtime belongs to.

The other data fields were already defined by the existing Calvin functions.

4.4.2 The link category

The value of the link category is determined in *calvin/runtime/south/plugins/transports/lib/twisted/twisted_transport.py*. This class creates a transport from a source runtime to a target runtime (one direction channel). This should be the class that receives the category value from TLS transport. Meanwhile TLS is implemented, fake transport will be simulated. Access to the configuration file is provided (added `_conf = calvinconf.get()`) such than the configuration fields `peers_in_domain` and `fake_transport` can be accessed. Moreover a new function is created, called `check_list(peer)`. This function gives the value of the link category. It finds `peer` in a peers list, that is obtained from the configuration file as explained above, and if it is in the list it means that is part of the domain (*intradomain*), otherwise is a *interdomain* link. If the list is empty, it will take the value of `fake_transport` directly from that file as well.

The existing class `CalvinTransport` is modified to have an attribute `_transport_category` set up with the function `check_list` when the constructor is called. And two more `set_transport_category(category)` and `get_transport_category()` to access the category variable.

After the value is obtained in this transport class, it will be passed to higher layer classes and the next modifications were performed. Figure 4.2 shows how the link category value is accessible for upper classes in the communication scheme. The goal is to provide this value to the actor manager that is the module in charge of creating actors, and this value should be checked when a migration is received.

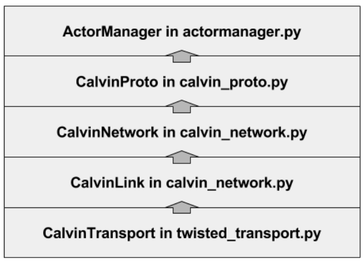

ActorManager in actormanager.py
CalvinProto in calvin_proto.py
CalvinNetwork in calvin_network.py
CalvinLink in calvin_network.py
CalvinTransport in twisted_transport.py

Figure 4.2: Accessibility to link category value.

Thus, in the file *calvin/runtime/north/calvin_network.py*, first the class `CalvinLink` incorporates the attribute `transport_category`. In addition, `set_transport_category(category)` and `get_transport_category()` have been created to modify and access this attribute.

The other class that was modified is `CalvinNetwork`. In the method `join_finished` after a `CalvinLink` object is created, and there is access to the transport `CalvinTranport`, the link category is passed from the transport to the link by `link.set_transport_category(transport.get_transport_category())`. This approach has been adopted to not modify the constructor interface.

At that moment the `CalvinLink` has the value permanently stored. We continue with *calvin/runtime/north/calvin_proto.py*. The class `CalvinProto` serves as the interface between runtimes. When a migration is issued, the method `actor_new_handler` is executed on the target side (when receives a message from the source runtime). It takes the transport category from the link object that participates in the communication and give it as parameter to the actor manager. This is possible because `CalvinProto` has one intance of `CalvinNetwork` as attribute variable and with this class the `CalvinLink` is accessible.

So, the next file is *calvin/runtime/north/actormanager.py*. The method "new" has a new parameter called `link_category`. It will be passed to the method `_new_from_state` as a parameter that has been added to the method definition. The method, `_new_from_state`, is really important because the translation is performed here and it will be explained in the next section.

4.4.3 The translation

The translation is realized in the `ActorManager` class. This class is the one that can create actor instances in a runtime. Concretely in `_new_from_state`, utilizing the translation module created for this case. This method is the correct place to translate because this function is designed to create an actor that owns an already created state, which is the case of a migration. The other available method `_new` for actors creation does not use state, it is for the first instantia-

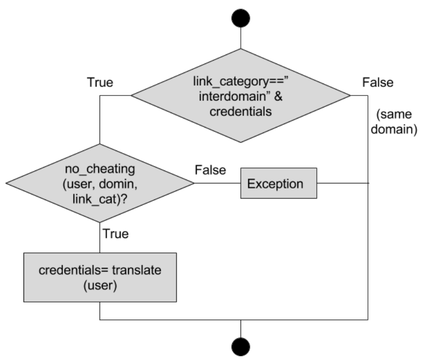

link_category==" interdomain" & credentials
True
False
(same domain)
no_cheating (user, domin, link_cat)?
False
Exception
True
credentials= translate (user)

Figure 4.3: Activity diagram regarding the translation code.

tion. Thus, one needs to change the identifier before the `Actor` object is instantiated. A small piece of code has been added following the algorithm in Figure 4.3. Thus if the link category is *interdomain* (which means domain crossing) and credentials are available (the actor is identified), possible cheating will be checked (`no_cheating function`) and the translation of the identifier executed (`migration_policy.translate(identifier)`). Please note that a `TranslationPolicy` object is created and stored when an `ActorManager` object is created, and takes the necessary parameters from the configuration file. Access to the configuration file has been added as well (applying `_conf = calvinconfig.get()`) Moreover, the user's identifier can be obtained from the credentials which at the same time can be obtained from the actor's state, one of the parameters of the method `_new_from_state` and that is given.

Note that the credentials are part of the actor's state, and the state is received in `CalvinProto` by the method `actor_new_handler` whose argument is the payload sent by the source runtime and the state is included there. When `CalvinProto` receives the order to create a new actor, it calls the actor manager and gives the actor, so the actor manager knows it is a translation and `_new_from_state` is issued.

4.5 Tests

Two scenarios have been used to test the domain crossing solution and test the code to make sure that everything behaves as we want. We used two runtimes and both runtimes were on the same machine so we set up the configuration file with the value of `peers_in_domain` empty and forced the transport with `fake_transport` value. With respect to the deployment policy, we have kept opened the policy so everything can be deploy, we are not interested in that part at all. Neither the attributes and requirements scheme is used.

The following counter application will be used in each scenario:

```
source :  std.Counter()
```

```
delay :  std.ClassicDelay(delay=0.5)
output :  io.Print()

source.integer > delay.token
delay.token > output.token
```

It is a counter starting in 1. The samples are delayed so it counts slowly. The `output` actor prints the result on the screen. This actor will be the target for migration considering that we could see how different terminals (runtimes) produced numbers.

First scenario: intradomain

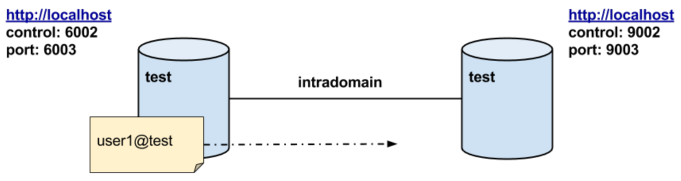
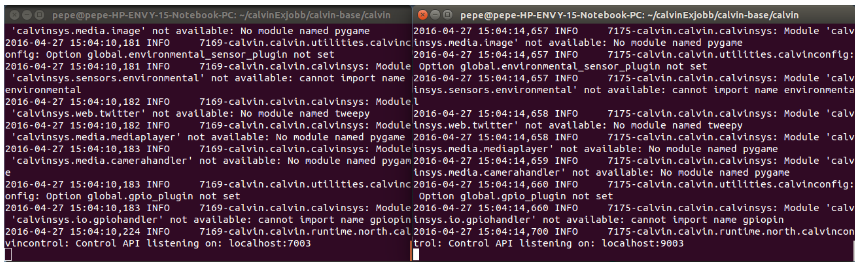


Figure 4.4: First scenario scheme.

This scenario tests that the identifier is not translated among runtimes that belong to the same domain. The scenario is described by Figure 4.4. In this scenario, `fake_transport` is `intradomain` and `domain` is `test`. Both runtimes were deployed, see Figure 4.5

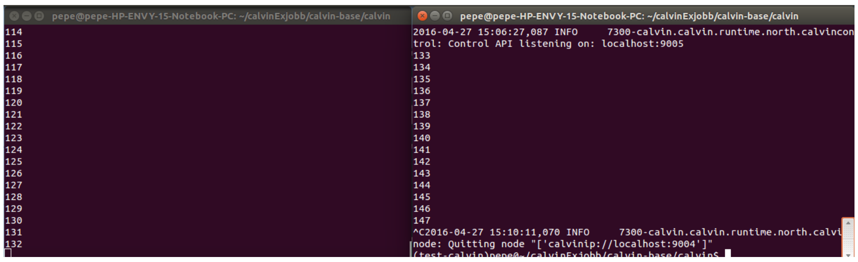


```
pepe@pepe-HP-ENVY-15-Notebook-PC: ~/calvinExjobb/calvin-base/calvin
 'calvinsys.media.image' not available: No module named pygame
2016-04-27 15:04:10,181 INFO     7169-calvin.calvin.utilities.calvinc
onfig: Option global.environmental_sensor_plugin not set
2016-04-27 15:04:10,181 INFO     7169-calvin.calvin.calvinsys: Module
 'calvinsys.sensors.environmental' not available: cannot import name
environmental
2016-04-27 15:04:10,182 INFO     7169-calvin.calvin.calvinsys: Module
 'calvinsys.web.twitter' not available: No module named tweepy
2016-04-27 15:04:10,182 INFO     7169-calvin.calvin.calvinsys: Module
 'calvinsys.media.mediaplayer' not available: No module named pygame
2016-04-27 15:04:10,183 INFO     7169-calvin.calvin.calvinsys: Module
 'calvinsys.media.camerahandler' not available: No module named pygam
e
2016-04-27 15:04:10,183 INFO     7169-calvin.calvin.utilities.calvinc
onfig: Option global.gpio_plugin not set
2016-04-27 15:04:10,183 INFO     7169-calvin.calvin.calvinsys: Module
 'calvinsys.io.gpiohandler' not available: cannot import name gpiopin
2016-04-27 15:04:10,224 INFO     7169-calvin.calvin.runtime.north.cal
vincontrol: Control API listening on: localhost:7003
```

```
pepe@pepe-HP-ENVY-15-Notebook-PC: ~/calvinExjobb/calvin-base/calvin
2016-04-27 15:04:14,657 INFO     7175-calvin.calvin.calvinsys: Module 'calv
insys.media.image' not available: No module named pygame
2016-04-27 15:04:14,657 INFO     7175-calvin.calvin.utilities.calvinconfig:
 Option global.environmental_sensor_plugin not set
2016-04-27 15:04:14,657 INFO     7175-calvin.calvin.calvinsys: Module 'calv
insys.sensors.environmental' not available: cannot import name environmenta
l
2016-04-27 15:04:14,658 INFO     7175-calvin.calvin.calvinsys: Module 'calv
insys.web.twitter' not available: No module named tweepy
2016-04-27 15:04:14,658 INFO     7175-calvin.calvin.calvinsys: Module 'calv
insys.media.mediaplayer' not available: No module named pygame
2016-04-27 15:04:14,659 INFO     7175-calvin.calvin.calvinsys: Module 'calv
insys.media.camerahandler' not available: No module named pygame
2016-04-27 15:04:14,660 INFO     7175-calvin.calvin.utilities.calvinconfig:
 Option global.gpio_plugin not set
2016-04-27 15:04:14,660 INFO     7175-calvin.calvin.calvinsys: Module 'calv
insys.io.gpiohandler' not available: cannot import name gpiopin
2016-04-27 15:04:14,700 INFO     7175-calvin.calvin.runtime.north.calvincon
trol: Control API listening on: localhost:9003
```

Figure 4.5: First scenario, runtimes deployment.

The application was deployed issuing `cscontrol http://localhost:7003 deploy test/counterapp.calvin -credentials '{"user":"user1@test", "password": "pass1"}'`. Then the printer actor was migrated and the result is in Figure 4.6. Nothing happens apparently and that is the expected result.



```
pepe@pepe-HP-ENVY-15-Notebook-PC: ~/calvinExjobb/calvin-base/calvin
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
```

```
pepe@pepe-HP-ENVY-15-Notebook-PC: ~/calvinExjobb/calvin-base/calvin
2016-04-27 15:06:27,087 INFO     7300-calvin.calvin.runtime.north.calvincon
trol: Control API listening on: localhost:9005
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
^C2016-04-27 15:10:11,070 INFO     7300-calvin.calvin.runtime.north.calvi
node: Quitting node "['calvinip://localhost:9004']"
(test-calvin)pepe@~/calvinExjobb/calvin-base/calvin$
```

Figure 4.6: First scenario, no migration.

Second scenario: interdomain

We simulated an *interdomain* transport between two runtimes installed in the same machine but separate virtual environments (see how to install Calvin in the documentation using virtual environments). Both configuration files were set correctly regarding the value of `domain` and setting `fake_transport` to `interdomain` in both configurations. The cases displayed in Figure 4.7 were tested and are explained below.

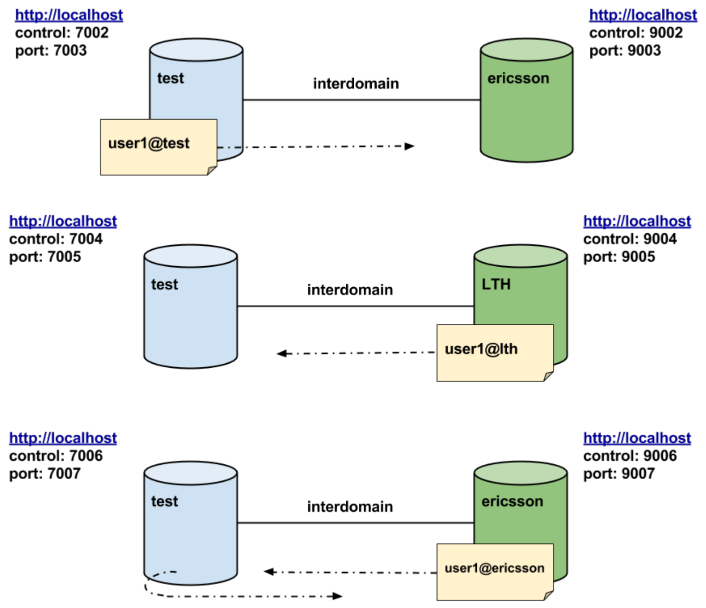
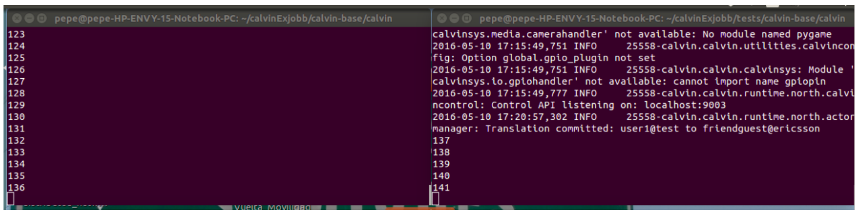


Figure 4.7: Second scenario scheme.

The first case tests a migration from test domain to ericsson domain. The application was deployed, for all the cases, issuing again `cscontrol http://localhost:PORT` `application deploy PATH/counter.calvin -credentials '{"user":"user1@DOMAIN", "password":"pass1"}'` which authenticated the user in each domain. Due to the policy implemented in ericsson domain,

```
...
"translation_category":"domain",
"source":["google", "test"],
"result":  "friendguest@ericsson"
...
```

the result obtained in Figure 4.8 is `friendguest@ericsson` since the identifier was coming from `test`.



```
pepe@pepe-HP-ENVY-15-Notebook-PC: ~/calvinExjobb/calvin-base/calvin
123
124
125
126
127
128
129
130
131
132
133
134
135
136
```

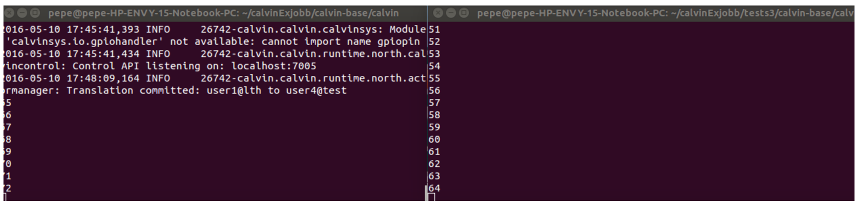
```
pepe@pepe-HP-ENVY-15-Notebook-PC: ~/calvinExjobb/tests/calvin-base/calvin
calvinsys.media.camerahandler' not available: No module named pygame
2016-05-10 17:15:49,751 INFO     25558-calvin.calvin.utilities.calvincon
fig: Option global.gpio_plugin not set
2016-05-10 17:15:49,751 INFO     25558-calvin.calvin.calvinsys: Module '
calvinsys.io.gpiohandler' not available: cannot import name gpiopin
2016-05-10 17:15:49,777 INFO     25558-calvin.calvin.runtime.north.calvi
ncontrol: Control API listening on: localhost:9003
2016-05-10 17:20:57,302 INFO     25558-calvin.calvin.runtime.north.actor
manager: Translation committed: user1@test to friendguest@ericsson
137
138
139
140
141
```

Figure 4.8: 2nd scenario, first case result. Test (left) and Ericsson (right).

During the second test, the result in Figure 4.9 was obtained. The domains involved were test and lth, who migrated the actor. The identifier policy category matched the migrated identifier:

```
...
translation_category":"identifier",
"source":["user1@lth"],
"result":  "user4@test"
...
```

Thus `user4@test` was the new identifier in test domain.



```
pepe@pepe-HP-ENVY-15-Notebook-PC: ~/calvinExjobb/calvin-base/calvin
016-05-10 17:45:41,393 INFO     26742-calvin.calvin.calvinsys: Module
'calvinsys.io.gpiohandler' not available: cannot import name gpiopin
016-05-10 17:45:41,434 INFO     26742-calvin.calvin.runtime.north.cal
incontrol: Control API listening on: localhost:7005
016-05-10 17:48:09,164 INFO     26742-calvin.calvin.runtime.north.act
rmanager: Translation committed: user1@lth to user4@test
5
6
7
8
9
0
1
2
```

```
pepe@pepe-HP-ENVY-15-Notebook-PC: ~/calvinExjobb/tests3/calvin-base/calv
51
52
53
54
55
56
57
58
59
60
61
62
63
64
```

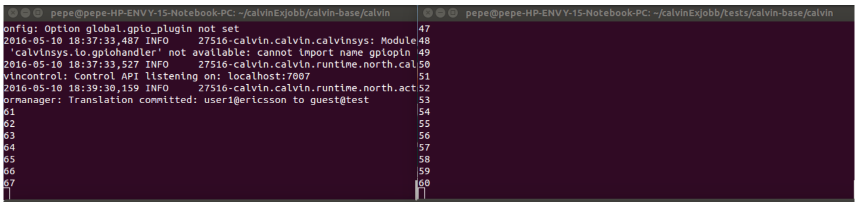
Figure 4.9: 2nd scenario, second case result. Test (left) and LTH (right).

Finally the third test, where the problem related to the solution was simulated. Again test and ericsson were the domains defined. Ericsson runtime migrated the actor to test where the default policy matched:

```
...
```

```
"translation_category":"default",
"result":  "guest@test"
...
```

Thus, `guest@test` was the new identifier, see Figure 4.10. Then ericsson runtime



```
pepe@pepe-HP-ENVY-15-Notebook-PC: ~/calvinExjobb/calvin-base/calvin
onfig: Option global.gpio_plugin not set
2016-05-10 18:37:33,487 INFO     27516-calvin.calvin.calvinsys: Module
 'calvinsys.io.gpiohandler' not available: cannot import name gpiopin
2016-05-10 18:37:33,527 INFO     27516-calvin.calvin.runtime.north.cal
vincontrol: Control API listening on: localhost:7007
2016-05-10 18:39:30,159 INFO     27516-calvin.calvin.runtime.north.act
ormanager: Translation committed: user1@ericsson to guest@test
61
62
63
64
65
66
67
```

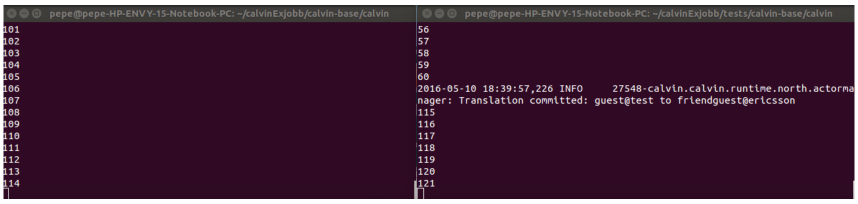
```
pepe@pepe-HP-ENVY-15-Notebook-PC: ~/calvinExjobb/tests/calvin-base/calvin
47
48
49
50
51
52
53
54
55
56
57
58
59
60
```

Figure 4.10: 2nd scenario, third case A result. Test (left) and Ericsson (right).

ordered the migration back and the following policy was applied:

```
...
"source":["google", "test"],
"result":  "friendguest@ericsson"
...
```

to give `friendguest@ericsson`, Figure 4.11. Please note that the so mentioned identity loss appears here.



```
pepe@pepe-HP-ENVY-15-Notebook-PC: ~/calvinExjobb/calvin-base/calvin
101
102
103
104
105
106
107
108
109
110
111
112
113
114
```

```
pepe@pepe-HP-ENVY-15-Notebook-PC: ~/calvinExjobb/tests/calvin-base/calvin
56
57
58
59
60
2016-05-10 18:39:57,226 INFO     27548-calvin.calvin.runtime.north.actorma
nager: Translation committed: guest@test to friendguest@ericsson
115
116
117
118
119
120
121
```

Figure 4.11: 2nd scenario, third case B result. Test (left) and Ericsson (right).

Chapter 5

Discussion

5.1 Discussion of some topics

First in the approach and later in the implementation, the identifier has been a key word. Let us return to the designed translation policies. It could be interesting to reconsider how the new translated identifier is created and if the chosen approach is appropriate. In a domain there will be identified users that deploy their applications. Their actors can be tracked using the identifier that is associated with the application/actor. Now consider actors being migrated to this domain. Also let us assume that there is a policy that translates any identifier to `guest@domain`, as it has been done in the example policy we discussed before. In this situation, many actors called 'guest' land up in the domain even if many different users have deployed them on their respective domain. Indeed, we are mapping many real users to one identifier. This seems not a real attractive situation. Maybe this situation can be avoided.

One approach is to make the translations more unique, by, for instance, assigning different guest identifiers such as `guest1@domain`, `guest2@domain`,... All of them are still "guests" in the domain, yet the new identifier is sort of individual, a kind of pseudonym and we can identify different external users in the domain. Such an approach may have its use. Indeed it is natural to think about identifiers that refer to one user and not a group. However, we finally adopted the described policy in Chapter 3 since we are not interested in the user once it has been translated. We only care about what kind of resources the actor can access, and the policies that achieve this. Note that our approach still allows a named user to be translated to a single user in the target domain but such a mapping requires that there is trust between the domains and a previous knowledge of the incoming identifier value.

Finally, it is also worth to mention that there is an idea of translation as a wider scope solution, not only for Calvin and the domain crossing. It is thought to be a procedure whose function is to link heterogeneous systems and provide interoperability and systems integration. Maybe two organizations that works with the same feature but each one implement it in different way, both of them could understand each other if the translation is performed. For example it could be used to link policies for different uses. Just mention that solution pattern to provide interoperability.

5.2 Future work

Regarding the future work that would build upon this project, we note that the translation policy implemented in this thesis work is an example and the policy could be modified to fulfil the final user's requirements. For example, adding more rules or other fields. Of course the class *translation policy* needs to be updated to process correctly the new JSON policy file. Furthermore, the procedure can be uploaded to be external, like the deployment policy, where other runtimes can be act as translation server. For example if unification of translation are required.

The transport category procedure is temporary since TLS transport is not implemented. The actual Calvin transport, based on Twisted, can use the SSL libraries provided by Twisted to get an improved version. Moreover, the transport category value must be generated after the TLS channel has been established. This requires functionality to be added to the implementation. With the authentication , the runtimes can known if the other party belongs to the same domain or not and set the category. This should be the procedure taken to decide *intradomain* or *interdomain* transport category.

Another future path could be, as mentioned in Section 3.5, to attend the problem that appears when an actor returns to its source domain. It would be interesting to continue the investigation within that subject, describe in depth the problem, figure out possible solutions and describe advantages and shortcomes. At least, the goal should be to find a solution that maintains the status of an actor in a domain independently of the number of migrations that the actor instance has made. For status it is understood the resource access that the actor used to have.

Finally, perform a true distributed deployment, since in the tests executed the runtimes where in the same machine.

Chapter 6 Conclusion

The Calvin system is still in its initial development stage and much new functionality is being added. Our study did not concern exactly the functionality of the Calvin system as defined by the Calvin code but assumed that the concept of domain is present. Assumptions have been made how the Calvin system will implement this and we checked our assumptions with the Calvin team at Ericsson. Where as the domain concept itself has been considered by the Calvin team there is no security solution, designed or even sketched, that guarantees trusted operations when domain crossing migrations are performed. The risk is to receive untrusted actors from another domain that could use our resources on behalf of untrusted users.

To enhance the security in Calvin and to cope with actors that can cross domains, solutions have been studied and we formulated on proposal and made a proof-of-concept for testing if the approach indeed can be realized in the Calvin framework. The solution has its fundament in an identification scheme for actors deployed in the runtime. Applications, and implicitly actors, will handle identifiers of the user that executes them. A policy will control the actor deployment regarding its identifier and the resources it is requesting. Identifiers will be part of the namespace valid in the domain. For actors identified by other domain, a translation will be completed such that the new identifier can be valid for the domain's namespace and therefore, for the runtime's policies. Moreover TLS transport will enhance communications channels features and give us a means to determine if actors come from a trusted domain, e.g. the same as I belong to, or that the incoming actors come from a runtime in a foreign domain.

The proposed solution has been implemented in Calvin, where a number of Calvin modules have been modified to adopt the new modules that were added. Runtime deployment policies and actor credentials were already implemented by Calvin engineers while carrying this project and our final implementation makes use of this.

The result demonstrates that actors migrated to a runtime in another domain can be meaningfully handled in the receiving domain. Through a translation process one can assign an identity to the incoming actor that is then used to perform the access control like any other actor in the receiving domain. However this approach has as a consequence that the actor that crosses the domain looses its original identifier which creates problems when, for example, coming back to its origin domain. Currently we do not see a simple way out here but we suggested approaches that could partly give a better solution, albeit at the price of higher complexity. The advantage of the translation scheme proposed in the thesis is its intuitive simplicity and that it does not require the origin or receiving domain to keep track of the actor. Therefore we believe that even if the solution has its limitations it can be use to handle many simpler use cases. For example where one does not have to handle that the actor returns to its origin domain or moves further.

Our work shows that the domain transition is in a general setting a very difficult use case to handle and more research is needed to find a solution that can handle complex use cases with actors moving back and forth or migrating over several domains. Besides the question how to handle such dynamic migration behaviour we also see here many challenges in how to specify what policies should followed by the runtimes when dealing with migrating actors. Especially if the domains belong to different organisation it is not clear how to easily define the policies without having some kind of common or standardised approach to to use the access control mechanisms in the Calvin system. Also here more research is needed.

Finally, we want to note that our observations on Calvin actors can, after proper reformulation, be applied to virtual machines or containers that are migrating. To pursue this line of thinking could be an interesting study item as well.

References

[1] Cisco. Internet of things. `http://www-cs-faculty.stanford.edu/~{}uno/abcde.html`, 2014.

[2] Ericsson. Ericsson mobility report. `http://www.ericsson.com/res/docs/2016/ericsson-mobility-report-2016.pdf`, June 2016.

[3] Alex Wood. The internet of things is revolutionizing our lives, but standards are a must. The Guardian (theguardian.com), March 2015.

[4] P. Corcoran. The internet of things: Why now, and what?s next? *IEEE Consumer Electronics Magazine*, 5(1):63–68, Jan 2016.

[5] R. Minerva, A. Biru, and D. Rotondi. Towards a definition of the internet of things(iot). Master's thesis, Politecnico di Torino, 2015.

[6] L. Tan and N. Wang. Future internet: The internet of things. In *3rd International Conference in Advanced Computer Theory and Engineering (ICACTE).*

[7] M Presser et al. *Inspiring the Internet of Things: The Internet of Things Comic Book*. Internet Of Things International Forum, 2012.

[8] Y. K. Chen. Challenges and opportunities of internet of things. In *17th Asia and South Pacific Design Automation Conference*, pages 383–388, Jan 2012.

[9] S. Ray, Y. Jin, and A. Raychowdhury. The changing computing paradigm with internet of things: A tutorial introduction. *IEEE Design Test*, 33(2):76–96, April 2016.

[10] G. S. Matharu, P. Upadhyay, and L. Chaudhary. The internet of things: Challenges amp; security issues. In *Emerging Technologies (ICET), 2014 International Conference on*, pages 54–59, Dec 2014.

[11] S. A. Kumar, T. Vealey, and H. Srivastava. Security in internet of things: Challenges, solutions and future directions. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 5772–5781, Jan 2016.

[12] M. Amadeo et al. Information-centric networking for the internet of things: challenges and opportunities. *IEEE Network*, 30(2):92–100, March 2016.

[13] F. J. Riggins and S. F. Wamba. Research directions on the adoption, usage, and impact of the internet of things through the use of big data analytics. In *System Sciences (HICSS), 2015 48th Hawaii International Conference on*, pages 1531–1540, Jan 2015.

[14] M. R. Palattella et al. Internet of things in the 5g era: Enablers, architecture, and business models. *IEEE Journal on Selected Areas in Communications*, 34(3):510–527, March 2016.

[15] Calvin web site. `http://www.ericsson.com/research-blog/cloud/open-source-calvin/`.

[16] Calvin gthub repository. `https://github.com/EricssonResearch/calvin-base`.

[17] G. Agha. Actors: A model of concurrent computation in distributed systems. *Cambridge, MA, USA: MIT Press*, 2016.

[18] Michael Armbrust et al. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[19] Noflo web site. `http://noflojs.org/`.

[20] Orleans web site. `http://dotnet.github.io/orleans/`.

[21] Netbeast web site. `https://netbeast.co/`.

[22] Wiki of calvin in github. `https://github.com/EricssonResearch/calvin-base/wiki`.

[23] P. Persson and O. Angelsmark. Calvin - mergin cloud and iot. *Procedia Computer Science*, 2015.

[24] T. Nilsson. Authorization aspects of the distributed dataflow-oriented iot framework calvin, 2016. Student Paper.

[25] E. Yuan and J. Tong. Attributed based access control (abac) for web services. In *IEEE International Conference on Web Services (ICWS'05)*, page 569, July 2005.

[26] Twisted web site. `https://twistedmatrix.com/trac/wiki/Documentation`.

[27] D. Gollmann. *Computer Security*. Wiley, 3 edition, 2011. Chapter 15.5 Public-Key Infraestructures.

[28] N. Lindskog. Consistent authentication in distributed networks, 2016. Student Paper.

[29] D.R. Stinson. *Cryptography, Theory and Practice*. Chapman Hall, 2006. Chapter 12.

[30] Rfc 5246 the transport layer security (tls) protocol version 1.2. `https://tools.ietf.org/html/rfc5246`.

Appendix A

Basics of needed security concepts

A.1 Secure Socket Layer

Secure Socket Layer is the name used to refer to the SSL as well as to the TLS protocol each designed to provide secure communications over a network. Today actually only TLS (Transport Layer Security) should be used. TLS is an improved version of the SSL family of protocols and is specified in the IETF RFC 5246, see reference [30]. Still SSL is used and it then refers to both SSL and TLS.

During the set-up, or handshake, of the TLS session, first the two parties will introduce themselves and perform an authentication and then agree on the cryptographic algorithms that will be used in the next step. During the authentication two things can happen. Either the server side will only be authenticated by the client side, or both client and server side authenticate each other. TLS supports different ways to perform the authentication during the handshake. For Calvin we consider the case that the two sides send each other their PKI certificate containing a copy of the public key, signed by a trusted certification authority of the given PKI. At the receiving side of the handshake, the certification authority signature can be verified using the authority's public (root) key that somehow should be available in a secure way. As subsequent part of the handshake both parties proceed to determine two common secret session keys. This is done by generating a random master key that is send to the other end using the other end's public key so that side can decrypt it. When the two parties have the master key, they independently generates the same two keys (K1 and K2) from the master key using a hash based key derivation function. The keys will be used to integrity protect and authenticate data using a message authentication code and to encrypt and decrypt data using a secret key crypto system, respectively.

The usefulness of the TLS comes from the fact that the certificate that is used in the handshake can be linked to a certificate of the runtime which per definition is linked to the domain. The linking of certificates can be done using a Public Key Infrastructure (PKI). In the next section we briefly summarize some important properties of such a PKI:

A.2 Public Key Infrastructure

Public Key Infrastructure is a term used to describe a system for issuing and managing certificates. The Certificate Authority (CA) has an important role in a PKI because it issues the certificates that make the binding between a subject and a cryptography key. The certificate contains a public key, mandatory and optional attributes and the CA digital signature. The said binding will couple an identifier to the subject. Before issuing the certificate the Certificate Authority should check that the subject is who it claims to be and that the certificate's attributes are correct. Furthermore, the certificate can be defined for a specific use as a signed instrument that empowers the subject, and containing at least an issuer and a subject attribute, validity conditions, as well as authorization and delegation information. In its simplest form it binds the subject name to a public key and to owner of the private can use this to prove he/she is indeed the subject named by the subject name. It is this binding that makes to certificate so useful in the Calvin domain concept.

Below an example of certificate is shown. Among the different information fields, some of the stand out. The domain and CA is indicated in the field issuer, where the values are testdomain and testdomainCA. The time for which the certificate is valid is indicated in Validity. The runtime owning this certificate is referred in the subject field, O=testdomain/dnQualifier=9b8627dc-bf7c-4034-b31b-0719f7b85014,
CN=org.testexample++++testNode1, where the first value is identifier in the domain the runtime belongs to, and the second one the name given by the runtime attributes. The public key of the subject is given by the Public Key field, where some aspects about the key are indicated as well. Finally, the certificate can be decrypted used the public key of the CA to check its truthfulness. `Certificate:`

```
        Data:
            Version:  3 (0x2)
            Serial Number:  4097 (0x1001)
        Signature Algorithm:  sha256WithRSAEncryption
            Issuer:  O=testdomain, CN=testdomainCA
            Validity
                Not Before:  May 26 08:33:30 2016 GMT
                    Not After :  May 26 08:33:30 2017 GMT
            Subject:  O=testdomain/dnQualifier=9b8627dc-bf7c-4034-b31b-0719f7b85014,
CN=org.testexample++++testNode1
            Subject Public Key Info:
                Public Key Algorithm:  id-ecPublicKey
                    Public-Key:  (256 bit)
                    pub:
                        04:58:8d:f8:47:9a:0b:9f:5b:7a:5e:6d:5e:8b:12:
                        a9:91:b2:63:b9:71:cf:09:0b:25:0e:8f:46:b8:6f:
                        1d:50:f6:3c:e2:cf:a6:ed:e3:03:60:e2:fd:b2:74:
                        65:77:fe:1d:6e:62:cf:d3:2b:05:52:7a:df:3b:df:
                        c8:75:bb:26:72
                    ASN1 OID: prime256v1
            X509v3 extensions:
                 X509v3 Subject Key Identifier:
                       E3:28:8B:B4:53:14:9A:A6:7F:C7:49:35:17:45:52:7E:13:A3:2C:7F
                 X509v3 Authority Key Identifier:
                       DirName:/O=testdomain/CN=testdomainCA
                       serial:C8:CE:77:33:6E:4C:3C:08
                 X509v3 Basic Constraints:
                       CA:FALSE
            Signature Algorithm:  sha256WithRSAEncryption
36:62:35:53:16:b3:ae:b7:ef:06:17:01:12:3b:fe:67:cd:37:
16:ea:0d:2b:1f:b9:98:78:c4:78:94:bc:76:60:ad:9f:cf:18:
1f:a4:8b:d1:4e:0a:64:96:d9:9d:b0:7f:2c:ea:a0:c6:9c:a5:
12:74:a2:14:59:ad:d2:a7:04:46:39:f1:b7:4f:55:c9:18:87:
f5:0b:24:e4:0f:d4:ac:22:05:ef:9f:86:ca:2c:17:cb:dc:f5:
eb:47:a6:51:91:fe:f0:6c:f8:9e:a8:5d:d4:f2:20:71:f7:c4:
38:a5:b5:0d:61:ec:7d:fa:57:92:d3:fd:22:d5:32:67:4b:35:
3d:f7:a1:6c:2a:f0:59:81:9c:b0:58:6c:a5:35:f8:d1:c4:23:
8f:6e:6f:11:b5:c6:1e:ca:45:b2:82:2f:93:95:10:68:8c:f7:
bd:af:2e:b4:fb:4a:37:ee:2f:ea:b1:a2:b8:6d:6e:02:33:ef:
6d:fd:c5:4c:6c:48:57:08:88:06:a2:59:72:ab:d2:2f:d5:c2:
ea:50:7a:36:a9:54:01:98:92:8b:e0:7c:2f:ac:a1:6e:bb:db:
c9:57:42:da:bf:0c:47:00:49:01:1a:c0:6c:32:21:7f:54:2e:
d0:1d:3a:b1:1d:2a:e3:6b:d7:79:45:5e:93:b4:d9:34:11:a6:
16:2c:d3:80
--BEGIN CERTIFICATE--
MIICvDCCAaSgAwIBAgICEAEwDQYJKoZIhvcNAQELBQAwLDETMBEGA1UEChMKdGVz
dGRvbWFpbjEVMBMGA1UEAxMMdGVzdGRvbWFpbkNBMB4XDTE2MDUyNjA4MzMzMFoX
DTE3MDUyNjA4MzMzMFowazETMBEGA1UECgwKdGVzdGRvbWFpbjEtMCsGA1UELhMk
OWI4NjI3ZGMtYmY3Yy00MDM0LWIzMWItMDcxOWY3Yjg1MDE0MSUwIwYDVQQDDBxv
cmcudGVzdGV4YW1wbGUrKysrdGVzdE5vZGUxMFkwEwYHKoZIzj0CAQYIKoZIzj0D
AQcDQgAEWI34R5oLn1t6Xm1eixKpkbJjuXHPCQslDo9GuG8dUPY84s+m7eMDYOL9
snRld/4dbmLP0ysFUnrfO9/IdbsmcqN0MHIwHQYDVR0OBBYEFOMoi7RTFJqmf8dJ
NRdFUn4Toyx/MEYGA1UdIwQ/MD2hMKQuMCwxEzARBgNVBAoTCnRlc3Rkb21haW4x
FTATBgNVBAMTDHRlc3Rkb21haW5DQYIJAMjOdzNuTDwIMAkGA1UdEwQCMAAwDQYJ
KoZIhvcNAQELBQADggEBADZiNVMWs6637wYXARI7/mfNNxbqDSsfuZh4xHiUvHZg
rZ/PGB+ki9FOCmSW2Z2wfyzqoMacpRJ0ohRZrdKnBEY58bdPVckYh/ULJOQP1Kwi
Be+fhsosF8vc9etHplGR/vBs+J6oXdTyIHH3xDiltQ1h7H36V5LT/SLVMmdLNT33
oWwq8FmBnLBYbKU1+NHEI49ubxG1xh7KRbKCL5OVEGiM972vLrT7SjfuL+qxorht
bgIz7239xUxsSFcIiAaiWXKr0i/VwupQejapVAGYkovgfC+soW6728lXQtq/DEcA
SQEawGwyIX9ULtAdOrEdKuNr13lFXpO02TQRphYs04A=
--END CERTIFICATE--
```

Appendix B
Modifications to Calvin

translation.py

```
import json
from calvin.utilities import calvinconfig
from calvin.utilities import calvinlogger
_log = calvinlogger.get_logger(__name__)


class TranslationPolicy:
""" This class reads a translation policy file,
and provides a translation function to a identifier.
"""
        def __init__(self, policy_file):
                try:
                        json_policy=open(policy_file, 'r')
                        self.json_data=json.load(
                           json_policy)
                        json_policy.close()
                        self.id=self.json_data["id"]
                        self.description=self.json_data["
                           description"]
                        self.rules=self.json_data['rules']
                except:
                        _log.exception("Failed opening/
                           loading JSON policy file.\n")

        def get_policy_id(self):
                return self.id

        def get_policy_description(self):
                return self.description

        def translate(self, identifier):
```

```
""" Translates identifier to a new value regarding the
    policy loaded.
"""
                #TODO: check the id format is ok.
                domain=identifier[identifier.find("@")+1:
                    len(identifier)]
                flag=False
                new_id=identifier
                for rule in self.rules:
                        if rule["translation_category"]=="
                            domain" and not flag:
                                for domains in rule["source
                                    "]:
                                        if domains ==
                                            domain:
                                                new_id=rule
                                                    ["
                                                    result"
                                                    ]
                                                flag=True
                                                break
                        elif rule["translation_category"]==
                            "identifier" and not flag:
                                for ids in rule['source']:
                                        if ids==identifier:
                                                new_id=rule
                                                    ["
                                                    result"
                                                    ]
                                                flag=True
                                                break

                        elif rule["translation_category"]==
                            "default" and not flag:
                                new_id=rule["result"]
                                flag=True
                return new_id

        def no_cheating(self, identifier_domain, domain,
            link_category):
""" Returns true if there is no cheating: the coming
    identifier's domain is different from the host domain
"""
                if link_category == "interdomain":
                        if identifier_domain == domain:
                                raise Exception('\nTried to
                                    cheat during the
                                    translation.')
                                return False
                    return True
```

twisted_transport.py

This file is allocated in *calvin/runtime/south/plugins/transports/lib/twisted/*.
The function `check_list` hast been added to the file.

```
_conf = calvinconfig.get()
def check_list(peer):
    """ This function finds a peer in the list. If it is
        found, returns True otherwise False.
        It is used for find an address in a peers domain,
            such that if it is found it signify it belongs
            to the same domain. """
    #The port is eliminated from the calvinip: calvinip
        ://1.1.1.1:80 to calvinip://1.1.1.1
    peer=peer[0:peer.rfind(":")]
    #For security reasons, default is interdomain.
    peers_in_list=_conf.get("security","peers_in_domain")
    found="interdomain"
    if not peers_in_list:
        found=_conf.get("security", "fake_transport")
    else:
        for elements in peers_in_list:
            if peer in peers_in_list:
                found="intradomain"
    return found
```

The class `CalvinTransport` incorporates the following new coded functions:

```
 def __init__(self, rt_id, remote_uri, callbacks, transport
    , proto=None):
        """docstring for __init__"""
        super(CalvinTransport, self).__init__(rt_id,
            remote_uri, callbacks=callbacks)

        self._rt_id = rt_id
        self._remote_rt_id = None
        self._coder = None
        self._transport = transport(self._uri.hostname,
            self._uri.port, callbacks, proto=proto)
        self._rtt = 2000  # Init rt in ms
        #FAKE TLS ~ TRANSPORT
        self._transport_category=check_list(remote_uri)
        # TODO: This should be incoming param
        self._verify_client = lambda x: True
        self._incoming = proto is not None
        if self._incoming:
            # TODO: Set timeout
            # Incomming connection timeout if no join
            self._transport.callback_register("disconnected
                ", CalvinCB(self._disconnected))
            self._transport.callback_register("data",
                CalvinCB(self._data_received))


def set_transport_category(self,category):
        self._transport_category=category

def get_transport_category(self):
        return self._transport_category
```

calvin_network.py

In class CalvinLink, it has been modified:

```
def \_\_init\_\_(self, rt\_id, peer\_id, transport, old\
    _link=None):
        super(CalvinLink, self).\_\_init\_\_()
        self.rt\_id = rt\_id
        self.peer\_id = peer\_id
        self.transport = transport
        # FIXME replies should also be made independent on
            the link object,
        # to handle dying transports losing reply callbacks
        self.replies = old\_link.replies if old\_link else
            {}
        self.replies\_timeout = old\_link.replies\_timeout
            if old\_link else {}
        if old\_link:
            # close old link after a period, since might
                still receive messages on the transport
                layer
            # TODO chose the delay based on RTT instead of
                arbitrary 3 seconds
            \_log.analyze(self.rt\_id, "+ DELAYED LINK
                CLOSE", {})
            async.DelayedCall(3.0, old\_link.close)
        self.transport\_category=None

def set\_transport\_category(self, category):
        self.transport\_category=category
```

```
def get\_transport\_category(self):
        return self.transport\_category
```

In class CalvinNetwork, it has been modified:

```
def join_finished(self, tp_link, peer_id, uri, is_orginator
    ):
        """ Peer join is (not) accepted, called by
            transport plugin.
            This may be initiated by us (is_orginator=True)
                 or by the peer,
            i.e. both nodes get called.
            When inititated by us pending_joins likely have
                 a callback

            tp_link: the transport plugins object for the
                link (have send etc)
            peer_id: the node id we joined
            uri: the uri used for the join
            is_orginator: did this node request the join
                True/False
        """
        # while a link is pending it is the responsibility
            of the transport layer, since
        # higher layers don't have any use for it anyway
        _log.analyze(self.node.id, "+", {'uri': uri, '
            peer_id': peer_id,
                                         'pending_joins':
                                             self.
                                             pending_joins,
                                         '
                                             pending_joins_by_id
                                             ': self.
                                             pending_joins_by_id
                                             },
                                         peer_node_id=
                                             peer_id, tb=
                                             True)
        if tp_link is None:
            # This is a failed join lets send it upwards
            if uri in self.pending_joins:
                cbs = self.pending_joins.pop(uri)
                if cbs:
                    for cb in cbs:
                        cb(status=response.CalvinResponse(
                            response.SERVICE_UNAVAILABLE),
                            uri=uri, peer_node_id=peer_id)
        return
    # Only support for one RT to RT communication link
        per peer
    if peer_id in self.links:
        # Likely simultaneous join requests, use the
            one requested by the node with highest id
        if is_orginator and self.node.id > peer_id:
            # We requested it and we have highest node
                id, hence the one in links is the peer'
                s and we replace it
            _log.analyze(self.node.id, "+ REPLACE
                ORGINATOR", {'uri': uri, 'peer_id':
                peer_id}, peer_node_id=peer_id)
            self.links[peer_id] = CalvinLink(self.node.
                id, peer_id, tp_link, self.links[
                peer_id])
            self.links[peer_id].set_transport_category(
                tp_link.get_transport_category())
        elif is_orginator and self.node.id < peer_id:
            # We requested it and peer have highest
                node id, hence the one in links is peer
                's and we close this new
            _log.analyze(self.node.id, "+ DROP
                ORGINATOR", {'uri': uri, 'peer_id':
                peer_id}, peer_node_id=peer_id)
            tp_link.disconnect()
        elif not is_orginator and self.node.id >
            peer_id:
            # Peer requested it and we have highest
                node id, hence the one in links is ours
                 and we close this new
            _log.analyze(self.node.id, "+ DROP", {'uri'
                : uri, 'peer_id': peer_id},
                peer_node_id=peer_id)
            tp_link.disconnect()
        elif not is_orginator and self.node.id <
            peer_id:
            # Peer requested it and peer have highest
                node id, hence the one in links is ours
                 and we replace it
            _log.analyze(self.node.id, "+ REPLACE", {'
                uri': uri, 'peer_id': peer_id},
                peer_node_id=peer_id)
            self.links[peer_id] = CalvinLink(self.node.
                id, peer_id, tp_link, self.f[peer_id])
            self.links[peer_id].set_transport_category(
                tp_link.get_transport_category())
        else:
            # No simultaneous join detected, just add the
                link
            _log.analyze(self.node.id, "+ INSERT", {'uri':
                uri, 'peer_id': peer_id}, peer_node_id=
                peer_id, tb=True)
            self.links[peer_id] = CalvinLink(self.node.id,
                peer_id, tp_link)
            self.links[peer_id].set_transport_category(
                tp_link.get_transport_category())

        # Find and call any callbacks registered for the
            uri or peer id
        _log.debug("%s: peer_id: %s, uri: %s\
            npending_joins_by_id: %s\npending_joins: %s" %
            (self.node.id, peer_id, uri, self.
            pending_joins_by_id, self.pending_joins))
        if peer_id in self.pending_joins_by_id:
            peer_uri = self.pending_joins_by_id.pop(peer_id
                )
            if peer_uri in self.pending_joins:
                cbs = self.pending_joins.pop(peer_uri)
                if cbs:
                    for cb in cbs:
                        cb(status=response.CalvinResponse(
                            True), uri=peer_uri,
                            peer_node_id=peer_id)

        if uri in self.pending_joins:
            cbs = self.pending_joins.pop(uri)
            if cbs:
                for cb in cbs:
                    cb(status=response.CalvinResponse(True)
                        , uri=uri, peer_node_id=peer_id)

        return
```

calvin_proto.py

In class *CalvinProto,* it has been modified:

```
def actor_new_handler(self, payload):
        """ Peer request new actor with state and
            connections """
        _log.analyze(self.rt_id, "+", payload, tb=True)
        #PASSES UP THE CATEGORY
        kwargs = {}
        kwargs['link_category'] = self.network.links[
            payload['from_rt_uuid']].get_transport_category
            ()
        self.node.am.new(payload['state']['actor_type'],
            None, payload['state']['actor_state'], payload[
            'state']['prev_connections'], callback=
            CalvinCB(self._actor_new_handler, payload), **
            kwargs)
```

actormanager.py

In class *ActorManager*, it has been modified:

```
def __init__(self, node):
        super(ActorManager, self).__init__()
        self.actors = {}
        self.node = node
        if _conf:
            self.migration_policy = translation.
                TranslationPolicy(_conf.get("security","
                security_conf")['translation']['
                policy_storage_file'])

def new(self, actor_type, args, state=None,
    prev_connections=None, connection_list=None, callback=
    None,
            signature=None, credentials=None, link_category
                =None):
        """
        Instantiate an actor of type 'actor_type'.
            Parameters are passed in 'args',
        'name' is an optional parameter in 'args',
            specifying a human readable name.
        Returns actor id on success and raises an exception
             if anything goes wrong.
        Optionally applies a serialized state to the actor,
             the supplied args are ignored and args from
            state
        is used instead.
        Optionally reconnecting the ports, using either
          1) an unmodified connections structure obtained
              by the connections command supplied as
              prev_connections or,
          2) a mangled list of tuples with (in_node_id,
              in_port_id, out_node_id, out_port_id)
              supplied as
    connection_list
"""
_log.debug("class: %s args: %s state: %s, signature
    : %s" % (actor_type, args, state, signature))
_log.analyze(self.node.id, "+", {'actor_type':
    actor_type, 'state': state})
try:
    if state:
        a = self._new_from_state(actor_type, state,
            link_category)
    else:
        a = self._new(actor_type, args, credentials
            )
except Exception as e:
    _log.exception("Actor creation failed")
    raise(e)

# Store the actor signature to enable GlobalStore
    lookup
a.signature_set(signature)

self.actors[a.id] = a

self.node.storage.add_actor(a, self.node.id)

if prev_connections:
    # Convert prev_connections to connection_list
        format
    connection_list = self.
        _prev_connections_to_connection_list(
        prev_connections)

self.node.control.log_actor_new(a.id, a.name,
    actor_type, isinstance(a, ShadowActor))

if connection_list:
    # Migrated actor
    self.connect(a.id, connection_list, callback=
        callback)
else:
    # Nothing to connect then we are OK
    if callback:
        callback(status=response.CalvinResponse(
            True), actor_id=a.id)
    else:
        return a.id
```

```
def _new_from_state(self, actor_type, state, link_category)
    :
        """Return a restored actor in PENDING state, raises
            an exception on failure."""
        try:
            _log.analyze(self.node.id, "+", state)
            credentials = state.pop('credentials', None)
            try:
                state['_managed'].remove('credentials')
            except:
                pass
            if link_category=="interdomain" and credentials
                :
                user=credentials['user']
                if self.migration_policy.no_cheating(user[
                    user.find("@")+1:len(user)], _conf.get(
                    "security", "domain"), link_category):
                     credentials['user'] = self.
                        migration_policy.translate(user)
                     _log.info("Translation committed: %s to
                        %s " % (user, credentials['user'])
                        )
            a = self._new_actor(actor_type, actor_id=state[
                'id'], credentials=credentials)
            if '_shadow_args' in state:
                # We were a shadow, do a full init
                args = state.pop('_shadow_args')
                state['_managed'].remove('_shadow_args')
                a.init(**args)
                # If still shadow don't call did_migrate
                did_migrate = isinstance(a, ShadowActor)
            else:
                did_migrate = True
            # Always do a set_state for the port's state
            a._set_state(state)
            self.node.pm.add_ports_of_actor(a)
            if did_migrate:
                a.did_migrate()
            a.setup_complete()
        except Exception as e:
            _log.exception("Catched new from state %s %s" %
                (a, dir(a)))
            raise(e)
        return a
```