Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

June 17, 2016

Online Data Center Modelling

INDaaS

Daniel Rivera

Abstract

Data centres are huge entities of interwoven parts in multiple layers, combining network, hardware, and software components. Finding links and dependencies between the layers quickly becomes a main issue, knowing them is crucial in order to find the most vulnerable parts of it. However, through INDaaS, easy to collect information analysed correctly can make this difficult task a lot easier. With the dependency information at hand and visualisable, finding out weak spots in the form of risk groups, makes preventing service outages a much more manageable task.

Advisor
Prof. Robert Soulé

Co-Advisor
Prof. Antonio Carzaniga

Assistant
Daniele Rogora

Advisor's approval (Prof. Robert Soulé): Date:

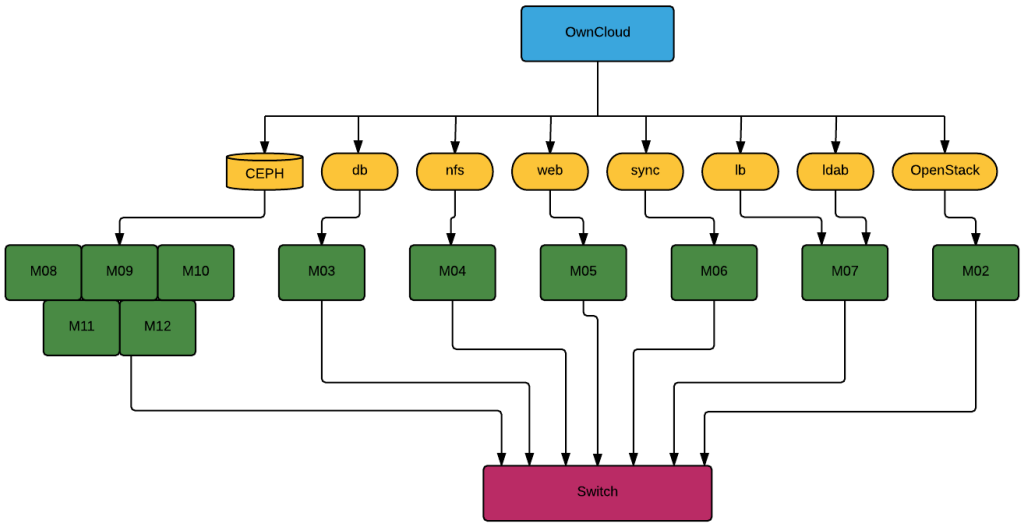
Contents

1 Introduction

I believe it is needless to say that reliability is one of the key factors in today's systems. Focusing on Data Centres, they have become behemoths of interrelated parts in multiple layers, being software, hardware and network. With such a kind of system, it is not uncommon for there to be unknown dependencies between parts, causing individual, or small groups of elements to be crucial to the system, in the sense that if said elements fail, the whole system breaks down. In order to deal with this problem, one must first find out it exists.
A possible solution for the above is INDaaS[7], or Independence as a Service, a proposed architecture to audit the independence of redundant systems, which we will explore throughout this report.
It is worth mentioning that this is only a part of the greater effort to create a general model of Data Centres, to remove their "black-box" attributes, focusing in reliability.

1.1 Data Centre and Service

Naturally, we need a Data Centre to take as an initial model, for this we previously recreated, in a joint effort with ETHZ, a smaller version of SWITCH's data centre and named it Mozart. This will act as our main reference for our model. Mozart is composed of 12 machines, which host an OpenStack implementation, with Ceph as storage.
We will be recreating SWITCH's cloud storage service, with ownCloud, as the main service on which to test these tools. I would like to take this opportunity to thank Daniele Rogora for setting this system up from scratch, for my entire project depended on it.
The following figure shows the topology of our data centre in a very simplistic manner, displaying the different services that ownCloud depends on, and the machines where they are.



1.2 Independence-as-a-Service (INDaaS)

INDaaS solution is to collect and audit structural dependency information, then uses it to evaluate the independence of it, one can then do with that information what he pleases. It achieves this through two elements: the collection is handled by dependency collectors, and the auditing part is handled by two different possible modules, a standard one and a privacy maintaining one, which we will disregard throughout this report.

1.2.1 Dependency collectors

Referred to as "dependency acquisition modules", the first part of the INDaaS architecture focuses con collecting all the necessary information to run analyses on. Each of these modules focus on a different layer of the system:

- **NSDMiner**: NSDMiner[3] is in charge of collecting the network dependencies. A network dependency outlines the path between a source and a destination, and the different components it encounters in the way.
- **HardwareLister**: Dealing with physical components, HardwareLister[6] is the module currently used to collect hardware dependencies.

- **apt-rdepends**: apt-rdepends[1] handles software components, closely relate to the hardware dependencies, for a key part of it is to know in which piece of hardware it runs.

1.2.2 Structural Independence Auditing (SIA)

From here on referred to as SIA, this protocol is tasked with generating the dependancy graph, determining risk groups (later mentioned), raking them, and generating the report. We will revisit this throughout the report.

A more detailed description of INDaaS and it's service is found in the next section.

1.3 Motivation

As the paper that influenced this project very coherently puts: "Today's systems pervasively rely on redundancy to ensure reliability. In complex multi-layered hardware/software stacks, however - especially in the clouds where many independent businesses deploy interacting services on common infrastructure - seemingly independent systems may share deep, hidden dependencies, undermining redundancy efforts and introducing unanticipated correlated failures." [7]. This is clearly a main issue in a lot of systems, not solving it can cause both partial and complete service outages[2].

1.4 Goals

The main goals of this project are 4:

1. Deployment of INDaaS on Mozart.
2. Collect measurements to identify correlated failures in the data centre.
3. Evaluation of the INDaaS tools on an actual deployment.
4. Optional improvement of the tools.

In short, understanding dependencies in order to have a better understanding of the system and thus take measures to prevent previously mentioned service outages is not the only goal of this project, we also analyse the usability of the INDaaS architecture in a general environment.

2 INDaaS

Developed by members of Yale University and Bell Labs/Alcatel-Lucent, INDaaS is a proposed architecture to audit the independence of redundant systems proactively. In an environment that requires high reliability, redundancy is often the go-to solution to improve a system's reliability, however, given the complicated and interrelated nature of such big systems, such as cloud services, the dependancies between different pieces of it are not always obvious. It is these hidden dependencies that go unnoticed that are often the cause of unexpected correlated failures, which the redundancy efforts did not take into account.
To better understand this, imagine you have a service that relies on two independent redundant servers in order to avoid service outages (one server might fail, yet your service will still work, giving you time to fix the issue in the failed server), however, the fact that both servers shared a common aggregation switch went unnoticed, and when the switch failed, your whole service fails. Finding this unexpected risk is the aim of INDaaS, in order to adequately deal with it before it can fail.
In the previous example, the group of servers was a "risk group" (RG), a set of components whose simultaneous failures could cause a service outage. The objective was to have all risk groups be of size 2, giving you enough room to deal with individual failures. The switch was an unexpected risk group of size 1, meaning that its individual failure brings the entire service down.
INDaaS solution to finding and dealing with correlation failures focusing on collecting and auditing structural dependency information, then evaluating the independence of its elements, and giving you the information in order to deal with them accordingly.

2.1 Dependency Acquisition

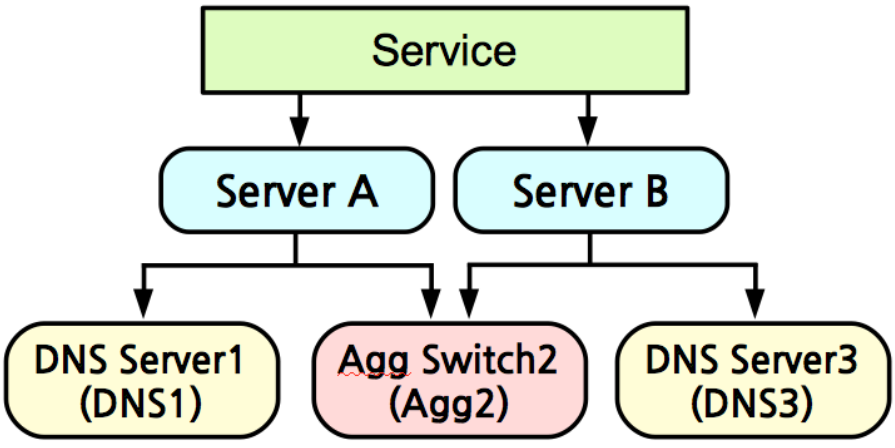
As stated in the original INDaaS paper: "Acquiring accurate structural dependency data within heterogeneous cloud systems is non-trivial, and realistic solutions would need to be adapted to different cloud environments.", to deal with this, INDaaS proposes using pluggable dependency acquisition modules, and using a uniform XML format that can be then directly inserted into the SIA protocol.
Format defintion:

Type	Dependency Expression
Network	`<src="S" dst="D" route="x,y,z"/>`
Hardware	`<hw="H" type="T" dep="x"/>`
Software	`<pgm="S" hw="H" dep="x,y,z"/>`

In the original prototype, they used the previously mentioned modules (NSDMiner, apt-rdepends, HardwareLister) to gather the information and then parse it into the required format.
NSDMiner collects data through analysing network traffic flows from individual packets or network devices. HardwareLister gathers each machines hardware configuration. Lastly, we use apt-rdepends to extract software packages and library dependencies. Note that while these are used both for their prototype and my implementation, the idea is to be able to use a wide range of different tools to collect dependency information, and then parse it to the input expected by SIA.

2.2 Structural Independence Auditing

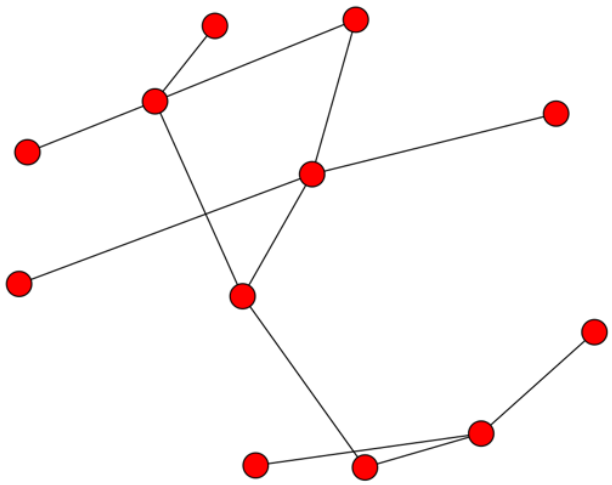
Once the dependency data has been acquired and parsed, SIA handles the auditing of it, to generate the desired dependency graph, generate risk groups, and rank said risk groups. Please refer to the paper previously cited by Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford, "Heading Off Correlated Failures through Independence-as-a-Serve" for an in-depth explanation on how it achieves this.
To demonstrate a simple case, assume you have the following topology:



Here a service is replicated in 2 separate redundant servers, Server A and Server B, and each of them relies on a separate DNS server, and have a common aggregate switch. After gathering the dependancy information of this system, parsing it and using it as input for SIA, SIA would output the following risk groups:

[['Agg2'], ['DNS3', 'DNS1']]

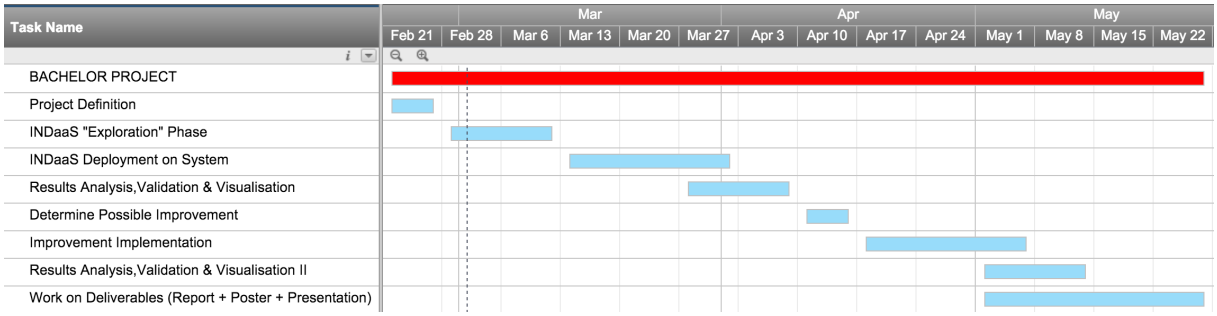
While probably obvious in this example, this tells us that either the failure of the aggregate switch, or of both DNS servers would cause a service outage.
For completion, it would also output a simple graph similar to the following one (improving the expressiveness of this graph was one of the factors improved).



3 Project Plan

3.1 Initial Plan

My initial plan looked like this:



- INDaaS "Exploration" Phase: A 2 week period to go through the existing code of the INDaaS architecture, understanding how it works, all of its different elements and how they relate to each other.
- INDaaS Deployment on System: After understanding it, I scheduled the actual deployment of it on Mozart would take 3 weeks, taking into account all the possible issues and hiccups I might find on the way.
- Results Analysis, Validation & Visualisation: This 2 week period should be enough to see the output of INDaaS, determine wether it's correct or not and find a way to display them properly.
- Determine Possible Improvement: The final goal of my project was not to only deploy and use this tool on Mozart, but to also extend it in a way that I saw fit. This decision was to be made in this time period.
- Improvement Implementation: Self explanatory, extend the tools with the previously decided improvement.
- Results Analysis, Validation & Visualisation II: Same as Results Analysis, Validation & Visualisation, but taking into account my added feature.
- Work on Deliverables: Report and poster writing, as well as the preparation of the poster presentation.

3.2 Revised Plan

Naturally, not everything went as planned, some general comments about the changes made to each of the phases:

- INDaaS "Exploration" Phase: This phase took a lot more time than expected, I would say it was ongoing throughout the whole project, when I found out things that were not obvious at first, through conversations with the original author of the code. One of the causes for this was the lack of documentation within the project.

- INDaaS Deployment on System: As with the previous point, the initial deployment in Mozart took more than it was foreseen, mostly due to the previous phase as well. From this point on the project was behind schedule by some weeks.

- Results Analysis, Validation & Visualisation: Because of the previous 2 points, this phase occurred at a later time, but it went as expected.

- Determine Possible Improvement: Same as before. The improvement decided was the implementation of probabilities and propagation throughout the fault graph, as well as a neater graphical display of the output.

- Improvement Implementation: Implementation went as planned, but with some hiccups.

- Results Analysis, Validation & Visualisation II: With the improved visual interfaced, this phase was somewhat merged with the "Improvement Implementation" phase.

- Work on Deliverables: Only change was the delay mentioned before.

4 Development

4.1 Development process

From the very start, it was decided that the development process would be closely monitured by both my advisor, Professor Robert Soulé, and my co-advisor, Professor Antonio Carzaniga, as well as with Daniele Rogora. We held weekly meetings where we discussed progress, tried to solve existing issues, and planned the best way to move forward with the project. These meetings proved to be very useful, and went on from the very first week up until the final week of development.

I will separate this section according to the points discussed in my project plan, thus following a chronologically linear story, and will outline the process, challenges and solutions found throughout.

4.2 INDaaS Exploration Phase

The exploration phase mainly involved me going through the original INDaaS code (again, excluding PIA), in order to be able to properly deploy it in Mozart. Understanding INDaaS was by far the most time consuming part of this project, for two main reasons:

- Lack of documentation: Understanding someone else's code is already a feat sometimes, however the issue was not the code structure, but the lack of any actual documentation of the project as a whole, the relationship between different components as well as within the individual files. While some files do have a brief description of what it does in general, I spent hours, if not days, figuring out how they all related to one another, what they depended on and sometimes wether they were even used. The author of the project agrees, and mentioned that it was an issue he was intending to work on during the summer, however I had to deal with the project without it. However I must say he was always helpful whenever I was truly lost with a piece of the code.

- Missing parsers: The parser of some dependencies (from dependancy data to XML for SIA) had been lost, which meant that I could not properly see connections between the input to SIA and the data collectors.

Additionally, I also had to analyse and understand other elements, more specifically, the dependancy acquisition tools used, for they were standalone elements not developed by them. While apt-rdepends and HardwareLister are quite common, and thus had plenty of documentation available, NSDMiner is not, and it also took a long time to understand.

Overall, this became a task that I never truly finished, finding out things about INDaaS that I initially missed at many points during later stages of development.

4.3 INDaaS Deployment on System

This was the first concrete goal of my project: to get INDaaS working on Mozart, and it being able to analyse properly Mozart's topology and dependencies, as well as hopefully showing us hidden ones that we were not aware of. Given the nature of INDaaS, this task can be divided in 2: the deployment of the collection tools in the system, and of SIA. Starting with the dependancy collection tools, the progress deploying them very closely mimics the previous exploration phase; HardwareLister and apt-rdepends had many resources I could use, thus their deployment was more than trivial, NSDMiner's resources were scarce and vague. Setting it up involved me going back and forth between figuring out what exactly does and how it does it, with a hint of playing in the dark, for I had not a real idea of the kind of output I was looking for.
To properly collect the information, I monitored the system while my colleague Emmanuele Sotta ran his performance tests on it, and I listened and collected pcaps from different machines, subsequently transformed said pcap files into flows, which I fed to NSDMiner.After spending more time that I should have on this I obtained an output that I could not make much sense of, but had the format I should expect from it. As I just mentioned, getting an output from the first HardwareLister and apt-rdepends was not an issue.
It was at this point I faced the biggest challenge I faced throughout this entire project: linking the dependency collector tools' output to SIA's XML format in order to obtain meaningful results. While I could understand what the collectors output meant, it was hard linking it to the available examples of input that I had for SIA, and even when I could see how I could transform it into it, it would not give me the results that I expected. The previously mentioned lack of the parsers played a crucial role in this, for I was without hints on wether I was missing something obvious or not, which I could have realised analysing them, unfortunately I was unable to.
In other words, while parsing the output into the XML format expected by INDaaS, I could not see the relation between the output and any meaningful input I could give it.
For reference, NSDMiner's output was of the following format:

```
Flows from 1459942201.81 to 1459942405.94 (10588 flows) -
10.0.0.42:80:TCP 100
    10.0.0.91                  5432  TCP        79.100
    195.186.1.101               123  UDP         0.000
    82.197.164.46               123  UDP         0.000
    195.186.1.100               123  UDP         0.000

10.0.0.51:443:TCP 100
    10.0.0.45                    80  TCP       100.000
    10.0.0.42                    80  TCP       100.000
    10.0.0.33                 10000  TCP        54.048
    195.186.1.100               123  UDP         0.000
    46.235.147.16               123  UDP         0.000
    212.51.144.44               123  UDP         0.000

10.0.0.91:5432:TCP 100
    195.186.4.100               123  UDP         0.000

10.111.1.67:22:TCP 100
    10.111.1.63                6789  TCP       100.000
    10.111.1.65                6789  TCP       100.000
    10.111.1.64                6789  TCP       100.000

129.132.12.13:4789:UDP 100
    129.132.12.2               9696  TCP        55.998
    129.132.12.22              6804  TCP        16.677
    129.132.12.3               4789  UDP        10.522
    129.132.12.28              6800  TCP        10.522
    129.132.12.20              6804  TCP         0.000
    129.132.12.14              6789  TCP         0.000
```

This shows a set of flows from some IP addresses to their destinations, and intermediary stops. One could see this as the network dependencies by parsing the path. However the set was often incomplete and changed with different

tests, and lacked many other network information that perhaps is not available through capturing flows, the reason is unknown to me, but I could not reliably use it to test INDaas.
I solved this by hard coding the topology information of Mozart that I had at hand, while less than optimal, this allowed me to move forward with the project, for I had wasted to much time trying to figure out what was happening and why.

On the other hand, deploying SIA was trivial, for as long as you format the input in the XML-format it expects as input, it will work. An example file it would take as input would be:

```
<node id="ownCloud"><gate>OR</gate><dep>Ceph</dep><dep>db</dep><dep>nfs</dep><dep>
web</dep><dep>sync</dep><dep>lb</dep><dep>ldap</dep><dep>OpenStack</dep><dep>
Network</dep></node>
<node id="Ceph"><gate>AND</gate><dep>M08</dep><dep>M09</dep><dep>M10</dep><dep>M11<
/dep><dep>M12</dep></node>
<node id="OpenStack"><gate>AND</gate><dep>M02</dep></node>
<node id="db"><gate>AND</gate><dep>M03</dep><dep>M04</dep></node>
<node id="nfs"><gate>AND</gate><dep>M03</dep><dep>M04</dep></node>
<node id="web"><gate>AND</gate><dep>M04</dep><dep>M05</dep></node>
<node id="sync"><gate>AND</gate><dep>M06</dep><dep>M05</dep></node>
<node id="lb"><gate>AND</gate><dep>M07</dep><dep>M06</dep></node>
<node id="ldap"><gate>AND</gate><dep>M07</dep><dep>M06</dep></node>
<node id="Network"><gate>AND</gate><dep>Switch1</dep><dep>Switch2</dep></node>
</list>
```

Where we list the elements of the system, along with the services they depend on and the machines where they are deployed in. One can tell there are 2 different kinds of gates: AND and OR:

- **AND** The node will only fail if ALL of its elements in its dependency list fails.
- **OR** The node will fail if ANY of its elements in its dependency list fails.

From this configuration, the resulting lists of risk groups is

[[’M02’], [’M04’, ’M03’], [’M04’, ’M05’], [’M06’, ’M05’], [’M06’, ’M07’], [’Switch2’, ’Switch1’], [’M11’, ’M10’, ’M12’, ’M08’, ’M09’]]

Accompanied by a graph almost identical to the one shown before.

While I mentioned that the deployment of SIA itself was trivial, I should clarify that I had numerous issues with different colliding library issues and such sort of implementation issues, but I assumed those are common in every project.

4.4 Results Analysis, Validation & Visualisation

As previously said, I hardcoded some of the input to the SIA tool, therefore I will not focus on the dependency collectors, for I believe the analysis and validation of them was a middle step in the previous section.
Once I had everything up and running with SIA, and ran it with the necessary input in the expected format, it worked like a charm. It outputted the risk groups mentioned above, as well as the graph.
While not very explicit, the potential to make more of it was there, but this was only a proof-of-concept prototype. The next step was to decide in which direction I wished to extend it.

4.5 Determine Possible Improvement

Once I reached this section I discussed with my advisors what direction we could take from here on. Some improvements proposed were:

- Go back and try to get the dependency collectors information properly parsed in order for it to not be mostly hardcoded.
  Of course, the major hiccup and issue I had was the linking between the 2 main parts of INDaaS, fixing this issue was extremely tempting. However we opted not to take this direction since I had already spent weeks trying to fix this with no real progress at hand, which really hurt morale and delayed everything.
- Improve the visual representation of the analysis.
  As we visited before, the graph outputted from the original prototype did not give too much information (if any), improving this part of the project would give it a friendlier look and make the output easier to understand.

- Add failure probabilities.
  If one were to add the probability of an element failing according to its kind, we could better quantify risk groups and give a total chance of failure to the entire system. Taking into account not only the size of the risk group but of how likely its elements are to fail.

- Real time processing of data collected by collectors.
  The most ambitious one initially proposed, this would entail being able to constantly run the collectors and see in real time changes made to the system. However, this entails the first improvement proposed in this list to be also implemented.

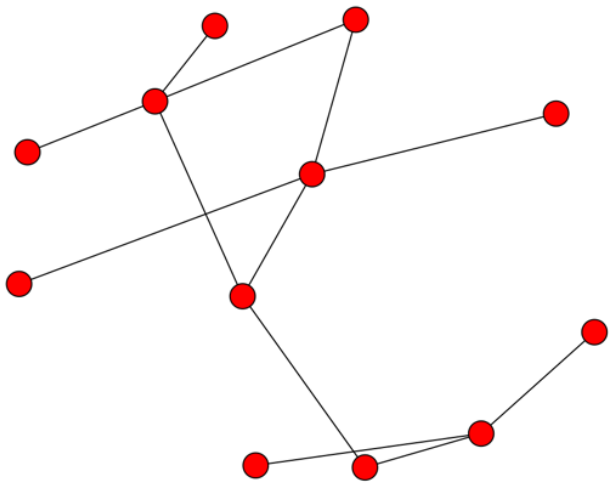
At the end, we decided to do both the second and third improvements of the list: improve visual representation of graph,

4.6 Improvement Implementation

As I just mentioned, we decided to improve INDaaS through a more explicit visual representation of the system, as well as adding probabilities of failure into the mix. This was done in that order, and at the end one worked thanks to the other.

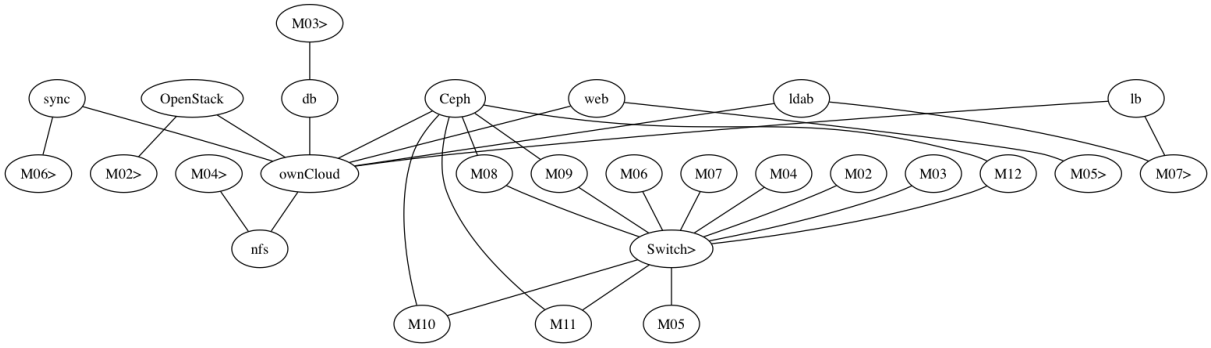
4.6.1 Visual Representation

To recap, the original graph displayed by INDaaS looked something like this:



One could not infer much from it, therefore this improvement was, in my opinion, the most important and interesting one.
The original graph was made using NetworkX[4], and it displayed all the nodes unlabelled and without any apparent order. Initially I worked on improving the graphical interface through NetworkX as well, exploiting its capabilities, however the more I researched I seemed to find a consensus that Pygraph[5] might be better for this, which was great for I have used Pygraph beforehand.

Moving from networkX to Pygraph proved to be more problematic than expected, since SIA built the networkX graph, I decided it was easier to transform the existing graph into a Pygraph graph, rather than figuring out how the undocumented code works and modifying it to build a Pygraph directly. I had a compatibility issue because of lacking information in the original graph which I solved by modifying the Pygraph library. Afterwards it went smoothly.
My initial attempts with it gave me something of the sort:



Which even though it's labeled, it is definitely not clearer than the previous one, still unordered and without much else information.
I further improved it by adding a proper hierarchical structure to the graph, as well as colours depending on wether or not they are part of a risk group. The final result is shown in subsequent sections.

4.6.2 Probabilities

There were multiple ways to implement probabilities into the mix. Since I had experience working with Pygraph and had just finished the visual improvement, I decided face the problem directly from the graph's perspective.
My steps to do this follow:

1. Give each element a probability of failure
   For this project I gave every element the same probability of failure, however with enough research one could give different probability of failures depending on what kind of "thing" it is, be it a switch, a machine, a rack, a plug, anything. I assigned for default a 1% chance of failure for everything.

2. Assign the probability of failure of each element to its node.

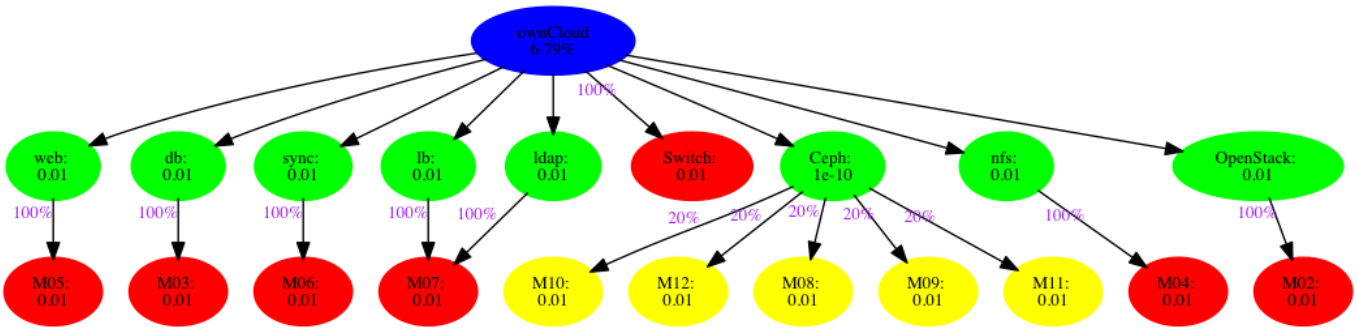
3. Run SIA with the normal input.

4. With the graph built by SIA containing the probability of failure of each node, I traverse it bottom up and propagate the probabilities, taking into account the kind of gate that it has (AND or OR, mentioned before).

5. Once all probabilities propagate to the root (service), update the nodes, as well as the risk groups previously obtained, to also account for probabilities.

6. Display the graph.

The final result implementing both implementations gave us a graph looking like this:



Regarding the colouring: in **Blue** is the root, or the service provided, in **Green** are elements not in a risk group, in **Yellow** you find elements belonging to risk groups with size greater than 1, and in **Red** are all elements whose individual failure would break the entire system. This is also supported by a textual more explicit breakdown of the risk groups in text format, for this graph it would be the following:

```
[['Switch'], ['M03'], ['M04'], ['M05'], ['M06'], ['M07'], ['M02'], ['M11', 'M10', 'M12', 'M08', 'M09']]
Individual probabilities for each risk group:
{'Switch': 0.01}
{'M03': 0.01}
{'M04': 0.01}
{'M05': 0.01}
{'M06': 0.01}
{'M07': 0.01}
{'M02': 0.01}
{'M11,M10,M12,M08,M09,': 1.0000000000000002e-10}
Probability of a service outtage (any risk group failing):
6.79%
```

4.7 Results Analysis, Validation & Visualisation II

As with the first analysis, validation and visualisation phase, I worked closely with the implementation of it, for it was all about the graphical output. The risk, both the total and the individual ones seem to work properly as well. The visualisation part ended in a way I am happy with, with the graph being informative and the complementary text providing some more information.
The results of both results align with what I know from our setup, which is not a surprise since I hardcoded the input, but it would be worrisome if it did not.

4.8 Work on Deliverables

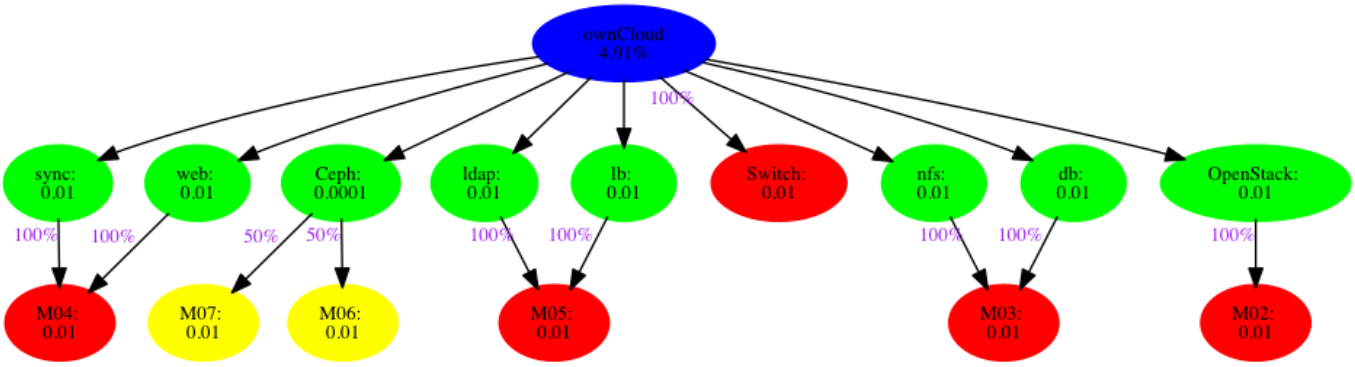
The final stage of the project, it has gone according to plan, of course given the delay caused by the first 2 phases I have dedicated extra time not accounted for in my original plan. The deliverables for my project are this report, an accompanying poster, as well as the extended INDaaS code.

5 Final Result Example

I have tested the reliability of SIA with different system topologies, in the form of the evolution of a system in order to showcase how changes to the topology modify both the graphical and textual output. The progression follows:

5.1 Stage 1

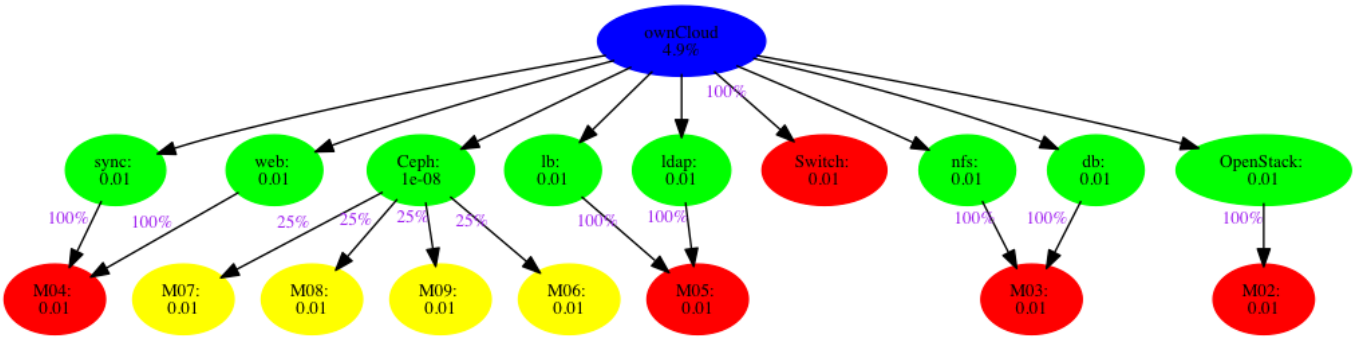
Initial configuration, shows a system with almost no redundancy, and with multiple services depending on common machines. Total risk of failure is 4.91%




```
[['M03'], ['M04'], ['M05'], ['Switch'], ['M02'], ['M06', 'M07']]
Individual probabilities for each risk group:
{'M03': 0.01}
{'M04': 0.01}
{'M05': 0.01}
{'Switch': 0.01}
{'M02': 0.01}
{'M06,M07,': 0.0001}
Probability of a service outtage (any risk group failing):
4.91%
```

5.2 Stage 2

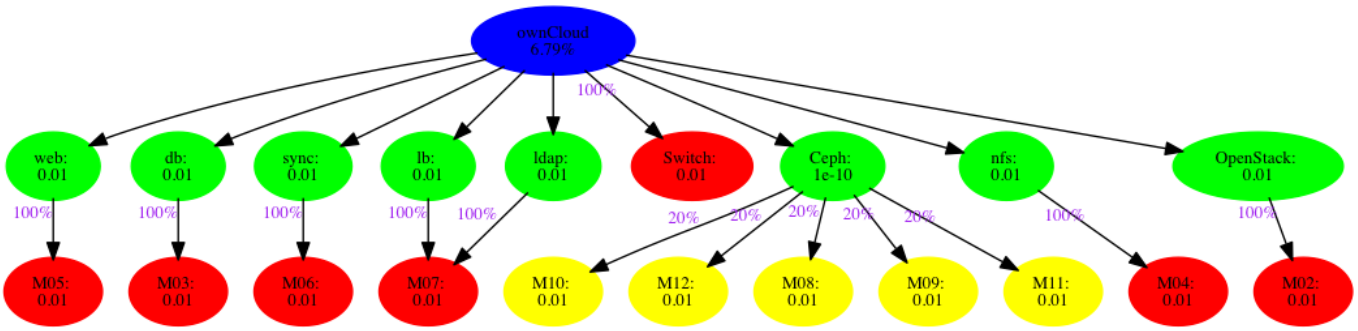
Here we have only added more redundancy to an already redundant system: Ceph, which handles storage in our Mozart cluster. This measure has only reduced the risk by 0.01%, this showcases how adding redundancy where it is not necessary won't change much.




```
[['M03'], ['M04'], ['M05'], ['Switch'], ['M02'], ['M06', 'M07', 'M08', 'M09']]
Individual probabilities for each risk group:
{'M03': 0.01}
{'M04': 0.01}
{'M05': 0.01}
{'Switch': 0.01}
{'M02': 0.01}
{'M06,M07,M08,M09,': 1.0000000000000002e-08}
Probability of a service outtage (any risk group failing):
4.90%
```

5.3 Stage 3

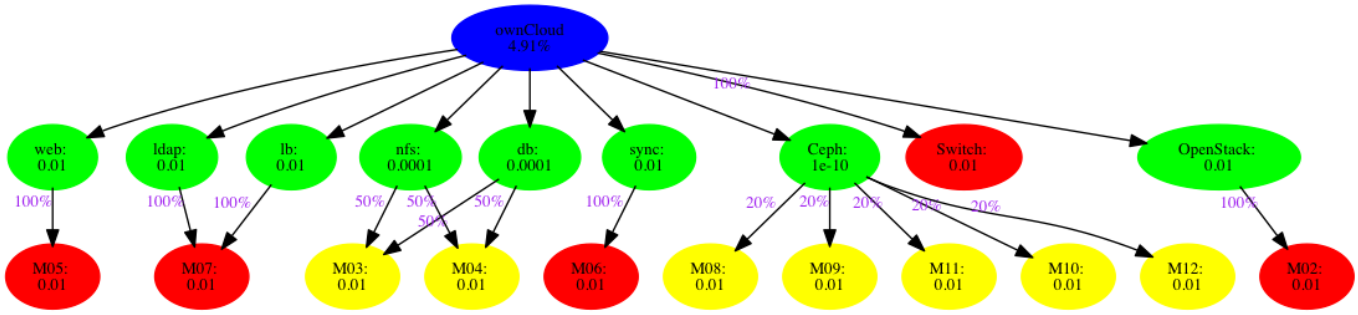
By removing multiple services relying on common machines, we have actually made things worse (by adding more things that could fail). The probability of failure has now increased to a 6.79%. It is worth noting that this is the actual state of our cluster.




```
[['Switch'], ['M03'], ['M04'], ['M05'], ['M06'], ['M07'], ['M02'], ['M11', 'M10', 'M12', 'M08', 'M09']]
Individual probabilities for each risk group:
{'Switch': 0.01}
{'M03': 0.01}
{'M04': 0.01}
{'M05': 0.01}
{'M06': 0.01}
{'M07': 0.01}
{'M02': 0.01}
{'M11,M10,M12,M08,M09,': 1.0000000000000002e-10}
Probability of a service outtage (any risk group failing):
6.79%
```

5.4 Stage 4

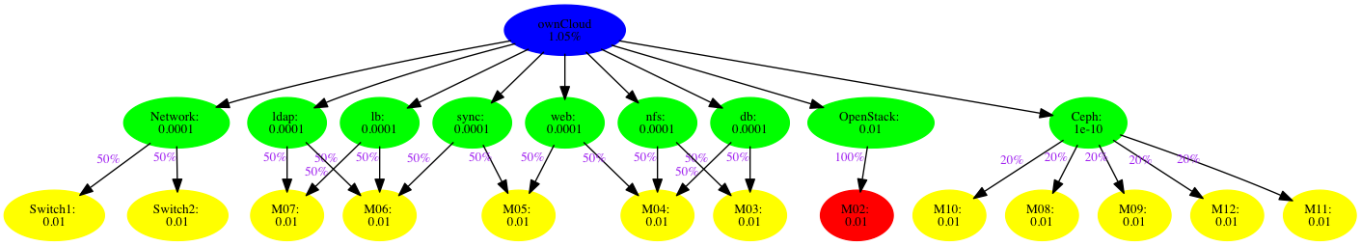
Without changing the number of machines, but duplicating services and adding redundancy within services, we have gone back to our initial value in stage 0.




```
[['Switch'], ['M05'], ['M06'], ['M07'], ['M02'], ['M04', 'M03'], ['M11', 'M10', 'M12', 'M08', 'M09']]
Individual probabilities for each risk group:
{'Switch': 0.01}
{'M05': 0.01}
{'M06': 0.01}
{'M07': 0.01}
{'M02': 0.01}
{'M04,M03,': 0.0001}
{'M11,M10,M12,M08,M09,': 1.0000000000000002e-10}
Probability of a service outtage (any risk group failing):
4.91%
```

5.5 Stage 5

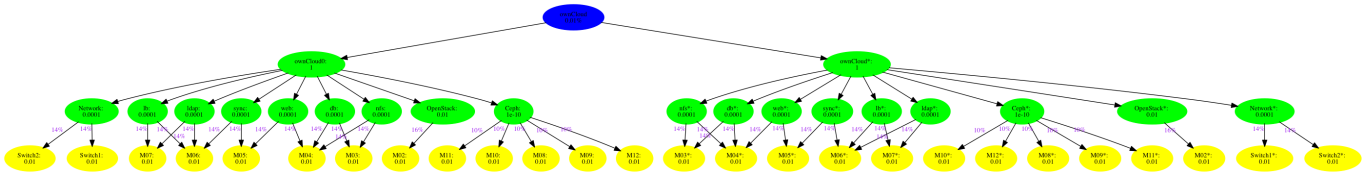
By adding even more redundancy within the existing machines (but no new ones) as well as relying in more than a single switch, we have completely eliminated single point of failures (a single machine crashing causing a service outage), and brought down the risk of failure to a more comfortable 1.05%.

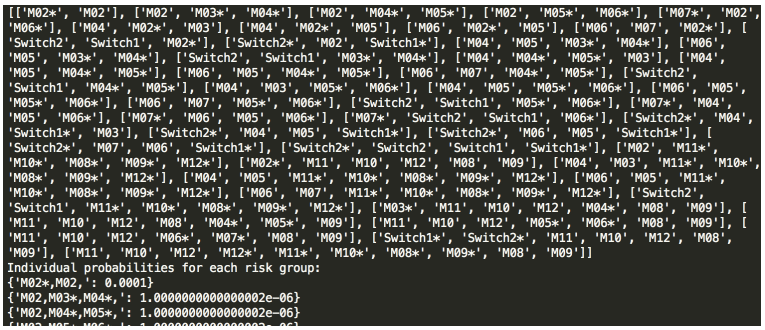



```
[['M02'], ['M04', 'M03'], ['M04', 'M05'], ['M06', 'M05'], ['M06', 'M07'], ['Switch2', 'Switch1'], ['M11',
'M10', 'M12', 'M08', 'M09']]
Individual probabilities for each risk group:
{'M02': 0.01}
{'M04,M03,': 0.0001}
{'M04,M05,': 0.0001}
{'M06,M05,': 0.0001}
{'M06,M07,': 0.0001}
{'Switch2,Switch1,': 0.0001}
{'M11,M10,M12,M08,M09,': 1.0000000000000002e-10}
Probability of a service outtage (any risk group failing):
1.05%
```

5.6 Stage 6

Mostly for testing, by completely duplicating the system, we have completely eliminated single point of failures, as well as reduced the risk of failure to 0.01%.





5.7 Disclaimer

While from Stage 3 on it gets more and more reliable, this in no way takes into account performance or cost effectiveness.

6 Conclusion

As we can tell from the final example, the results of the analysis from SIA appear to be quite reliable, matching what we know from our data centre and common sense, but this is not the point of the tool. One can see how with more dependency information collected through the dependency acquisition modules, not only will it be more descriptive of the data centre audited, but will also show unknown correlations of every type proactively, allowing the time to fix them before they are found out about after they cause a service outage.

6.1 Possible Future Work

- Documentation: As mentioned in various parts of this report, I believe that most of the challenges I experienced were partly caused by a lack of documentation, fortunately the author of the code has expressed to me his interested on doing this in the near future.
- Generalisation: INDaaS is currently in a early stage. As far as I am aware of, it has only been implemented in Yale and in our Mozart cluster, however its modular nature will hopefully allow for it to be deployed almost anywhere, reducing the amount of failures due to unknown correlations and dependancies significantly.
- Parsers: As I experienced, in order for the previous point to be achieved, a clear connection between the dependency acquisition modules and the auditing tools (both SIA and PIA) should be available.

6.2 Acknowledgements

I want to thank both my advisor Prof. Robert Soulé and Co-advisor Prof. Antonio Carzaniga, as well as my assistant Daniele Rogora, for giving so much time and being so helpful throughout this project, enduring weekly meetings both with me and colleague Sotta and constantly helping us find the right direction. And for letting us partake in the exciting project along such renowned institutions as ETHZ and Yale are. It was truly a unique experience.
As well as to my classmates, who had to endure my bad moods when things were not always going my way, and who supported me by saying that we were all on the same boat, it helped, kind of.

References

[1] Debian. Package apt-rdepends: Recursively lists package dependencies., 2016. [Online; accessed 15-June-2016].

[2] S. Kuranda. The 10 biggest cloud outages of 2013. crn, 2014. [Online; accessed 15-June-2016].

[3] A. Natarajan, P. Ning, Y. Liu, S. Jajodia, and S. E. Hutchinson. *NSDMiner: Automated discovery of network service dependencies*. IEEE, 2012.

[4] Python. networkx 1.11 - python package for creating and manipulating graphs and networks, 2016. [Online; accessed 15-June-2016].

[5] Python. pygraph 0.1.0 - a graph manipulation library in pure python, 2016. [Online; accessed 15-June-2016].

[6] L. Vincent. Hardware lister (lshw), 2016. [Online; accessed 15-June-2016].

[7] E. Zhai, R. Chen, D. I. Wolinsky, and B. Ford. Heading off correlated failures through independence-as-a-service. *OSDI 14*, 2014.