Calculating Total System Availability

Hoda Rohani, Azad Kamali Roosta
Information Services Organization
KLM-Air France
Amsterdam
Supervised by Betty Gommans, Leon Gommans

***Abstract*— In a mission critical application, "Availability" is the very first requirement to consider. Thus understanding what it is, what would affect it, and how to calculate it is vital. Although many methods have been proposed to calculate the availability of a device and/or a simple system, calculating the availability of a Business Application within a very complex organization is not still easily achievable. In this project, we would be proposing a method to enable the IT management team of KLM, to predict their Business Application availability based on the configuration and the components used in their infrastructure.**

System Availability, Reliability Engineering, MTBF, MTTR, Failure, Network, Application

Contents

I. Introduction

In a mission critical application, “Availability” is the very first requirement to consider. Thus understanding what it is, what would affect it, and how to calculate it, is vital. Proper functioning of a system can be evaluated based on different factors. Among those, “Reliability” and “Availability” are two close measurements in use. While these terms might have slightly different meaning in different contexts (e.g., in information security, “integrity” and “availability” are examined separately [1]) but we’ll be binding ourselves to the “Reliability Engineering” definition of these terms within this project. “Reliability Engineering” is a sub-disciplinary of System engineering that emphasizes dependability in the lifecycle management of a product [2].

Said so, “reliability” is a function of time, defined as the conditional probability that the system will perform correctly throughout the interval [t_0, t_1], given that the system was performing correctly at the time t_0 [3], while “availability” is considered as a function of time, defined as the probability that system is operating correctly and is available to perform its function at the instant of time “t” [3]. The major difference between these two terms is in the time, which is considered as an interval in former and instantly in the latter [4]. For example, if you consider a reservation system with 98.3% availability, we expect that it will be operating successfully for 59 minutes in each hour (statistically speaking). But reliability of such a system can be as bad as 10 minutes, which means that it will be considered “not working”, 10 seconds after each 10 minutes of working. Although the system availability is rather high, if a customer needs 15 minutes to book a ticket, she/he will never find the chance!

The likelihood of a system to fail, is often measured by its MTTF and/or MTBF parameters. MTBF (Mean Time between Failures) is the average (expected) time between the two successive failures of a component. It is a basic measure of a system’s reliability and availability and is usually represented as units of hours. Similarly, MTTF is defined as the expected time for the first failure [4].

“Recovery” is yet another main concern about any service. Having a correct recovery procedure and being prepared to recover from any failure in a defined amount of time via defined amount of energy and resources spent, one may decide not to lower the likelihood of the system to fail, but just simply recover it in case of a failure as soon as possible. All in all, what matters is to have the service do what is it supposed to do at the right time. MTTR (Mean Time to Repair) is the main term when determining how a system would behave in case of recovery. It is another major factor of determining a system “Availability”.

While these terms are highly interdependent, they have similar building blocks. In this project, we will be focusing on the *availability*. Although similar methods can be used to calculate the *reliability* as well.

In the term of our case and within KLM environment, a “service” is a component defined and used in company’s business process to play a specific role. Each Service consists of different software, hardware, people and processes in different layers. It is obvious that the total amount of availability a service has, would be highly dependent on its components.

II. Problem Statement

KLM is the flag carrier airline of the Netherlands and the oldest airline in the world still operating under its original name with its hub being at Amsterdam Airport Schiphol. Within KLM IT infrastructure hierarchy, there exists a couple of top level applications which are supposed to meet corporate business functions’ requirements. These applications are referred to as “Business Applications”. Electronic Booking System (EBT) is one of those applications which we will be focusing on throughout this project. Although the proposed model can also be applied to any other Business Application as well.

In this project, we will be creating a framework which allows the calculation of a Business Application Availability in various environment defined by the management. Live infrastructure data will be received from the AITIH database and Availability is calculated based on these data. It can also provide the required data for analysis of critical points (those having the most negative effect on the availability) in the infrastructure.

A. Effective Parameters

Each Business Application consists of other application software as its building blocks. Said so, a Business Application is considered "available" if all its building blocks are available and can communicate together (where required) correctly. If the Business Application is supposed to be *accessed* by the end-user (namely the customer), this accessibility should also be taken into account.

Of course for each application to operate correctly, the underlying hardware are supposed to be working correctly as well and so should the underlying software layers (like operating system). The communication layer is yet another obvious component we need to be available for each two components needing to pass data to each other.

When one talks about a piece of hardware (or software) availability, she/he is considering that it is being operated under the situation that the component is supposed to be working. This means that the probability of network switch failing, does not include the probability of power outage, cooling failure, misconfiguration by the switch administrator or even wrong port selection by the user due to lack of proper manuals. This means that the availability of these components should also be taken into the account separately. We would be categorizing these parameters as "people" and "processes" within our project.

These parameters and their relation can be seen in Fig. 1.

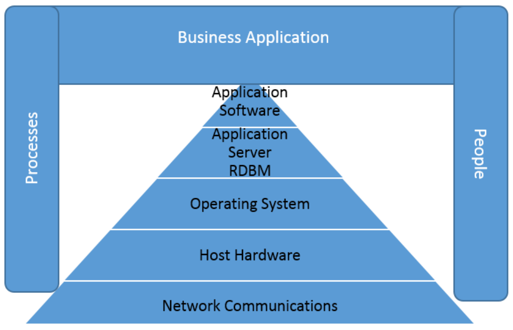


Figure 1-Availability Dependency Model in abstract.
Each layer's availability is affected by those in lower layers.

It is obvious that the more details we include in the model, the more precise result we will be having. But this might end up trading the calculation time and the model complexity for some precision that we don't really need. So we have to select only those components we want to consider their effect carefully. One way of determining which to select can be to rely on experienced expert's opinion. Another good practice can be going through the historical data of incident management system. If we're dealing with a system that has been in place for a rather long time and the incident records are accessible for it, the latter might be more useful. On the other hand, if we have a new system with most of its behaviors are yet to be known, the former is a better choice.

III. Terms and Definitions

Before going into more details, we want to introduce the terms we will be using along the project.

- *Mean Time between Failures (MTBF)*

Mean Time between Failures is the average (expected) time between two successive failures of a component. It is a basic measure of a system's reliability and availability and is usually represented as units of hours.

- *Mean Time to Repair (MTTR)*

Mean Time to Repair (or Recover) is the average (expected) time taken to repair a failed module. This time includes the time it takes to detect the defect, the time it takes to bring a repair man onsite, and the time it takes to physically repair the failed module. Just like MTBF, MTTR is usually stated in units of hours.

The following equations illustrates the relations of MTBF and MTTR with reliability and availability [5].

$$Equation1: \qquad Reliability = e^{-\frac{Time}{MTBF}}$$

$$Equation2: \qquad Availability = \frac{MTBF}{MTBF + MTTR}$$

The following conclusions can be reached based on these formulas [6]:

- The higher the MTBF value is, the higher the reliability and availability of the system.
- MTTR affects availability. This means if it takes a long time to recover a system from a failure, the system is going to have a low availability.
- High availability can be achieved if MTBF is very large compared to MTTR.

- *Inherent Availability*

The probability that an item will operate satisfactorily at a given point in time when used under stated conditions in an ideal support environment. It excludes logistics time, waiting or administrative downtime, and preventive maintenance downtime. It includes corrective maintenance downtime. Inherent availability is generally derived from analysis of an engineering design and is calculated as the mean time to failure (MTTF) divided by the mean time to failure plus the mean time to repair (MTTR). It is based on quantities under control of the designer [7].

When a system occurs a failure, the point is that how quickly the system can be recovered. In that case, the most important consideration is returning the failed processes up and running as fast as possible.

- *Achieved Availability*

The probability that an item will operate satisfactorily at a given point in time when used under stated conditions in an ideal support environment (i.e., that personnel, tools, spares, etc. are instantaneously available). It excludes logistics time and waiting or administrative downtime. It includes active preventive and corrective maintenance downtime [7].

- *Operational Availability*

The probability that an item will operate satisfactorily at a given point in time when used in an actual or realistic operating and support environment. It includes logistics time, ready time, and waiting or administrative downtime, and both preventive and corrective maintenance downtime. This value is equal to the mean time between failure (MTBF) divided by the sum of mean time between failure and the mean downtime (MDT). This measure extends the definition of availability to elements controlled by the logisticians and mission planners such as quantity and proximity of spares, tools and manpower to the hardware item [7].

This flavor is what we refer to as availability through this project.

Also according our SLA, all delays (like logistic times) are included in MTTR, the MDT and MTTR are used interchangeably.

- *Application*

The term “Application” is used as the general term within this project to present both “an application software” and/or an application service. A website, a database management service, a web-service, an executable file which is supposed to run on a server and be accessible throughout the network, any network service (such as DNS), and a storage service (software on top of a storage in a SAN) are all examples of an application.

Applications are main pieces of our puzzles as we’re actually trying to investigate their impact on each other. In other words, although there are many other components involved, our goal is to see if:

- Applications are alive;
- They can communicate together (and to the End User).

More exact details of this evaluation is presented later.

- *Host*

A host, is the main hardware and any software on top of it that is required to run the “Application”. Unless explicitly stated (like NICs) a “Host” includes the hardware, any firmware, virtualization environment, operating system and web server on top of it.

Hosts are important as they act like a containers for their applications, and no application can survive without having a working host.

- *NIC*

Network Interface Cards are the cards connecting a “host” to a “network device”. Although they can be considered as part of Host, but due to the fact that their failure may affect the host’s availability in separate ways, we have considered them as independent components.

If we had a homogenous environment (on which every host had identical number of NICs, and those NICs were bridged together and/or connected to a stacked switch), then we could consider the NICs as part of the hosts and so reduce the complexity of the model and also the execution time of simulation.

- *Network Device*

A network device is an equipment that provides network connectivity for hosts or other network devices. Although they may have other roles in the network (like firewalling) but the only role that is interesting for us in this project is their connectivity function.

For instance, assume that a firewall is connecting two hosts together, but an administrative rule inside the firewall blocks some part of traffic. This scenario in our situation is called a failure for this network device as it is not available for its intended service (which is connecting those two hosts). This is mostly important when calculating the availability of network device itself. This point should be mostly reflected as configuration errors (part of “human error”).

- *Application Replica (Instance) and Application Process (Clone):*

Applications can have multiple instances running on either different hosts, or even same host. These instances are assumed to be completely similar from a user’s prospective. This means that a user (be it a human or another service which is using this application) does not care which instance is answering its service. These instances have the same source code, so their (component) availability amount is identical.

However, there is a slightly difference between those instances running on a machine and those running on different machines. All those instances running on a same host are completely transparent to the user from network prospective. So, if the user can reach the host and any of the instances on the host is alive, the user will get its answer. But this is not the case for those instances running on different machines. A user should be able to get to "any" of those hosts, and that host and its "own" instance should be running. So we can assume all clones on a host ONE (new) application with a higher availability than the application itself.

We will refer to the application instances on a single machine as "clones" and those instances on different hosts as "replicas". Note that a replica may contain multiple clones.

- *Cluster*

A clusters is a group of applications which are identical in function and the system is considered available even if one of those in a group is still available. Replicas are an example of clusters.

- *Failure*

A failure happens when a component is not available. Components may fail, because they have been randomly chosen and marked as fail to evaluate their effect, or they may fail because any other component they were depending on, has failed.

In reliability engineering (and within this project), a *Failure* is said to happen when a component/system is not doing its desired function and considered as being *unavailable*.

- *Error*

In reliability engineering, an *Error* is considered a malfunction which is the root cause of a *Failure*.

- *Fault*

In reliability engineering, a *Fault* is considered a malfunction which is the root cause of an *Error*. But within this project, we may refer to a *component failure* as a *fault* that may lead to the *failure* of the system. This is done where there is a risk of ambiguity between a *failure* which is happening in intermediate levels (referred to as a *fault*) and one which is happening eventually (referred to as *failure)*.

- *Component*

A component in our model is the building block of calculations. It is either a hardware or a software which we want to consider its effect on system availability and its availability is calculated independently.

- *Dependency*

We define the dependency between two component *A* and *B* and say that *A* depends on *B* when the (un)availability of *A* will be completely affected by *B* being (un)available. In other words, if *B* is unavailable, then *A* is unavailable.

A good example of dependency is that a web application *depends* on its database service. Another example is that all components rely on the *environmental availability*[1].

- *End User*

In the business era and at the end of the day, all that matters is the customer and the ultimate goal is that the users be able to access the system. In order to ensure that, we introduce a virtual component placed on a part of network which is supposed to access some of other components. This virtual

[1] Although environmental availability will affect the whole system in general, because there might be different environment for different components (i.e. having more than a datacenter, which is the case for KLM), we have to consider this parameter in component level. If this was not the case, we could have simply considered this parameter at the final level at once.

component is called the “End User” and it depends on all major non-database applications. *End User*’s availability is considered 1.0 (always available).

- *Arbitrary Failure*

Beside the different sources, a failure may be of different types, one being *Arbitrary Failures.* These types of failure have undetermined result and a component/system facing such failures will be showing unpredictable behavior. For instance, a simple calculator in presence of such failures, might give you different results for an identical equation and parameters over the time. Such behavior, in general, will make it hard to find if a component has failed. This specially becomes more important when there are other replicas of a component.

As an example, assume that a user is using either of two instances of a redundant applications through a dispatcher. The dispatcher’s role is to determine which of the instances are currently available to serve. Normally, this job is easily done via a simple tests, hence making the dispatcher simple and relatively reliable. But in presence of arbitrary failures in the applications, the dispatcher may mistakenly route the user to the wrong instance. In this case, the dispatcher has failed because of a failure in the application. Such dependent availability calculations are not permitted in our model.

In order to avoid such situation, it is important to calculate those application’s availability more accurately. Also the dispatcher should become wiser so that it can cover such failures. Finally, the availability of the dispatcher should be calculated independently of the applications.

In general, it is best to eliminate the sources of such failures as much as possible. In software, this type of failures are mostly because of a fault in design and/or coding phases, and can be reduced by in-depth reviewing and testing of software artifacts.

- *Peaceful Degradation*

Clusters of components might be made as a way of masking their failures, but it also might be a way of increasing their processing power. While the former case is being referred to as *High Availability*, the latter is called *Load Balancing*.

It is obvious that as soon as a node in a load balanced cluster fails, the availability of that cluster would become dependent on the load on the remaining nodes. If the node becomes higher than what they’re capable of deliver, there would be interruptions on the service (i.e. users may experience a slower response). This situation is referred to as *Peaceful Degradation*. In such situation, the service might still be considered available, but with a limited functionality.

An example of such situation is when a 4 engine aircraft flying at 30,000 feet, loses 3 of its engine. The plane is still able to fly, but cannot keep its altitude at 30,000.

Although one may consider such degraded state as still available, but because subsequent failures of nodes in the cluster may result to failure of the cluster (although there might still be some available nodes in it) we will not mark the degraded state as available and will mark it as failed.

This being said, throughout this project, when we’re talking about clusters, it means that they are meant solely for high availability and not load balancing.

- *Failure rate*

Failure Rate, is the frequency by which the system fails.

For the components without moving parts, assuming a constant failure rate is not far from reality and since –except for the Hard Disk Drives- all of components engaged in our model are solely electrical, we don’t have any moving part. Disk drives however, are devices without repair. So any kind of failure would lead to the failed disk being replaced and there would be no consecutive failures. Hence for these devices also the constant failure rate holds (they only fail once during their operational life) [4].

- *Bathtub*

Hardware failures are usually described by “bath tub curve”. The first period is called infant mortality. During this period, the hardware failure is high. The next period is called the normal life. Failures usually occur randomly during this time but the point is that the rate of failures is predictable and constant and is almost low. The cause of failures may include undetectable defects, design flaws, higher random stress than expected, human factors, and environmental failures. The last period called wear out period, is when the units are old and begin to fail at a high rate due to degradation of component characteristics [4, 5, 8].

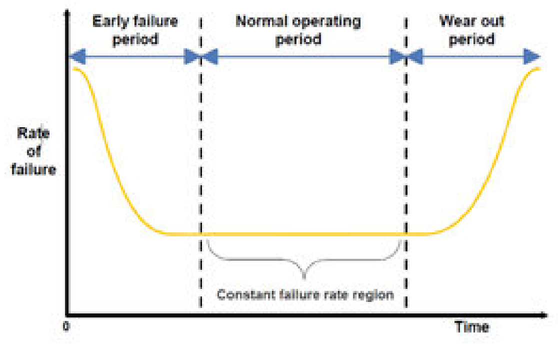


Figure 2-Hardware Failure Rate

- *MTBF vs Useful Life*

Sometimes MTBF is confused with a component’s useful life. Consider, a battery has a useful life of four hours and MTBF of 100,000 hours. This means that in a set of 100,000 batteries, there will be about one battery failure every one hour during their useful lives [4, 5].

The reason of sometimes these numbers are so much high is that these numbers are calculated based on the failure rate of usefulness period of component, and it is assumed that the component will remain in this stage for a long period of time. In the above example, wear out period mitigates the life of component, and the usefulness period becomes much smaller than its MTBF so there is not necessarily direct correlation between these two.

Consider another example, there are 100,000 20-year-old humans in the sample. We monitored this sample for one year. During that time, the death rate became 100/10,0000 = 0.1%/year. The MTBF is the inverse of the failure rate or 1/0.001 = 1000. This example shows that high MTBF values is different from the life expectancy. As people become older, more deaths occur, so the best way to compute MTBF would be monitor the sample to reach their end of life. After that, the average of these life spans are computed. Then we reach to the order of 75-80 which would be very realistic.

- *Availability and Number of Nines*

Availability is typically described in nines notation. For example 3-nines means 99.9%. Obtaining 5 nines or 99.999% availability is an ambitious goal for many vendors when producing hardware and software modules [9].

Table 1-Downtime of availability

Availability	9s	Downtime
90%	One	36.5 days/year
99%	Two	3.65 days/year
99.9%	Three	8.76 hours/year
99.99%	Four	52 minutes/year
99.999%	Five	5 minutes/year
99.9999%	Six	31 seconds/year

IV. Failure Sources

The system outages are put in two major categories: Unplanned outages (failure) and planned outages (maintenance). Both of them result in downtime. Unplanned outages are most costly compared with the planned one but it may be mitigated by using the redundant components [9].

Usually a planned outage has a tolerable impact on the availability of the system, if they are scheduled appropriately. They are mostly occur as a result of maintenance. Some causes of planned downtime can be periodic backup, changes in configuration, software upgrades and patches [6].

According to Sage Research Studies 44% of downtime in service providers is unscheduled. This period of downtime can cost a lot. Average cost of network downtime is estimated $21.6 million per year or $2169 per minute [10].

Another categorization can be:

- Internal Outage
- External Outage

Internal factors like specification and design flaws, manufacturing defects and wear out. External factors like radiation, electromagnetic interference, operator error and natural disasters. However a system is well designed or the components are highly reliable, the failures are inevitable, but it is possible to reduce their impact on the system [12].

V. Single Component Calculations

There are various factors that influence the total availability of the system. These factors are [11]:

- Hardware
- Software
- Environment
- Human Errors

Hardware failures like File System Full error, Kernel In-Memory Table Full error, disk full, power spike, power failure, and LAN infrastructure problem. Software failures like problem caused by source code and structure of software, software defects, application failure and firmware defects. Natural disaster like fire, flood and Human errors like operator or administrator error (pilot error) [13].

A. Hardware

For calculating the availability of hardware component in network, we need to know its MTBF and MTTR. The MTBF values are usually obtained from the vendor for off-the-shelf or hardware team for in-house components based on the component's configuration and design. It is the value which is estimated by the manufacturer before a failure occurs in a hardware component.

MTTR is based on the response time of our service contract or vendor. In our model for critical component this value is four hours. It is the time to get a service man on-site and replace the failed module [8].

B. Software

The definitions for software availability and hardware availability is the same even though their related failure occur for different reasons.

Obtaining high available software is much harder than the hardware ones. For example, software availability cannot be increased by using redundant component. The errors may occur by incorrect logic, statement or input data. Sometimes the software needs infinite time for testing/debugging which is not reasonable when have to ship the product to the customers in a timely manner.

We have to keep the history to get the software defect density in the system. Defect density or defect rate means the number of detected defects divided by the size of the software (this value is stated as thousands of lines of code or KLOC) during a specified period of time. This value is depending on many factors such as:

- Complexity of software
- Size of software
- Experience of team developer
- Percentage of the code which is used before in a stable project
- How much test/debug is done before releasing the product

MTBF for software modules can be computed by multiplying the defect rate by KLOCs executed per second.

As most of software failures can be eliminated by rebooting the system, MTTR can be considered as average time for reboot time [8].

Said so, obtaining MTBF of the software requires measuring the software failures for a large set of same modules over a long period of time. For example Cisco has decided 30,000 hours MTBF for mature software and 10,000 hours for a recent ones [11].

C. Procedure

According to Gartner, "Through 2015, 80% of outages impacting mission-critical services will be caused by people and process issues [10], and more than 50% of those outages will be caused by change/configuration/release integration and hand-off issues" [9].

As human intervention is not always error-free, it is a good policy to automate as many as processes as we can. Defining proper procedures would lead to a better availability. This parameter can be evaluated separately or can be integrated into the human factor as well. Sometimes these automation can be achieved by writing scripts like: routine backups, and software upgrades [9].

D. Environment

These fault can be occur by power outages, fires, earthquakes, tornadoes and other events. The point is that these events cannot be predicted and when encountering with them, the whole system becomes down for some hours or even months depending on the damages they impose [4, 9].

E. People (Human Factor)

These error are mostly occur as result of changes like adding, upgrading and reconfiguring the network components. When we want to execute the human factor in calculating the availability of the component, we have to consider the human and the task itself. What kind of task is the human executing, whether it is simple or hard, routine or non-routine, what is the stress factor (it means how much time is available for doing the task), is there any procedural guidance for assigned task, is the human experienced or has special training for doing the task. All these factors are weighted to compute the human factors [9].

VI. Simple Models

Assume we have two component, A_1 and A_2 and from the previous section, we know how to calculate their availability. Now we want to calculate the availability of a system, consisting only these two components in different configurations.

A. Serial Configuration

Serial configuration happens when two (or more) components are required to be available in order for the system to be available (Fig. 3). In this configuration, if either of those components fail, the system is considered to be unavailable.

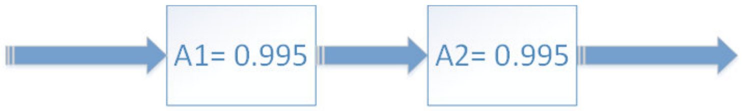


Figure 3- A system with two serial components

A typical example of this configuration is a “RDBMS server” and its “storage”. It is obvious that to be able to serve a database, both parts are meant to function properly at an instance of time. Another example are the HDDs in a RAID0 (striping) configuration in which a failure of either of the disks will result in losing all the data on the array.

In this configuration, as the system will be considered working as far as all the components are working, the total availability would be the multiplication of each component’s “independent” availability [14].

$$Equation\ 3: \qquad A(Serial) = \prod A(Component_i)$$

As availability of each component is a number lower than 1, the total system availability would become lower than any of its components. For example, the total availability of the system in Fig. 2 would be:

$$A(Fig.3) = A_1 \times A_2 = 0.990025 < 0.995$$

This actually makes sense as it is said that “A chain is only as strong as its weakest link”.

B. Parallel Configuration

Sometimes, system designers would put identical (or even similar) components together, in a way that as far as one of the components are available, the system can survive. Component in this configuration are said to be made redundant (Fig 4.).

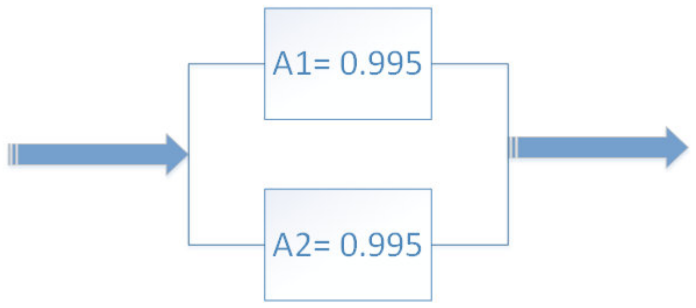


Figure 4- A system with two parallel components

Unlike the Serial configuration, components in a parallel configuration are either identical components, or two (or more) components with the same function. This main idea here is that the components failures are not arbitrary and components are fail-safe. This means that the component is either operating correctly, or it stops from working. If this is not the case, there should be a 3rd component as the output evaluator which considers which of the components placed in parallel should be used at a time (if this is not the case, then the "voting" mechanism is used which requires at least 3 component in parallel).

The system is this configuration fails, if all of its components fail [11, 14].

$$Equation\ 4{:}\quad A(Parallel) = 1 - Unavailability(Parallel) = 1 - \prod [1 - A(Component_i)]$$

For the system in Fig. 4, the availability would be:

$$A(Fig.4) = 1 - (1 - 0.005)(1 - 0.005) = 0.999975$$

Since the unavailability of this system is smaller than the unavailability of each of its components, the total availability would be even higher than the most available component.

The key for these calculations are that each component's availability should be calculated independently. On the other hand, the failure sources that are unique among components of a parallel configuration should be excluded from components availability and calculated separately.

Examples of this configuration are an aircraft with 4 engines, which can also fly by 1 engine operating, or disk mirroring which is considered in RAID 1 configuration.

It is also worth mentioning that we will not consider peaceful degradation as complete parallel configuration.

C. Hybrid Configuration

Hybrid Configuration happens when the system consists of multiple components, from those some are serial and the others are parallel. In order to calculate the availability of such a system, one may calculate any consecutive serial/parallel components and replace them with blocks with new availability in order to be able to complete the calculation.

This method is known as Reliability Block Diagram [15] modeling and works best as far as we can see each components either as serial or parallel (and not both) in a system. Fig. 5 is an example of a system that deriving RBD for it is not so easy.

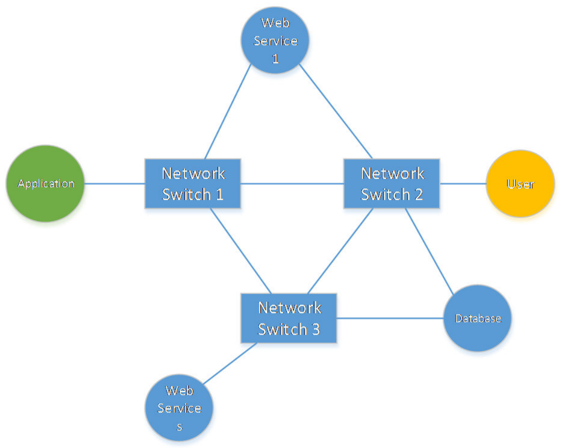


Figure 5-A hybrid, layered system with interdependency between components

Figure 5 is a hybrid configuration in which the user wants to access the application in green circle, while this application needs either of Web Services to operate correctly. The Web Services are also relying on the Database.

As we can see, the Network Switch 1 and 2 are seen as redundant from the "Web Service 1" point of view, however, this is not the case when user is going to access the Application.

VII. Related Works

A. Markov Modeling

Markov Probability Model [5, 15] is a stochastic models that is function of system's state and time. A state machine is drawn to describe the behavior of model. The major property of Markov Modeling is that changes between states i and j happens with a probability of P which only depends on the i, j and time t. In other words, the previous states have no effect on the p.

Each of these two variables can be either discrete or continues which would result into 4 different type of models. Markov Process is the variation of model with discrete state and continues time. The major property of a Markov Model states that

Now if we consider a state, as an array of 0s and 1s for each of the components in the model, in which a 0 means that component has failed, and a 1 means that it has survived and find the transitions that can happen between these states with their respecting probability, we can finally calculate the systems availability.

The problem with this modelling is that with having too many components, the state machine would become quite big and complex to create.

On the other hand, the benefit of such a model is that it would also cover the probability of a system NOT being repaired in its defined time duration. Another benefit is that this model can also consider consecutive failures.

Figure 6 shows the Markov state machine of a Triple Modular Redundancy (TMR) in which the probability of each component failing is considered as $\lambda\Delta t$ and is independent of machine's state.

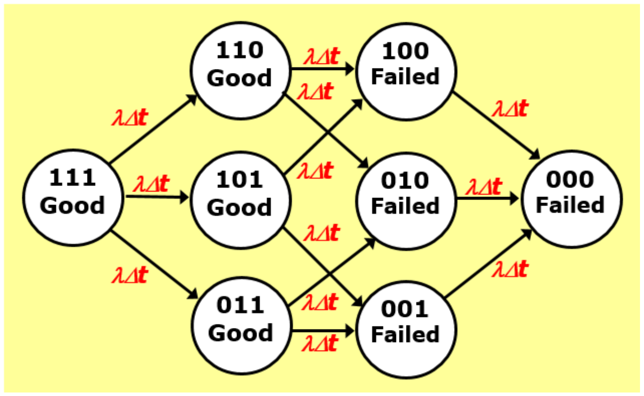


Figure 6-Markov Process of a TMR system

B. Fault Tree Analysis

Another Proposed Model for prediction of failure and calculation of reliability/availability is Fault Tree Analysis (FTA) [16].

In this method, first the undesired event (i.e. the failure of the main module of our system) is defined, then component failures that can cause this failure are extracted and put in the second level. The third level consists of those that may cause the 2nd level failure and this would continue until we reach the component level (often referred to as elementary faults).

Although this model has its usefulness like [15]:

- Forces the analyst to actively seek out failure events (success events) in a deductive manner;
- Provides a visual display of how a system can fail, and thus aid understanding of the system by persons other than the designer;
- Points out critical aspects of systems failure (system success).
- Provides a systematic basis for quantitative analysis of reliability/availability.

But the same problem with Markov Modeling still holds. When the system becomes complex, drawing the Fault Tree is not that easy and requires a lot of manual works. In other words, the complexity of our system will not allow us to create a reusable Fault Tree that can be adopted for several changes in the system.

C. Failure Mode and Effect Analysis

The FMEA method and its deviations (i.e. FMECA) are meant to specify the modes of failure and evaluating their consequences. Unlike the FTA, in a FMEA method, we start from the component level, and examine the different failures each component may face. Then we will check what consequences such failure may have on the system.

This model would nicely fit to our criteria, as it allows us to evaluate different causes of failures independently. This makes more sense when some causes are meant to be evaluated independently, but others are not. A good example is failure of an application, which might be based on the code bug (this way all its replica will fail) or as of an inappropriate configuration (which will affect a single replica only).

Although the number of possible component failure modes that can realistically be considered is limited [16], it is sufficient enough to cover our causes.

The main problem with using such a model is that determining the causes of a failure in our complex environment is not so easy. Especially as redundancies are in place on different layers, that may prevent some component failures to eventually cause the system to fail.

VIII. The Approach

The method that we are proposing here will be using the ideas from other methods mentioned in previous sections. In brief, we first specify the components we want to consider and then evaluate the effect of its failure on the system.

The components we're considering are the "Applications", "Hosts", "Network Interface Cards" and "Network Devices" as the components, assuming that either independent availability function or the MTBF and MTTR parameters of each component are in hand. We'll also consider the network connections between different components. We consider the dependency between different components (mostly applications) as mandatory rules of survival (being available).

Finally, we try to simulate component(s) failure and see if our Business Application survives. The total amount of availability would be calculated based on the failure scenarios and their probability to happen.

A. Component Selection and Layering

A component or system failure may be the result of many faults, including:

- An application may have bugs;
- An application server may run out of resources;
- An operating system may fail;
- A hard disk may fail;
- A server hardware may fail;
- A network cable may get disconnected;
- A switch may malfunction;
- An administrator may make a mistake while configuring something;
- You may have power outage;
- Your cooling system may fail.

As stated before, the more we go into the detail, the more accurate result we will be having and the more complex and time consuming our calculations would become. In order to overcome this tradeoff, first we decide which components should be considered and partition the components into different categories. The typical component type that we have in our environment are pictured in Fig. 7. (Note that there is no OS shared between hardware devices).

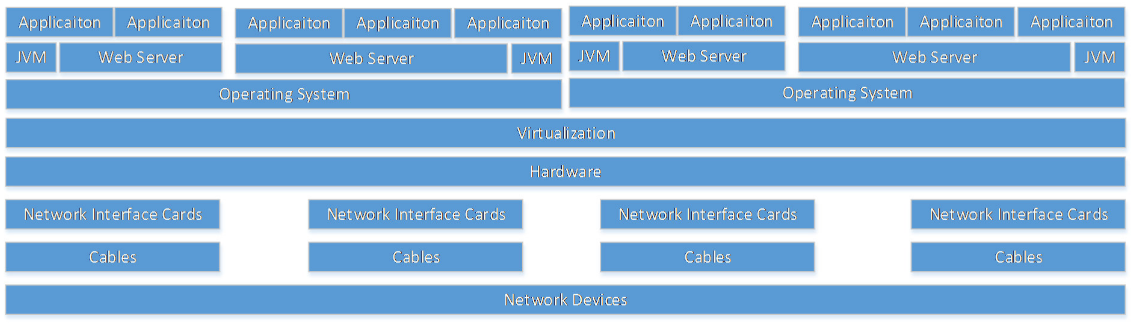


Figure 7-Components in a layered view

There are two major categorize available:

- *Network Category*, which consists of all those components providing network connectivity;
- *End Point Category*, containing all other components.

The *End Point Category* is itself subcategorized as:

- Application itself;
- Container, which includes underlying components of an application.

The main idea here is that dependency rules only applies between components inside Application subcategory and never to the components in Container. The Network Category is also subcategorized into Container and Interfaces.

After defining and categorizing our components, we inspect their failure on the system by simulating each component's failure and examining its effect on other components and the *End User* as the decisive factor of our model. This is done based on the following rule:

A component is considered as failed if either of the following happens:

- There is a failure in *environment*;
- It is chosen by the simulator to fail;
- Its container had failed;
- What it depends on, had failed;
- There is no network connectivity between this application and what it depends on.

Note that is an application is in a cluster, the dependency rule should be examined for all nodes in the cluster.

The simulation is done in rounds, and at each round, a total of n component are randomly chosen to be assumed as failed, with n being from 1 to the total number of components. The chosen components are called *faulty components*.

The inspection of a round will continue until the system reaches a steady state, that is no new failure is found. In this state, if the *End User* component fail, we will call this a *failure scenario*.

It is obvious that the probably of a component being a *faulty component* is equal to the unavailability of that component. This being said, the probability of such round happening would be the multiplication of unavailability of all *faulty components* and availability of other components. For a failure scenario, we call this a *round unavailability* for round *i*.

$$Unavailability_{round\ i} = \prod 1 - A(fualty\ Components) \prod A(nonfaulty\ Components)$$

When all the failure scenarios have been found, our desired unavailability (the availability of End User) is the summation of *round unavailabilities*.

$$System\ Availability = A(End\ User) = 1 - \sum_{i=0}^{Total\ of\ Failure\ Scenarios} Unavailability_{round\ i}$$

Finally, we introduced a criticality function in order to evaluate the role of each component in the total (un)availability of the system and find the component with the most influence on such (un)availability.

The criticality function of a component is defined as *the total number of that component's appearance as a faulty component* multiplied by *component's unavailability*.

$$Criticality(c) = (1 - A(c)) \times \sum_{i=0}^{Total\ number\ of\ c'appearance\ \ as\ a\ faulty\ component} 1$$

Eventually, the more the criticality of a component is, the more effect it has on the system's unavailability.

IX. Proof of Concept

In order to see the proposed model in action, we prepared a program in Python which receives AITIH data in multiple comma separated text files in order to calculate the availability of a subsidiary of EBT business application. Wherever the real data were not available, we have made an assumption to make the model consistent and complete. These assumption are clearly marked within the report.

Using the mentioned input data, we drew an abstract model of real world with the "Applications", "Hosts", "Network Interface Cards" and "Network Devices" as the components, assuming that either independent availability function or the MTBF and MTTR parameters of each component is in hand.

A. Situation

The chosen subsystem of EBT consists of the following Applications:

- appT
- appCSA
- appEUI
- appEBC
- appEDB
- appCS
- appkia

Each of these applications might be running on multiple hosts and might have multiple instances running on each host. Table 2 shows this test case hosts, applications, and their clones.

Table 2-Application, Host, Clone Relations

Application Name	Host	No. of Clones Running
appCSA	hst01	1
appCSA	hst02	1

appEUI	hst03	5
appEUI	hst04	5
appEUI	hst05	5
appEBC	hst06	3
appEBC	hst07	3
appEBC	hst08	3
appEBC	hst03	3
appEBC	hst04	3
appEBC	hst05	3
appCS	hst06	1
appCS	hst07	1
appCS	hst08	1
appCS	hst03	1
appCS	hst04	1
appCS	hst05	1
appKIA	hst06	1
appKIA	hst07	1
appKIA	hst08	1
appKIA	hst03	1
appKIA	hst04	1
appKIA	hst05	1
appT	hst09	1
appT	hst10	1
appEDB	hst11	1

“appT” and “appEDB” are two database services supporting these applications. Table 3 represents these relations (dependencies).

Table 3-Dependency Rules

Application Name	Database Service	Hosted on
appCS	appT	hst09
appkia	appT	hst10
appEBC	appEDB	hst11

A visualization of all components in relation can be found in Appendix 1.

As not all parts of input data were in hand, we made some assumptions to prepare a valid initial state:

- There is a 3rd switch called “Switch_3” which provides connectivity between “Switch_2” and “Switch_1”;
- The “hst11”, “hst10” and “hst09” hosts have two separate NICs which are being connected to “Switch_1” and “Switch_2” to provide redundancy;

“End User” is connected to “Switch _3” as well. It is a computer with Availability function of 1.0 (It doesn’t fail).

In this experiment, a “Business Application failure” is said to happen when the user (End User) cannot access either of the following applications: “appT", “appCSA”, “appEUI” “appEBC”, “appEDB”, “appCS”, “appkia”.

B. The input data

The input data (which are supposed to be exported from AITIH database and show our current situation) are fed into the calculation engine via five simple csv files. These files are explained in Table 4.

Table 4Input File Structures

File Name	**Record Format**	**A Sample Record**	**Description**
apps.csv	*Host Name, Application Name, Clone Counts*	hst01,appCSA,1	Contains Host's, Application and clone counts
netnods.csv	*Network Node, Direct Neighbor1, Direct Neighbor2, … ,Direct Neighbor n*	Switch_1,Switch_3	list of network devices and their direct neighbors
Clusters.csv	*Cluster's node 1, Cluster's Node 2, Cluster's Node 3, …. , Cluster's Node n*	N/A	Explicit Clusters
hostnicsw.csv	*Host Name, Ethernet Card, Switch Name*	hst08,eth2,Switch_1	Network connectivity of components
dep.csv	*Application A, Application B*	appEBC,appEDB	Showing that A depends on B
availability.csv	*Component name, MTBF, MTTR, Availability*	hst08->eth2,,,0.999944	*

*) The availability file contains component's availability parameters. It is assumed that each single component availability is calculated before and presented in this file. However, as many of the component we're dealing with, have the MTBF and MTTR in hand, if the availability is not provided directly for a component, the program will calculate it on the fly, based on the known $A = \frac{MTBF}{MTBF+MTTR}$ formula. If these parameters are not presented either, the availability of the component is assumed 1.0. Fig. 8 shows component availability flowchart.

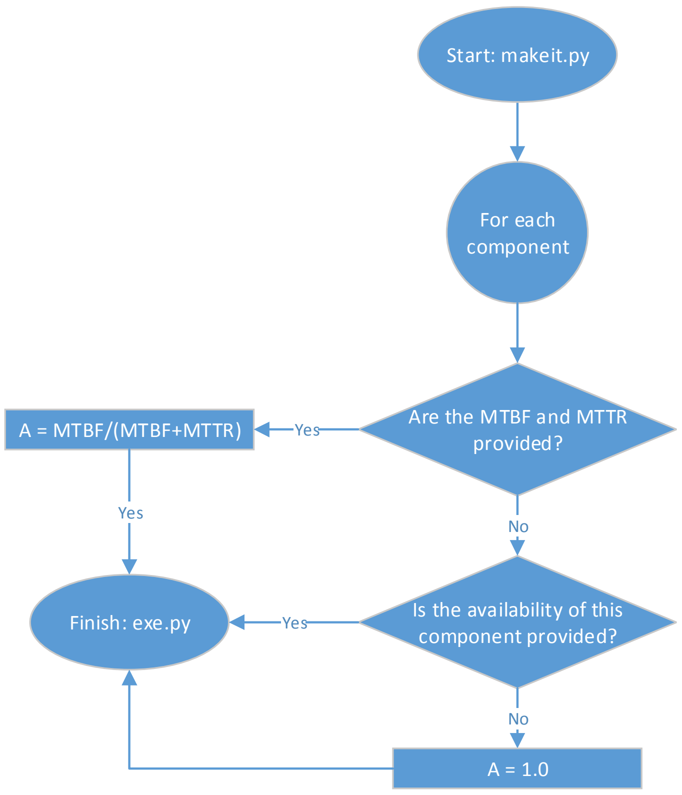


Figure 8-Compoennt Availability FLowchart

Availability is entered as a number between 0 and 1 and each of these parameters can be left empty (except for the component name). Note that if the application is being served by a 3rd party whom we have a contract (and probably SLA) with (like Amadeus), the availability mentioned in the contract can be a good choice as that application's availability [17].

It is also worth mentioning that as these availability values are "Independent" of each other, availability of application replicas would be the same.

For this experiment, we generated 54 random numbers between (0.9999 and 1.0) as availability parameters, with 0.9999 being the smallest, 0.999997 being the largest number and an average of 0.99995.

C. Processing Files

The program was organized into three different executable files, among those one calculates component's availability (acalculator.py), one prepares the input files (makeit.py) and eventually the other (run.py) runs the main program over the data structures. There are some other auxiliary files that facilitate the execution. We can see all these files and their relation in Fig. 9.

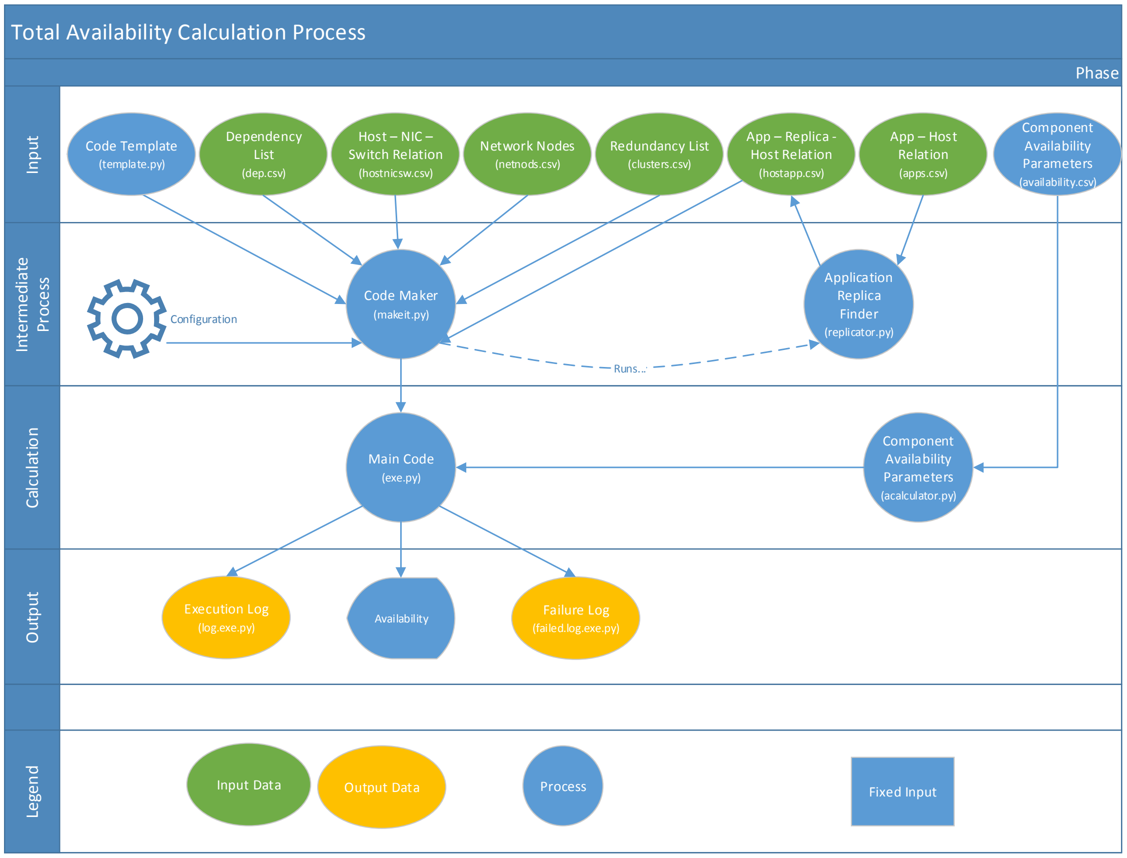

Total Availability Calculation Process
Phase
Input
Intermediate Process
Calculation
Output
Legend
Code Template (template.py)
Dependency List (dep.csv)
Host – NIC – Switch Relation (hostnicsw.csv)
Network Nodes (netnods.csv)
Redundancy List (clusters.csv)
App – Replica - Host Relation (hostapp.csv)
App – Host Relation (apps.csv)
Component Availability Parameters (availability.csv)
Configuration
Code Maker (makeit.py)
Application Replica Finder (replicator.py)
Runs...
Main Code (exe.py)
Component Availability Parameters (acalculator.py)
Execution Log (log.exe.py)
Availability
Failure Log (failed.log.exe.py)
Input Data
Output Data
Process
Fixed Input

Figure 9-The Availability Calculation Process

D.Assumptions

Apart from the general assumptions for our model, we have made the following assumptions regarding this experiment:

- Each single node's independent Availability is either pre-calculated, or its MTBF and MTTR parameters are present. If none were present, a random number between 0.9999 and 0.999997 were assigned as the availability.
- Whenever there is a physical network path between two network nodes, it illustrates a network connection between them. In other words, no network segmentation exists in upper layers.
- Physical connectors (like cables) are considered as always available.
- Network devices are seen as a single component even if they are modular.
- There is no virtualization involved.
- There is only one web server on each OS.
- Hosts include: Web Server, Operating System and Host hardware (except for the NIC).
- All network cards of a server are able to take-over other cards.
- In the network layer, Redundancy is made by using separate paths. There is no Stacked Switch.
- Environmental and Human Related Factors are rolled out for simplicity

E. Execution

We ran the program 6 times, starting from a maximum of 1 failure at a time, up to a maximum of 6 simultaneous failures. The result was as follow:

1) Maximum of 1 simultaneous failure

In this case, 5 different failure scenarios occurred, which were caused by the failure of:

- hst11
- appEDB
- Switch_3
- Switch_1
- *End User*

Note that, as "End User's" availability is 1.0, it will not affect the total amount of availability, but as it is considered a component, it is shown in the result (This is true for all components with availability of 1.0).
And total availability of "End-User" were calculated to be: **99.9781477%**

2) Maximum of 2 simultaneous failures

When considering up to more 2 concurrent failures, it is obvious that the result contains single failures, as well as all component failures that include one of the single failures components.

There are 55 components: 5 for each single failure alone, 50 for each couple containing a single failure and $\binom{5}{2}$ for combinations of those 5, which will make: 5+5*50+10=265

Hence, we expect this case to be at least 265. The final amount is: 280 failure scenarios. Those 15 scenarios are caused by failure of the components shown in Table 5:

Table 5- Failure Scenarios in presence of maximum 2 Failure

#	Component A	Component B	Description
1	'hst09'	'appT.REP2'	appT.REP2 on hst10
2	'hst09'	'hst10'	
3	'appT.REP2'	'appT.REP1'	appT.REP1 on hst09
4	'appT.REP1'	'hst10'	
5	'hst02'	'hst01->eth2'	
6	'hst02'	'appCSA.REP1'	.REP1 on hst01
7	'hst02'	'hst01'	
8	'hst01->eth2'	'appCSA.REP2'	.REP2 on hst02
9	'hst01->eth2'	'hst02->eth2'	
10	'appCSA.REP2'	'appCSA.REP1'	
11	'appCSA.REP2'	'hst01'	
12	'appCSA.REP1'	'hst02->eth2'	
13	'hst02->eth2'	'hst01'	
14	'Switch_2'	hst11->eth1	
15	'hst11->eth1'	'hst11->eth2'	

Note that "A.REPx" shows the xth replica of the application "A".

3) Maximum simultaneous failures > 2

We continued the simulations up to maximum number of 7 which ended up quite similar to the previous results.

Table 6-Test Case Result

Maximum Concurrent Failures	Total Failure Scenarios	Total Availability	Time required to calculate
1	5	99.9781476669 %	< 1 min
2	280	99.9780993579 %	< 1 min
3	8,192	99.9780993065 %	< 1 min
4	136,153	99.9780993064 %	5 min
5	1,769,375	99.9780993064 %	46 min
6	17,919,053	99.9780993064 %	> 11 hours

It is worth mentioning that although the concurrent failures came into account, the more precise availability would be calculated, but there is point where the precision we gain is far from what we really need. On the other hand (considering the high availability function of single components used in enterprise networks) the probability of multiple failure happening together is negligible.

So we can say that the current test case has got an availability of "99.97809%" with even maximum concurrent failure equal to sum of components.

According to our result, the most critical component were turned out to be "Switch_1" switch.

X. Future Works

The method we proposed here can be improved in (specially) two aspects.

A. Optimizing the algorithm

The program were written as a proof of concept, with around 600 line programming in Python. Neither the choice of the programming tool, nor the method of programing is prepared is so efficient.

On the other hand, it is best if the algorithm could become smarter in detection of failure scenarios in case of examining the redundancies and also the merge of some components and summarizing their availability.

Passing the data via text files are not a really good idea in production. It would be good if the program could read its data directly from AITIH database.

B. More criticality options

The *criticality function* defined here is not the only way we can find the weakest link of our chain. One can define some other functions (even more accurate ones) to highlight the effect of some special components on the whole system.

There can also functions being defined to find any over qualified components (if any), those that though having a high availability, are not affecting the whole system as expected. This function can be used in decision making process to save some costs.

Acknowledgment

We would like to take the chance to thank all KLM AITIH department and especially Betty and Leon Gommans, Peter Huisman, Maarten Hogendoorn and David van Leerdam, who helped a lot during different parts of this project, either by their supportive behavior or cooperative actions.

References

1. ISO/IEC, *Information technology - Security techniques - Management of information and communications technology security*, in *Part 1: Concepts and models for information and communications technology security management*. 2004.

2. Wikipedia. *Reliability engineering*. [cited 2014 January]; Available from: http://en.wikipedia.org/wiki/Reliability_engineering.
3. *Fault-tolerant computer system design*. ed. K.P. Dhiraj. 1996, Prentice-Hall, Inc. 550.
4. Vargas, E. and S. BluePrints, *High availability fundamentals.* Sun Blueprints series, 2000.
5. Torell, W. and V. Avelar, *Mean time between failure: Explanation and standards.* White Paper, 2004. **78**.
6. Cisco. [cited 2014 January]; DESIGNING AND MANAGING HIGH AVAILABILITY IP NETWORKS]. Available from: https://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6550/prod_presentation0900aecd8031069b.pdf.
7. Wikipedia. *Availability*. [cited 2014 January]; Available from: http://en.wikipedia.org/wiki/Availability.
8. eventhelix. [cited 2014 January]; Available from: http://www.eventhelix.com/realtimemantra/faulthandling/reliability_availability_basics.htm#.UvfKjJA1iM9.
9. Weygant, P.S., *Clusters for High Availability: A Primer of HP Solutions*. 2001: Prentice Hall Professional.
10 Colville, R.J. and G. Spafford, *Configuration Management for Virtual and Cloud Infrastructures.* Gartner, http://www. rbiassets. com/getfile. ashx/42112626510, 2010.
11. Oggerino, C., *High Availability Network Fundamentals: A Practical Guide to Predicting Network Availability*. 2001: Cisco Press. 256.
12. Vallath, M., *Oracle real application clusters*. 2004: Access Online via Elsevier.
13. Weygant, P.S., *Primer on Clusters for High Availability.* Technical Paper at Hewlett-Packard Labs, CA, 2000.
14. Xin, J., et al., *Network Service Reliability Analysis Model.* CHEMICAL ENGINEERING, 2013. **33**.
15. Shooman, M.L., *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*. 2003: Wiley.
16. Vesely, W.E., U.S.N.R.C.D.o. Systems, and R. Research, *Fault tree handbook*. 1981: Systems and Reliability Research, Office of Nuclear Regulatory Research, U.S. Nuclear Regulatory Commission.
17. Fishman, D.M., *Application Availability: An Approach to Measurement.* Sun Microsystems. Recuperado el, 2000. **8**.

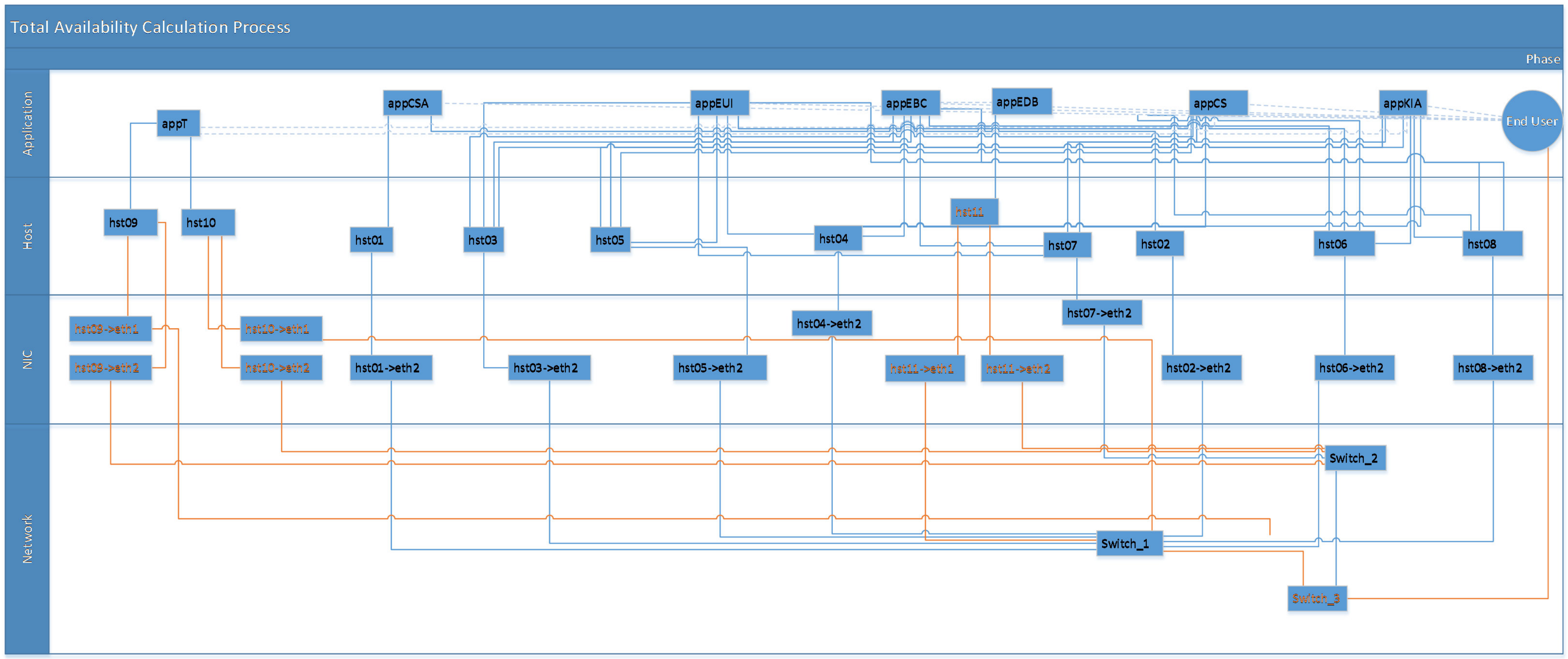
XI. Appendix 1.

The diagram in this attachment is showing all components in our proof of concept experiment, and their relation.

There are four different categories of nodes, including: Applications, Host Hardware, Network Interface Cards and Networking Devices (Switches). The lines between nodes represent a “relation” which is interpreted based on type of components in the relation, as shown in Table 3.

Table 7- Relation Definition

Line between	Shows
Application <-> Application (dotted line)	Dependency Rule
Application <-> Host	The “Application” running on this “Host”
Host <-> NIC	The “NIC” belongs to this “Host”
NIC <-> Network Devices	A network connection between two
Network Device <-> Network Device	A network connection between two

Total Availability Calculation Process
Phase
Application
Host
NIC
Network
appT
appCSA
appEUI
appEBC
appEDB
appCS
appKIA
End User
hst09
hst10
hst01
hst03
hst05
hst04
hst11
hst07
hst02
hst06
hst08
hst09->eth1
hst09->eth2
hst10->eth1
hst10->eth2
hst01->eth2
hst03->eth2
hst05->eth2
hst04->eth2
hst11->eth1
hst11->eth2
hst07->eth2
hst02->eth2
hst06->eth2
hst08->eth2
Switch_2
Switch_1
Switch_3