UNIVERSITEIT VAN AMSTERDAM

Module Deployment and Management within the Grid-based Virtual Laboratory for e-Science

Vincent Buijtendijk
Faculty of Science
Universiteit van Amsterdam
the Netherlands
vbuijten@science.uva.nl

Thesis Supervisors
Adam Belloum
adam@science.uva.nl
Dmitry Vasunin
dvasunin@science.uva.nl

May 2005

vl·e virtual laboratory for e·science

Abstract

The Grid has the potential to couple resources, such as pc's, supercomputers, mass-storage systems, and laboratory instruments all over the world together. The Virtual Laboratory for e-Science (VL-e), which is built around the Globus toolkit, is a solution for scientists who wish to make use of the benefits of the Grid. It provides an intuitive way for designing complex experiments, without knowledge of low-level Grid issues. VL-e experiments can be decomposed into a graph, which connects generic processing elements called modules. A connection represents the flow of data between modules. These modules need to be deployed to individual VL-e resources. The objective of this thesis, is to propose an automated module deployment and module management system for VL-e. An architecture for module deployment is presented. Furthermore, the aspect of module management is researched from a scientific perspective. An important task of module management is to remove the least needed module in case there is no more space left on a VL-e resource. Trace-driven simulations are performed on real and artificial data to find the best VL module replacement strategy. The strategies used here, were originally designed for implementation in hardware and web caching systems. It is interesting to observe their behavior and performance in a completely different environment.

Contents

Chapter 1

Introduction

In the last years, large scale computing in the scientific community has become more and more common. In the past, all mass computing was carried out on supercomputers or parallel computers/networks very controlled environments. Parallel networks had reliable connections between individual nodes, and were mostly homogeneous.

However, more and more computers, laboratory instruments and mass-storage systems are connected through the internet or other international networks. If these could be linked together, there would be an enormous capacity for mass computation and collaboration.

The concept of collaboration between heterogeneous resources is called the *Grid* [1]. The Globus toolkit [2] is an implementation of the Grid concept. It has emerged as a standard for building Grid applications.

In order to provide an extra layer which makes collaboration easier for scientists and programmers, the Grid-based Virtual Laboratory Amsterdam (VLAM-G) [3] was first developed. VLAM-G will be followed up by the Virtual Laboratory for e-Science (VL-e), which is currently in development. VLAM-G and VL-e are built around the Globus toolkit.

VL-e experiments are composed out of modules, and are the *basic building blocks* of the system. They contain all necessary information to enable VL-e resources to perform a sub-task of the experiment such as software libraries and executables. An example of a VL-e experiment, in the chemo-physical application domain, is the chemical analysis of the surface of degrading old master paintings like those by Rembrandt [4]. This particular experiment can be divided into four general steps: data acquisition from a storage system, data analysis using a Fourier transform, quality control and visualization. These tasks are distributed to independent resources by the Resource Manager. All of these resources need modules to be installed and deployed prior to performing a task. When all tasks have been successfully distributed to a resource, the experiment can be started.

Suppose that, during run-time, one of these resources fall out for common reasons like losing network connection or experiencing hardware failures. In the experiment concerning the aging paintings, the resource which performs analysis may fail. As a result, the entire experiment stalls, since we are unable to analyze the raw data and therefore can produce no visual output. In the worst case, if no resource is available containing a pre-installed analysis module, we need to reinstall the same module completely manually on another system. Manual installation consumes a considerable amount of time. First, a suitable new resource needs to be found to install the module on. Second, the module needs to be transported to the new resource. Third, taking into account the specifications of the new resource, the module needs to be installed. Tedious as this manual process seems, it can get even worse. The only copy of the required module may have been on the machine which is down. Additionally, the average (non-computer) scientist does not have the skills, or privileges, to perform these manual steps. By the time the system administrator has been found and is willing to do the job, the paintings have aged so much that we need to start over the entire experiment.

As we can learn from the example above, the requirement of manual module deployment contrasts with all design goals of VL-e. An automated module deployment system is therefore absolutely *essential* for VL-e to function efficiently. The goal of this thesis is to propose such a system. It should be both simple and flexible, from the perspective of the user. As little user interference as possible should be needed to deploy a module to a resource.

Our deployment system incorporates all of the above requirements. Furthermore, components in our system are designed to have a high level of fault tolerance; when one of the components fail, others can take over. For example, a part of our system is the Global Module Repository. It contains all modules in a domain. A resource may retrieve a required module from there. When a Global Module Repository is down, the resource may also retrieve the module from other resources where the module is installed already. Additionally, it is possible to retrieve Modules from Global Module Repositories in other domains. All of these resources can be found using another component in a VL-e domain, the Resource Index.

We research a selection of popular software deployment and/or retrieval systems, which can be used for module deployment. Their individual advantages and disadvantages are weighed to see which system is best suited for integration into the deployment

Another very important aspect is module management. Once modules are deployed they reside on the hard drive of the resource. However, there is only a limited amount space available for the modules. This is why there needs to be some sort of a removal policy, which uninstalls a module when the quota is reached. We review several replacement policies, which perform this sort of task within the traditional scientific area of caching. Our aim is to determine if these policies are suitable for usage in a module management system on a VL-e resource. We perform trace-driven module replacement simulations, using real and artificial traces. From the results, we try to decide which of the replacement strategies is the most suitable for our environment.

Chapter 1 Introduction.

Chapter 2 A general description of the Virtual Laboratory for e-Science and an introduction to the concept of modules.

Chapter 3 Requirements for a module deployment and management system. Module deployment architecture. Module management. Introduction to classic cache replacement policies: LFU, LRU, Size, Hybrid and LUV and their usability for module management.

Chapter 4 Simulation of module replica management. Simulations performed using a Netherlands UNIX User Group download log, and artificially generated trace files. Results and discussion of the results.

Chapter 5 Conclusions and Future Work.

Chapter 2

The Virtual Laboratory for e-Science (VL-e)

The Virtual Laboratory for e-Science is an ongoing project. The Faculty of Science of the Universiteit van Amsterdam and many other organizations and companies participate in this project. It is based on the completed VLAM-G project. Both projects share the goal of developing a middle-layer which provides scientists an intuitive platform for developing grid-applications. It is based on the popular Globus toolkit, which is an implementation of the grid concept.

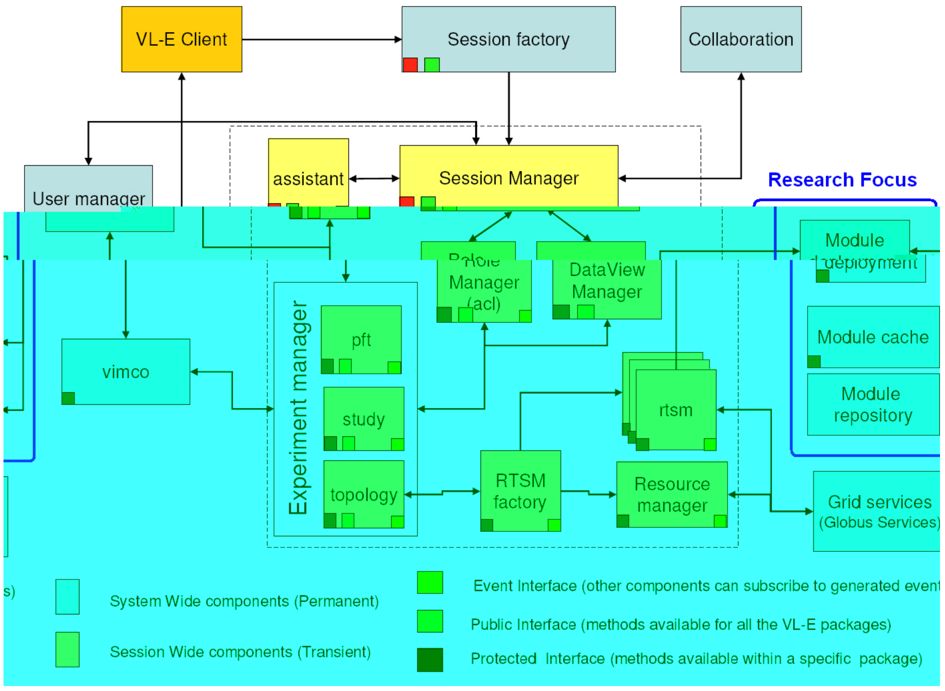
Creating grid applications directly, using Globus, is of course a possibility. However, some scientists, specialized in areas of science which can benefit from grid-technology, don't have the necessary programming skills. VL-e aims to be a solution for this group.

In addition to making the development of Grid applications easier for scientists, VL-e is also designed to be as generic as possible. The project does not focus on one scientific discipline in particular. Researchers from, for example, biology, chemistry and physics should all be able to benefit from this technology.

An application created within VL-e has a modular design. These modules may accept input and generate some kind of output and may be connected using the topology editor. We explain these concepts in the next sections.

2.1 VL-e Architecture

To give an impression about the environment where our module deployment system will be used, we will very briefly describe the major components



VL-E Client
Session factory
Collaboration
User manager
assistant
Session Manager
Research Focus
Role Manager (acl)
DataView Manager
Module deployment
Module cache
Module repository
vimco
Experiment manager
pft
study
topology
rtsm
RTSM factory
Resource manager
Grid services (Globus Services)
System Wide components (Permanent)
Session Wide components (Transient)
Event Interface (other components can subscribe to generated event
Public Interface (methods available for all the VL-E packages)
Protected Interface (methods available within a specific package)

Figure 2.1: VL-e Architecture [5]

of the Virtual Laboratory for e-Sciences architecture. The architecture is displayed in figure 2.1 [5].

The components which are blue-colored are permanent, while the yellow-colored components only exist within the scope of a session. These components are called System Wide and Session Wide components, respectively.

The Session Wide components are created when an experiment is started and terminated when the experiment is finished.

Session Factory Creates an instance of the Session Manager.

Session Manager Controls all session activities

Experiment Manager Consists of the Process Flow Template, the Study and the Topology. The components of the Experiment Manager are covered in section 2.2.

Collaboration Provides a way for VL-e users to interact with other users.

VIMCO Contains all information about virtual experiments in VL-E.

Assistant Assists the user in composing an experiment, by providing templates and information about previously conducted experiments.

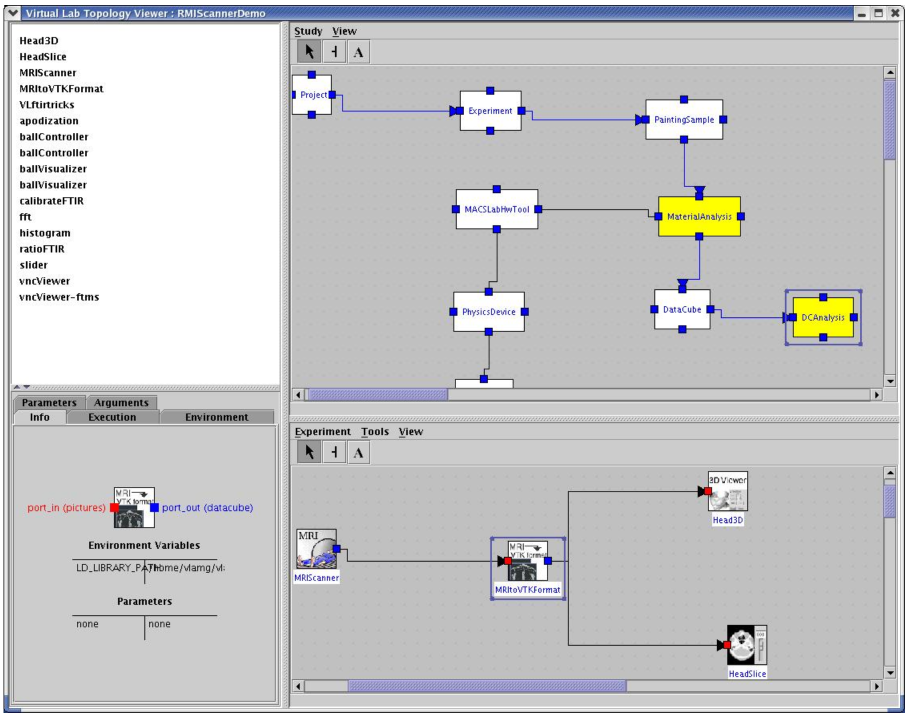


Figure 2.2: Process Flow Template (PFT) [5]

RTSM Factory The Run-Time System Manager factory creates an instance of the RTSM for each experiment.

RTSM The Run-Time System Manager distributes the VL-e experiment over several resources (VL-e nodes) by dividing it into tasks. Starts and monitors the experiment as well.

Resource Manager Discovers, locates and selects suitable resources according to the requirements of modules. Described in more detail in section 3.2.1.

***Module Deployment, Cache and Repository** The area of focus in this thesis, our new automated module deployment and management system.

The above VL-e components are described extensively in the publication about the Grid-based Virtual Laboratory Amsterdam (VLAM-G) [3].

2.2 Process Flow Template and Topology Editor

The Graphical User Interface (GUI) provides the main interaction with developers and users. It consists of the Process Flow Template (PFT) editor, the PFT viewer and the Topology editor. The PFT is displayed on the upper right side in figure 2.2. All experiments created in VL-e start with a domain-specific PFT. The Process Flow Template (PFT) is created by a domain expert. It defines a certain type of scientific experiment. The PFT consists of a formalized abstraction of the data and processing steps involved in such an experiment. A PFT can be created using the PFT editor. Users in such a scientific domain can create an instance of the PFT, called the Process Flow Instance (PFI) or study. A selection of attribute values may be modified by the user to accommodate his/her specific experiment.

In the Topology editor, the end-user is allowed to define the processing elements composing his experiment. Such an experiment is represented by a number of *modules* in a directed data flow graph, which is introduced in the next section.

2.3 Modules

Modules are an important concept within VL-e. A study is represented by a directed data flow graph (DFG) or by a Kahn diagram. In VL-e, the software entities within such a graph are called modules. Modules can be experiment-specific or generic. They contain software (libraries) which allow the module to run on a VL-e node. Modules can be connected by arrows, which represent data streams between them. An example of the decomposition of an experiment into modules, a topology, is displayed on the lower right side in figure 2.2.

An example of a typical module task is a Fast Fourier Transform. The module operates an fft on some data set and generates output. Modules may be used in multiple topologies. Modules are the building blocks for VL-e applications. A module needs to be transported to, and installed on the resources you would like to use the module on. We call this module deployment. Modules are stored in a database called the module repository.

At the moment, module deployment has to be performed manually within the VL-e environment. This imposes several restrictions on the VL-e system: all clients have to be known and we need to know their specifications, OS, and whether a module may have already been deployed. It is also very time consuming.

It is clear that it is desirable to automate this, since a Grid environment is usually very heterogeneous and dynamic. Nodes can join and leave the grid frequently. When the VL-e network grows, manual deployment becomes more and more impractical.

In this thesis we propose a method to deploy modules automatically. It is also important to manage the modules after they are deployed. These subjects are described in detail chapter 3.

Chapter 3

Module Deployment and Management

In this chapter, we present our architecture which can be used within the VL-e environment to deploy modules automatically. We first state requirements (section 3.1).

In section 3.2.1, the individual components of the proposed architecture and their relations are described.

There are many existing systems which could potentially be used for module retrieval and/or module deployment. We made a selection of the most popular software deployment tools and compare them in section 3.2.3.

The local module repositories on the nodes need to be maintained. Module management takes care of removing modules from the repository when there is no more disk space (or quota) for modules left on the node. The LFU, LRU, Size, Hybrid and LUV strategies, used in caching systems, are introduced and their relevance for module management is explained.

In section 3.3.4 the issue of dependency relations between modules and their impact on module management is raised.

3.1 Requirements

There are some general requirements which the automated module deployment and management system needs to satisfy. They are in line with the requirements of the VL-e system in general. Furthermore, the automatic deployment system should address all shortcomings of the current manual deployment system. The example of the analysis of paintings in the introduction raises some of these shortcomings and we translated them into our set of requirements.

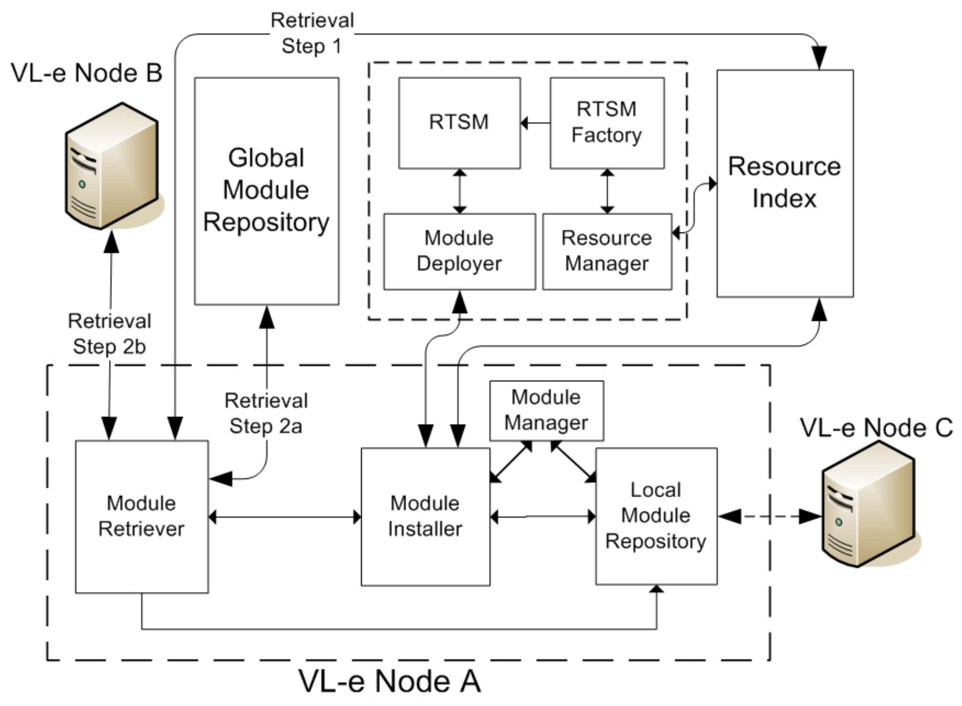


Figure 3.1: Deploying Modules

Heterogeneity The system should work on as many platforms as possible. This is in line with a general requirement, which aims to make the VL-e project as generic as possible. This increases the number of potential resources.

Fault tolerance The system should have a high degree of fault tolerance. We need to build in some redundancy, in case components in the system fail. This can be achieved by setting up the system in a flexible and distributed manner.

Scalability It should be easy to add new nodes, institutions and subnetworks, which extend the VL-e network.

Automation The user, and even the administrator, should need as little knowledge as possible to be able to use the system. The (domain) administrator only needs to set some (one-time) basic settings for the resources, such as quota.

3.2 Module Deployment

As was mentioned in the introduction, module deployment in the VL-e environment is performed manually currently. We propose an architecture in this thesis to automate this. Module deployment consists of 3 general steps.

1. Finding the right resource
2. Retrieving the module
3. Installing the module

The framework presented will facilitate these steps and is intended for integration into the VL-e project.

3.2.1 Architecture Components

In figure 3.1 an overview of the module deployment architecture is displayed. In this subsection, each component in the figure is introduced.

Run-Time System Manager Factory The Run-Time System Manager factory creates an instance of the RTSM for each experiment when it receives a request to run an application (composed of a set of connected modules). The modules need to be distributed to individual VL-e nodes. The RTSMF uses the Resource Manager to find them. When the Resource Manager returns suitable nodes, the RTSM factory translates the schedule to the RTSM for execution.

Run-Time System Manager The Run-Time System Manager (RTSM) is an existing component within the current VL-e architecture. It performs the distribution of tasks to Grid-enabled resources, starts the experiment and monitors its execution. The distribution schedule is received from the RTSMF. In the case that modules have to be deployed to one or more VL-e nodes, this is passed on to the Module Deployer which performs the job. The Module Deployer reports back to the RTSM whether deployment was successful.

Resource Manager The Resource Manager is an existing component within the current VL-e architecture. The Resource Manager handles resource discovery, location and selection according to module requirements. It maps tasks to resources to optimize experiment performance utilizing a number of algorithms and scheduling techniques. Several resource allocation algorithms were presented and compared in [6].

There are many factors which may be considered to decide to which VL-e node a task should be mapped. These factors include performance requirements set by modules to ensure efficient execution:

- Architecture
- CPU clock speed
- Operating system
- Available memory
- Available disk space
- Bandwidth
- Location

All of the above information is available from the Resource Index. Another factor in deciding where a task should be executed is the availability of resources where the required module is already deployed. We must give priority to distributing tasks to these resources, since it saves time and network load by preventing unnecessary module deployment. The Resource Manager communicates with the Resource Index to retrieve information whether the required module has already been deployed on one or more resources.

After all tasks of a experiment have been deployed by the Run-Time System Manager, the Resource Manager continues to monitor the resources during the execution phase in case rescheduling is needed.

Resource Index The Resource Index is a global component which provides information about which modules are deployed on which VL-e resources. Each module has a unique fingerprint which makes sure that indexing is non-ambiguous. In addition to module information, the Resource Index also keeps information about the characteristics of each resource (CPU, OS, memory etc.).

The Resource Manager uses the information from the Resource Index to decide how to distribute experiment tasks to VL-e resources. To keep the Resource Index up-to-date, VL-e nodes report to the Resource Index whenever a change takes place in the installed modules.

Module Deployer The Module Deployer receives the command from the Run-Time System Manager to deploy a certain module on one or more VL-e nodes, which have been selected by the Resource Manager. The Module Deployer initiates contact with individual nodes in a parallel fashion. It reports back to the RTSM if the node was reachable. If this was the case, the Module Deployer indicates whether or not installation was successful. If it was not successful, an error code is passed on to the RTSM.

The RTSM (together with the Resource Manager) may always decide, in the meantime, to deploy the module to more VL-e nodes.

Module Installer The Module Installer is a component on the target VL-e node. It receives information from the Module Deployer (in the outside world) to retrieve and install a certain module. If there is enough disk space in the Local Module Repository, the command is passed on to the Module Retriever. This component retrieves the module and passes it to the Installer. The installer installs the module to the Local Module Repository. When a module is not needed anymore, it is uninstalled by the Module Installer as well.[1] The result of the installation attempt is reported back to the Module Deployer. The Module Installer is partly implemented by using an existing packaging and deployment system, see section 3.2.3.

Module Retriever The Module Retriever is a local component on the target VL-e node, and receives a command from the Module Installer to retrieve a specific module. It contacts a Resource Index and checks if there is another VL-e node which already possesses the module. If this is the case, the Local Repository of the node is contacted and the module is retrieved. If there is no VL-e node available containing the module, the Global Module Repository may be contacted to retrieve the module from there.

When the module has been retrieved, a copy is sent to the Module Installer and a copy is placed directly into the Local Repository.

The Module Retriever may be partly implemented by using an existing packaging and deployment system, see section 3.2.3.

Global Module Repository The Global Module Repository is the component where all existing modules are stored. Only in the case that a module is not available from another VL-e node, may the module be retrieved from

[1]Most uninstalls are performed because of lack of disk space or quota. This is covered from section 3.3, about module management.

the Global Module Repository. This is to balance the load.

If a new module is introduced in VL-e, it should be submitted to the Global Module Repository first. From there the module can spread over the VL-e network, when nodes start requesting it.

Local Module Repository The Local Module Repository a local component (present on all VL-e nodes) where all retrieved and deployed modules are stored. There are in fact two copies of each module. One is the packed module file itself and the other is the unpacked and installed module. This ensures that the node can always be contacted by another VL-e node to provide one of the installed modules.

Module Manager The Module Manager basically acts as a referee. It decides whether a module is allowed to be installed and which modules should be uninstalled in case the resource runs out of hard drive space or quota. A replacement policy is used to determine which module should be uninstalled when a new module needs to be deployed, and there is a lack of space.

These subjects are described in detail in section 3.3 and chapter 4.3.

3.2.2 Fault Tolerance and Scalability

Two requirements of the module deployment system are fault tolerance and scalability. They are tightly related to each other. In order to meet both requirements, we must focus on the Global Module Repository and the Resource Index.

Having just one Global Module Repository might suffice in case of a small VL-e network. However, problems will arise when the VL-e network grows in size. In case that the Global Repository is down, modules can only be downloaded directly from VL-e nodes. Problems are even more serious when the Resource Index is down. No module information can be retrieved and the status (which modules are installed where) can't be updated.

Apart from possible downtime issues, bandwidth and capacity limitations can become a problem in a growing VL-e system.

Distributing both components ensures that both the fault tolerance *and* the scalability requirements are met. A setup may be devised where each domain has its own Resource Index and Global Module repository. The distributed components may communicate with each other if a module can't be found within the own domain. Each VL-e node should have a list of multiple Resource Indexes and Global Module Repositories, in case that these components are down in the local domain.

3.2.3 Packaging and Deployment System

Our aim is to automatically place software on VL-E nodes. This means that the software should be packed, transported to the node and installed. This may seem trivial. However, there are many issues and requirements to address.

3.2.3.1 Requirements

Simplicity The software should be packed as one single file; the module. This makes transport simple and efficient.

Dependency Most software is component based, which means that a piece of software *depends* on one or more other software components. These components may in turn depend on other components. Such a set of dependency relations are called a dependency tree. When deploying a module, these software dependencies need to be resolved as well. Dependency resolving is a very complex subject. A lot of research has been done on this subject. All existing deployment systems have their own dependency resolving methods. The issue of dependencies within module *management* is discussed in detail in section 3.3.4.

Heterogeneity Deployment of modules should be possible on as many architectures and operating systems as possible. Of course, this does not only depend on the deployment system, but also on the compatibility of the modules themselves. There are two ways to make sure that a module can be deployed on multiple platforms. The preferred way is to include the source code of the software in the module, so that it can be compiled on many platforms. The VL-e group promotes this option. It is currently researching if there are any obstacles for using source-based modules within the VL-e architecture.

However, including the source code is not always possible. There might simply be no access to the source code, because the module uses third party or legacy software. In this case we must add binaries of all platforms we want to support to the module. A module deployment system must therefore be able to support both sources and binaries.

3.2.3.2 Existing Systems

There are many available packaging and deployment systems. The most well known are Debian's APT (Advanced Packaging Tool) [7] and RPM (Red Hat Package Manager) [8]. Many other (commercial) packaging systems are based on one of these systems. Yum is also well known, and is based on RPM.

Red Hat Package Manager [8] The Red Hat Package Manager installs, updates, uninstalls, and veri es software. RPM has been designed originally for Red Hat Linux, however, it is available for many Linux distributions now.

Each RPM le contains a package label which has the following information:

Software name

Software version (the version of the software which is packaged)

Package release (number of times the package has been rebuilt. Also often used for indicating the distribution the package is intended for)

The architecture the package was built under

Source code may also be distributed in RPM packages. Such package labels obviously do not have an architecture part.

Some packages will not operate properly unless some other package is installed too. RPM makes sure that the package being installed will have its dependency requirements met. It will also insure that the packages installation will not cause dependency problems for other, already installed, packages. In case a dependency problem arises, the user receives feedback from RPM and should resolve it him/herself before the package can be installed. Possible dependencies of a package are stated in the RPM itself.

In the background, the RPM database is continuously maintained. It contains the information of all installed RPMs. Additionally, it tracks all les which are created and changed when a package is deployed. Therefore, when a package needs to be uninstalled, everything can easily be reverted to the situation prior to deployment.

RPM is very popular and powers Fedora Core, Red Hat, SuSE Professional, SuSE Enterprise, Novell Linux Desktop, Novell Open Enterprise Server, Mandriva, CentOS and many more.

Advanced Packaging Tool [7] The Advanced Packaging Tool is a packaging and deployment tool originally created for Debian. However, like RPM, it is now available on many more Linux distributions.

APT greatly simplifies the process of installing and removing software on Unix systems, by automating the retrieval (from internet, network, or cd), the configuration, the compiling and the installation of software from APT sources. Package upgrades may also be performed using a single command.

Software is retrieved from APT repositories which are listed in a configuration file. It is very easy to create your own APT repository. The list of packages which are available on the repositories is cached locally. This makes it faster to find a resource which has the requested software. The list may be updated by issuing an update command.

APT is also able to resolve dependencies completely automatically. No user intervention is needed. Necessary software packages can be retrieved and deployed automatically. When an installation or upgrade command is provided, APT will first search its cached list of packages and list dependent packages (dependencies) it needs to install or upgrade, and automatically fetches, configures and installs them.

A modified version of APT which can handle RPM files is available as well. This tool is called apt4rpm.

APT comes standard with Linux distributions such as Debian GNU/Linux, Ubuntu Linux, Xandros, MEPIS, Knoppix and Linspire.

Yellowdog Updater, Modified [9] Yum (the Yellowdog Updater, Modified) is derived from yup, an automated package updater originally developed for Yellowdog Linux, hence its name. Yum stands for "Yellowdog Updater, Modified". Yum is an extension to RPM, which it uses for packaging and dependency checking.

Yum has many similarities with APT. Like APT, yum has the ability to retrieve, install and resolve dependencies automatically. Yum uses the headers of RPM files to find out dependencies and retrieves and installs these RPMs for the user. Yum uses a list of repositories as well, which can also be easily set up.

Yum is only available on Red Hat and Fedora distributions.

3.2.3.3 Choosing a VL-e Packaging and Deployment System

There are not many differences between yum and APT. The choice is basically to use one of these systems, or the more low-level RPM. Yum and APT have the advantage over RPM that they perform many tasks completely automatically such as retrieval and dependency resolving. Yum and APT deploy all software components on which the main component depends without the need of user input. This is a feature which is very useful if we would like the deployment system to be fully automated.

Both systems are available as open source under the GNU public license, which is beneficial for customizing them for VL-e, if necessary. Creating a repository is straightforward for both systems as well. This is important because when a VL-e node retrieves (and installs) a module, the module should end up in a local repository. A module will not only be available from a global repository, but also from one or more nodes.

APT and yum are also able to update all packages using only one command, which may be useful for updating modules in VL-e.

There are some advantages of APT over yum. APT seems to be faster and is available on more Linux distributions then yum.

Using RPM also has its advantages. Since it is more low level, RPM is better customizable for the VL-e environment. For example, in the future, we may want to split up a module into multiple parts during retrieval. A module may then be downloaded from multiple locations at once. Since RPM does not take care of retrieval itself (unlike Yum and APT), we need to implement this independently from RPM.

When weighed all advantages and disadvantages, we recommend using APT or yum, and customizing them for VL-e. Both systems support many Linux distributions. However, they don't support Windows or Macintosh. This leaves out a large number of potential VL-e nodes, which unfortunately conflicts with our heterogeneity requirement.

3.3 Module Management

The local Module Repository needs to be maintained by the Module Manager. Each VL-e resource will run a module managing process. One of its functions is to clear up space in the repository when it is full. The system administrator of each node may decide for him/herself how much quota he wants to allocate to the Module Repository. A policy needs to be found to automate module removal. It should decide which module(s) is/are the least needed. This is a well know problem in caching systems. When a decision is made, one or more modules (depending on the space needed) are uninstalled and this is reported back to the Resource Index.

In the remainder of this chapter a module management scenario is sketched. Several cache replacement policies and their relation to the module manage-

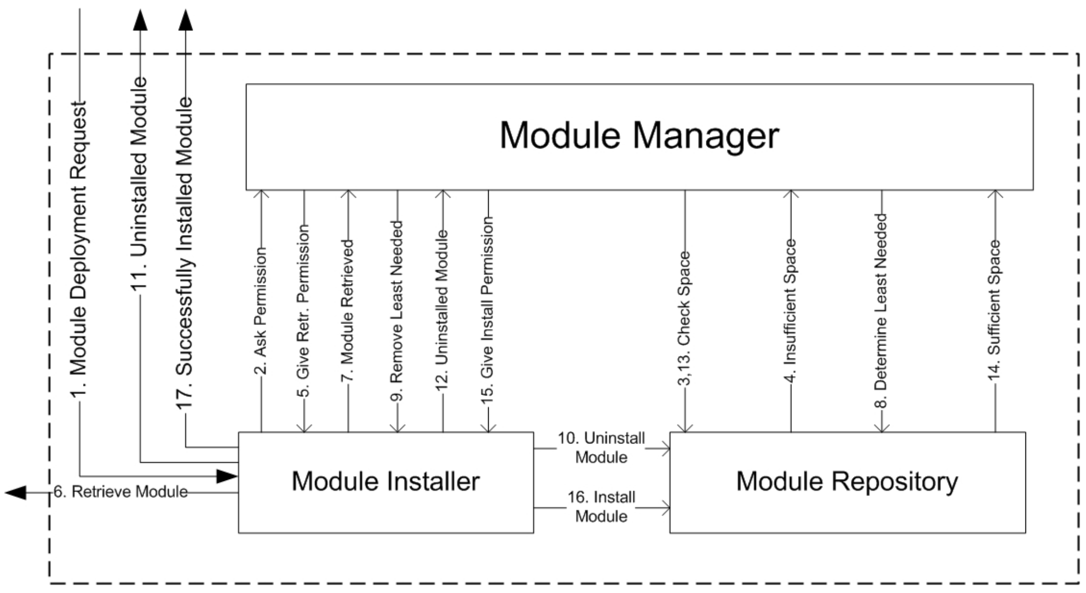


Figure 3.2: Module Management Scenario

ment are discussed. Furthermore, we consider dependency issues in module management. In chapter 4 the cache replacement policies are analyzed.

3.3.1 Module Management Process

The Module Manager has the (complicated) task to manage module installations and uninstalls. A representative scenario of the installation of a module is depicted in figure 3.2. It describes what happens in the case that there is not enough space in the Module Repository to install the new module. Since the focus lies on module management in this section, only the Module Installer, Module Manager and Module Repository are displayed.

In figure 3.2, a request arrives from the Module Deployer[2] to retrieve and install a certain module (step 1). Before the Module Installer may retrieve and install the module, it must first ask the Module Manager for permission. In this scenario, the Module Manager checks if the Module Repository has enough space left for installation of the new module (and for the packed module). It finds out it does not. This implies that modules should be removed to clear up space in the Repository. Now the Module Manager gives permission to retrieve the module. Only when this is successful, the Module Manager starts looking for modules to be removed. This is a safe-

[2]This component is not depicted in figure 3.2. See figure 3.1.

guard to make sure that no modules are uninstalled, in case the new module could not be retrieved properly. Now the module manager decides, using a replacement policy, which old module is least needed. Now the Module Installer uninstalls that module. This is reported to the Resource Index in step 11 (figure 3.2). The Module Manager checks if there is sufficient space now. In our scenario there is and the module may be installed by the Module Installer. Installation does not only mean unpacking and configuring the module. An original (packed) copy of the module is also be placed in the Module Repository (see section 3.2.1). If there is still not enough space left, steps 8 to 13 are be repeated until there is. Finally, after installation, the Module Deployer is notified of the successful module deployment.

3.3.2 Caching and Replacement Policies

A replacement policy is needed by the Module Manager to determine which modules can be removed. We therefore look at other scientific areas where replacement strategies are already used. The areas where most research on this subject has been done are hardware caching and web caching.

When the concept of using caches in computers to decrease access times to data was introduced, a way to discard data when the cache becomes saturated was also needed. Several techniques where developed to improve the so called hit ratio.

In caching systems, when the processor requests a page and the cache can provide it, a cache hit is registered. If this is not the case, we register a cache miss. The hit ratio is defined by equation 3.1:

$$\frac{\sum_{i \in R} h_i}{\sum_{i \in R} f_i} \tag{3.1}$$

where R is the set of all accessed data/documents, h_i is the total number of hits for page/document i and f_i is the total number of requests for page/document i.

Web caching aims to reduce latency by bringing documents closer to the requestor. Many replacement policies were designed for this form of caching. In web caching we have more aspects to consider than just optimizing the hit ratio, which complicates matters:

- Object sizes are usually not of equal size
- Retrieval latency in case of a miss is not constant

Therefore we need to introduce some additional metrics. The byte hit ratio is defined by equation 3.2.

$$\frac{\sum_{i \in R} s_i.h_i}{\sum_{i \in R} s_i.f_i} \tag{3.2}$$

where s_i is the size of document i

The delay saving ratio (or reduced latency) is defined by equation 3.3.

$$\frac{\sum_{i \in R} d_i.h_i}{\sum_{i \in R} d_i.f_i} \tag{3.3}$$

where d_i is the mean fetch delay from server for document i

These metrics better reflect the properties of web caching [10]. Additionally, module caching is similar to web caching, which is explained in section 3.3.3. When implementing a cache, usually one of the metrics is chosen for optimization. If one would, for example, want to save network bandwidth, the delay saving ratio should be optimized.

Now we descibe the replacement strategies which are used in our simulations.

3.3.2.1 Least Frequently Used (LFU)

LFU simply evicts the module[3] which is requested least frequently first. In other words, LFU relies on the assumption that a module which has been requested often in the past stands a big chance being requested in the future. Implementation of this algorithm is not complicated. The computational (time) complexity of LFU is logarithmic, $O(log(n))$.

However, there are some drawbacks. The policy is not very adaptive. The cache may become polluted with unpopular modules. When a very popular module suddenly becomes unpopular, it could stay in the cache for a long time.

Occurrences of ties are an issue to consider as well. There should be a selection criterium in case of a tie, which complicates implementation.

3.3.2.2 Least Recently Used (LRU)

LRU attempts to solve the problem from a different angle. It assumes that if a module has been requested recently, it will probably be requested again

[3]From now on we will use the word "module" or "package" which is the equivalent of a "document" or "page" in hardware and web caching.

in the near future. The module which has been requested the longest time ago will therefore be evicted in the case of cache saturation.

Implementation of this algorithm is very simple. Computational time complexity is linear, $O(1)$, which is a considerable improvement over LFU.

A module which is requested frequently but with big time intervals may still result in misses for this module. This is a weakness of LRU.

3.3.2.3 Size

The Size [11] algorithm evicts the largest module first. A tie may occur when there are multiple largest modules. The original implementation of this algorithm does not include additional criteria for resolving these.

Because Size evicts all large modules, many small modules fit in the cache. This results in the hit rate being optimized. However, because the cache will be polluted with small modules only, Size usually performs much worse in the byte hit rate and the reduced latency metrics. The computational complexity of Size is logarithmic, $O(log(n))$.

3.3.2.4 Hybrid

The Hybrid [12] policy aims to optimize the access latency. Hybrid uses a function to assign a value to each module. Wooster and Abrams defined this function as follows in equation (3.4) for module p:

$$u(p) = \left(rtt_s + \frac{W_b}{b_s}\right) * \frac{(nref_p)^{W_n}}{S_p} \tag{3.4}$$

where rtt_s is the round trip delay to server s where the module originates, b_s is the bandwidth between the cache and the server s, $nref_p$ is the number of times module p was requested since it is in the cache and S_p is size of module p.

W_b and W_n are constants. W_b can be adjusted to reflect the importance of the connection time against the bandwidth. W_n represents the importance of module frequency versus module size.
Hybrid has a logarithmic computational complexity, $O(log(n))$.

3.3.2.5 Least Unified Value (LUV)

The Least Unified Value (LUV) algorithm [13], designed by Bahn et al., uses frequency, recency and size information. Most other policies use just one or two of these aspects. Another difference with most other replacement policies is that LUV can be adjusted to optimize the desired metric: hit ratio, byte hit ratio and reduced latency.

LUV attempts to combine the good properties of LRU and LFU. It can therefore be categorized as a policy in the LRFU category (Least Recently/Frequently Used) [14].

The LUV algorithm assigns a value to modules which is calculated as follows:

$$Value(i) = p(i) * Weight(i) \tag{3.5}$$

where $p(i)$ is the estimated re-reference potential of a module i.
$Weight(i)$ is the retrieval cost of module i per unit size and is formally defined as:

$$Weight(i) = c_i/s_i \tag{3.6}$$

where c_i represents the cost and s_i the size of module i.
$p(i)$ is a function which estimates the re-reference potential using time differences

$$p(i) = \sum_{j=1}^{k} \left(\frac{1}{2}\right)^{\lambda . t_{diff}(j)} \tag{3.7}$$

where k is the number of previous appearances of module i and $t_{diff}(j)$ is the time elapsed since occurrence j of module i. λ is a parameter which can be adjusted to optimize a desired metric. The $(1/2)^{\lambda . x}$ part of the function was used from the original LRFU implementation by D. Lee et al. [14].

In other words, $p(i)$ takes into account all past references (and the time of occurrence) of a module. Furthermore, more recent references have bigger weights than older references. Retrieval cost and module size are taken into consideration by the multiplication of $p(i)$ with $Weight(i)$.

The constant may be set to values $0 \leq \lambda \leq 1$. If the value is set closer to 0, frequency information becomes more important. If λ approaches 1, the accent shifts to more recent references. λ must be computed offline and its value depends on a combination of the properties of the trace and the desired result.

When looking at this algorithm, it may appear as if the complexity of calculating and recording the $Values$ makes implementation impractical. This is not the case, as the authors proved in [13]. They proved that the $Value$ associated with a module only changes when the module is requested. It is also possible to calculate the new $Value$ of a module at the kth reference easily from the value of a module at the $(k-1)$th reference:

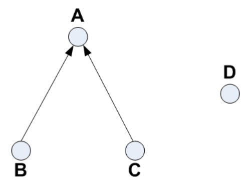


Figure 3.3: Dependency relations between modules

$$Value_i(k) = p(\theta).Value_i(k-1) + c_i/s_i \tag{3.8}$$

where θ is the time since the last request of module i at time $(k-1)$.

These properties give the policy a very reasonable time complexity of $O(log(n))$.

3.3.3 Module Replica Management

Module replica management in the VL-e environment is comparable to web caching. However, the properties are of a different order. Modules are usually bigger in size than the average web page. Other characteristics may also differ.

The effectiveness of existing replacement policies within a module management environment must be researched. We need to run simulations to analyze their performance, since these policies were traditionally used in the areas of hardware or web caching.

For a simulation, we need input data. VL-e traces are, of course, not yet available since modules are deployed manually. We therefore have to find a similar alternative. It is possible to create an artificial trace file which we can use as input. However, for a completely new application it is very hard to estimate the parameters to generate a realistic trace file. yet big enough predict these properties for the future.

In chapter 4 we discuss how to tackle this problem, and simulate and analyze the results.

3.3.4 Dependency Issues

As we discussed in section 3.2.3.1, there are dependency issues to consider in module deployment. However, there are possible implications for module management as well.

In figure 3.3, it can see why this might be the case. The figure represents a

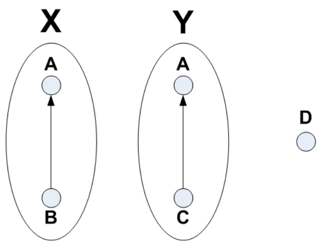


Figure 3.4: Avoiding dependency relations

simplified view of a module repository on a VL-e node. The dots represent modules, distinguished by characters A, B, C, and D. The arrows visualize dependency relations between modules.

Modules B and C depend on module A. This means that in order for B and C to function correctly, module A needs to be deployed. D on the other hand, is completely independent and needs no other modules to be installed.

Suppose that the module repository has reached its space or quota limit, and we need to remove a module. If a replacement policy judges that module D is the least needed, there arise no problems. However, when it is decided that A is the least needed module at the moment, we are left with *two* non working modules. B and C depended on A. Now B and C are useless and should be uninstalled. As can be seen, this conflicts with the replacement policy. Modules are removed which, according to the replacement policy, should *not* be removed.

There are several solutions to this problem. There is a replacement policy which can handle these kind of dependency issues. The Least Frequently Used policy counts the number of requests and removes the policy with the least amount of requests. This ensures that a ”parent” module (A) is never removed *before* a ”child” module (B or C). In the worst case scenario, a tie may occur. As a tiebreaker, we remove the module in the dependency tree, on which no other module depends. In other words, we remove the leaf of the tree. There is a drawback to this method, the tiebreaker may increase time and space complexity somewhat (since we need to maintain dependency trees).

Another method is to make each module independent. This is accomplished by packing the module the user would like to install, together with the complete module tree it depends on. This is depicted in figure 3.4. As can be seen, module X and Y both contain module A, on which B and C depend. X and Y are now completely separate entities since they both depend on their *own* A. While this resolves dependency problems, it is quite inefficient because there exist multiple copies of A in the repository. However, this could be partly resolved by including modules, which are very often needed, in the standard installation on a VL-e node. Modules included in the standard installation are immune from removal by a replacement policy.

Since we would like to compare different replacement policies in this thesis, and for simplicity reasons, we decide to assume the last scenario.

Chapter 4

Simulation and Results

In this chapter we discuss how we performed our simulations, which data we used as input and present the results of the simulations. In the final sections of this chapter, the results will be analyzed and discussed.

4.1 Simulating Module Caching

An artificial trace alone can not capture all of the behavior of a real trace. This is why we used a trace which we expect to have similarities with real future data of module requests.

We used a download log from the ftp-site of the *Netherlands UNIX User Group (http://www.nluug.nl)* for our trace-driven simulations. It contains all Linux RPM downloads (from several distributions) for a period of 11 days. It contains information such as size, url, time-to-download and time of download.

Although the download log is similar to a trace, there are some differences. NLUUG users download their files from a single server. There is no caching is involved, either. A VL-e trace would come a VL-e node receiving requests for modules. In case of a miss they are retrieved from another resource. While we're not denying that there are differences, the logs *do* have desirable properties which could reflect a real VL-e trace. First of all, a Linux RPM can be compared to a VL module. It is a software entity which adds functionality to a system, just like a VL-module. We expect a similar behavior and size distribution. In addition to this, we can use the time-to-download data in our trace. VL-e retrieves the modules from different resources. This means that bandwidth and latency may vary between these resources just as it does between users and the NLUUG ftp-site. We can therefore use time-to-download in our trace as the module retrieval latency

Table 4.1: Characteristics of the NLUUG trace

Trace	Trace period	Total Requests	Unique Requests
NLUUG	1/3/2005 - 12/3/2005	203252 (371559 MB)	51067 (89110 MB)

in case of a miss.

Because of the above reasons, we use the NLUUG log file as a basis for our trace-driven simulations. For some characteristics of the NLUUG trace, see table 4.1.

We are also interested in changing some aspects of the trace to observe possible (relative) variations in policy performances. We extract the unique modules from the NLUUG trace and use them as a basis for our artificially generated traces. In this way we can safely change a specific parameter to extremes, while the other parameters needed to build a trace are taken from the original NLUUG trace. This is clarified in the *Results* section (4.3).

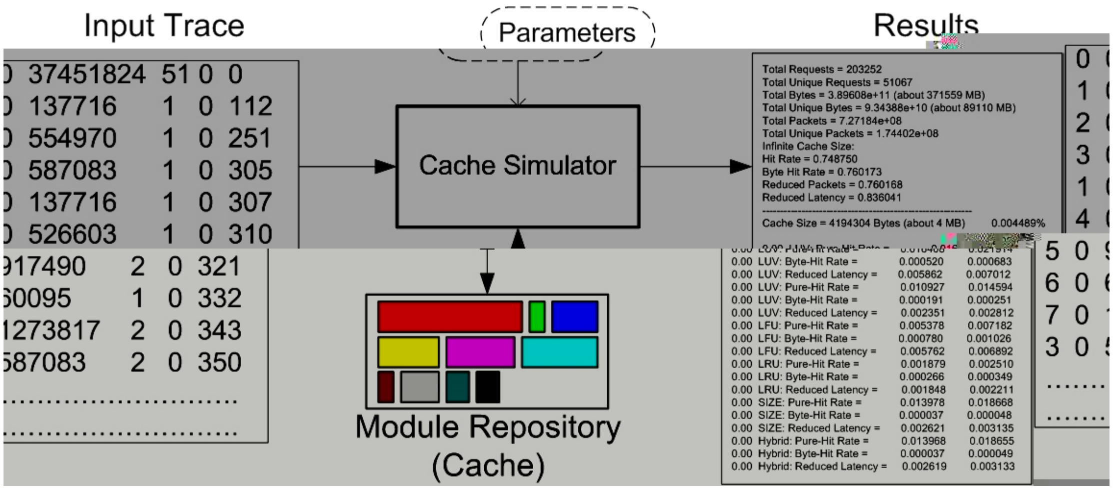
4.2 Cache Simulator

The open source simulator we use for our simulations was developed in 1997 by Pei Cao of the University of Wisconsin, and was programmed in C. It features a number of cache policies, LRU, Size and Hybrid.

We extended the simulator with the Least Unified Value and Least Frequently Used policies. LFU is interesting since it can be used as a reference policy because of its simplicity. LUV is one of the promising LRFU policies on the web caching front. *Bahn et al.* established that LUV was superior on all fronts (hit ratio, byte hit ratio and reduced latency) compared to other policies. We are interested to see if this is also the case in our type of VL-e trace.

An extra output format was added to the simulator as well, which can be used to easily generate plots.

We preprocessed the original log file so that it could be used for the simulator. All unique URLs were assigned a unique numerical id and we removed all incomplete downloads from the trace. One could argue that these incomplete downloads should not be removed: however, the simulator then assumes that the size of the file/module changes, which is not the case. Since there is no arbitrary way to distinguish between an incomplete download and a file-change from the trace file, we had to remove them.



```
) 37451824  51 0  0
) 137716     1  0  112
) 554970     1  0  251
) 587083     1  0  305
) 137716     1  0  307
) 526603     1  0  310
917490    2  0  321
50095     1  0  332
1273817   2  0  343
587083    2  0  350
.........................
.........................
```

```
Total Requests = 203252
Total Unique Requests = 51067
Total Bytes = 3.89608e+11 (about 371559 MB)
Total Unique Bytes = 9.34388e+10 (about 89110 MB)
Total Packets = 7.27184e+08
Total Unique Packets = 1.74402e+08
Infinite Cache Size:
Hit Rate = 0.748750
Byte Hit Rate = 0.760173
Reduced Packets = 0.760168
Reduced Latency = 0.836041
--------------------------------------------------------
Cache Size = 4194304 Bytes (about 4 MB)        0.004489%
```

```
0.00  LUV: Pure-Hit Rate =          0.010408     0.021914
0.00  LUV: Byte-Hit Rate =          0.000520     0.000683
0.00  LUV: Reduced Latency =        0.005862     0.007012
0.00  LUV: Pure-Hit Rate =          0.010927     0.014594
0.00  LUV: Byte-Hit Rate =          0.000191     0.000251
0.00  LUV: Reduced Latency =        0.002351     0.002812
0.00  LFU: Pure-Hit Rate =          0.005378     0.007182
0.00  LFU: Byte-Hit Rate =          0.000780     0.001026
0.00  LFU: Reduced Latency =        0.005762     0.006892
0.00  LRU: Pure-Hit Rate =          0.001879     0.002510
0.00  LRU: Byte-Hit Rate =          0.000266     0.000349
0.00  LRU: Reduced Latency =        0.001848     0.002211
0.00  SIZE: Pure-Hit Rate =         0.013978     0.018668
0.00  SIZE: Byte-Hit Rate =         0.000037     0.000048
0.00  SIZE: Reduced Latency =       0.002621     0.003135
0.00  Hybrid: Pure-Hit Rate =       0.013968     0.018655
0.00  Hybrid: Byte-Hit Rate =       0.000037     0.000049
0.00  Hybrid: Reduced Latency =     0.002619     0.003133
```

```
0 (
1 (
2 (
3 (
1 (
4 (
5 0 9
6 0 6
7 0 1
3 0 5
.......
.......
```

Figure 4.1: Using the Cache Simulator for VL-e Module Management

Figure 4.1 provides a general overview of how the cache simulator works. It simulates Module Management on one (arbitrary) VL-e node. On the left side of the picture is the trace, which serves as input for the cache simulator. The simulator also takes as input the following parameters:

- Replacement policies to simulate
- Cache sizes to simulate
- Number of simulated requests
- Output format
- (Optional) Replacement policy parameters

The simulator starts by reading the input trace file to memory. For each cache size and replacement strategy, one-by-one, the simulation is run.

The simulator adds the module of each request in chronological order to the Module Repository, which is the VL-e equivalent of the *cache*. The colored rectangles in figure 4.1 represent these modules. Each time the Module Repository is full, the least needed module is removed by the replacement policy. A hit is counted if the requested module is present in the Module Repository, at the time the request occurs. If this is not the case, a miss is counted. When each simulation finishes, statistics such as the hit ratio, the byte hit ratio and the reduced latency are written to the output file(s).

Depending on the replacement policy, information of all modules which passed is kept, or only of the modules present in the Repository. All information is maintained in dynamic arrays. These are cleared after each simulation. The next simulation (featuring a new cache size and/or replacement policy) starts with a "clean sheet".

When we obtain the results, they are processed by a Matlab function written by us. This function generates the plot presented in the *Results* section.

4.3 Results

The results of our simulations are discussed in this section. The results are represented in graphs, which are divided in three categories: hit ratio, byte hit ratio and reduced latency. On the x-axis we plot the cache size (in bytes). The simulation was performed using the following cache sizes: 4, 64, 512, 1024 and 2048 MB. These sizes represent the amount of storage available on a VL-e resource for the module repository. On the y-axis we plot the corresponding ratio.

The results of the following replacement policies are presented in all simulations:

- Least Unified Value (LUV)
- Least Frequently Used (LFU)
- Least Recently Used (LRU)
- Size
- Hybrid

Additionally, LUV is plotted with λ-parameters 0 and 1 in each graph, since these are the border values of λ. In some cases we plot LUV using a different λ. We refer to "LUV-λ" in the legend of the plots and in the text.

4.3.1 NLUUG Trace

In figure 4.2, on page 35, we present the results of the original NLUUG trace. As mentioned before, some characteristics of this trace file can be found in table 4.1.

Size and Hybrid perform very well in the hit ratio plot (figure 4.2(a)). Size discriminates large modules since they get evicted first, which explains the high hit ratio: the cache is filled with many small modules. Both Size and Hybrid perform much worse when we look at the byte hit ratio (figure 4.2(c)) and the reduced latency (figure 4.2(e)). LRU does not perform well in any metric in contrast with LFU. This implies that the emphasis lies on frequency information, as opposed to recency information in this particular trace file. LFU is the best in the byte hit rate graph, while LUV-0 outperforms all other policies when it comes to reduced latency.

When all metrics are taken into account, LUV-0 seems to perform best on average, which confirms the observation that frequency information is important here. As we explained earlier, if λ approaches 0, LUV is similar to a weighted LFU. Normalizing the value with the weight ($cost/size$) of a module thus has a positive influence on the performance of LUV-0, in the reduced latency plot.

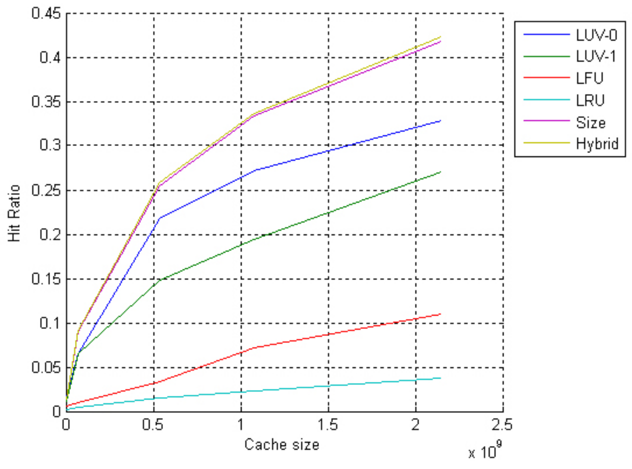
In [13] it is explained that LUV performance may be optimized by changing the value of λ. We tried to find some values which increase performance for the different metrics. They are plotted in figures 4.2(b) (hit ratio), 4.2(d) (byte hit ratio) and 4.2(f) (reduced latency). When λ is raised only a fraction above zero, performance immediately drops to LUV-1 level (which performs considerately worse than LUV-0). However, when we make λ small enough, we do see a minor improvement over LUV-0. In the plots differences are hard to see. For example, for $\lambda = 3 * 10^{-7}$, the hit ratio gives a slightly better performance than $\lambda = 0$ for a cache size of 2 GB. To give an idea, the hit ratio for a 2 GB cache size with $\lambda = 0$ is 0.327736, while the hit ratio for $\lambda = 3 * 10^{-7}$ is 0.327790.

The same applies for the byte hit ratio, where there is only a very small improvement. There are bigger improvements for the reduced latency metric. Although still not very spectacular, for a cache size of 2 GB the performance benefit for $\lambda = 6 * 10^{-8}$ versus $\lambda = 0$ is almost 0.3% (which is an improvement of 1.5%).

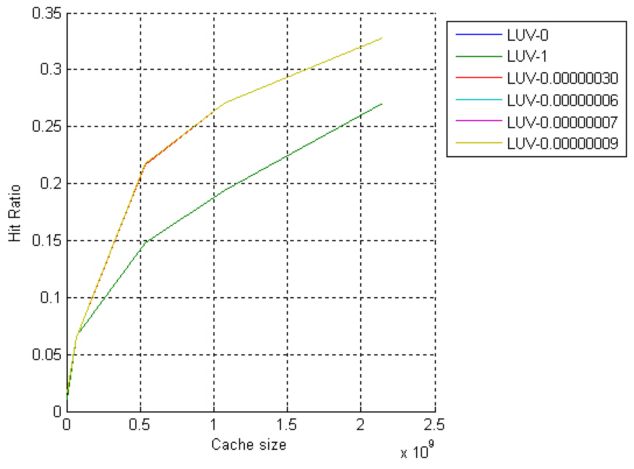
4.3.2 Artificial Traces

The NLUUG trace *may* have the same characteristics as a future VL-e module trace. However, since we can't establish this claim as a fact, a study needs to be done on traces with different properties.

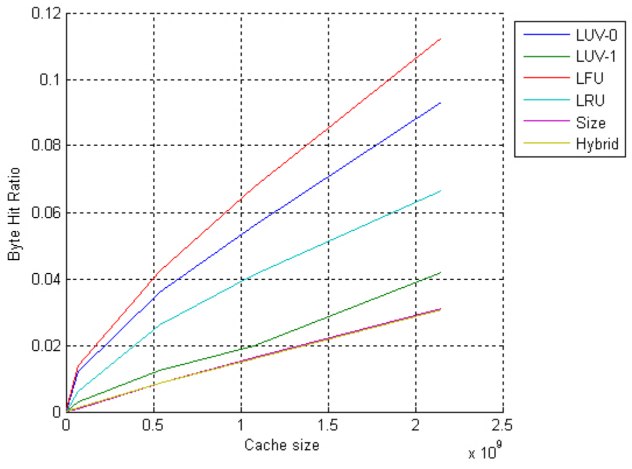
We developed a program which is able to generate a trace file, using a list of unique modules as input. For this research, we extracted the unique mod-



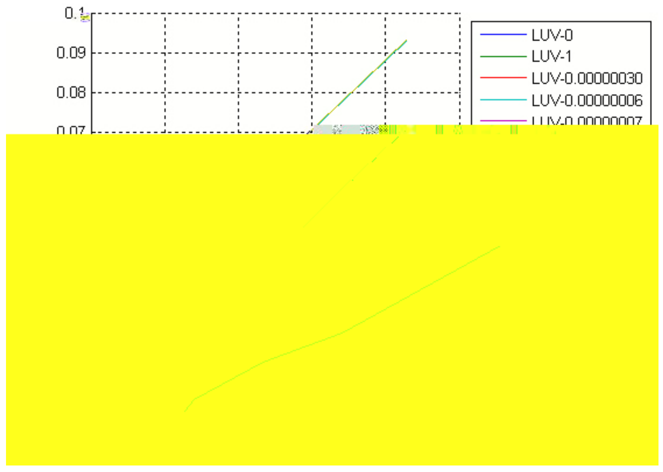
(a) NLUUG Hit Ratio



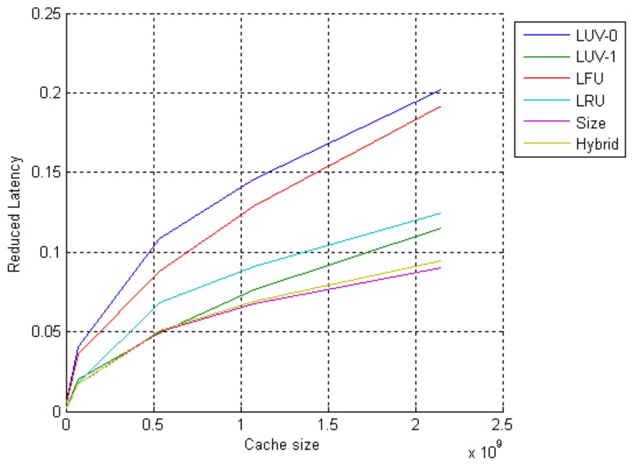
(b) NLUUG Hit Ratio (LUV)



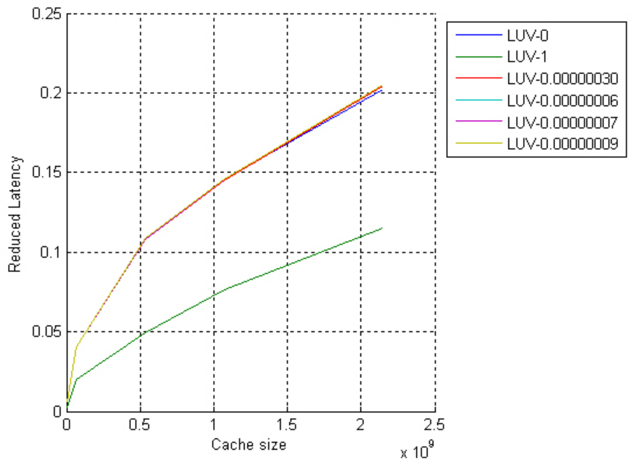
(c) NLUUG Byte Hit Ratio



(d) NLUUG Byte Hit Ratio (LUV)

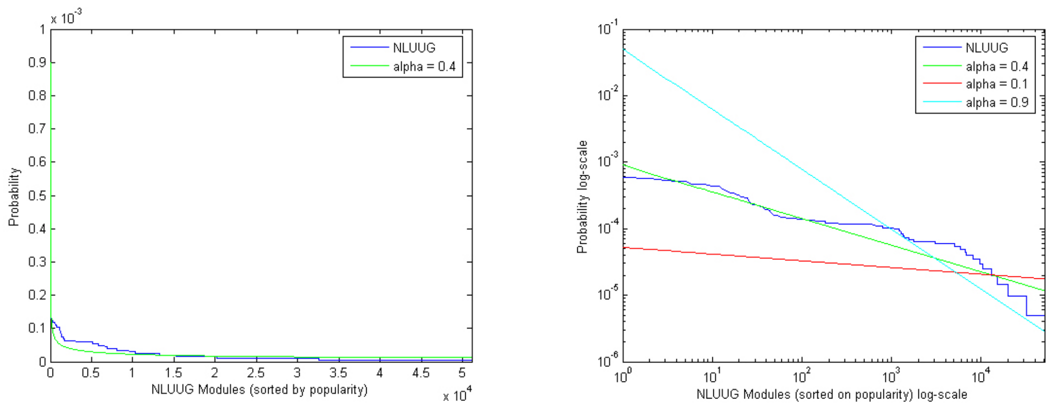


(e) NLUUG Reduced Latency



(f) NLUUG Reduced Latency (LUV)

Figure 4.2: NLUUG Trace Results



(a) Zipf-like Fit (linear axes)

(b) Zipf-like Fit (log axes)

Figure 4.3: Using the Zipf-like distribution to model module popularity

ules (in order of appearance), together with their sizes, from the NLUUG trace. The module file can then be used as a basis for generating our artificial traces. Parameters can be adjusted to generate a customized trace file:

- Time between trace occurrences
- Module retrieval bandwidth (in case of a miss)
- Module selection

Time and bandwidth can be set to a constant value. All three parameters can also be set to follow a certain *Probability Density Function* (PDF) such as the exponential distribution, normal (Gaussian) distribution or the Poisson distribution. A library for these distributions was used, which was written by Steve Park and Dave Geyer in 1998. We added the Zipf-like distribution ourselves, which is explained below.

4.3.2.1 Zipf-like Distribution

Our first objective is to find the frequency distribution of the NLUUG trace, since this is an important property which may affect the (relative) performance of different policies.

It is impossible to create a good fit of the original trace with any of the well known Probability Density Functions (Gauss, exponential etc.). However, L. Breslau, Pei Cao et al. proved in their publication [15] that the popularity of web documents usually follows a Zipf-like distribution. The relative access frequency for a document is inversely proportional to the rank of the document. In other words, some of the modules get a large percentage of the queries, leaving others to be asked for quite rarely. We would like to find out if the Zipf-like distribution fits our type of trace as well.

The Zipf-like distribution is defined as follows in equation 4.1:

$$P_N(i) = \frac{\Omega}{i^\alpha} \tag{4.1}$$

where

$$\Omega = \left(\sum_{i=1}^{N} \frac{1}{i^\alpha} \right)^{-1} \tag{4.2}$$

given the arrival of a new request in a trace, $P_N(i)$ is the probability that module i is requested. i represents the rank in popularity, where a lower i is more popular. N represents the total number of unique modules.

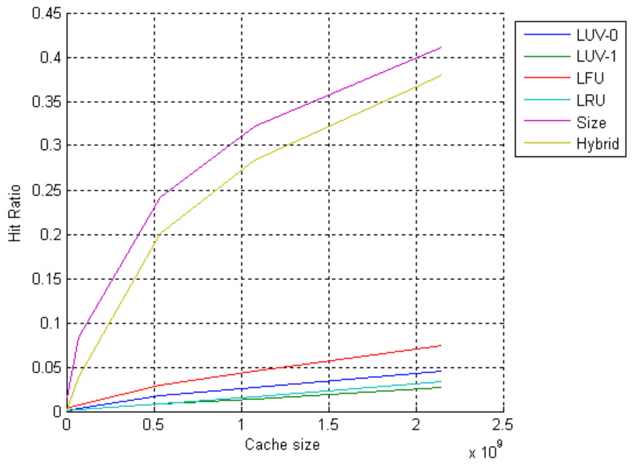
α is a parameter where $0 < \alpha < 1$. A higher α makes popular modules even more popular, while a lower α gives more weight to the tail. Breslau and Cao observed that in the case of web caching, α ranges from 0.6 to 0.8. To give an impression, in the case that $\alpha = 0.8$, 0.3% of the objects get 40% of the requests. In the original Zipf distribution [16], α was fixed to 1.

If we look at figure 4.3 we can see clearly that the Zipf-like distribution is a good fit for representing the popularity of modules. α is set to 0.4 for our trace.

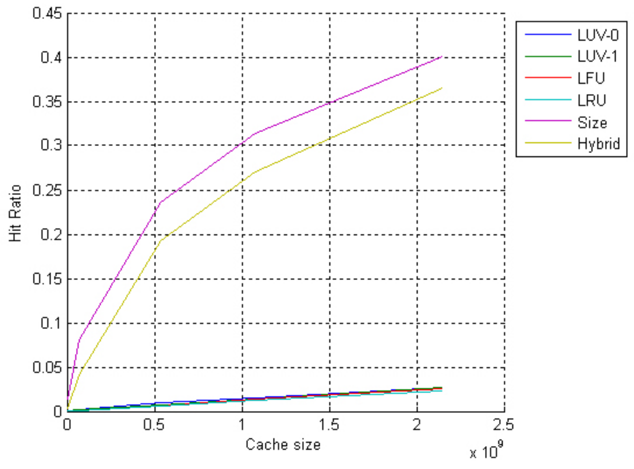
4.3.2.2 Zipf-like Traces

We start by generating artificial traces where the modules are selected using the Zipf-like distribution.

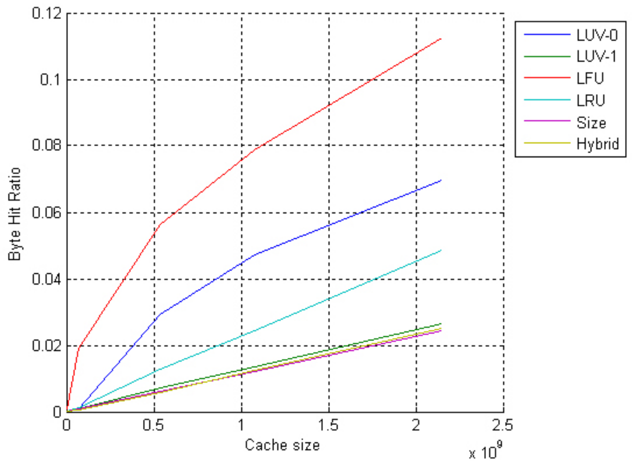
Trace 1 uses $\alpha = 0.4$, which is a good fit for the frequency distribution of the NLUUG data. The results of this trace are plotted in figure 4.4 in the left column. We observe the hit ratio (figure 4.4(a)), the byte hit ratio (figure 4.4(c)) and the reduced latency (figure 4.4(e)) metrics, and compare these to the results of the NLUUG trace (figure 4.2). While both sets of results look reasonably similar, there is a relative drop in performance of the recency based algorithms. Recency based algorithms are LRU and LUV. LUV is partly recency based. Recency information is clearly missing in our



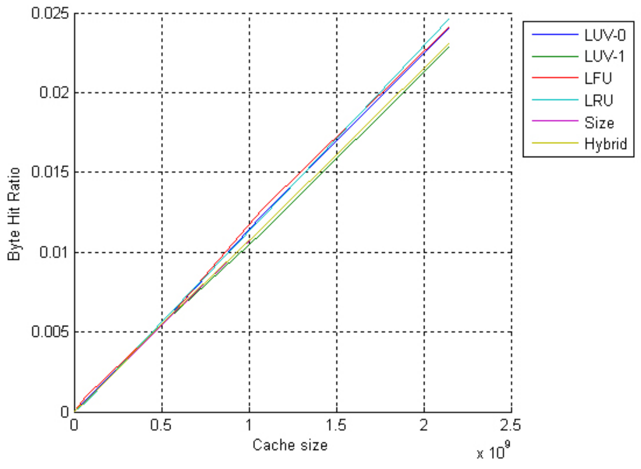
(a) Trace 1: Hit Ratio ($\alpha = 0.4$)



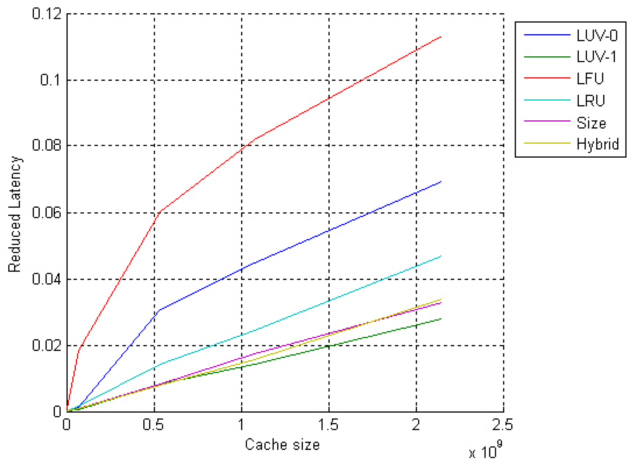
(b) Trace 2: Hit Ratio ($\alpha = 0.1$)



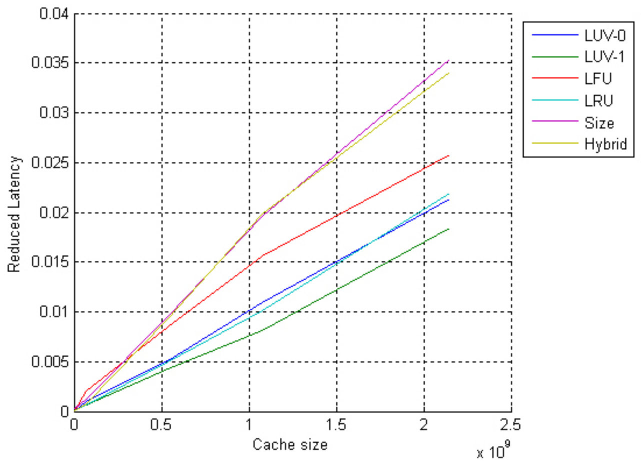
(c) Trace 1: Byte Hit Ratio ($\alpha = 0.4$)



(d) Trace 2: Byte Hit Ratio ($\alpha = 0.1$)

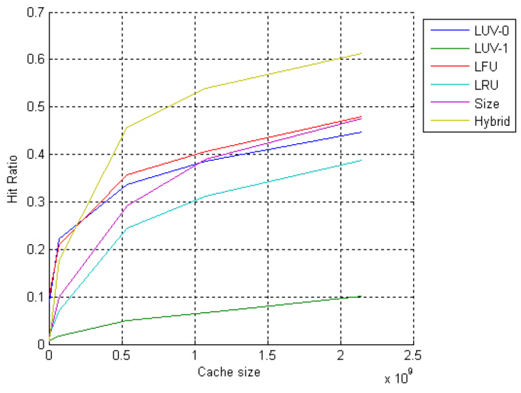


(e) Trace 1: Reduced Latency ($\alpha = 0.4$)

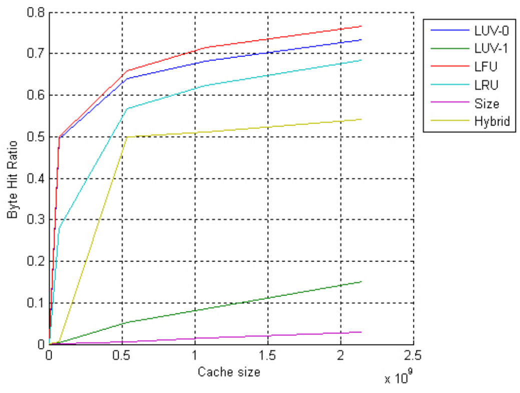


(f) Trace 2: Reduced Latency ($\alpha = 0.1$)

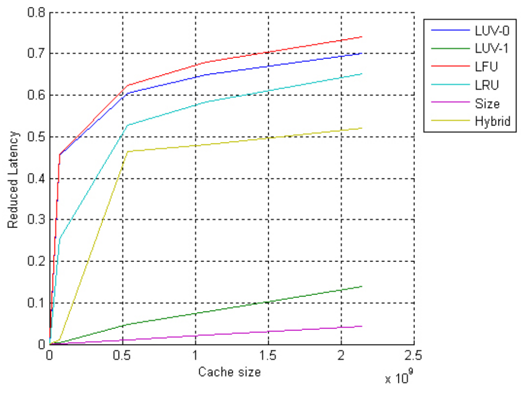
Figure 4.4: Zipf-like traces simulation results (Part 1)



(a) Trace 3: Hit Ratio ($\alpha = 0.9$)



(b) Trace 3: Byte Hit Ratio ($\alpha = 0.9$)



(c) Trace 3: Reduced Latency ($\alpha = 0.9$)

α α

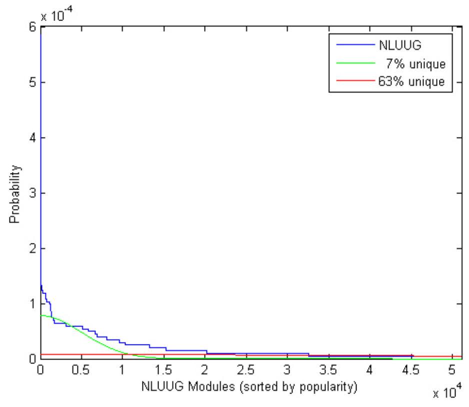


Figure 4.6: Using the normal (Gaussian) distribution to model module popularity

result in a somewhat uniform distribution. Then α is set to a very high value, which makes popular modules occur less frequently. The Zipf-like distributions are plotted in figure 4.3(b) with both the x and y-axis in log scale. This makes comparison easier.

Trace 2 (right column of figure 4.4) uses $\alpha = 0.1$. The distribution now has characteristics of a uniform distribution. Caching policies have much difficulty caching uniform distributions, since all modules occur at (almost) the same frequency. All policies have trouble coping with this kind of distribution. Only Size and Hybrid seem to be the positive exception when it comes to the the hit rate (figure 4.4(b)) and the reduced latency (figure 4.4(f)). Size only caches very small modules, which explains its superiority in the hit rate plot. The huge performance difference there also accounts for the small advantage of Size in the reduced latency metric, compared to the other policies. In the byte hit rate (figure 4.4(d)) all policies perform equally bad.

While a (nearly) uniform distribution has a devastating impact on cache policy performance, we expect policies not to have any problems with a trace where there are just a few very popular modules. This is the case for Trace 3 (figure 4.5), where $\alpha = 0.9$. For all 3 metrics we see very good results for most policies. Especially the frequency based policies perform very well. A large frequency of just a few modules also has a positive influence on recency based modules such as LRU. Only LUV-1 lags behind, which *is* recency based. However, a weight factor is included in the LUV policy. This may account for its poor performance.

4.3.2.3 Gaussian Traces

Since it is not sure that the popularity model in a real VL-e trace will comply to a Zipf-like distribution, we now do some tests with a Gaussian distribution.

The Gaussian (normal) distribution has a very different shape, like a bell shaped curve. The tail of a Gaussian distribution is relatively small, while the popular modules are not as popular as in a Zipf-like distribution. The normal distribution is symmetric, with the highest probability on the mean. The standard deviation (σ) determines the width of the distribution. Because we would like to be able to compare the results in an easy way with the Zipf-like distribution, we decided to retain the same popularity order. This implies that we set the mean at 0 and ignore negative values when generating the trace.

Our objective is to change the percentage of unique requests versus total requests. We would like to see if changing this parameter to extremes influences the relative performances of the replacement policies. In the original trace, the ratio of unique requests/total requests was 25%.

In the first Gaussian distributed trace, the curve is relatively narrow. The second trace has a much wider curve. In figure 4.6, a plot of the narrow, wide and original distribution curve is displayed for illustrative purposes.

The results of running the traces through the cache simulator are displayed in figure 4.7. On the left side of the page the results of Trace 4, which is the narrow curve, are displayed. On the right side the results of Trace 5 can be found.

In Trace 4, the unique request ratio is 7%. When we observe the plot of the hit ratio (figure 4.7(a)) we notice that Size and Hybrid again perform best here, just as in the NLUUG trace (figure 4.2). LUV-0 still follows in second place. This may be explained by the fact that there is a frequency benefit again. LFU-like algorithms (LFU and LUV-0) perform well. In the byte hit rate (figure 4.7(c)) LFU and LUV-0 also perform best, while Size and Hybrid end last. In the reduced latency (figure 4.7(e)) plot, LFU again takes the lead. The second place is for LUV-0, which just edges past Hybrid in performance. Hybrid here does reach its goal of optimizing reduced latency, where it didn't in the real trace.

Trace 5 is on the other end of the spectrum with its unique request ratio of 63%. To boost the unique request ratio, this trace only consists of 51000 requests, while all other artificial traces consist of the original 203252 requests from the NLUUG trace. Such a large percentage of unique requests make it very hard for policies to cache efficiently. Since the Gaussian curve is very wide (to boost the number of unique modules occurring) it approximates a uniform distribution, as we can see in figure 4.6. We see all policies struggling with this. Hybrid does a relatively good job in the byte hit rate (figure 4.7(d)) and the reduced latency (figure 4.7(f)). LFU drops to the last place, since frequency information has become almost useless. LRU is doing better. Both LUV's perform reasonably when we look at the reduced latency. It seems that the weight factor in LUV accounts for this, since it prevents having to retrieve modules with a relatively large cost again. In the byte hit rate all algorithms are very close.

Changing the λ parameter for the two previous traces again did not have much effect on the results, so we did not plot them.

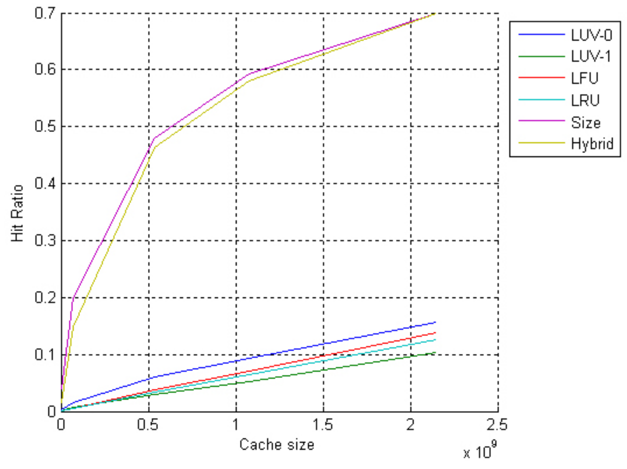
4.3.2.4 Shifting Gauss

The previous artificial traces were biased towards frequency based policies. We would now like to create a trace which is more sensitive to recency based policies and analyze the influence on our results. We have devised a method we call "Shifting Gauss" to accomplish this.

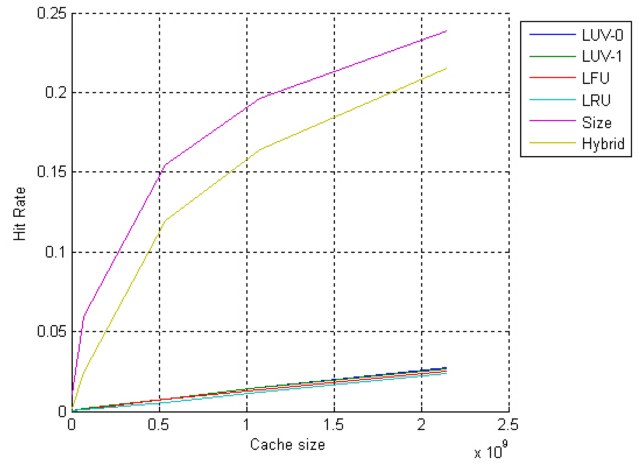
The method shifts a Gaussian distribution over a set of unique modules (in our case NLUUG) every few requests, to generate a trace. This means that the mean of the distribution shifts x modules every n requests. While generating the trace, the Gaussian distribution maintains a constant standard deviation (σ), which is relatively small. The idea behind "Shifting Gauss" is that it mostly eliminates frequency information. When we keep n small enough, frequency based policies are unable to adapt to the regularly changing set of modules. Recency based methods should not have any problems since they adapt quicker to these kinds of changes.

In figure 4.8 we have plotted an example of generating a trace using the Shifting Gauss method. Request 0 to 999 are generated from the red-colored distribution. Request 1000 to 1999 are generated using the green-colored distribution, etcetera. Each distribution has a σ of 1000. The distance between each gaussian curve equals $7 * \sigma$, which makes sure that the curves don't overlap too much. In this specific example the distance between each curve is 7000.

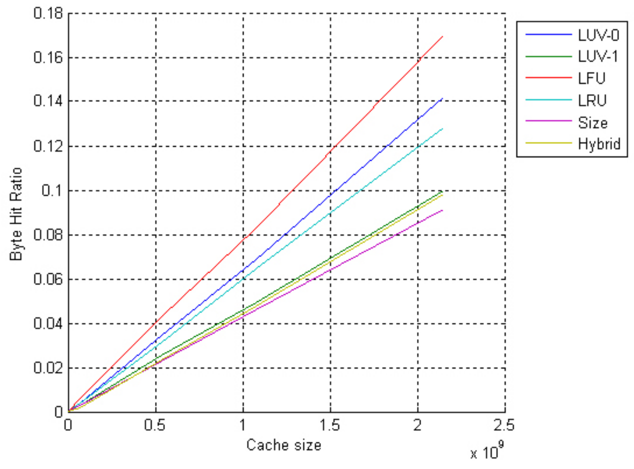
We have to keep in mind some issues while generating a trace to successfully prevent sensitivity to frequency based policies:



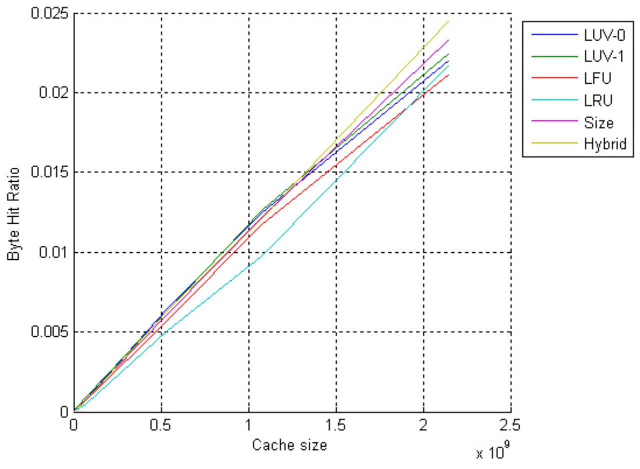
(a) Trace 4: Hit Ratio



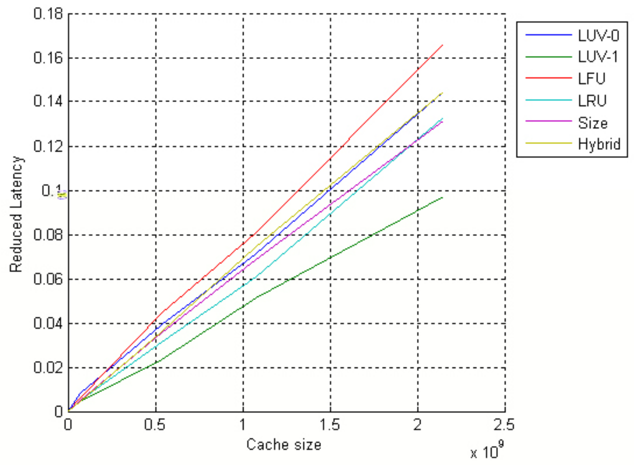
(b) Trace 5: Hit Ratio



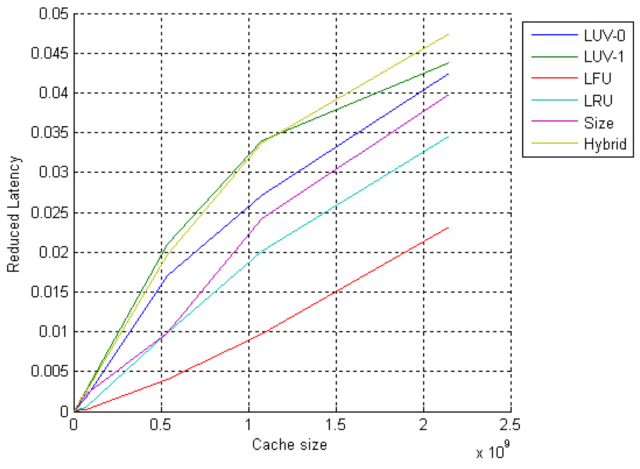
(c) Trace 4: Byte Hit Ratio



(d) Trace 5: Byte Hit Ratio



(e) Trace 4: Reduced Latency



(f) Trace 5: Reduced Latency

Figure 4.7: Artificial Trace Results: Unique/Total Requests Trace 4: 7% Trace 5: 63%

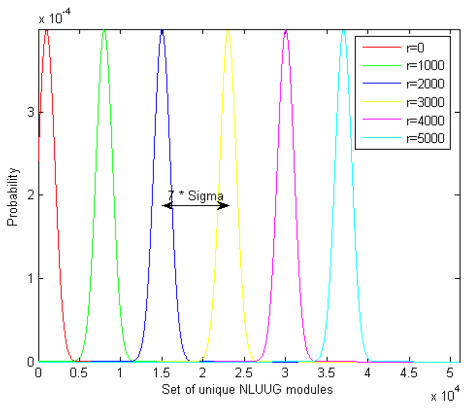


Figure 4.8: Shifting Gauss Example

- Shift the Gaussian distribution often
- Use a small σ
- Don't overflow the total amount of unique modules

Trace 6 uses a σ of 100 (modules) and shifts 700 modules every 1000 requests. The results are displayed in figure 4.9. In figure 4.9(a) we see that Size and Hybrid are defeated for the first time in a hit rate plot. LRU benefits strongly and gains a number 1 position. This is in line with what we expected. LRU performs best in the byte hit ratio (figure 4.9(b)) and the reduced latency (figure 4.9(c)) as well. All other algorithms perform much worse. The best of the other algorithms is LUV-1, which is the recency based variant of LUV.

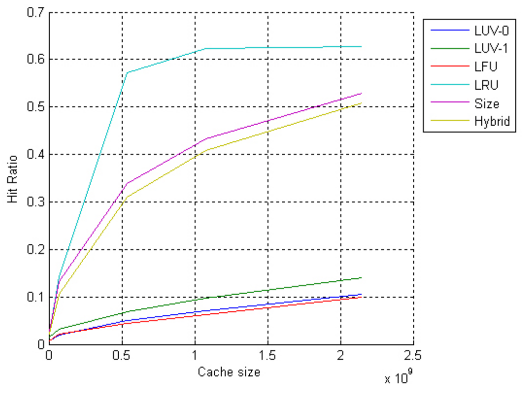
4.4 Discussion

We have performed many experiments, but there is one question we are most interested in.

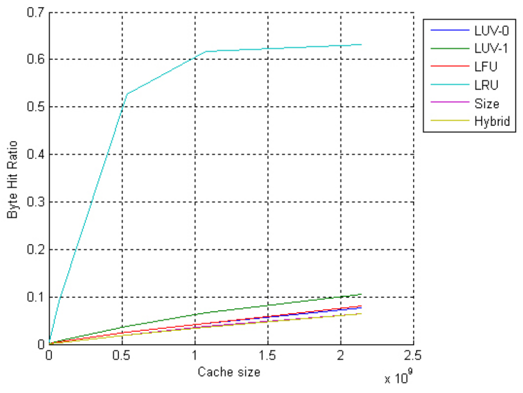
Which replacement policy should we use in our module management system?

Even after our experiments there is no clear answer to this question. The answer is that it really depends on the objective.

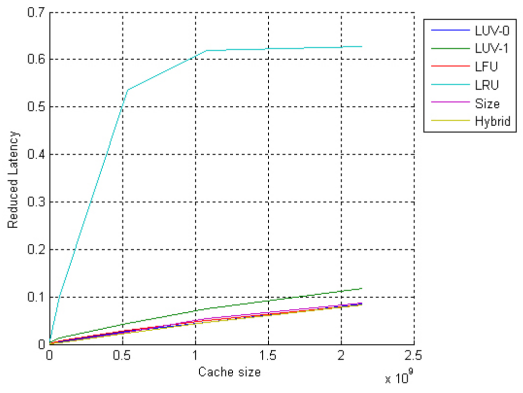
If we would like to optimize the hit ratio, Size or Hybrid are a very safe choice, since in almost every case they are the winners. Optimizing the hit ratio is nice for the user, since they usually don't have to wait for a small



(a) Trace 6: Hit Ratio



(b) Trace 6: Byte Hit Ratio



(c) Trace 6: Reduced Latency

Figure 4.9: "Shifting Gauss" Method

module to be retrieved and installed. However, a user does have a big chance to get a miss, if he/she requests a bigger module.

Our experiments show that when we optimize reduced latency, this comes at the cost of the hit ratio. The byte hit ratio and the reduced latency don't differ much in most cases.

So which metric would we like to optimize in the case of VL-e? In this case it would probably be the reduced latency, since optimizing this metric prevents unnecessary traffic. This saves money and prevents network saturation.

If we would like an algorithm which performs reasonable in all metrics, this would probably be LUV. Its performance is not as superior in our type of simulations as in [13] and [10] for web caching. However, the average performance still is impressive. The logarithmic computational and linear space complexity are very reasonable as well.

In most cases LUV-0 outperforms LUV-1, except for traces which have a very strong recency characteristic (such as in the Shifting Gauss method). Unfortunately we did not once measure any worthwhile benefit by adjusting the λ parameter between 0 and 1. We think that this has something to do with equation 3.7. The factor 1/2 may be too small for our type of trace. In web caching, time between occurrences is much smaller. Setting this value somewhat closer to 1 would give more weight to past occurrences, and may improve performance. This is something that we have not tested and could be part of future research. Currently, it is just a hypothesis.

Finally, the traditional policies LRU and LFU also performed well. However, they are not very suitable for broad range of traces since they only focus on recency and frequency information, respectively.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, we discussed a component in the Virtual Laboratory for e-Science which should be automated, namely module deployment and management. Using an representative example of the problems which arise in the current system, we concluded that it is absolutely *essential* to automate module deployment. An architecture was introduced which can be used as a blueprint for implementation.

Existing software deployment and packaging systems were compared and analyzed for their suitability for module deployment.

From a scientific perspective, we were more interested in optimizing module management. This is why we focused our research on this aspect. The question was raised how modules should be removed in case we run out of hard disk space. Several replacement policies from hardware caching and web caching were introduced.

We investigated, with the help of simulations, if these policies can be used effectively within the VL-e module management environment. In the simulations, we applied these policies to a download trace from the Netherlands UNIX User Group (NLUUG). In addition to this, artificial traces were generated using a program we wrote. These artificial traces were used to set parameters to extremes and measure the effects on performance of the policies.

There are many differences between caching systems in general and module replica management. We may conclude from our simulations however, that using traditional caching policies in a VL-e environment, does result in a good performance.

As we discussed in section 4.4, it is hard to pick which is the best policy. It depends on the nature of the trace. As a starting point, we would recommend using LUV with $\lambda = 0$, which shows an above average performance on almost any trace and metric. When the VL-e system is up and running, a suitable policy should be chosen by analyzing real traces and by taking into account the metric we would like to optimize. We recommend maximizing the reduced latency, since this spares download time and saves on network load.

The way we have addressed the problem is general enough to make our work widely usable. The design may be incorporated into existing grid-based projects, possibly with some modifications.

5.2 Future Work

The system we introduced is a blueprint which still has to be implemented. The module management software could be integrated in a module itself and made a part of the standard VL-e installation package for new nodes. The simulator can be easily adapted so that it could be used for real traces and as part of the management system.

One of the most important goals of VL-e is to be as generic as possible. Therefore, an important part of future research should be focused on this requirement. We already briefly touched some issues. In fact, most limitations considering module deployment and management are in the package managers which are currently available. The current packaging systems have some limitations, such as limited computer language support and limited OS support. Windows and Macintosh are not supported. It might be worth researching other packaging systems, or extending current systems. This makes a more generic VL-e possible.

Future research may also include a system which implements dynamically changing replacement strategies, when the nature of the trace changes. Also, a better module replacement strategy may be found if it is specifically designed for VL-e traces. However, existing policies seem to give good results.

A completely satisfying solution to the dependency issues addressed in section 3.3.4 was not found. We found that little research has been done in the area of caching with dependency relations. The examples presented in section 3.3.4 on page 27 were very simple. Dependency trees may be much more complex. Also, issues such as version management should be considered: does each version of a module have its own dependency tree? This could be a very interesting future research topic.

Finally, in the area of module deployment, more research could be done on efficient module retrieval. Currently, the system downloads from only one source (a node, or a global repository). In order to decrease download time for large modules and spread network load, something similar to BitTorrent could be implemented. BitTorrent divides a file (or module) in several parts and downloads it from several sources at a time. To implement multi-threaded downloads, the current deployment and packaging systems should be modified.

5.3 Acknowledgements

I would like to thank my supervisors Adam Belloum, Dmitry Vasunin from the VL-e project group. Their constructive ideas and honest comments ensured that this thesis is at least *somewhat* readable ;-). I would also like to extend my gratitude to the Netherlands UNIX User Group who kindly provided me with a log file to base our simulations on. Finally, a word of thanks to my family, who supported me when writing this thesis.

Any errors, idiocies and inconsistencies in this thesis remain my own.

Bibliography

[1] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150, 2001.

[2] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[3] Adam S.Z. Belloum, David L. Groep, Zeger W. Hendrikse, Bob L.O. Hertzberger, Vladimir Korkhov, Cees T.A.M. De Laat, and Dmitry Vasunin. VLAM-G: A grid-based virtual laboratory. *Future Generation Computer Systems*, 19:209–217, 2003.

[4] G.B. Eijkel, H. Afsarmanesh, D. Groep, A. Frenkel, and R.M.A. Heeren. Mass Spectometry in the Amsterdam Virtual Laboratory: Development of a High-Performance Platform for Metadata Analysis. In *Proceedings of the 13th Sanibel Conference on Mass Spectrometry*, Sanibel Island, FL, USA, 2001.

[5] Vladimir Korkhov, Adam S.Z. Belloum, and Bob L.O. Hertzberger. VL-E: Approaches to Design a Grid-Based Virtual Laboratory. In *Proceedings of the DAPSYS 2004 Conference*, Budapest, Hungary, 2004.

[6] Vladimir Korkhov, Adam S.Z. Belloum, and Bob L.O. Hertzberger. Evaluating Meta-scheduling Algorithms in VLAM-G Environment.

[7] Advanced Packaging Tool (Debian). http://www.debian.org/doc/manuals/apt-howto/apt-howto.en.pdf.

[8] Edward C. Bailey. Maximum RPM: Taking the Red Hat Package Manager to the Limit. 1997.

[9] Yellow dog Updater, Modified. http://www.linux.duke.edu/projects/yum/.

[10] Abdullah Balamash and Marwan Krunz. An Overview of Web Caching Replacement Algorithms. *Communications Surveys and Tutorials*, 6:44–56, 2004.

[11] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of the ACM SIGCOMM '96 Conference*, Stanford University, CA, 1996.

[12] R. Wooster and M. Abrams. Proxy Caching that Estimates Page Load Delays. In *Proceedings of the 6th International World Wide Web Conference*, pages 325–334, Santa Clara, CA, 1997.

[13] Hyokyung Bahn, Kern Koh, Sam H. Noh, and Sang Lyul Min. Efficient Replacement of Nonuniform Objects in Web Caches. *Computer*, 35:65–73, 2002.

[14] D. Lee et al. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Computers*, 50(12):1352–1361, 2001.

[15] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM (1)*, pages 126–134, 1999.

[16] George Kingsley Zipf. Relative frequency as a determinant of phonetic change. *Reprinted from the Harvard Studies in Classical Philiology*, Volume XL, 1929.