Winter 2013

Using power-law properties of social groups for cloud defense and community detection

Justin L. Rice

USING POWER-LAW PROPERTIES OF SOCIAL GROUPS FOR CLOUD DEFENSE AND COMMUNITY DETECTION

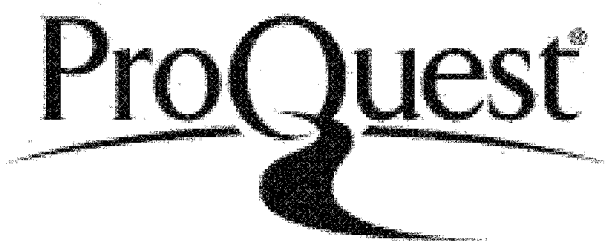
by

Justin L. Rice, B.S., M.S.

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

COLLEGE OF ENGINEERING AND SCIENCE
LOUISIANA TECH UNIVERSITY

March 2013



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

LOUISIANA TECH UNIVERSITY

THE GRADUATE SCHOOL

1/14/2013

Date

We hereby recommend that the dissertation prepared under our supervision by Justin L. Rice

entitled

Using Power-Law Properties of Social Groups for Cloud Defense & Community Detection

be accepted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Vir Vasandr Phoha

Supervisor of Dissertation Research

Head of Department

College of Engineering and Sciences

Department

Recommendation concurred in:

Advisory Committee

Approved:

Director of Graduate Studies

Approved:

Dean of the Graduate School

Dean of the College

GS Form 13a
(6/07)

ABSTRACT

The power-law distribution can be used to describe various aspects of social group behavior. For mussels, sociobiological research has shown that the Lévy walk best describes their self-organizing movement strategy. A mussel's step length is drawn from a power-law distribution, and its direction is drawn from a uniform distribution. In the area of social networks, theories such as preferential attachment seek to explain why the degree distribution tends to be scale-free. The aim of this dissertation is to glean insight from these works to help solve problems in two domains: cloud computing systems and community detection.

Privacy and security are two areas of concern for cloud systems. Recent research has provided evidence indicating how a malicious user could perform co-residence profiling and public to private IP mapping to target and exploit customers which share physical resources. This work proposes a defense strategy, in part inspired by mussel self-organization, that relies on user account and workload clustering to mitigate co-residence profiling. To obfuscate the public to private IP map, clusters are managed and accessed by account proxies. This work also describes a set of capabilities and attack paths an attacker needs to execute for targeted co-residence, and presents arguments to show how the defense strategy disrupts the critical steps in the attack path for most cases. Further, it performs a risk assessment to determine the likelihood an individual user will be victimized, given that a

successful non-directed exploit has occurred. Results suggest that while possible, this event is highly unlikely.

As for community detection, several algorithms have been proposed. Most of these, however, share similar disadvantages. Some algorithms require apriori information, such as threshold values or the desired number of communities, while others are computationally expensive. A third category of algorithms suffer from a combination of the two. This work proposes a greedy community detection heuristic which exploits the scale-free properties of social networks. It hypothesizes that highly connected nodes, or hubs, form the basic building blocks of communities. A detection technique that explores these characteristics remains largely unexplored throughout recent literature. To show its effectiveness, the algorithm is tested on commonly used real network data sets. In most cases, it classifies nodes into communities which coincide with their respective known structures. Unlike other implementations, the proposed heuristic is computationally inexpensive, deterministic, and does not require apriori information.

APPROVAL FOR SCHOLARLY DISSEMINATION

The author grants to the Prescott Memorial Library of Louisiana Tech University the right to reproduce, by appropriate methods, upon request, any or all portions of this Dissertation. It is understood that "proper request" consists of the agreement, on the part of the requesting party, that said reproduction is for his personal use and that subsequent reproduction will not occur without written approval of the author of this Dissertation. Further, any portions of the Dissertation used in books, papers, and other works must be appropriately referenced to this Dissertation.

Finally, the author of this Dissertation reserves the right to publish freely, in the literature, at any time, any or all portions of this Dissertation.

Author Justin L. Rice

Date 02/08/2013

DEDICATION

To my family.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

There is an old African proverb which states: "it takes a village to raise a child." If this is indeed true, then imagine how many more people would be involved in producing a minority Ph.D. in the STEM discipline. Considering this, it is safe to say that many thanks are in order for I stand on the shoulders of those who have paved the way for me. There is simply not enough space and time to express the full extent of my gratitude, but I would be remiss if I did not recognize a few special individuals. First and foremost, I would like to thank God. There are some things that happened along the way that were nothing short of miracles. So, thanks goes out to the Invisible Force which shapes my destiny and guides me along my path of purpose. I would like to thank my mother and friend, Colette Rice, for her patience and support. She sacrificed her life's goals, aspirations, and dreams to raise her children. I just want her to know that I really appreciate her presence, her advice, her strength, and her wisdom. She is truly one of a kind. All a son could ask for and more. I want to thank my late father, William Rice, III. He provided for the family, and taught me many life lessons. I would like to thank my youngest brother, Timothy Rice. His strength and will to live in spite of the prognosis of his disease gives me the motivation needed to press on through any of life's obstacles. Special thanks goes to Jackson State University: Robert Whalin, Angie Jackson, Ora Rawls, Khalid Abed, Gerald Morris, Mahmoud Manzoul, Gordon Skelton, and Kamal Ali. They have truly shown their support both during my tenure at Louisiana Tech University and during my family's hour

of need in the passing of my father in 2009 and my brother's cancer diagnosis in 2011. To my "Board of Directors," thanks for support and encouragement: Miguel Gates, Ilija Pjescic, Divya Narayan, Lamarious Carter, and Ivan Walker. I would especially like to thank Miguel for his assistance while I was physically absent from Tech's campus. I would also like to thank the NASA Goddard Space Flight Center family for the moral support and for helping me with the school/work balance: Damon Bradley, Janine Dolinka, Lori Moore, Kenneth Rehm, Jerome Bennett, and Charles Wildermann. Thanks to Louisiana Tech University for all the academic support and assistance: Bala Ramachandran, Galen Turner, Terry McConathy, Anita Young, Gloria Skains, Marilyn Robinson, Rachel Parker, and Brenda Stapleton. Last, but certainly not least, much gratitude goes to my advisor Vir V. Phoha. The invaluable lessons I learned from this experience will undoubtedly help me to become a better researcher, professional, and individual. Thanks.

CHAPTER 1

INTRODUCTION

Nature is one of life's most fascinating mysteries. It is self-regulated, highly decentralized, and ever-evolving. Understanding how nature works may be the key to solving some of man's most challenging research questions. In recent years there has been a growing interest in using interdisciplinary studies to investigate group behavior and the underlying dynamics of within-group interactions. Towards this end, researchers have begun to explore social animals - populations of interacting individuals that operate in a cooperative fashion to survive. Examples of group activities include: foraging, nest building, and group defense. Often times these interactions are asynchronous and decentralized - exhibiting emergent properties, where basic communication and action on local scales lead to complex phenomena on a global scale. Examples of emergent behavior include: collective harvesting, flocking of birds, self-organization, standing ovations, and traffic jams.

1.1 Research Questions

This research focuses on social groups from two different perspectives and seeks to answer two fundamental questions.

1. From an insider's perspective, in what ways do communities emerge? That is, what internal processes have to occur on the micro-level to have group formation emerge

on the macro-scale? Can mussels and Lévy walks be used to describe these processes? How can this type of behavior be used as a defense strategy?

2. From an outsider's perspective, how to detect communities once they have formed? Given each individual's local connections only, is it possible to classify individuals into their respective known global communities? How can the scale-free properties of social networks help shed light on this problem?

1.2 Limitations and Assumptions

The idea is to use insight gained from social group behavior to develop algorithms and social simulations that are, in turn, used to help solve real world problems. However, one well-known characteristic of this type of approach is local convergence. Since individuals use peers within their vicinity to make decisions, this may lead to premature results - local optima where local best is taken as global best. Though this phenomena is inherent for populations of interacting individuals, this research does present ways to mitigate its occurrence.

For community detection, this work assumes non-overlapping communities - the notion that a network can be divided into disjoint communities where each individual belongs to one (and only one) group. However, in reality, it is quite possible for groups to overlap where individuals belong to one of several groups.

1.3 Dissertation Overview

This dissertation aims to use power-law properties of social groups to develop: a defense strategy for cloud systems and a greedy heuristic for community detection. Chapter 2 introduces an aspect of the technical cloud framework used by the work done in

Chapter 3. Among other things, it details the notion of using a proxy server to serve as an intermediary between Internet users and multiple virtual machines. In light of an attack strategy recently described by researchers in [1], Chapter 3 proposes a defense strategy, in part inspired by mussel self-organization behavior. Chapter 4 presents a greedy heuristic for community detection which exploits the scale-free nature of social networks. Lastly, Chapter 5 provides concluding remarks and ideas for future directions.

CHAPTER 2

WEB FARM-INSPIRED CLOUD FRAMEWORK

In this chapter, we introduce a web farm-inspired framework for dynamic and concurrent computational processing in the cloud. We compare and contrast this with the Hadoop-cloud framework, discuss the main problems associated with our approach, and give suggestions on ways to overcome said challenges. To implement the web-inspired framework, we use Node.js - a lightweight, single threaded, server-side framework which uses asynchronous callbacks to allow nondependent operations (parallel-like sections) to execute while waiting for I/O events such as "fetching a file" or "writing a file to disk." We perform experiments to reveal two preliminary results that showcase the framework's functionality and scalability.

One, for non-blocking operations, worker nodes which use Node.js servers are significantly faster than those which use traditional servers. In particular, a single Node.js is (on average) 2.11 times faster than one Ruby Webrick server, and is (on average) 1.88 times faster than two Ruby Webrick servers. Two, we find that increasing the number of worker nodes improves overall performance for blocking computational operations. As the number of worker nodes increases, the total execution time decreases exponentially and the number of requests per second increases linearly.

2.1 Introduction

Cloud computing has recently emerged as an information technology solution which facilitates access to on-demand utility storage and computing services. Customers are afforded the economy of scale at a low operational overhead. These features prove to be especially advantageous to small companies which lack financial resources and physical manpower. For larger companies, security and the privacy are more issues of concern; so, some organizations have developed private clouds, or even hybrid clouds, purposed to serve internal communities and trusted constituents. Research communities have also considered the cloud environment as a viable alternative platform for scientific workflows. Though promising, the jury is still out on whether this technology marks the next generational shift in platform of choice for low-cost computational processing and analysis [2–4].

The Internet, on the other hand, has found a formula that works - a formula which is efficient and highly scalable. This success is, in part, due to the way in which web services are hosted. More often than not, they take the form of web server farms. These farms are usually comprised of three basic components: load balancer, web servers, and a shared database server. The load balancer receives all the requests and forwards them to the web servers according to some scheme, i.e., round robin, least connection, source. The same content is duplicated on each server; so, it does not matter which server handles the request. Since this is the case, all servers must use a shared database for storage. This type of infrastructure grants websites the ability to accommodate the dynamic and bursty nature (sudden and dramatic increase in requests) of web traffic.

We would like to bring these types of concepts to the science domain. In particular, we envision a web farm-like framework for dynamic and concurrent computational processing in the cloud - an approach which allows researchers to quickly build, configure, and perform experiments using tools traditionally designed for web applications. For web services, developers create and deploy web applications to hypertext transfer protocol (HTTP) web servers which, in turn, receive, process, and deliver web content. We propose a similar framework whereby researchers develop and deploy computational applications to compute servers which, in turn, receive computational tasks (via representational state transfer (RESTful) requests over HTTP), perform computations, and deliver science products.

With this, researchers will have the freedom to work on either a system level, an application level, or both. Working on a system level gives one direct control over the infrastructure, i.e., the interface to computational services, the job routing scheme, the total number of nodes, the number of compute servers per worker node, etc. On the other hand, the application level is more concerned with developing the computational algorithm, which will be deployed as a job to run on the system - what executes, how it executes, and the resources it requires. There are several reasons why this framework would be beneficial. One, the basic infrastructure for communication between heterogeneous-roled components already exists. Two, most of the tools for infrastructural development, experimentation, and analysis are freely available (open-source), widely used, and rigorously tested. Three, this "building on existing web components approach" provides an infrastructural abstraction, and allows the researcher to focus on the algorithm itself, thereby increasing developer productivity.

There are a number of other frameworks which leverage distributed resources for computational processing, i.e., Hadoop [5] with MapReduce [6], Pegasus [7], Swift [8], SAGA [9], Condor [10], etc. However, the current state of the art does not easily lend itself to the degree of flexibility we require. Our ultimate goal is to dynamically upload and process custom web coverage processing service (WCPS) [11] algorithms on heterogeneous platforms. Platforms of interest include: cloud, unmanned aerial vehicles (UAV), satellites, etc. In most of these cases, the upload bandwidth will be extremely limited. Implementing the framework on the cloud provides a basic proof of concept. We later plan to expand the implementation to other platforms.

This chapter presents a web farm-inspired framework for dynamic computational processing in the cloud. It is divided into eight sections. Section 2.2 compares and contrasts this approach with Hadoop-cloud framework. Section 2.4 presents and explains the web farm-inspired design, and goes into detail about the Node.js framework. Section 2.5 describes the cloud services we use, and provides a general overview on our approach toward implementing the web farm-inspired framework. Section 2.6 conducts experiments (for blocking and nonblocking operations) to reveal preliminary results which showcase the framework's functionality and scalability. Section 2.7 reviews the results, discusses the challenges associated with this approach, and identifies ways in which they can be overcome. Finally, Section 2.8 provides concluding remarks.

2.2 Hadoop-Cloud Framework

Most existing distributed application frameworks (i.e., Hadoop) are designed to handle large data processing jobs in a static batch-like fashion. When combined with a

web service such as Amazon Elastic MapReduce [12], compute nodes could be provisioned on-demand to complete the job as fast as possible. To do this, individual nodes have to be assigned individual tasks in a coordinated and organized manner. Once each node receives and completes its respective task, each result must be collected. This is accomplished via MapReduce. Here, a one (master) to many (workers) relationship is assumed. Furthermore, there are two distinct phases: Map and Reduce. In the Map phase, the master node splits the big job into smaller tasks and assigns them to available worker nodes. In the Reduce phase, the master collects all the individual results and combines them in such a way to generate the desired overall output. Additionally, there is a distributed file system for scalability, data reliability, and fault tolerance.

The Hadoop approach caters specifically to big jobs; or small jobs which have been bundled together to form big jobs. Either way, the processing engine will not be able to process any new job until the current job has been completed. Thus, while processing, there are two ways to handle dynamic requests:

1. **Option One**:
   (a) Reject them OR
2. **Option Two**:
   (a) Collect requests
   (b) Form batch jobs which consists of many requests
   (c) Queue batch jobs (first in, first out (FIFO) default) or use custom schedulers, i.e., CloudBATCH [13]
   (d) Process batch jobs (when available)

If we choose Option Two, we would have to determine whether all batch jobs should be heterogeneous or homogeneous in size. If jobs are homogeneous, then problems could arise in a situation where the demand goes down and there are not enough requests to make a standard job. Perhaps determining an efficient "job padding" technique should suffice. If, on the other hand, jobs are heterogeneous, a priority scheduling scheme would be needed to prevent small jobs from being processed at the mercy of large jobs. As such, scheduling this many (big jobs) to one (processing cluster) computational paradigm may not be the best approach when processing smaller sized job requests ($\ll$ Hadoop job), i.e., requests made via web.

As it stands, there is no standard framework to handle dynamic computational requests. Usually, researchers have to build the infrastructure from the ground up [14]. Towards this end, we propose web farm-inspired computational cluster.

2.3 Related Research

Existing research to date primarily focuses on pleasingly parallel scientific application dataflows with large datasets. This is attractive because it caters to two strengths of the cloud computing paradigm: on-demand scalability and minimal to no communication between compute nodes. Below, we give a few examples.

Wall et al. implemented reciprocal smallest distance (RSD), a comparative genetics algorithm, using Amazon's Elastic Computing Cloud (EC2). They executed approximately 300,000 tasks on 100 high capacity compute node in about 70 hours at a cost of $6,302 USD [15]. Zhang et al. [16] use Hadoop to develop a cloud application which processes sequences of microscopic images of live cells. They conclude that "Hadoop allows to speed

up calculations by a factor that equals the number of compute nodes." Vecchiola, Pandey, and Buyya use Aneka, a cloud computing solution, to classify gene expression and execute an fMRI brain imaging algorithm [17]. From their findings, they state: "... large high performance applications can benefit from on-demand access and scalability of compute and storage resources provided by public Clouds." Lu, Jackson, and Barga [14] seek implement the AzureBLAST, their parallel version of the Basic Local Alignment Search Tool (BLAST) algorithm, on Windows Azure. In regards to the scalability of their design, they report: "... the throughput of AzureBlast increases almost linearly when given more instances." They also find that the read throughput of the Azure blob storage increases with the number of instances. Lastly, Gunarathne et al. deploy pleasingly parallel biomedical applications to the cloud environment [18]. They maintain that "cloud infrastructure based models as well as the Map Reduce based frameworks offered very good parallel efficiencies given sufficiently coarser grain task decompositions."

From reviewing those above as well as others not presented here [19–21], we conclude that application-specific algorithmic development is a non-trivial task. Issues such as overall system architecture, job partitioning and allocation, inter-node communication, etc. must be addressed. The approach to these issues may vary significantly from one application to the next. Thus, developers should be well-versed and familiarized with the application itself and the inner-workings of the cloud environment, as naïve implementations usually will not render impressive results.

2.4 Web Farm-Inspired Framework

2.4.1 Assumptions

Before we go into the details of the system design, we feel it is appropriate to point out our assumptions. First, we take nodes to be representative of virtual machines (VMs); and we also take algorithms to be representative of computational kernels. We use these terms interchangeably. Second, users submit jobs asynchronously via some web application. However, when testing to find the limitations of the system, the worst case scenario is assumed - all users submit jobs simultaneously. Third, the system adopts a one master to many workers distributed paradigm, whereby the master ONLY distributes jobs to multiple worker nodes. Fourth, individual jobs are independent of each other, execute on one worker node, and do not need to be partitioned. Given this, there is no need for MapReduce. Fifth, the master uses some scheduling technique to distribute jobs, i.e., round-robin, least connections, source, etc. Sixth, we assume a NO SHARE architecture. This means that there is no inter-worker node communication or inter-worker node dependencies. Seventh, if a worker node receives new jobs while it is processing, it simply queues and processes them in a FIFO fashion (when available).

2.4.2 System-Level Components

As shown in Figure 2.1, the design contains the same basic components one would expect to find in a traditional web farm. Here, the master HAProxy [22] node distributes (load balance) jobs to worker nodes. Each worker contains a compute server which wraps some computational kernel. HAProxy provides a password protected web status page which allows system administrators to view how jobs are being distributed across the

worker nodes. Monit [23] monitors (stop, start, restart) critical processes, files, devices, and remote systems. It also provides a password protected http interface which allows one to monitor the system(s) via a web browser. Node.js [24] presents a server-side framework which we use to build scalable networked compute servers. We discuss Node.js in detail later. ApacheBench [25] (not shown) benchmarks the system's raw performance; and it generates a summary report which contains important measurements and statistics such as achieved throughput (requests/second) and total execution time.

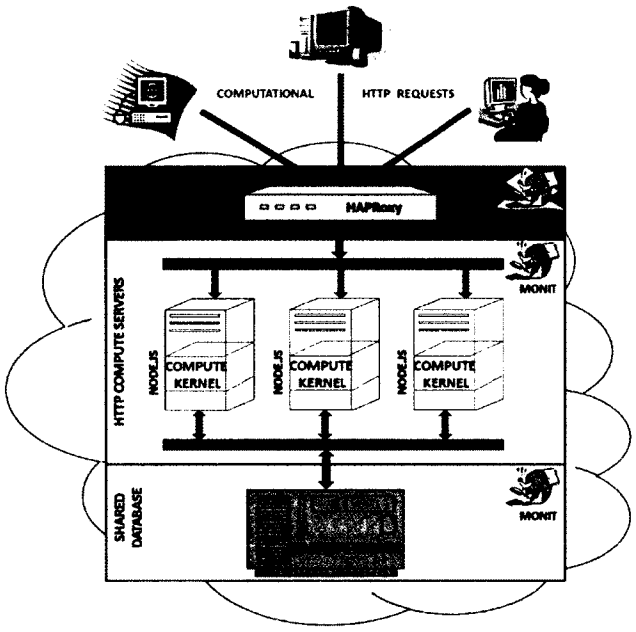


Figure 2.1: Web farm-inspired system design

As mentioned, there are multiple http services that need to be accessed: HAProxy's status page, Monit's system monitor, and the compute cluster itself. To only use one uniform resource locator (URL), we take advantage of HAProxy's access control list (ACL) feature. This allows us to use HAProxy to make decisions based off of information extracted from the uniform resource name (URN). In particular, we define rules such that

HAProxy routes requests to different server pools depending on the URN. Thus, a request can mean "fetch a status page" or "submit a job to the compute cluster." In our setup, as shown in Figure 2.2, a request which ends in "/" is taken to be a job and is load balanced across the compute servers; "/haproxy-stats" is directed to the HAProxy web status page; and "/monit" is routed to the Monit http interface to view the health of the system. To improve the system's fault tolerance, Heartbeat [26] and an additional backup HAProxy master node can be introduced for failover and failback. Since the Node.js server plays a crucial role in the system design, we now turn to: describe the framework in detail, elaborate on the properties which make the assignments to it more advantageous (when compared to more traditional servers), and explain its role in job processing.

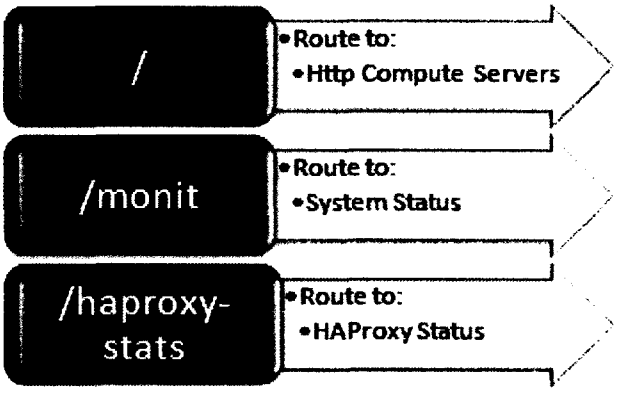


Figure 2.2: HAProxy ACL

2.4.3 Node.js

Our design calls for a lightweight server-side framework which acts as a simple I/O wrapper - a middleman which sits between HAProxy (the request dispenser) and the computational kernel (the job to be performed). In particular, its role is to receive requests,

parse them into inputs, and appropriate the inputs to the desired computational kernel. Though somewhat non-traditional, the focus here is to use a simple framework for I/O rather than employ a full stack web application framework. Node.js is an evented I/O server-side framework (built on top of Google's V8 JavaScript Engine [27]) which runs a single thread in an evented loop. To keep the loop from blocking, it uses asynchronous callbacks for input/output (I/O).

There are three advantages to using Node.js. First, it adheres to our requirements. It a lightweight framework which specializes in I/O. Second, asynchronous callbacks allow nondependent operations (parallel-like sections) to execute while waiting for I/O events such as "fetching a file" or "writing a file to disk." This is in direct contrast with the nature of traditional http servers (and serial programs in general) which execute commands in a linear, top-down, fashion. The main argument here is that CPU cycles are wasted, especially for I/O operations, because non-dependent commands do nothing since commands are executed sequentially. Third, is the difference in the number of threads used to handle multiple connections. Node.js is single threaded. However, it executes callbacks to accept and possibly service multiple requests in parallel. The caveat here is that one non-blocking service operation blocks all servicing operations from other requests. Traditional servers, on the other hand, create a new thread for each accepted request. Hence, they are called multi-threaded servers. In Section 2.5, we implement the web farm-inspired framework; then in Section 2.6 we observe and analyze the performance for both blocking and non-blocking operations. However, next, we discuss what a job is, how it is formed, and how it is handled once it is distributed to a compute server.

2.4.4 Job Submission & Processing

We take a job to be a request for some type of computational processing. To make a request, users will either select from existing algorithms in the database or upload a new custom algorithm (which will be added to the database). A custom algorithm must be written according to some agreed standard. The actual details of this standard is beyond the scope of this chapter. Nonetheless, users will also have to provide the necessary inputs which correspond to the selected algorithm. Once done, the request or "job" will be sent to a worker node via HAProxy. Algorithm 1 gives a general idea of how a compute server handles a job request. Each worker node has at least one compute server which listens for jobs on some specified port (line 9). Once a job arrives, the server immediately accepts it. When ready, the server begins to process the job by storing the input data to a local variable (line 3). Next, it parses that variable and passes the parameters to the execution function (lines 4-6). The execution function, as indicated by the constructor call (line 1), executes the request when given the necessary parameters (line 7). Once the request is fulfilled, the connection terminates (line 8).

Algorithm 1: Pseudocode for node.js compute server

```
1 execute ← new job() ;
2 createSERVER
3 |   parameters ← receiveDATA() ;
4 |   data ← stringTokenize(parameters) ;
5 |   input ← data [0] ;
6 |   algorithm ← data [1] ;
7 |   execute.run(algorithm, input) ;
8 |   end connection ;
9 listen (port, IP address) ;
```

2.5 Cloud Description and Setup

To realize the web farm-inspired framework, we require the cloud infrastructure as a service (IaaS) paradigm. The model we use, in particular, is made accessible via the Eucalyptus [28] application program interface (API). This API grants the ability to provision, configure, manage, or deprovision VM instances - as needed. In terms of specifications, each VM uses CentOS 5.5 (version 2.6.32-5-amd64) [29], has 15 GB of memory and 19 GB of storage space, and is mounted with 42 TB of shared storage.

We now give a general overview on our approach toward implementing the web farm-inspired framework. The experiments we will run (in this chapter), at most, require 11 VMs: one master node and up to 10 worker nodes. Instead of deploying all 11 nodes at once, we start by deploying one node and configuring its environment with all the necessary software components, i.e., HAProxy, Monit, Node.js, etc. Once done, we create an image of the running instance, and use the Eucalyptus API to bundle the image, upload the bundle, and register the image. Now that we have a base image, we deploy 10 additional VMs. Regardless the role, each VM uses the same image.

We use screen [30] to multiplex the terminal so that we can secure shell (ssh) [31] into each VM. Since we use a private cloud, each VM has a private internet protocol (IP) address which takes the form of 10.101.10.- -. We take the VM the lowest IP address to be the master node, and all the other nodes to be workers. We configure HAProxy on the master node and create a server pool consisting of the IP:PORT of each compute server which will run on the worker nodes. Next, we develop the both Node.js server and the computational function which executes the job and deploy them to the worker nodes (discussed in Section 2.6). We then start the compute servers on the ports as indicated in the

HAProxy configuration file. All workers are now ready to receive and process jobs. Next, we start HAProxy on the master node, and now it is ready to receive requests and distribute them to the compute servers, which execute on their respective worker nodes.

2.6 Experiments and Preliminary Results

In this section, we do experiments to reveal preliminary results which showcase the framework's functionality and scalability. Towards this end, our immediate objective is to implement the web farm-inspired framework and assess its overall performance for both blocking and nonblocking operations. We covered most of the general framework implementation in the prior section. So, here (in addition to assessing performance) we devote some attention to discussing both the blocking and non-blocking jobs which execute within their respective Node.js servers. Also, we do not implement the front-end web application interface for end-users. To test this framework, we instead vary one of two parameters - either vary the number of job requests or the number of VMs. We use Apachebench to vary the number of concurrent requests - parameter one. As aforementioned, HAProxy receives the requests and distributes them to the worker nodes. We modify HAProxy's configuration file to vary the number of VMs to which it routes requests - parameter two; and use HAProxy's status page to confirm that Apachebench did send the desired amount of requests.

2.6.1 Non-Blocking Operation

For the non-blocking experiment, we imagine a scenario where users submit jobs which perform some I/O operation. Let's say that the operation is to fetch large files, and the time required to fetch each file is 10 seconds. Given this, we want to observe the overall

task completion time given a certain amount of concurrent requests. So, for example, if we have 20 users that each submit a request simultaneously, we expect the overall task completion time to be about 200 seconds (at the very worst). A traditional single-threaded server which accepts requests, queues and fetches files in a FIFO fashion would mostly likely give these results. A natural question which comes to mind is: how do both the Node.js server and a traditional multi-threaded server fare in this situation?

To answer this question, we implement a Ruby Webrick server (adapted from [32]) and a Node.js (adapted from [33]). We use a 10 second timeout function to simulate the act of "fetching a file." This means that for each request, each server waits for 10 seconds and delivers a "job completed" message at the end of the waiting period. Though theoretically the same, each implementation is fundamentally different in approach. The Ruby Webrick approach, as shown in Algorithms 2, is straightforward. The server accepts requests, "fetches a file" (sleeps for 10 seconds), then responds.

Algorithm 2: Ruby webrick, fetch file

```
1 class
2     Timeout ;
3     define
4         call(env) ;
5         sleep 10 ;
6         return {200, ; {Content-Type => text/plain},
7         {"The timeout function has completed"}} ;
8 run Timeout.new ;
```

The Node.js implementation (Algorithm 3), on the other hand, is not as intuitive. So let us describe this in a bit of detail. Node.js is evented. This means that a sequence of events direct the nature of the program. Thus, in lines 1-4, we write a sleep function such

that it creates an event when the timeout expires. The function, in particular, ingests a status parameter and passes it to the callback function. Since we set the timeout value to 10000 milliseconds, the callback function fires when the timeout value expires after 10 seconds. The server is implemented in lines 5-9. There is a listener, on line 9, which triggers lines 6-9 every time a request is made. Lines 6-9 execute sequentially. Line 6 sends an "OK, request received" response. On line 7, we pass a "completed" status to the sleep function. After 10 seconds, an event fires and line 8 executes. Line 8, gives a "request completion" response and closes the connection. As for the order of events, when the server starts, the print statement on line 10 executes immediately. If we use Apachebench to send N concurrent requests, the listener on line 9 detects N requests. In response to this, all N requests are accepted. Next, the server executes line 6 in parallel for each request, then line 7 (in parallel), and lastly line 8 (in parallel).

Algorithm 3: Node.js, fetch file

```
1  sleep
2      sleep ← function(data, callback) ;
3      var timeout ← 10000 ;
4      setTimeout(function(){callback(null, data)}, timeout) ;
5  createSERVER
6      function(request, respond) ;
7      respond.writeHead(200, {Content-Type: text/plain }) ;
8      sleep("completed", function(err, data){ ;
9      respond.end("The timeout function has: " + data)}) ;
10 listen(3000, 127.0.0.1) ;
11 console.log(Server running at http://127.0.0.1:3000/) ;
```

Using the web farm approach, we setup one worker node with one Node.js server. We then use Apachebench to vary the number of concurrent requests from 20 to 500, and

observe the performance. The results, shown in Figure 2.3, are as expected. The average total execution time 10.28 seconds. Next, we repeat the same experiment with one Webrick server. Again, the results are as expected. The Webrick server does not fare as well. The average execution time here is 21.74 seconds. This means that one Node.js sever is on average 2.11 times faster. We try to experiment again using two Webrick servers. The results improve slightly. This time the average execution time is 19.24 seconds. Thus, a single Node.js server is on average 1.88 faster than two Webrick servers. Notice that each experiment in Figure 2.3 follows a similar trend. The total execution time increases from 20 to 100 concurrent requests, decreases from 100 to 200, and begins to increase again from 300 to 500.

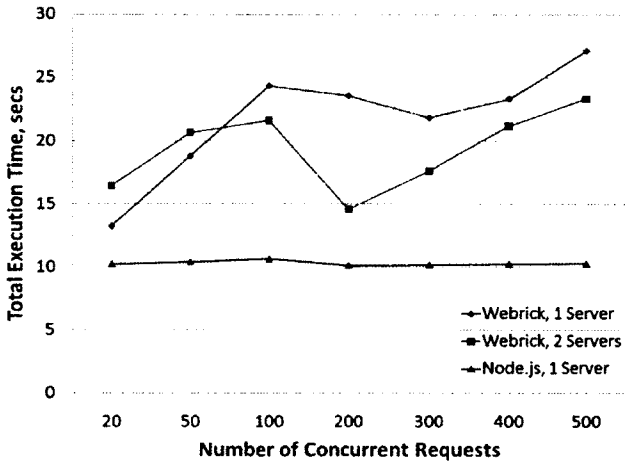


Figure 2.3: Single node.js outperforms multi-threaded webrick servers for non-blocking operations.

What happens if we actually replace the timeout function with an actual computation? How would the overall system perform? If the computation is a blocking

operation, can we increase the number of servers to improve performance? We entertain these questions next.

2.6.2 Blocking Operation

In this experiment, we build on the contrived example above by replacing the "wait statement" with an actual computation. Our goal is to determine whether adding worker nodes to the web farm-inspired compute cluster lead to improved performance for blocking computations. We decide to embed a normalized difference vegetation index (NDVI) algorithm within the Node.js compute server, as indicated in Algorithm 4. The flow of this program is similar to that described earlier (Algorithm 1). Thus, we will not go into as much detail.

Algorithm 4: Node.js, perform NDVI

```
1  var sys ← require(sys) ;
2  var http ← require(http) ;
3  job ← require(./job) ;
4  var runJOB ← new job.RuntimeEngine() ;
5  createSERVER
6      (function(request, respond) ;
7      respond.writeHead(200, {Content-Type: text/plain}) ;
8      respond.write(Please Wait) ;
9      runJOB.execute("ams_11g_ndvi_png", "AMS.19nov09.16.3_47", 0) ;
10     runJOB.addListener(result, function(result)) ;
11     foreach var attr in result do
12         sys.puts(attr + " : " + result(attr));
13     respond.end("Completed") ;
14 listen(3000, 127.0.0.1) ;
```

Nonetheless, lines 1-4 include a few libraries. Line 4, in particular, is a custom built library which has functions related to job execution. The nature of the library itself is

beyond the scope of this discussion. Lines 6-7 sends a response to the client once a request is detected in line 15. Line 8 accepts three parameters: the NDVI algorithm, the image, and a cache option. The cache option simply checks the database to determine whether or not the results already exist for this algorithm/image combination. Setting this value to one, turns the option on; and setting this value to zero, turns the option off. We want this option off for testing purposes; so we set the value to zero. Lines 10-13 listens for the result from each of the three image bands and outputs the result. Once done, line 14 sends a "completed status" and closes the connection. The main thing to note here is this is not an I/O or non-blocking operation. Thus, the computation blocks all other computations from executing in parallel.

Since image processing, in general, can be computationally intensive, we determine that there should only be one compute server per worker node and that each worker node would have the same compute server. So, for example, 10 worker nodes mean 10 duplicate Node.js http servers. In order to establish some baseline for performance, we take all the jobs to be the same. This means that all jobs executes the same NDVI algorithm on the same image. Next, we use Apachebench to simulate an arbitrary number of concurrent computational job requests (we choose 20). HAProxy then receives and distributes the requests to worker nodes in a round-robin fashion. We form 10 test cases by varying the cluster size from one to 10 worker nodes and observe the overall system performance.

Again, this time, instead of waiting, each Node.js compute server responds to a request by executing the algorithm. In the previous example, each test case executed the concurrent requests (from 20 up to 500) in approximately 10 seconds, the time it took to execute one single request. This let us know that the concurrent requests were accepted and

processed in parallel. The operations did not block the single Node.js thread. However, in this example, our initial observation is as expected. Node.js blocks while executing a single request, but instead of rejecting, each worker node queues subsequent requests if received while processing. Below, we indicate how we deduce this.

One computation takes approximately 5.51 seconds to complete. Instead of taking close to 5.51 seconds to accept and execute 20 duplicate concurrent computational requests, it takes 88.94 seconds (for one worker node). This means that each computation takes about 4.45 seconds (on average), and that each request blocks the evented thread and prevent other requests from being executed. To ensure that the thread is indeed blocked due to the computation (as opposed to the problem being the same jobs waiting for the same resources - not a blocked thread), we compose another simple function. If the function executes after the computation, this would indicate that the computation blocks the thread. If the function executes concurrently with the computation, this would indicate that the computation did not block the thread. The results indicate the former. The function did not execute until after the computation. The good news is, in spite of this, the compute servers accept and queue the requests and execute them in a FIFO manner.

Given that the computation blocks the thread, the only way to improve the overall performance is to increase the number of threads (or the number of Node.js servers). Since we limit each worker node to 1 compute server, we have to increase the number of worker nodes. By varying the number of worker nodes from one to 10, the overall performance does improve.

The results are shown in Figures 2.4 and 2.5. The total execution time decreases exponentially, and the number of requests/second increase linearly as the number of worker

nodes increase. Also, in both Figures 2.4 and 2.5, notice that the results for test cases six and nine (number of virtual machines) deviate slightly from the trend, as suggested by the other test cases. The result for these two cases is slightly higher than what the trend suggests in Figure 2.4, and is slightly lower than what the trend suggests in Figure 2.5.

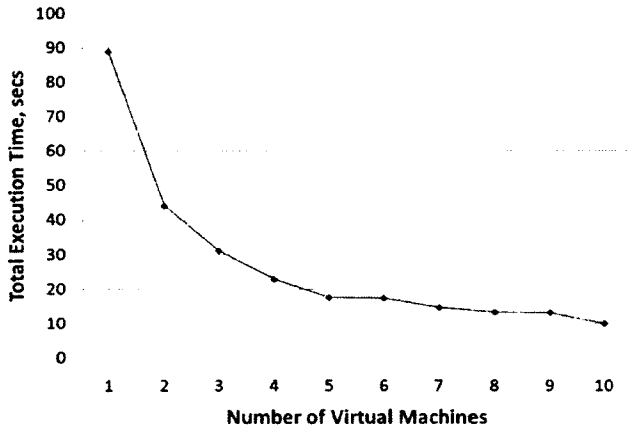


Figure 2.4: For blocking operations, the total execution time decreases exponentially as the number of VMs increase.

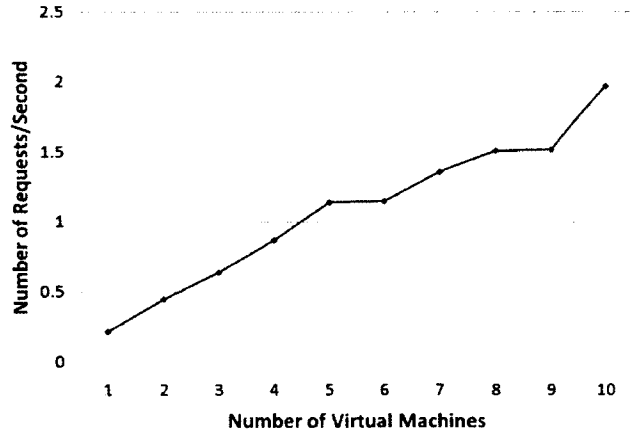


Figure 2.5: For blocking operations, the number of requests/second increases linearly as the number of VMs increase.

2.7 Discussion

In the last section, we claim that the total execution time decreases exponentially and the requests/second increase linearly as the number of VMs (or worker nodes) increase. However, this conclusion is just an initial assessment based on the trend that the data suggests. To test our hypothesis, we make a scatter plot of the data, and fit an exponential and linear curve to their respective graphs. As shown in Figures 2.6 and 2.7, each curve fits well. The R^2 for the exponential fit is 0.87. Notice, in Figure 2.6, that one VM has a total execution time of about 89 seconds, and two VMs have a total execution time of about 44 seconds. Clearly, two VMs perform twice as good as one. However, the big disparity in these two results most likely explains why the R^2 value is slightly below 90%. The R^2 value for the linear fit is 0.97. Thus, we can say that these results provide supporting evidence in favor of our hypothesis.

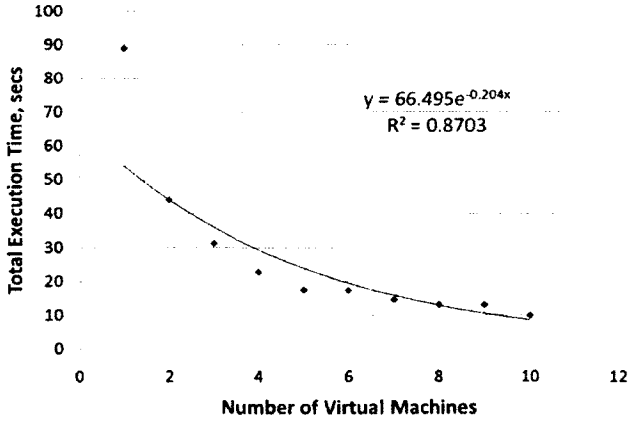


Figure 2.6: Exponential curve fitting for the total execution time

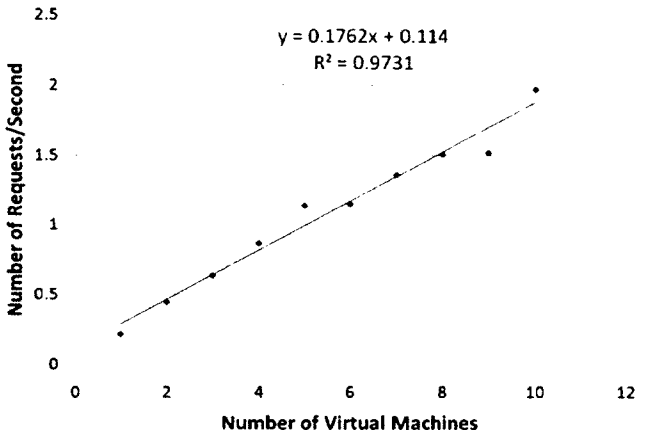


Figure 2.7: Linear curve fitting for the number of requests/second

Moving forward, we now discuss the inherent challenges associated with using the web farm-inspired framework for computational processing, and give suggestions researchers should consider when mapping computational kernels using this approach. Web farms, and the tools associated with them, are designed to serve lightweight requests which can be executed on the order of milliseconds to seconds. Computational processing, on the other hand, can be compute and time intensive. Given this, developers should seek to minimize communication and the number of compute servers per worker node (servers/node). In regards to communication, the best thing to do is to follow the web farm approach: maintain uni-directional communication from master node to worker nodes and have NO communication among worker nodes. To ensure no inter-worker node communication, use problems which are pleasingly parallel.

The number of worker nodes and the server/node ratio together determine the number of requests the system can handle in parallel. To handle more concurrent blocking computational requests, simply add more worker nodes. Like a web farm, each worker node can support multiple compute servers; but since the computational kernel may command

significant resources, researchers should look to keep the number of servers/node to a minimum. However, in general, the resource requirements of each computational request and the anticipated peak request traffic will be the determining factors in choosing the server/node ratio and the necessary number of worker nodes. Also, try to keep in mind that there is an inverse relationship between the number of servers/node and the number of worker nodes. A decrease in the server/node ratio means an increase in the number of worker nodes.

2.8 Conclusion

In this chapter, we proposed a web farm-inspired infrastructure for dynamic and concurrent computational processing in the cloud. We described this framework in detail and gave reasons as to why, and under what circumstances, it proves beneficial. We did experiments to reveal two preliminary results which showcased the framework's functionality and scalability. The first result showed that, for non-blocking operations, worker nodes which use Node.js servers are significantly faster than those which use traditional servers; and from the second result we concluded that, increasing the number of worker nodes improves the overall system performance for blocking computational operations.

2.9 Using Proxies for Cloud Security

The framework presented in the this chapter mostly centered on the use of a proxy. One important property of proxies is the ability to maintain 1-to-n mappings between clients and servers. In the next chapter, we couple the idea of proxies with the self-organizing behavior of mussels to develop a cloud defense strategy. This strategy addresses a vulnerability which exploits a 1-to-1 public to private IP mapping.

CHAPTER 3

MUSSELS, LÉVY WALKS, AND CLOUD SECURITY

Recent research has provided evidence indicating how a malicious user could perform co-residence profiling and public to private IP mapping to target and exploit customers which share physical resources. The attacks rely on two steps: resource placement on the target's physical machine and extraction. Our proposed solution, in part inspired by mussel self-organization, relies on user account and workload clustering to mitigate co-residence profiling. Users with similar preferences and workload characteristics are mapped to the same cluster. To obfuscate the public to private IP map, each cluster is managed and accessed by an account proxy. Each proxy uses one public IP address, which is shared by all clustered users when accessing their instances, and maintains the mapping to private IP addresses. We describe a set of capabilities and attack paths an attacker needs to execute for targeted co-residence, and present arguments to show how our approach disrupts the critical steps in the attack path for most cases. We then perform a risk assessment to determine the likelihood an individual user will be victimized, given that a successful non-directed exploit has occurred. Our results suggest that while possible, this event is highly unlikely.

3.1 Introduction

Equipped with the ability to leverage virtual resources on-demand, cloud computing systems have recently emerged as a viable low-cost alternative to traditional computing

platforms. This has sparked widespread interest, adoption, and/or research initiatives from all institutions alike (e.g., academic, industrial, government, etc.), which in turn, has led to myriads of success stories [34–36] that give credence to its potential and effectiveness. Though promising, this technology suffers from the same fate as any other new development in its infancy stage. It solves some problems while newly introducing unanticipated and not readily understood challenges [37]. At the core of these concerns lies privacy and security [38–42]. Recent research [1] has shown that it is possible to identify and target a cloud user, launch malicious virtual machines (VMs) which perform co-residence checks, and possibly extract confidential information once co-residency with the victim has been established. An example such as this exposes the volatility of cloud security.

To gain insight on how to best solve this problem, we look towards nature - for research shows that social animals have the tendency to solve distributed problems (e.g., foraging, nest building, defense) optimally, robustly, and efficiently [43–47]. Similar to how mussels form small clusters to decrease water stress and minimize the risk of predation [48–52], we hypothesize that cloud providers can strategically cluster users to mitigate the chance of targeted exploits via malicious co-resident users. The general idea of using clustering to address security vulnerabilities is expressed in various works [53–56]. However, these mostly use mix networks. We briefly consider one example. In [54], Reiter and Rubin create a system to conceal the identity of clients when performing web transactions. This system is based on the notion of crowd blending and "operates by grouping users into large and geographically diverse groups that collectively issues requests on behalf of its members."

In this work, we propose a framework where the cloud provider uses mussel behavior to cluster users according to individual preferences, e.g., computational requirements (small, medium, or large VM), workflow duration (hours, days, weeks), or cluster size (small, medium, large). Each cluster contains members with similar preferences or workload types - subscribed in a best effort manner. This means that assignment to the cluster which exactly matches user preferences is not always guaranteed. To obfuscate the public to private IP map, each cluster is managed and accessed by an account proxy. Each proxy uses one public IP address, which is shared by all clustered users when accessing their instances, and maintains the mapping to private IP addresses. To prevent attacks aimed at users belonging to a particular proxy, clusters periodically dissolve. That is to say, on occasion, members are disbanded and are subscribed to new clusters. If an individual's preferences have not changed, then the new cluster will be similar to the prior cluster. Otherwise, the new cluster will reflect the individual's new interests. The period of updates is a user defined parameter. Thus, preference is used to determine how often clusters dissolve e.g. (hours, days, weeks).

The rest of this chapter is organized into seven sections. Section 3.2 gives the contributions of this chapter. Section 3.3 presents the system model, threat model, and the exploit description. Section 3.4 provides a brief background on mussel behavior, and details how we make mathematic and algorithmic modifications to a model which describes density-dependent interactions between individual mussels. Section 3.5 covers the technical aspects of our proposed solution. Section 3.6 performs a risk assessment to determine the likelihood an individual user will be victimized, given that a successful non-directed exploit has occurred; and lastly, Section 3.7 yields concluding remarks.

3.2 Contributions

Ristenpart et al. conduct research which uncovers security vulnerabilities in the cloud. They use standard network probing tools to decode the 1-to-1 public to private IP map, and use this map to identify and target other cloud users. Next, they conduct experiments and find that the private IP addresses are statically assigned according to launch parameters - availability zone and instance type. They then use the same launch parameters as that of the victim to maximize the chance of co-resident placement. Each VM performs co-residence checks to determine whether it shares the same physical machine as its victim. If not, it terminates. Otherwise, it proceeds with extraction - the next phase of the attack. This is the nature of the adversarial model we consider.

This chapter, on the other hand, firstly introduces a defensive mussel- inspired strategy to address cloud vulnerabilities. We obfuscate public to private IP mapping by having account proxies perform 1-to-n random mapping of public to private IP addresses. This decreases the risk of adversary targeting and significantly reduces the amount of public IP addresses needed for users to access their VM instances. By having clusters periodically dissolve, our strategy decreases the chances of directed attacks towards random users belonging to a particular account proxy. Lastly, our approach takes advantage of the cloud's intrinsic features. VM instantiations are inherently transient. Even with co-residency and a successful breach, the victimized VM can be terminated at any time, only to be redeployed later - possibly to another physical machine. This feature coupled with account proxies helps prevent a user from being consistently targeted, tracked, and victimized across multiple physical machines.

3.3 System Model, Threat Model, and Exploit Description

Cloud computing systems provide innovative solutions while introducing new avenues for research direction. One aspect of cloud systems which serves in this capacity is hardware virtualization - the ability for multiple customers to share the same physical resources at the same time. Though providers benefit from resource consolidation, this feature poses new security challenges and possibly serves as a significant system vulnerability. Consider two competing organizations which both lease resources from the same cloud provider. It is foreseeable that one customer's motive could consist of exploiting the shared nature of the cloud to identify, target, and victimize its competitor. Possible attacks could include: monitoring workflow patterns, extracting valuable information, conducting denial of service (DoS), distributed DoS (DDoS), or EDoS (Economic Denial of Service), where the victim's bill causes a shock at the end of the accounting period because they have had to use more instances than planned. Given this, we consider customer VMs, data, and information to be assets.

3.3.1 System and Threat Model

From a system model perspective, we classify customers based on intent. *Malicious users* are those with malevolent intent - those who target other users and seek physical machine co-residence for unauthorized surveillance and/or data extraction of via certain exploits, i.e., side channel attacks. We consider these type of users to be threats which launch attacks comprised of two steps: virtual machine placement on the machine upon which the target resides and data extraction. Below, we identify four types of attackers and list the possible goals for each.

1. ***Eavesdropping non-directed attacker*** goal is to read data or find out about any target.
2. ***Malicious non-directed attacker*** goal is to cause a DoS on any or all instances.
3. ***Eavesdropping directed attacker*** goal is to get data from a specific competitor's instance or learn about their workload pattern.
4. ***Malicious directed attacker*** goal is to cause one of the following attacks on a particular target: DoS, DDos, or EDoS.

Honest users, on the contrary, are those that use cloud resources for their intended purposes. These users have sincere intent. They abide by the protocols, procedures, and regulations as outlined in the terms of service agreement. We would like to prevent these users from being identified and targeted by malicious users. A *peer* is simply one that shares the same physical resources - a co-resident user. A peer can either be a malicious or honest user. We assume the cloud provider to be trusted and honest - providing the services to its customers as outlined in the service license agreement.

3.3.2 Exploit Description

Since the inception of cloud services, the possibility of users being exploited by a rogue peer has always been a major issue of concern. However, the realization of these fears never quite materialized until researchers began to uncover the extent of cloud user vulnerability. The exploit we consider is described by Ristenpart et al. In [1], they use Amazon's EC2 [57] " as a case study to demonstrate that careful empirical mapping can reveal how to launch VMs in a way that maximizes the likelihood of advantageous placement." To investigate this notion, they assume a placement and extraction attack strategy. They use domain name system (DNS) resolution queries and traditional network

tools, i.e., nmap, hping, wget, to determine the external name of an instance and to derive a map which exposes the correlation between the external public IP address and the internal private IP address of an instance. They additionally found that the internal IP addresses are statically assigned to physical machines according to availability zone and instance type. Thus, the map could be used to deduce the availability zone and instance type for any given target - effectively reducing both the search space for finding a target and the number of "probe instances" needed to be deployed before achieving co-residence. A probe instance is simply a malicious VM that performs a co-residence check to determine whether or not a target is a peer. If the target is a peer, it proceeds with data extraction - the next phase of the attack. Otherwise, it terminates.

Ristenpart et al. identify three different methods which could be used to determine co-residence, and present two strategies an attacker could use to exploit placement in EC2 - brute-forcing placement and placement locality. The brute-forcing placement strategy deploys a large number of instances over time in same zone and of the same type as that of the instances belonging to a large target set. They conduct an experiment using this strategy and receive a success rate of 8.4%. This means that 8.4% of the probe instances actually achieved co-residence with instances of the target set. The placement locality strategy, on the other hand, assumes a smaller target set, and also presumes that the attacker can launch probe instance soon after a targeted victim's instances are launched. They conduct another experiment, and find that this strategy yields a success rate of 40%. They make the following conclusions concerning Amazon's VM placement algorithm.

1. N parallel instantiations launched from a single account tend to result in placement on N different machines.

2. If a VM which runs on machine A is terminated and another VM is launched immediately thereafter, then that new VM tends to be placed on machine A. This may explain why the brute-forcing strategy did not fare as well.
3. Two VMs launched around the same time, from two different accounts, tend to be assigned to the same machine.
4. There is a small inherent bias in assigning new VM instances to machines with light loads.

They later outline possible ways to extract information from target victims once co-residence with the target is achieved. However, for the sake of space, we will not cover this here. One thing should be noted. Though the case study is specific to Amazon, Ristenpart et al. believe that modified variations of their technique can be extended to services supplied by other cloud providers.

3.3.3 Discussion

From the above conclusions it seems the placement algorithm may inadvertently assist miscreants in their mission to target and exploit other users. The first conclusion helps maximize the search space for a particular victim. A machine co-residence check only requires one probe instance. So ideally one VM, at most, should be assigned to each machine. Further, the first and third conclusions together ensures heterogeneous VM ownership per machine.

Two main problems form the central issue. The first - Amazon's VM placement algorithm is predictable and manipulable. The second - VM instances are directly connected to the Internet via port 80, and a DNS service, which translates public IPs to private

IPs, is a major line of defense for preventing malicious users from targeting, locating, and exploiting honest users. Ristenpart et al. found a way to decode the map and presented the details of their findings. To address these issues, we propose a solution in part inspired by mussel self-organization. We describe the details of this behavior in the next section.

3.4 Mussel Behavior

It is forseeable that a combinatorial rise in the possible combinations of user preferences could result in large computational overhead with deterministic or complete enumeration algorithms. Thus, the use of heuristic algorithms may prove to be beneficial. We extend the self-organization behavior of mussels to develop an algorithm to address such a problem.

3.4.1 Background

Interactions between organisms, themselves, and the environment in which they live leads to feedback which affects both the organisms and the environment. For mussels, the magnitude of this feedback varies with distance - a phenomenon known as scale-dependent feedback (SDF) [58]. There are two types of SDF: positive and negative. Mussels experience positive SDF over short-range distances with respect to peers. This leads to cooperation between individuals in the vicinity. If there is short-range density, or a certain number of peers per unit area in its immediate surroundings, an individual mussel tends settle, or maintain its current position. It then secretes byssal threads to attach itself to the shells of peers, rocks, or other various substrates. On the other hand, mussels experience negative SDF ver long-range distances with respect to peers. This leads to competition which restricts survival over long distances. If there is long-term density in its not so

immediate surroundings, an individual mussel tends to move to a new location. The interplay between positive and negative SDF ultimately results in patches of optimal sized clusters - large enough to decrease the risk of predation and water stress yet small enough for the groups to withstand the risk of food depletion.

3.4.2 The Model

Our research takes interest in this natural phenomenon and, in part, builds on the work done by de Jager et al. In [59], they use empirical observations, theoretical and computer modeling to determine that the Lévy walk (LW) best characterizes mussel movement strategy. When compared to the Brownian walk or ballistic motion strategies, the LW provides the best fit to experimental step length data, minimizes the time needed for pattern formation, and is evolutionary stable to mussels differing in movement strategy.

To investigate the role of density dependence in pattern formation, de Jager et al. observe mussel movements under laboratory conditions and meticulously extract step length data. Mussels are evenly spread on a PVC sheet, and a webcam is positioned to record their activity. Over time, the individual mussels move around to search for nearby conspecifics. They use byssal threading to attach themselves to the bed when they find a position which best balances neighbors with food availability. This local clustering behavior ultimately leads to a global spatial pattering in the mussel bed.

A histogram of the step length data reveals a heavy-tailed probability distribution. This infers that when the step length data is plotted on a log-log scale, the general power-law function, shown in Equation 3.1, results in a straight line with slope -μ,

$$p(l) = Cl^{-\mu} \quad . \tag{3.1}$$

The parameter μ determines the movement strategy. When $\lim \mu \rightarrow 1$, the strategy is ballistic (straight-line) motion; and the likelihood of taking a large step is equal to that of taking a small step [60]. When $\mu > 3$, the movement approximates a Brownian walk; and when $1 < \mu \leq 3$, the strategy is taken to be a LW, where small steps are occasionally alternated with larger ones [59,60]. The LW most commonly found in nature is $\mu \approx 2$ [59]. For the LW, the normalization constant, C, is expressed by Equation 3.2 [61],

$$C = (\mu - 1) l_{min}^{\mu - 1} \quad , \tag{3.2}$$

where l_{min}, $(0 < l_{min} < 1)$, is a constraint which represents the minimum step length. An l_{min} value of 0.42 provides the best results when fitting the actual data to the Levy walk. For the truncated LW, the normalization constant C, is expressed by Equation 3.3,

$$C = \frac{\mu - 1}{l_{min}^{1-\mu} - l_{max}^{1-\mu}} \quad , \tag{3.3}$$

where the steps, l, are only defined on the open interval $l_{min} < l < l_{max}$. An l_{min} value of 0.42 and an l_{max} value of 58.84 provided the best results when fitting the actual data to the truncated LW. De Jager et al. use the Goodness-of-fit (G) value, shown in Equation 3.4, to determine how well the frequency distributions of the movement strategies fit the actual data. A value closer to zero indicates a better fit,

$$G = 2 \sum O_i \ln \frac{O_i}{E_i} \quad . \tag{3.4}$$

Here, O is the inverse cumulative distribution of the actual data; and E is the inverse cumulative distribution of the fitted movement strategies. Table 3.1 shows how well the strategy models fit the actual data. Notice, the Lévy and truncated Lévy strategies both

yield comparable results, while the Brownian walk does not fare as well. In the end, the G value suggests the truncated Lévy strategy provides the best fit.

Table 3.1: Fitting movement strategies to actual experimental data

	Movement Strategy		
	Truncated Lévy Walk	Lévy Walk	Brownian Motion
G	22.45	47.22	-190.09
AIC weights	0.443	0.428	0.129
Adjusted R^2	0.997	0.997	0.837
Levy exponent	2.01	2.06	-

To investigate how the LW movement strategy effects the rate of pattern formation, de Jager et al. develop an individual-based model, where the density of conspecifics determines whether or not individual mussels decide to move. Towards this end, multivariate regression analysis of the experimental data reveals that mussels are more likely to stay if there is short-range density - mussel density within a 3.3 cm radius, and are more likely to move if there is long-range density - mussel density within a 22.5 cm radius. The linear expression, shown in Equation 3.5, relates the chance of movement (P_M) to long (D_L) and short-range (D_S) densities. By performing linear regression on experimental data, it is determined that $a = 0.63$, $b = 1.26$, and $c = 1.05$. As indicated in Table 3.2, an individual decides to move when its chance of movement is greater than a random value drawn from a uniform distribution. Otherwise, it maintains its current position. If an individual mussel decides to move, it takes a LW. That is to say, its direction, θ, is drawn from a uniform distribution (depicted in Equation 3.6); and its step length, l, is drawn from a power law distribution (shown in Equation 3.7).

$$P_M = a - bD_S + cD_L \quad . \tag{3.5}$$

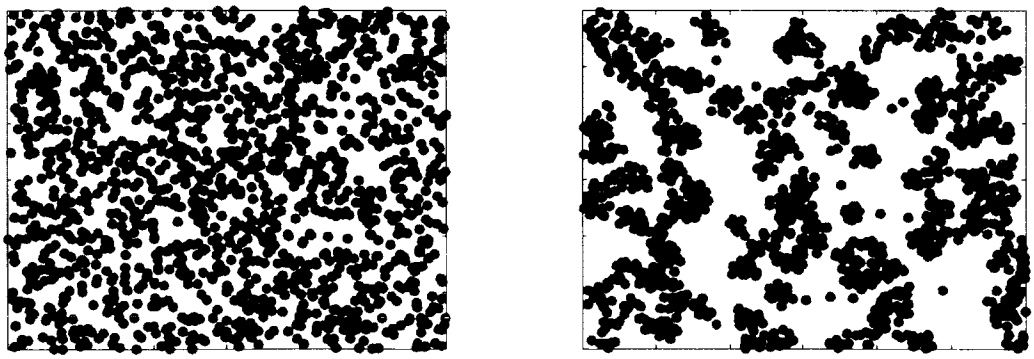
$$\theta = 2\pi w, \; w \sim U(0,1) \quad , \tag{3.6}$$

$$l = \frac{x_{min}}{(1-w)^{\frac{1}{\mu-1}}}, \; w \sim U(0,1) \quad . \tag{3.7}$$

Table 3.2: Mussel behavior in response to scale-dependent feedback

SDF Type	Action	Condition
Positive	Settle	$P_M \leq rand(0,1)$
Negative	Move	$P_M > rand(0,1)$

Over time, local density dependent interactions between individual mussels and their peers lead to the emergence of a distinct global spatial patterning in the mussel bed. Figure 3.1 shows the state of the mussel bed before the computer model starts and just after it ends.



(a) Before (b) After

Figure 3.1: Original mussel bed clustering computer model

3.4.3 The Algorithm

We use the model which describes mussel behavior to develop a bio-inspired heuristic clustering algorithm for multi-agent systems. We assume that agents make independent decisions based on local asynchronous interactions and have limited memory. Thus, they cannot remember past decisions or movements. Initially, agents with various basic abilities are randomly distributed throughout the interaction environment. As a result, agents must actively search for similiar agents and form specialized groups in order to be collectively effective. Agents with similar abilities are said to be homogeneous, and agents with dissimilar abilities are said to be heterogeneous. The highlights of the algorithm are:

1. first bio-inspired algorithm based on mussel behavior
2. decentralized - no central control
3. scalable to large agent set

Figure 3.2 shows the high level logic flow diagram. Each mussel agent initially observes its surroundings and calculates the short and long range density which are, in turn, used to determine its chance of movement. If it does not decide to move, it simply maintains its current position for some time before re-observing its surroundings. On the other hand, if it decides to move it selects a new position. If the new position is currently occupied, it takes a truncated LW; otherwise, it takes a LW. The details are further explained in Algorithm 5. As shown in lines 1-2, the onset is marked by setting the necessary parameters, i.e, the number of mussel agents, agent attributes, sensing radii, etc., and initializing the agent set to random (X, Y) positions. The main parameters of the mussel alogorithm are summarized in Table 3.3. Notice that each agent uses three different sensing radii to determine the density of: homogeneous agents within a short range, heterogenous agents

within a medium range, and any agent within a long range. The following steps are repeated until the number of maximum steps is reached. Each mussel agent uses the Euclidean distance to determine the distance of all other agents - line 7; determines the positive SDF by counting only the number of homogeneous peers in its short range sensing radius - lines 8-9; determines the negative SDF by counting both the number of hetergeneous peers in its medium range sensing radius and the number of any peer in its long range sensing radius - lines 10-11; uses the positve and negative SDF to determine the short and long range density - lines 12-13; calculates the chance of movement - line 14; selects new position if it decides to move - lines 15-19; takes a truncated LW if the new position is occupied - lines 20-21; and takes a LW if the new position is not occupied - lines 23-25.

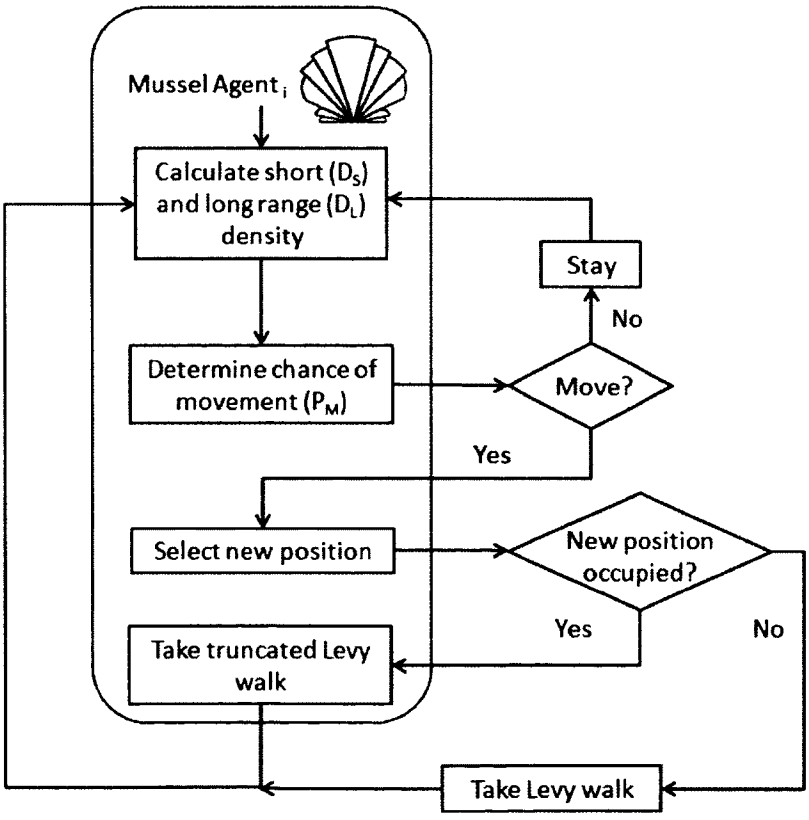


Figure 3.2: Logic flow diagram

Algorithm 5: Mussel agent clustering algorithm

```
set the necessary parameters ;
initialize mussel (x, y) positions;
repeat
    foreach mussel i in 0 to |T| do
        SDF_pos ← 0 , SDF_neg ← 0 ;
        foreach mussel j in 0 to |T| do
            d_{i,j} = sqrt((x_i − x_j)^2 + (y_i − y_j)^2) ;
            if d_{i,j} < D_1 ∧ g(i) = g(j) then
                SDF_pos + + ;
            if D_1 < d_{i,j} < D_2 ∨ d_{i,j} < D_3 ∧ g(i) ≠ g(j) then
                SDF_neg + + ;
        D_s = (SDF_pos − 1)/(πD_1^2) ;
        D_l = (SDF_neg − 1)/(πD_2^2) ;
        P_m ← according to Equation 3.5 ;
        if P_m > rand then
            θ_i ← according to Equation 3.6 ;
            L_i ← according to Equation 3.7 ;
            px_i = x_i + cos(θ_i) * L_i ;
            py_i = y_i + sin(θ_i) * L_i ;
            if px_i ≠ occupied ∧ py_i ≠ occupied then
                x_new ← px_i , y_new ← py_i ;
            else
                l_min ← calculate truncated step length ;
                x_new = x_i + cos(θ_i) * L_i * l_min ;
                y_new = y_i + sin(θ_i) * L_i * l_min ;
until maxSteps;
```

Table 3.3: The main parameters of the mussel algorithm

T	set of agents
g	set of agent attributes
D_1	sensing radius used for short range density of homogeneous agents
D_2	sensing radius used for long range density of any agent
D_3	sensing radius used for medium range density of heterogeneous agents
x_{min}	minimum walking distance
μ	exponent for Lévy movement strategy

3.4.4 Example Mussel Subscription

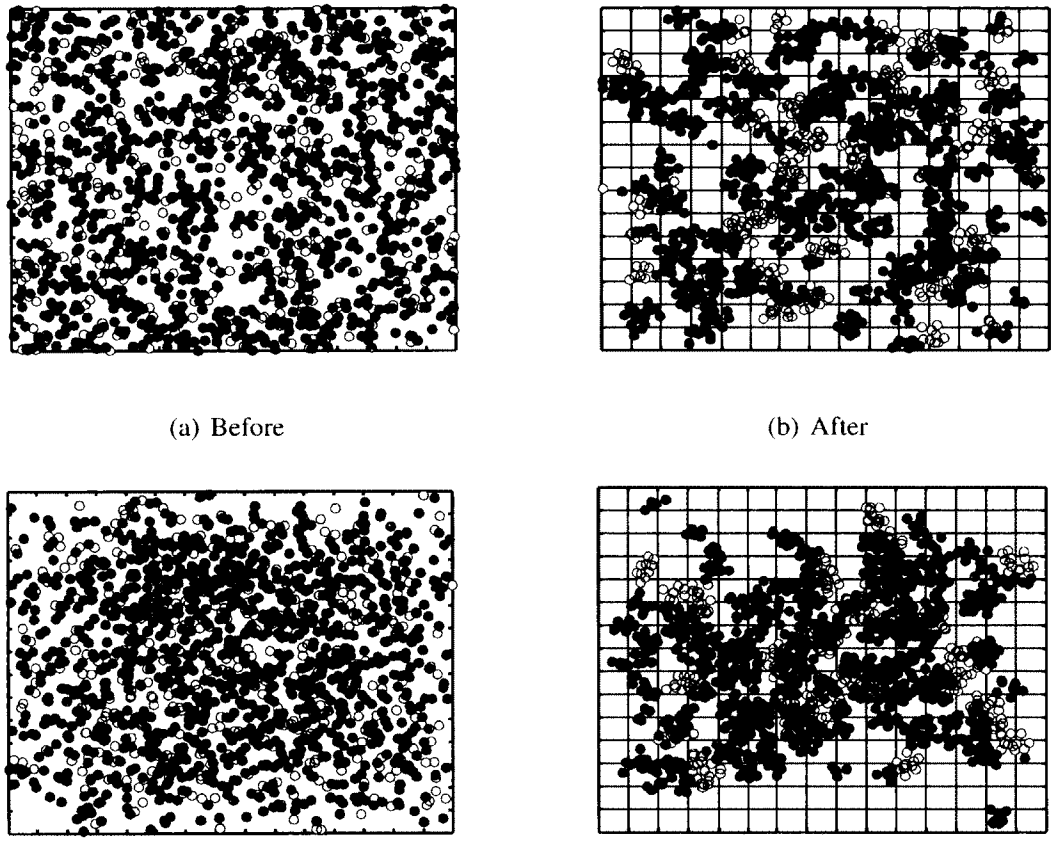
Now, suppose that we have 1800 users. Each of which is able can choose from two preferences: VM size and workflow duration. Further consider that each preference has two options. A VM's size can either be large or small, and the duration of workflows can either be based on hours or days. As shown in Table 3.4, users can belong to one of four categories. If a user intends to use a small VM for a few hours, then he/she will be placed in workload category I. The color red is used to identify those users in this category. Similarly, those users who prefer a large VM for a few hours will be placed in workload category II - denoted by the color blue; those who desire a small VM for a few days will be placed in workload category III - denoted by the color green; and those who wish to use a large VM for a few days will be placed in workload category IV - denoted by the color yellow.

Table 3.4: Assigning categories to user preferences

Length	VM Size: small	VM Size: large
hours	I (red)	II (blue)
days	III (green)	IV (yellow)

For this example, we assume that users are indifferent to the preferences. That is to say, the odds of being placed in any of the four categories are the same. Figure 3.3(a) shows the initial state of the logical field once each user is placed in a workload category based on preferences and assigned random (X, Y) coordinates. The system reaches steady state after some time, and gives rise to an emergent clustering pattern between users with

like preferences - as shown in Figure 3.3(b). Notice the cells. Here, cells are analogous to accounts. Users are subscribed to the cell in which their indicator settles. After some time account membership dissolves - Figure 3.3(c); and users are subscribed to new accounts - Figure 3.3(d).



(a) Before (b) After

(c) Dissolve (d) Regroup

Figure 3.3: Users cluster according to mussel behavior.

3.5 Proposed Framework

We now describe the technical analysis of the mussel-inspired self-organization approach towards reducing the risks of adversary exploitation as described by Ristenpart

et al [1]. They conclude by stating cloud providers should obfuscate the internal structure of their services and placement policies in order to complicate the adversary's attempts. However, obfuscation of topology and placement policy leads to additional computational overhead when doing VM placement, CPU load balancing, traffic shaping and workload migrations. They additionally state that such obfuscation techniques should be demanded only by customers with strong privacy requirements, but this additional differentiation in user classification and infrastructure configuration leads to more complex registration, preference analysis, and configuration options. We suggest defining a single user management and placement solution that comes with low-computation placement and topology obfuscation inherently, without causing a change in the familiar interface exposed to cloud users. Figure 3.4 provides an overview of the integrated solution's technical architecture.

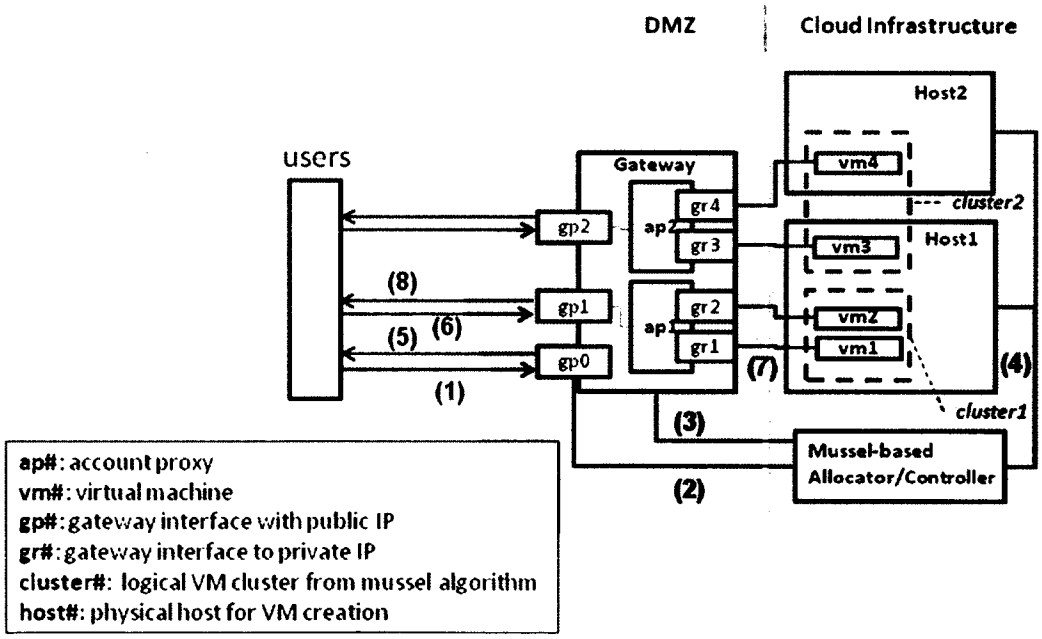


Figure 3.4: Technical architecture of the account proxies and mussel-based account allocation

Here, each logical cluster (cluster#) is managed and accessed by the same account proxy (ap#). An account proxy has one public IP address, which is hence shared by all account owners in a cluster when accessing their instances, and the account proxy maintains the mapping to private IP addresses. There is hence no 1-to-1 mapping of public to private IP addresses or dependence on a sequential allocation of private IP addresses. A 1-to-n mapping of public to private IP addresses is implemented by most modern Application-Level Gateways that include Network Address Translation (NAT) and traversal. The sequence of interactions of a typical user is as follows:

1. Subscription of user with the cloud infrastructure via an accessible gateway interface, gp0, with a static public IP address. The user provides a username, password and collection of preferences (duration, CPU, memory, availability zone), encrypted with the public key of the cloud infrastructure provider.
2. The user information is checked against subscription policies and forwarded to the Mussel-based Allocator/Controller, which is responsible for creating/dissolving groups and account proxies, as well as assigning users and VM instances to account proxies, groups, and physical hosts respectively. VMs with similar workload and access preferences are assigned to the same physical host when possible.
3. The Allocator/Controller creates a new account proxy (ap#) if necessary and assigns or adds the user to an existing account.
4. Asynchronously, the Allocator/Controller selects a host to create the requested VM instance and starts the VM instance assigning it a random IP address from a pool of unassigned private addresses.

5. The public IP of the newly created VM is mapped to the private IP and returned to the user as a uniform resource identifier (URI) of form /{ip of gp#}/{userid}/{vm_id}.
6. The user uses URI to send requests to VM including start, stop, modify or ssh.
7. The account proxy translates URI into a private IP and forwards the requests to VM.
8. Responses from the VM are returned to the user as if the target was the public IP address of the account proxy.

We assume that each user and the cloud provider are able to generate and maintain non-compromised public-private key pairs (e.g. RSA [62]) and symmetric keys (e.g. AES [63]) such that the above interactions can be secured using protocols like transport layer security (TLS) [64]. This is among the current best practices from leading cloud providers such as Amazon [65], and is an effective approach for minimising cloud communications risks such as man-in-the-middle, session high-jacking and replay attacks - as also denoted in [66–68] These types of attacks are hence not the focus of the solution as these are part of best practices in cloud security. On the other hand, we are interested in mitigating the impact co-resident placement and data extraction have on an attacker's ability to carry out successful exploits against a given target set; or, said another way, we would like to determine the maximum amount of co-residence knowledge an attacker can infer when there is workload similarity amongst peers. Towards this end, we specify 3 breach impact levels: low, medium, and high. A low level breach impact means that an attacker cannot differentiate the workload owner and the workload type from that of other peers. A medium level breach impact means that the enumeration of workload owner and workload

type is possible; and a high level breach impact means that an attacker can differentiate workload owner and type.

We now describe the set of capabilities and attack path an attacker needs to execute for targeted co-residence. As shown in Figure 3.5, we do not provide a solution to stopping step 1 - malicious VMs or scripts from being installed in the cloud infrastructure, as this depends on the types of pre-installation scanning mechanisms the provider implements. Our solution aims to remove the usefulness of public-to-private IP address mappings observable by the attacker, which impacts on steps 2, 3, and 4 in the attack path shown in Figure 3.5. Mapping one public IP address to n randomly assigned private IP addresses reduces the specificity of knowledge gained by an attacker with the capability to do internal domain name resolution. The records of mappings will have collisions, which serve to impede targeted co-residence by introducing additional effort and cost for the attacker, in that more brute-force attempts and malicious instances need to be deployed.

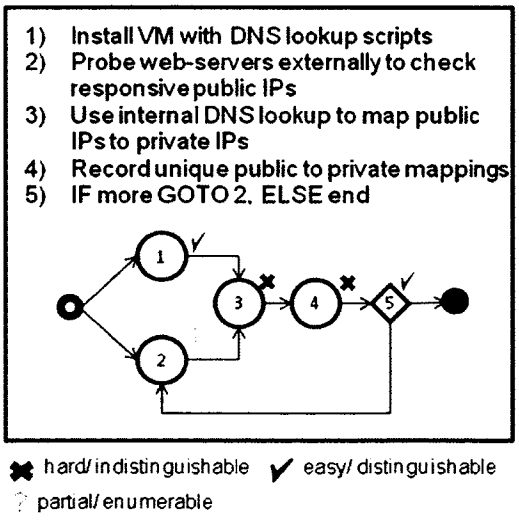


Figure 3.5: Attack capabilities and path to map public to private IPs

Figure 3.6 shows that the critical step 5 in the attack path is disrupted by our approach, as there is no pattern used for private IP address assignment. The assignment of IP addresses by a dynamic host configuration protocol (DHCP) server will follow a predictable sequence by default, but this can be configured to randomly select from the pool of available IP addresses. There is no need for an administrator to allocate IP addresses per availability zone as groups are assigned responsibility for specific IP addresses.

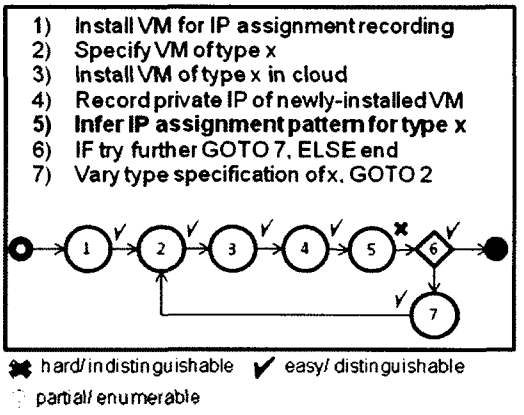


Figure 3.6: Attack path to determine mapping of VM types to IP ranges and availability zones

Notice Figures 3.7 and 3.8. In each case, step 5 is not explicitly addressed by our solution. It is still possible for the attacker to execute tracert on randomly selected private IP addresses and test for co-residence based on equivalent Dom0 addresses or relatively short round trip times. However, in both cases the attacker is forced to follow a random selection as opposed to following a sequence. Therefore a successful co-residence detection does not reveal knowledge about other IP addresses that are numerically close.

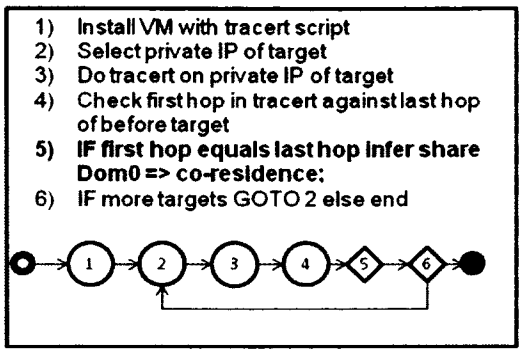


Figure 3.7: Determination of co-residence using Dom0 equivalence check

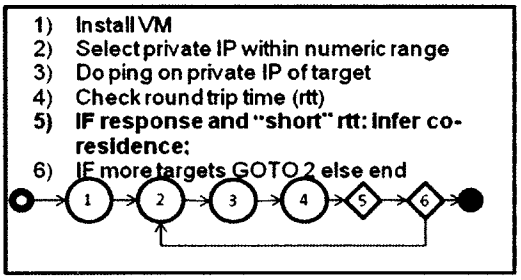


Figure 3.8: Determination of co-residence using relative round trip time estimate

3.6 Risk Assessment

Up until this point, we have discussed how our solution provides measures to prevent users from being targeted and exploited. However, it is quite possible for users to be random victims of non-directed exploits. We now perform a risk assessment to determine the likelihood of this event. With that said, suppose a malicious user decides to randomly target users belonging to any account proxy. Let user A denote a particular user amongst those which could be victimized; and let group B describe all users aside from user A. Below, we list three events.

Event A_1: User A is not victimized

Event A_2: Group B users are not victimized

Event B: A successful exploit occurs

Further,

$P(A_1) = \alpha$	$P(A_2) = 1 - \alpha$
$P(B \mid A_1) = \beta$	$P(B \mid A_2) = 1 - \beta$

Here, $P(A_1)$ is the likelihood that user A is not victimized; $P(A_2)$ is the likelihood group B users are not victimized; $P(B \mid A_1)$ is the likelihood a successful exploit occurs, given that user A is not victimized; and $P(B \mid A_2)$ is the likelihood a successful exploit occurs, given that group B users are not victimized. We assume that events A_1 and A_2 are mutually exclusive. ***We now use Bayes' Theorem to determine the likelihood user A is NOT victimized, given that a successful exploit occurred.***

$$\begin{aligned} P(A_1 \mid B) &= \frac{P(A_1)P(B \mid A_1)}{P(A_1)P(B \mid A_1) + P(A_2)P(B \mid A_2)} \\ &= \frac{\alpha\beta}{\alpha\beta + (1-\alpha)(1-\beta)} \quad . \end{aligned} \tag{3.8}$$

Further, suppose that all users each have a VM deployed, each account proxy has M_j members, and there are N total account proxies. Further consider that each proxy has the same chance of being targeted, and that members assigned to each account have the same chance of being victimized. Then, the chance of user A being victimized is the likelihood that his particular account proxy is targeted, $1/N$, times the likelihood that user A will be randomly targeted, $1/M_j$. This yields $1/NM_j$. Thus, the chance that user A will not be victimized is expressed in Equation 3.9. Notice, here, we use μ to denote the product of N and M_j. Moreover, as denoted in Equation 3.10, we say that the likelihood of user A

being victimized is the same as the chance that users from group B are not victimized.

$$\alpha = 1 - \left[\frac{1}{N}\frac{1}{M_j}\right] = 1 - \frac{1}{\mu} \quad . \tag{3.9}$$

$$1 - \alpha = 1 - \left[1 - \frac{1}{\mu}\right] = \frac{1}{\mu} \quad . \tag{3.10}$$

Substituting Equations 3.9 and 3.10 into Equation 3.8, we receive

$$\begin{aligned} P(A_1 \mid B) &= \frac{\left[1 - \frac{1}{\mu}\right]\beta}{\left[1 - \frac{1}{\mu}\right]\beta + \left[\frac{1}{\mu}\right](1 - \beta)} \\ &= \frac{\beta\mu - \beta}{\mu\left[\frac{\beta\mu - 2\beta + 1}{\mu}\right]} \\ &= \frac{\mu - 1}{\beta^{-1} + \mu - 2} \quad . \end{aligned} \tag{3.11}$$

In a similar sense, ***we use Bayes' Theorem to determine the likelihood that users from group B are NOT victimized, given that a successful exploit occurred.***

$$\begin{aligned} P(A_2 \mid B) &= \frac{P(A_2)P(B \mid A_2)}{P(A_1)P(B \mid A_1) + P(A_2)P(B \mid A_2)} \\ &= \frac{(1 - \alpha)(1 - \beta)}{\alpha\beta + (1 - \alpha)(1 - \beta)} \quad . \end{aligned} \tag{3.12}$$

Substituting Equations 3.9 and 3.10 into Equation 3.12, we find

$$\begin{aligned} P(A_2 \mid B) &= \frac{\left[\frac{1}{\mu}\right][1 - \beta]}{\left[1 - \frac{1}{\mu}\right]\beta + \left[\frac{1}{\mu}\right][1 - \beta]} \\ &= \frac{\frac{1-\beta}{\mu}}{\left[\frac{\mu-1}{\mu}\right]\beta + \frac{1-\beta}{\mu}} \\ &= \frac{1 - \beta}{\beta(\mu - 2) + 1} \quad . \end{aligned} \tag{3.13}$$

Given an exploit, events A_1 ***and*** A_2 ***are equally likely when***

$$P(A_1 \mid B) = P(A_2 \mid B)$$

$$\frac{\mu - 1}{\beta^{-1} + \mu - 2} = \frac{1 - \beta}{\beta(\mu - 2) + 1} \quad . \tag{3.14}$$

Setting $\epsilon = \mu\beta$ and solving for μ we receive

$$\begin{aligned}
(\mu - 1)(\epsilon - 2\beta + 1) &= (1 - \beta)(\beta^{-1} + \mu - 2) \\
\mu\epsilon - 2\epsilon - 1 &= \beta^{-1} - 3 \\
\epsilon(\mu - 2) &= \beta^{-1} - 2 \\
\mu &= 2 + \frac{\beta^{-1} - 2}{\epsilon} \quad .
\end{aligned} \tag{3.15}$$

The parameters for Equations 3.11 3.13, and 3.15 are N, M_j, and β. To understand how the number of members (M_j) affect $P(A_1|B)$ and $P(A_2|B)$, we arbitrarily choose $N = 36$, vary the values of parameters M_j and β, and plot the results. The interpretation of the graphs is quite intuitive. As shown in Figure 3.9, it is highly likely a user A will be NOT victimized, given a successful exploit has occurred.

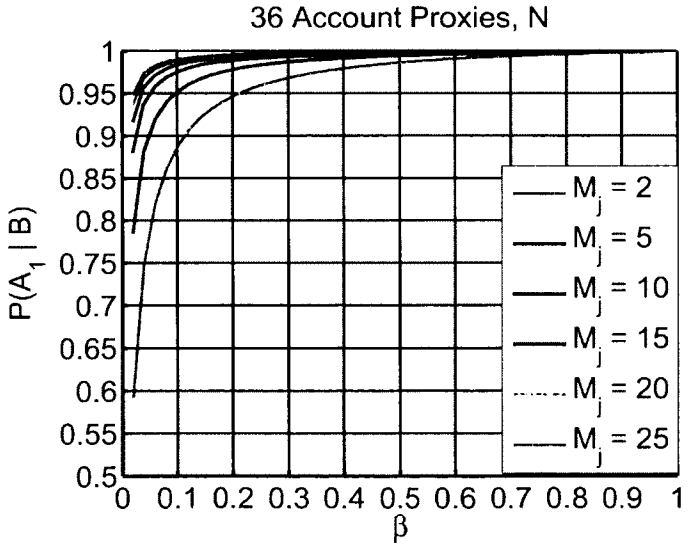


Figure 3.9: $P(A_1|B)$, The likelihood user A is not victimized.

The chances increase both as the number of account proxy members and the number of account proxies increase. That is to say, if you had virtual resources in the cloud and a random non-directed exploit occurred, chances are someone else was affected. This likelihood increases as the number of members subscribed to your account proxy increases - especially for low β values. In Figure 3.9, for instance, notice that having ≥ 15 members ensures that you have a 90% chance of being unaffected when $\beta \leq 0.1$. However, when $\beta \geq 0.5$, these odds are $\geq 90\%$ regardless of proxy membership. The reverse is true for a member from group B. If an exploit occurred, one of the many users in group B is most likely the victim. As shown in Figure 3.10, the chances of a user from group B not being the victim decrease as β increases.

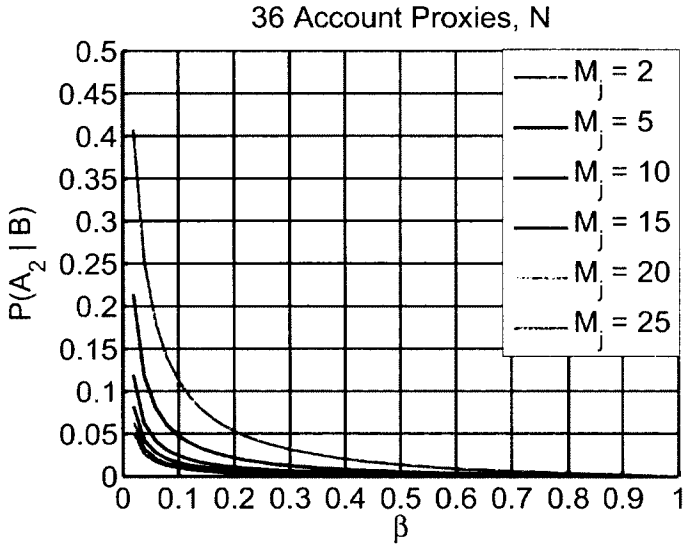


Figure 3.10: $P(A_2|B)$, The likelihood that users from group B are not victimized.

Further, having more members subscribed to one's respective proxy does little to alter these odds when $\beta \geq 0.5$. For events A_1 and A_2 to be equally likely, μ, the product of the number of members, M_j, and the number of proxies, N, has to be an integer - since M_j

and N are both integers. This only occurs for a select values of M_j and β when $\beta \leq 0.4$ - as shown in Figure 3.11. When $\beta > 0.4$, the events are equally likely when $\mu = 2$. The values of M_j are negligible in this case.

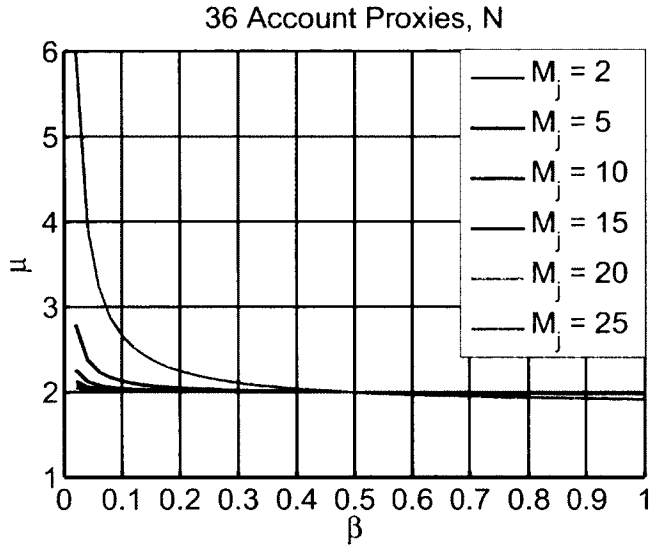


Figure 3.11: $P(A_1|B) = P(A_2|B)$, When events A_1 and A_2 are equally likely

To understand how the number of account proxies (N) affect $P(A_1|B)$ and $P(A_2|B)$, we arbitrarily choose N values above and below 36, and use the same set of values for parameters M_j and β. If N is much lower than 36, we expect individuals to be more at risk of being randomly victimized for lower values of M_j (when compared to the case where $N = 36$). The opposite is true for N values much higher than 36. In this case, we expect individuals to be less at risk of being randomly victimized for lower values of M_j (when compared to the case where $N = 36$). Figures 3.12 and 3.13 tend to support each of these claims. We do not present $P(A_2|B)$ for these graphs - as each decays exponentially (similar to the way that Figure 3.10 corresponds to Figure 3.9).

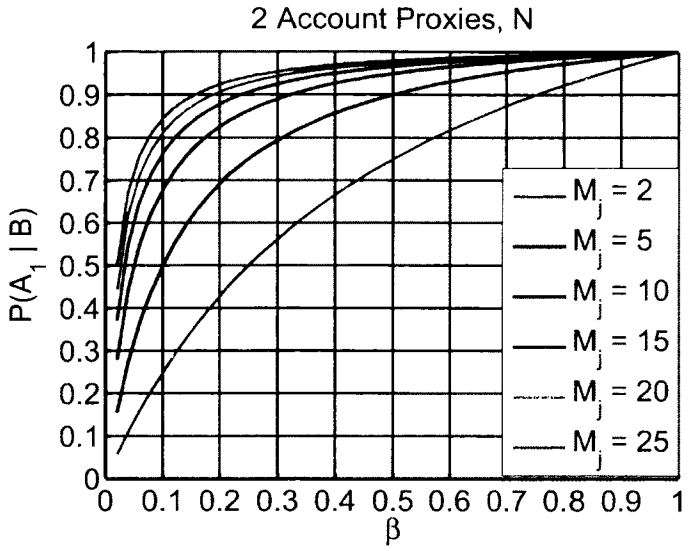


Figure 3.12: $P(A_1|B)$ - The likelihood user A is NOT victimized.

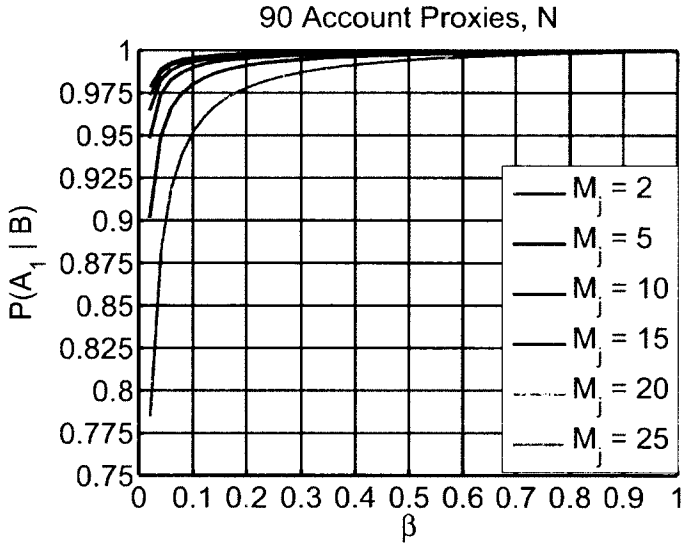


Figure 3.13: $P(A_1|B)$ - The likelihood user A is NOT victimized.

3.7 Conclusion

In this chapter, we considered an exploit which targets cloud users - as outlined by Ristenpart et al [1]. We proposed a solution which relies on mussel-inspired user account and workload clustering and account proxies to obfuscate the public to private IP map. We

then presented arguments to show how our strategy increases the effort required for an adversary to carry out a directed attack against a target set. Further, we gave results from a risk assessment that suggest a reduced per-individual chance of being victimized given a non-directed attack.

3.8 From Cloud Security to Community Detection

The next objective involved extending the mussel algorithm to the community detection domain. However, after conducting various experiments, we determined that the mussel algorithm was not flexible enough to handle network datasets of varying size. It worked well for large populations, where there was a lot of interacting individuals. However, it did not fare as well for small populations. Some social networks can have less than 50 nodes. In such cases, the system never reaches steady state - as there are not enough peers to meet each individual's short and long-term density thresholds. For this reason, we derived another algorithm to detect communities. We present our greedy heuristic for community detection in the next chapter.

CHAPTER 4

SCALE-FREE NETWORK CONNECTIVITIES AND COMMUNITY DETECTION

Recent research suggests that node connectivities in social networks follow a scale-free power-law distribution. On *Twitter*, for instance, large communities tend to form around popular celebrities. Some followers possess an avid interest, while others either preferentially attach or follow those who the majority of their peers follow. We propose a greedy community detection heuristic which exploits these characteristics. Like celebrities, we hypothesize that highly connected nodes, or hubs, form the basic building blocks of communities. We assume that each community has one global hub, and that nodes with lower degrees preferentially attach to hubs in their vicinity. We then frame community detection as a node to hub assignment problem. To show its effectiveness, we test our algorithm on four commonly used real network data sets. We obtain the following modularity (Q) values for three data sets: karate network - Q = 0.3715, dolphin network - Q = 0.3735, and political books - Q = 0.4492. In each case, our algorithm consistently classifies nodes into communities which coincide with their respective known structures. We additionally receive Q = 0.4592 for the college football data set. Unlike other implementations, ours is computationally inexpensive, deterministic, and does not require apriori information.

4.1 Introduction

Many complex networks have underlying community structures which could prove useful when attempting to understand the dynamics of within-group interactions. The potential benefits have broad impacts in a variety of fields, i.e., computer science, biology, marketing, sociology, etc. This has recently led to a strong demand for the development of methods which are effective at detecting and/or discovering community structures. Towards this end, several community detection algorithms have been proposed [69–73]. Most of these, however, share similar disadvantages. Some algorithms require apriori information, such as threshold values or the desired number of communities, while others are impractical and computationally expensive [74]. A third category of algorithms suffer from a combination of the two.

Our objective is to develop a more intuitive and more practical method for community detection. Recent research [75–77] has suggested that social networks tend to follow a heavy-tailed distribution - where a few nodes, called hubs, possess high connectivity, while the overwhelming majority of other nodes possess low connectivity. We hypothesize that hubs form the basic building blocks of communities, and thus are the key to community detection. A detection technique which exploits hubs and the scale-free properties of social networks remains largely unexplored throughout recent literature. This hence provides the necessary motivation for our work. We outline our contributions below.

1. Our algorithm is practical, deterministic, easy to implement, and scalable. It discovers community structures without the need of apriori information, i.e., threshold values, community size restrictions, or the desired number of communities.

2. In [78], the authors define modularity (Q) - a metric for measuring the strength of community structure that since has become well-known and commonly used. They state that in practice values "typically fall in the range from about 0.3 to 0.7." We obtain the following modularity values for three datasets: Zachary's Karate Network [79] - $Q = 0.3715$, bottlenose dolphin network [80] - $Q = 0.3735$, and books about U.S. politics [81] - $Q = 0.4492$. In each case, our algorithm consistently classifies nodes into communities which coincide with their respective known structures. We also apply our method on the American college football data set, and receive $Q = 0.4592$.

The rest of this chapter is divided into seven sections. Section 4.2 offers the necessary background information (i.e., commonly used definitions for network, community, and modularity). Section 4.3 discusses previous work. Section 4.4 defines the problem, gives the approach, and presents our algorithm. Section 4.5 details the real-world network data we consider. Section 4.6 provides our results and compares them to those reported by other researchers. Section 4.7 covers a small discussion on various issues we encountered while validating our algorithm; and Section 4.8 yields concluding remarks.

4.2 Background

4.2.1 What is a Community?

For community detection, a network is typically represented by an undirected and unweighted graph. This graph, $\mathbf{G = (V, E)}$, is an ordered pair comprised of a set of vertices (sometimes referred as nodes), V, interconnected by a set of edges. The goal of community

detection is to find a disjoint partition $G = C_1 \cup C_2 ... \cup C_k$, where each C_k represents a community.

There are various definitions for what constitutes a community. However, a commonly used definition states that a community is a subset or group of nodes where the number of internal connections (between nodes within the group) are exceedingly dense and the number of external connections (to nodes outside the group) are exceedingly sparse. In [82], Radicchi et al. use the degrees of nodes inside and outside a group to define community - in terms of both a weak and strong sense. Assuming that d_i^{in} represents the degrees of node i inside its group C_k, and d_i^{out} represents its degrees of node i outside group C_k, then group C_k is said to form a strong community if the internal degree is greater the external degree for all nodes i in group C_k [83] - as shown in Equation 4.1.

$$d_i^{in} > d_i^{out}, \forall i \in C_k \tag{4.1}$$

On the other hand, group C_k is said to form a community in a weak sense if the sum of the internal degrees of all nodes in group C_k is greater than the sum of the external degrees of all nodes in group C_k [83] - as denoted in Equation 4.2.

$$\sum_{i \in C_k} d_i^{in} > \sum_{i \in C_k} d_i^{out} \tag{4.2}$$

4.2.2 Measuring the Strength of Community Structure

In [78], Newman and Girvan propose a metric for measuring the strength of community structure, which they call modularity (Q). The main idea behind this metric is that the fraction of within-community edges should be greater than the expected number of edges found in a random null model. This model preserves the order of the graph and

individual node degrees, but forms random connections between nodes without regard for community structures. If the number of within-community edges is no better than its random counterpart, then $Q = 0$. On the other hand, a modularity value closer to one denotes strong community structure. They define modularity to be

$$Q = \sum_i (e_{ii} - a_i^2) = Tr\,\mathbf{e} - ||\mathbf{e}^2|| \quad . \tag{4.3}$$

where e_{ii} is the fraction of edges that fall within the same community and a_i is the expected fraction of randomly distributed edges in that community. In [84], Clauset et al. further expand on the definition of modularity and define Q to be

$$Q = \frac{1}{2m} \sum_{vw} \left[A_{vw} - \frac{k_v k_w}{2m} \right] \delta(c_v, c_w) \quad . \tag{4.4}$$

where m is the number of edges, A_{vw} is the adjacency matrix, k_v and k_w are the degrees of nodes v and w, respectively, and c_v and c_w are the communities to which nodes v and w belong, respectively. $\delta(c_v, c_w)$ denotes whether node v and w belong in the same communities. If one then, they both belong to the same community. Otherwise, they each belong to different communities.

4.3 Previous Work

To date, several solutions have been proposed to address the community detection problem, i.e., Girvan-Newman algorithm [78, 85], label propagation algorithm [71, 83, 86], modularity maximization [72, 87], genetic algorithms [70, 88, 89], etc. Most of these, however, share similar disadvantages. Some algorithms require apriori information, such as threshold values or the desired number of communities, while others are impractical and computationally expensive [74]. A third category of algorithms suffer from a combination

of the two. We give two examples of previous research. The first is a community detection method based on divisive hierarchical clustering; and the second is a detection method based on nearest neighbors interactions.

In [78], Newman and Girvan hypothesize that edges which lie between communities have the highest "betweenness" value. They derive three methods for calculating betweenness scores (shortest path, resistor networks, random walk), separately incorporate each of these into a divisive method which removes the links with the highest value, and apply their algorithms to real-world network data. To determine how well their algorithms performed, they develop a metric (modularity) to denote the quality of network partitions. The main drawbacks of the shortest path algorithm, the best of the three, is that it is computationally intensive and does not scale well. The time complexity for each iteration is $O(m^2n)$, where n is the number of nodes and m is the number of edges.

The technique which most resembles ours is the label propagation algorithm (LPA). In [83], Raghaven et al. use this method to assign each node unique labels. Then, in an iterative fashion, nodes adopt the label which coincides with the majority of their neighbors. In the event there is a tie, nodes randomly choose among eligible candidates. The nodes with the same label are assigned to the same community when the maximum number of steps is reached. The authors state that their algorithm, "uses the network structure alone as its guide and requires neither optimization of a pre-defined objective function nor prior information about the communities." Further, it runs near linear time and is thereby less computationally expensive. The downside with this method is that it invokes a stochastic process for breaking ties. This leads to non- deterministic results, i.e., different runs produce different community structures. To remedy this, the authors combine multiple

results to form an aggregate solution. Our solution, on the other hand, comes with similar benefits, but yields consistent results.

4.4 Greedy Heuristic for Community Detection

4.4.1 Problem Definition

We describe community detection as a node assignment problem. ***Node to Hub Assignment Problem (N-HAP):*** *Given an adjacency matrix, assign nodes to hubs such that each node belongs the closest hub with the maximum number of connections, subject to the constraint that the majority of each node's peers are members of the same hub.*

4.4.2 The Approach

To solve this problem, we first hypothesize that hubs, or highly connected nodes, form the basic building blocks of communities; and thus are key to community discovery. The notion that node degrees follow a scale-free power-law distribution for social networks was first proposed by Barabási and Réka. In [77], they found that this distribution was the consequence of two primary mechanisms: continual network expansion by the addition of new nodes and preferential attachment of new nodes to highly connected nodes. In other words, for social networks, a "rich gets richer" effect leads to a handful of nodes having an unusually high number of connections and the majority of nodes having an unusually low number of connections. Given this, we assume that each community has a single hub, and that nodes with lower degrees preferentially attach to hubs in their vicinity. Hence, our approach for solving N-HAP is to have each hub start a community, and have nodes of lower degrees use their nearest neighbors to find and join their respective local communities.

4.4.3 The Algorithm

Our algorithm can be divided in four phases. The first phase is *initialization*. In this phase, each node compares its degree with those of its nearest neighbors. If a node has the highest local degree, it declares itself as a hub. Otherwise, it joins its local hub. The second phase is *oblivious bandwagoning*. In the event a hub is connected to a node that is associated with a hub of a higher degree, it along with its followers join the community associated with that node. This is done to prevent local convergence - for not all local hubs end up being global hubs. The third phase is *majority rules*. Each node then verifies whether it belongs to the same community as the majority of its peers and changes membership in the event that this is not the case. The final phase is to repeat phase two - *oblivious bandwagoning*. This is done to propagate the changes (if any) invoked by phase three throughout the network. In phase one and two, each node carries out the objective - as denoted in Table 4.1; and in phase three and four, each node enforces the constraint - also denoted in Table 4.1.

Table 4.1: N-HAP from global and local perspective

input:	Global	adjacency matrix
objective:	Local	join closest hub with max connections
constraint:	Local	majority of peers are members of the same hub
output:	Global	assignment of nodes to a set of hubs

Our algorithm's pseudocode is given in Algorithm 6. Phase one is implemented in the first for loop - lines 1 - 7. Based on the degree of self and the degree of peers, a node will either: declare itself as a hub, join the hub of a peer with equivalent degree, or join the hub of the peer with the maximum degree. Phase two is implemented in lines 8 - 14. There

are three cases where a node will change hubs. Sometimes a node declares itself as a hub, but has little to no members. In this case, it joins the hub that coincides with the majority of its peers - line 10. There also may be a case where a node has peers whose memberships are equally divided between two or more hubs. Here, a node does a continual refresh each iteration to ensure it joins the hub that has the highest degree - line 12.

Algorithm 6: Greedy heuristic for community detection

Input: Adjacency Matrix
Output: Assignment of Nodes to a Set of Hubs

```
1  foreach node in i in 1 to |V| do
2      if degree of self > degree of peers then
3          declare self as hub;
4      else if no max degree among peers then
5          join hub of 1st tied peer (or take 1st peer as hub);
6      else
7          join hub of peer with max degree;
8  while at least 1 node changes hubs do
9      if hub ∧ most peers belong to another hub then
10         join hub that coincides with majority of peers;
11     if peer membership divided between ≥ 2 hubs then
12         join hub that has greatest degree;
13     if hub joins new hub then
14         join hub that former hub joined;
15 foreach node in i in 1 to |V| do
16     if majority of peers belong to another hub then
17         store ID of node_i;
18         store ID of majority_hub_i;
19         determine priority using max relative hub frequency;
20 use priority to sort nodes that need hub change;
21 use priority to sort new majority_hubs;
22 foreach node in i in 1 to |nodes_to_change| do
23     node_i joins new majority_hub_i;
24 Repeat steps 8 - 17;
```

In the event, a hub joins another hub, its followers update their membership to reflect the new change - line 14. This phase continues until all nodes stop changing hubs. Phase three is denoted by lines 15 - 23. In the *for loop*, nodes determine whether they belong to the same hub as the majority of their peers. If this is not the case, they associate themselves with a ratio which will be used for prioritization. This ratio is simply the number of majority peers which belong to the same hub divided by the total number of peers. Next, according the priority, nodes change hubs to that which coincides with the majority of their peers. Here, prioritization is key - for the act of randomly reassigning nodes to hubs could lead to a misclassification ripple effect. We will cover this in detail in the discussion section. The last phase, shown in line 24, propagates the changes (if any) of hub reassignment due to majority rules.

There is one thing we should note. The constraint specified in N-HAP coincides with the definition of a community in the strong sense as outlined in Equation 4.1. However, there is one case which will prevent full compliance with this constraint - line 10 - a case where a node has the number of peers inside its group equal to the number of peers outside its group. Typically this involves nodes that are situated between communities. A node which falls in this category will not be able to comply with the constraint - for majority rules is not applicable. We refer to this as deadlock. There is no best way to resolve this issue. In our algorithm, a node responds to deadlock by carrying out the objective and disregarding the constraint. In other words, the hub with the highest degree is the tie breaker. For example, if a node of degree six had three peers in one community and three in another, it would simply join the community of the peers which are associated with the hub of the highest degree. This is rather arbitrary, but tends to garner good results. Nonetheless,

given the special case of deadlock, we say that our algorithm creates strong communities in a best effort fashion.

As shown in Algorithm 6, our method contains: three *for loops*, two *while loops*, and two *sorts*. In the initialization phase, one of three basic statements execute per iteration. Assuming that $|V|$ = n, this requires $T = n$ time. In the oblivious bandwagoning phase, again, one of three basic statements execute per iteration. Given that there are less hubs than there are nodes, and that all nodes will not change hubs, this requires $T = kn$ time - where $k < n$. For phase three, a *for loop* is used to determine the priority of node changes. Here, there are two basic store statements, and one statement which stores the maximum value of an array - $T = 2w + \sum_i^w d_i$. Next, there are two *sorts* and another *for loop*. Assuming a quicksort method, this requires $T = 2w * logw + w$. Since some nodes will already belong to the same hub as the majority of their peers, the time phase three requires is $T = 3w + \sum_i^w d_i + 2wlogw$, where $w < n$. Phase four requires the same time as phase 2 - $T = n$. Hence, overall, our algorithm requires time

$$T = 2n + kn + 3w + \sum_i^w d_i + 2wlogw \quad . \tag{4.5}$$

4.4.4 Detecting Community Structure for Complete Graphs

To demonstrate how our algorithm handles complete graphs (a case where there are no hubs), we present a K_5 graph in Figure 4.1 Notice, there are five nodes, and every pair of nodes are connected by a unique edge. The labels denote each node's ID. This graph has strong community structure - for the internal degree (4) is greater than the external degree (0) for all nodes. Thus, all nodes should be assigned to one community.

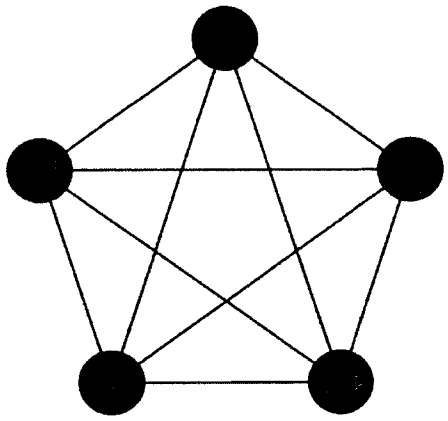


Figure 4.1: K_5 complete graph

Following our algorithm, Algorithm 6 - line 2 does not hold for any node, thus no node would declare itself as hub. In this case, each node would follow line 3, and would take its first peer as hub. Table 4.2 presents the attributes for all nodes. Notice, node 2 is the first neighbor of node 1; and for all other nodes, node 1 is the first neighbor. So in the initialization phase, node 1 would declare node 2 as hub; and all other nodes would declare node 1 as hub. Next, in the oblivious bandwagoning phase, node 1 would notice that most of its peers belong to a different node, line 9, and would declare itself as hub - line 10. In this case, only lines 1-17 are required for community detection. Here, the algorithm's total running time is $O(n)$.

Table 4.2: Node attributes for K_5 complete graph

Node ID	Degree	Neighbors	Max Peer ID	Join Hub
1	4	2,3,4,5	-	2
2	4	1,3,4,5	-	1
3	4	1,2,4,5	-	1
4	4	1,2,3,5	-	1
5	4	1,2,3,4	-	1

4.5 Real-World Network Data Sets

In this section, we present three real-world network data sets commonly used in community detection literature: Zachary's karate club [79], bottlenose dolphin network [80], and books on U.S. politics [81]. For each data set, we give a brief overview, show the known community structures (as discovered by our method), and display the histogram.

4.5.1 Zachary's Karate Club

Zachary's karate club data set describes a three year observation of social interactions between members of a karate club. Towards the beginning, a dispute emerged between the club president and the karate instructor over the cost of lessons. Over time, this dispute caused a rift between the club members. Some aligned themselves with the president, while others rallied support for the instructor. Thus, there are two known communities in this network. The graph of this network's known structures, as discovered by our method, is shown in Figure 4.2. Notice, there are two nodes with high connectivity - node 1, which represents the instructor, and node 34 which represents the president. The other nodes have low connectivity and are connected directly or indirectly to either node 1 or node 34. The degree distribution is shown in Figure 4.3. As expected, the majority of nodes with low degrees appear with great frequency, while the nodes with high degrees appear great infrequency. This tends to support the idea that social networks have scale-free properties. Further, the fact that the club divided into two factions - each centered around the two most connected people tends to give credence to our initial hypothesis that communities tend to form around hubs - in this case nodes 1 and 34.

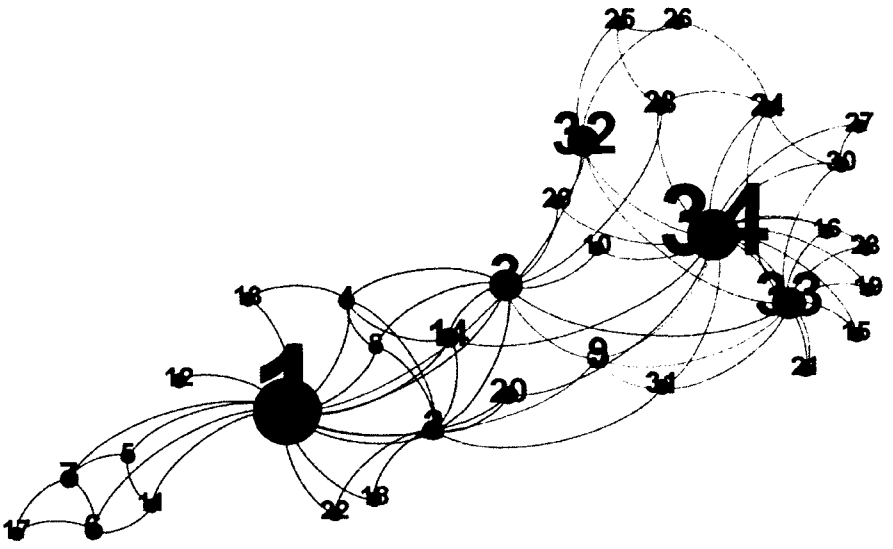


Figure 4.2: Zachary's karate network: known structure as discovered by our method

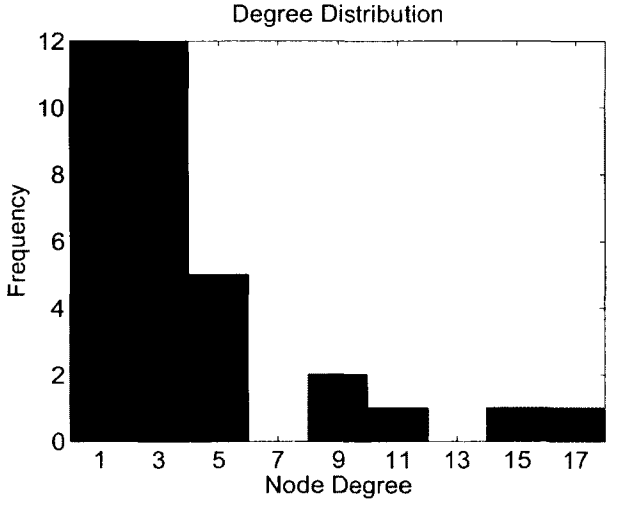


Figure 4.3: Zachary's karate network degree distribution

This data set is extensively used in community detection literature. The network is small - consists of 34 nodes and 78 edges. Thus, the results from a community detection algorithm can easily be verified by hand. Methods which use this data set tend to misclassify one node - typically either node 3 or 10. The authors in [78] misclassify node 3. They place it in the pink community instead of the green community. The authors in [90]

misclassify node 10. Further, the authors in [74] consistently place 33 out of 34 nodes in the correct community. A question which naturally comes to mind is: why are these particular nodes hard to correctly classify? Notice nodes 3 and 10 in Figure 4.2. They are both deadlocked - as both have an equal number of neighbors in each community. Node 3 has five neighbors in each community; and node 10 has one neighbor in each community. This poses a problem because it violates the quantitative definition for a community in the strong sense. For other algorithms, such as the LPA, a random selection may be used to select a community given these circumstances. In the discussion section, we explain how our algorithm handles this problem for this data set.

4.5.2 Bottlenose Dolphin Network

The bottlenose dolphin network data set describes a seven year observation of social interactions among dolphins in Doubtful Sound, New Zealand. Here, the school divided into two groups when the individuals connected at the boundaries of both groups suddenly disappeared. This network contains 62 nodes and 159 edges. The graph of the known community structures, as discovered by our method, is shown in Figure 4.4. Notice, the degree distribution in Figure 4.5. There seems to be a few nodes with degrees higher than expected. This may lead to local convergence/subgrouping - as we shall see. Though there are two known communities, many detection algorithms divide this data set into four communities [78, 86–89].

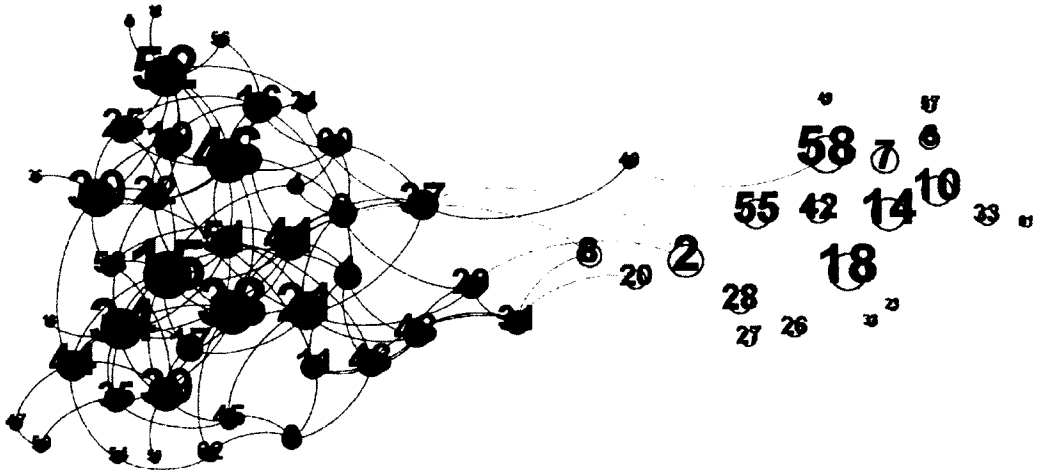


Figure 4.4: Bottlenose dolphin network: known structure as discovered by our method

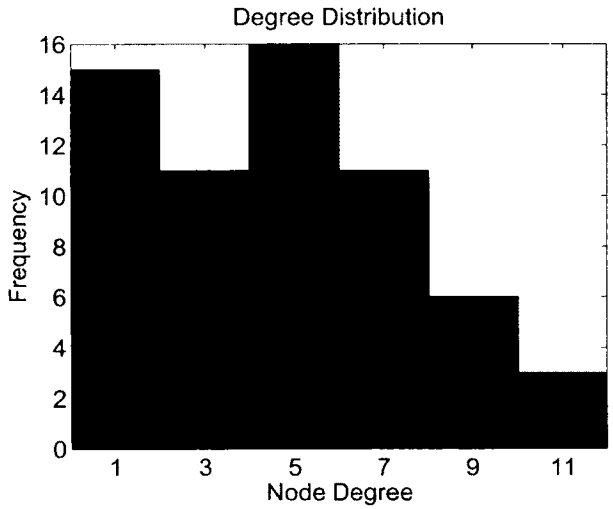


Figure 4.5: Bottlenose dolphin network degree distribution

4.5.3 Books on U.S. Politics

This data set describes political books purchased on Amazon.com [91] during the 2004 U.S. presidential election. Relationships between books represent books frequently purchased by the same buyers. As shown in Figure 4.6, the book communities divide along political preference and affiliation - i.e., liberal or conservative. The degree distribution is shown in Figure 4.7. Here, the disparity between the frequency of nodes with low

degree to that of nodes with high degree is quite noticeable. Though there are two known communities, researchers tend to detect anywhere from three to five groups in this data set [86–88].

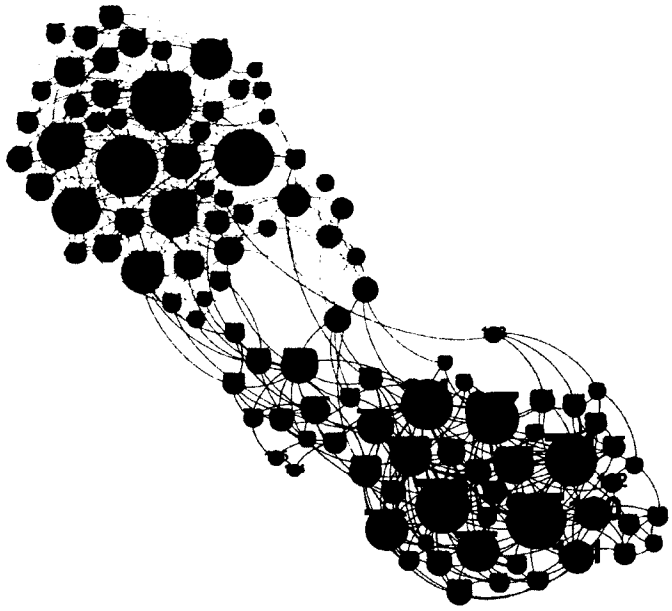


Figure 4.6: Books on U.S. politics: known structure as discovered by our method

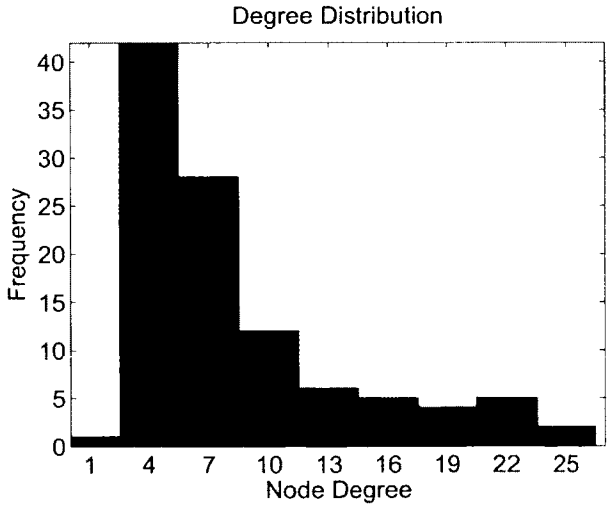


Figure 4.7: Books on U.S. politics degree distribution

4.6 Results

In this section, we present our results along with a host of others - as published by researchers for the same data set. We test our algorithm on the three data sets mentioned in the prior section and an additional data set - American college football. Further, we use Equation 4.4 to measure the strength of community structures - as detected by our method on these data sets.

In [78], the authors state that in practice modularity values "typically fall in the range from about 0.3 to 0.7." We obtain the following modularity values for 3 datasets: Zachary's Karate Network [79] - ***Q = 0.3715***, bottlenose dolphin network [80] - ***Q = 0.3735***, and books about U.S. politics [81] - ***Q = 0.4492***. In each case, our algorithm consistently classifies nodes into communities which coincide with their respective known structures. Table 4.3 shows our method's time complexity for each of these data sets according to Equation 4.5. Notice that our method executes in linear time for each data set. For the

Table 4.3: Our method's time complexity for each data set

	k	w	$T(n)$
Karate	2	3	$4n + 6\log 3 + \sum_i^3 d_i + 9 = O(n)$
Dolphins	3	1	$5n + d_i + 3 = O(n)$
Books	3	1	$5n + d_i + 3 = O(n)$

fourth data set, American college football, we receive a Q value which indicates strong community structure - ***Q = 0.4592***. However, our algorithm only detects four communities - where the number of known communities, for this data set, ranges anywhere from eight to 12. Figure 4.8 provides a possible explanation. The histogram for this data set is the complete opposite from what we have seen in the other cases. Nodes with high degrees are

the majority. They appear with great frequency, while the nodes with low degrees appear with great infrequency. Despite this, the Q value we obtain is consistent with those reported by other researchers.

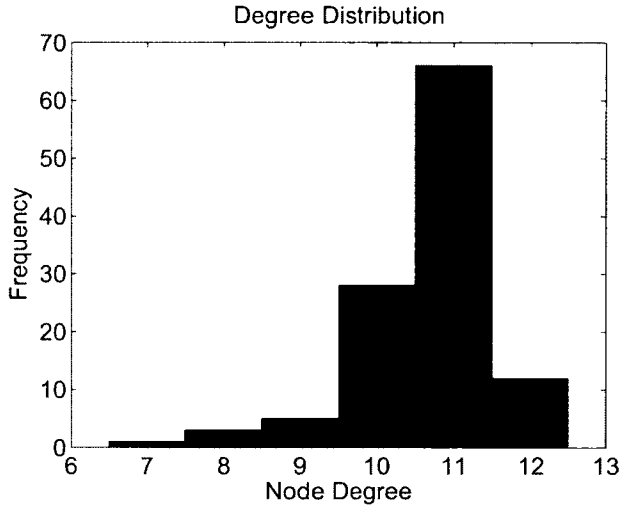


Figure 4.8: American college football degree distribution

We present the properties for each data set and the community attributes as detected by our method and previously published methods in Table 4.4. Dashes in the table mean that we were unable to obtain the designated information for a given method. *Cm* in column two is the number of known communities for each data set; and *# Cm Detected* in column five is the number of communities detected by each method. Notice that communities detected by different methods may or may not coincide with the known number of communities.

Table 4.4: Data set properties and community attributes as detected by our method and previously published methods.

Data Set	# Nodes, Edges, Cm	Method in Paper	Apriori?	# Cm Detected	Q
Zachary's Karate Club	34, 78, 2	SP betweenness [78]	Yes	2	0.3500
		Random Walks [92]	No	-	0.3710
		Greedy Heuristic [this work]	**No**	**2**	**0.3715**
		Mod Maximization [93]	-	2	0.3810
		Label Propagation [86]	-	2	0.4180
		Extreme Optimization [94]	-	4	0.4188
		Eigenvector-based [95]	-	-	0.4190
		Mod Maximization [87]	-	4	0.4200
Bottlenose Dolphin	62, 159, 2	**Greedy Heuristic [this work]**	**No**	**2**	**0.3735**
		Genetic Algorithm [88]	-	-	0.5050
		Genetic Algorithm [89]	-	-	0.5070
		SP betweenness [78]	Yes	4	0.5200
		Label Propagation [86]	-	-	0.5230
		Mod maximization [87]	-	-	0.5290
Books on U.S. Politics	105, 441, 2	**Greedy Heuristic [this work]**	**No**	**2**	**0.4492**
		Fast Algorithm [69]	-	-	0.5020
		Genetic Algorithm [88]	-	-	0.5180
		Label Propagation [86]	-	-	0.5270
		Mod maximization [87]	-	5	0.5272
American College Football	115, 663, 8-12	**Greedy Heuristic [this work]**	**No**	**4**	**0.4592**
		Genetic Algorithm [88]	-	-	0.5150
		Mod maximization [93]	-	6	0.5460
		Genetic Algorithm [89]	-	-	0.5770
		Label Propagation [86]	-	-	0.6040
		Mod maximization [87]	-	10	0.6460

Although we present our results alongside that of others, there are subtle differences which limit direct comparison. Two things should be noted. One, some of the Q values reported by other researchers are the results of averages or rounding, whereas the values we present are not. Our method is deterministic in nature. Therefore, multiple runs consistently yield the same results. Two, some of the methods require apriori information, whereas ours does not. Our method only uses the network structure, i.e., the adjacency matrix, as a guide. The common trend observed throughout literature is to generically compare Q values - with the basic assumption that a greater Q value denotes a more effective algorithm. We maintain that this type of brute force comparison can be slightly misleading - as more information may be needed. For instance, for the American college football network our method detected the four communities and obtained Q = 0.4592. How does this compare to an apriori method which detected less than the known number of communities it was given, but received a higher (averaged/rounded) Q value? Perhaps, the solution is to compare like methods - a priori with apriori methods and non-apriori with non-apriori methods. However, this is no easy task. Rarely do researchers clearly state how well the communities their method detected coincides with the known community structures - let alone whether their method require apriori information. This just goes to show the convoluted nature of establishing and executing a fair, direct, thorough result comparison.

The bottlenose dolphin network contains biggest disparity between the results we present and those reported by others. This is due to the fact that our method exactly detected the two known community structures, and the others most likely detected four communities. For instance, in [78], Newman and Girvan detect four groups and report $Q = 0.52 \pm 0.03$; but they also give the Q value for the two known communities - $Q = 0.38 \pm 0.08$. The latter

coincides with the results we receive. Further, the fact that all other researchers received Q values in the range of the former, leads us to conclude that their methods detected four groups as well - though not explicitly stated.

Another thing to note is that the modularity maximization method, proposed by authors in [87], yields the highest Q value for each data set. This is not happenstance - as the goal of the technique is to do just as its name suggests - find the community structures which maximize modularity for a given data set. This is done in hopes that the resulting group structures closely coincide with the known group structures. In [87], the number of community structures which maximized modularity is as follows: Zachary's karate club, four, where the known number is two; books on U.S. politics, five, where the known number is two-three; bottlenose dolphin network - number not reported; American college football, 10, where the known number is eight-12. The authors also report the Q value found by their method for the known structures of Zachary's karate club. They state, "the bipartition found by the VP method has a modularity of 0.3718, whereas the partition corresponding to the actual factions in the club has a lower modularity of 0.3715." Please note that this exactly coincides with the Q value we obtain using our method for this data set - Q = 0.3715. The authors go on to assert that the higher modularity value, "explains the misclassification of node 10, and also emphasizes that no clustering objective can be guaranteed to always recover the semantically correct community structure in a real network. The latter should be taken as a cautioning against accepting modularity-maximizing clusterings as ground truth." Thus, a higher Q value does not always mean a better method for detecting known group structures, it simply may be the result of misclassifications.

4.7 Discussion

Given that our method implements the objective (without regard for the constraint first), then adheres to the constraint after, some nodes end up displaced. The nature of the constraint itself is the reason we do this. It requires that nodes belong to the same community as the majority of their peers. Instead of deriving techniques to tackle the constraint directly (from the beginning), we assume that this implicitly mandates some basic pre-existing community structure - for once communities have already been established, this becomes a rudimentary exercise, i.e, it is easy to find where your friends are if they already are grouped together. It turns out that our oblivious bandwagoning technique is an effective way to establish basic community structures. In the cases we've seen, this alone defaultly places the majority of nodes in same communities as that of their peers. There may be a few exceptions. By actively invoking the majority rules constraint, we identify and resolve these exceptions. The process of exception resolution cannot be done haphazardly or it may lead the a misclassification ripple effect. Therefore, we derive a way to prioritize node changes to minimize the chance of this occurring. We present two examples we encountered while validating our algorithm. One deals with misclassification ripple effect; and the other deals with continual refreshing for deadlocked nodes. We also discuss measures we took to avoid local convergence.

4.7.1 Misclassification Ripple Effect

As mentioned earlier, some community detection algorithms tend to misplace a node for the Zachary's karate club data set: either node 3 or node 10. Depending on how community structures are defined, both of these nodes could end up deadlocked. Since our

method handles this case by implementing the objective only, we did not have a problem classifying node 10. It joined the community with the highest degree - and that coincided with the correct classification. However, after exercising oblivious bandwagoning, we noticed that three nodes were displaced - nodes 3, 14, and 20 (as shown in Figure 4.9).

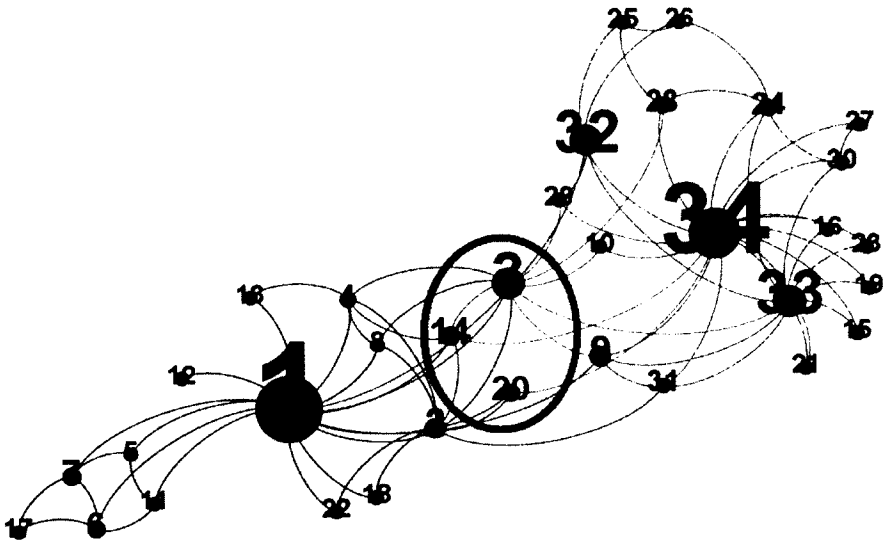


Figure 4.9: The lack of prioritization may lead to misclassification ripple effect.

Further, we found that implementing majority rules in a haphazard fashion did not remedy the situation. Initially, the exception resolution happened by node ID in ascending order. So node 3 would invoke majority rules, then node 14, then node 20. What we noticed was a misclassification ripple effect - nodes initially misclassified via oblivious bandwagoning tend to propagate misclassification if resolved haphazardly. As shown in Figure 4.9, node 3 is connected to node 14; and node 14 is misclassified. If node 3 invokes majority rules first, it will end up deadlocked. In this case, it would join the community with the highest degree - which, in this case, would be the wrong classification. Node 14 and node 20, on the other hand, can be resolved without issues - majority rules works just

fine for these nodes. Thus, leaving things unaltered, we would end up with the same result as other researchers - one misclassification - node 3. The correct thing to do would be to resolve node 14 before node 3. In order to do this, we came up with a way to prioritize node changes based on the ratio of the number of majority peers belonging to a single community to the total number of peers. This way, those nodes who do not in deadlock always invoke majority rules first. This is done in hopes to break the tie of a deadlocked node. Following this, node 20 changes first - its ratio is $2/3$. Node 14 goes next. Even though, node 3 is misclassified its ratio is $3/5$. Then, with a $1/2$ ratio, node 20 goes last.

4.7.2 Continual Refreshing for Deadlocked Nodes

For the bottlenose dolphin network, we initially ended up with one misclassification, and this was due to the lack of continual refreshing. As shown in Figure 4.10, node 40 is deadlocked. Our rules state that node 40 should go to the hub with the highest connections. Since node 58 had a higher degree than node 37, node 40 joined node 58 or the hub node 58 belonged to. Though it took time to propagate the changes, node 37 ultimate joined the hub associated with node 15. At this moment, node 40 was supposed to switch to the hub associated with node 37, because it had a higher number of connections than the hub associated with node 58; but this did not happen because node 40 did not observe both its peers after the initial observation. It joined node 58 and did whatever it (node 58) did. To resolve this, we simply created a rule to enforce continual refreshing. Nodes involved in deadlock should continuously observe peers to avoid potential misclassifications.

4.7.3 Avoiding Local Convergence

Initially, for the bottlenose dolphin network, we received the same results as those presented in [78]: four communities. Each community centered around four highly connected nodes - nodes 15, 46, 18, and 58. As shown in Figure 4.10, nodes 15 and 46 are subgroups which combine to form a larger community on the left half. The same goes for nodes 18 and 58 on the right half. To avoid local subgroup convergence, we added the rule that nodes which declare themselves as hubs should periodically observe their peers and join the hub associated with the majority of their peers. This ultimately resulted in two groups centered around nodes 15 and 18.

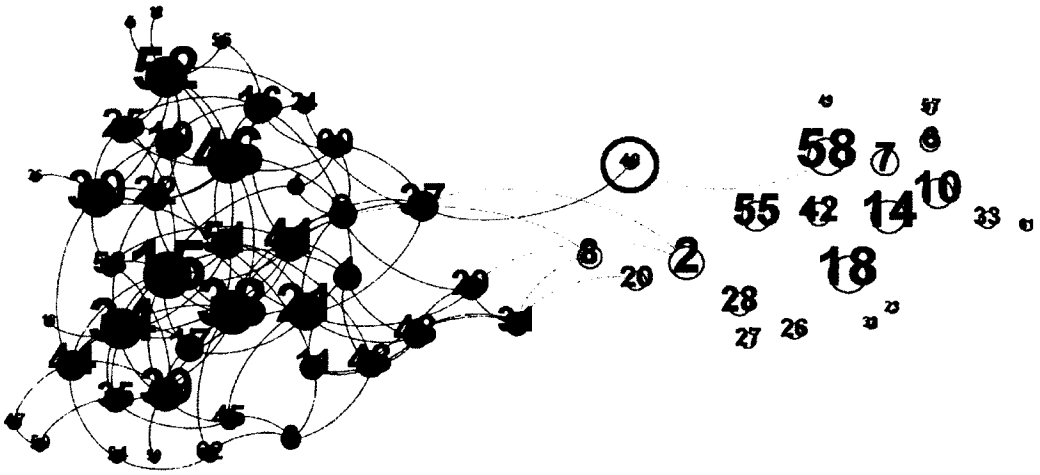


Figure 4.10: A misclassified node due to the lack of continual refreshing.

4.8 Conclusion

In this chapter, we framed community detection as a node to hub assignment problem and developed a greedy heuristic which exploits the scale-free nature of social networks to solve this problem. Our heuristic is practical, easy to implement, and deterministic. Further, it discovers community structures without the need of a priori information, i.e.,

threshold values, community size restrictions, or the desired number of communities. The results are promising - as our method, in some cases, outperforms apriori algorithms when tested on the same data set. This tends to give credence to our initial hypothesis that hubs play a key role in community discovery.

CHAPTER 5

CONCLUSION

The work contained in this dissertation sought to use power-law properties of social groups to answer two research questions.

1. From an insider's perspective, in what ways do communities emerge? That is, what internal processes have to occur on the micro-level to have group formation emerge on the macro-scale? Can mussels and Lévy walks be used to describe these processes? How can this type of behavior be used as a defense strategy?
2. From an outsider's perspective, how to detect communities once they have formed? Given each individual's local connections only, is it possible to classify individuals into their respective known global communities? How can the scale-free properties of social networks help shed light on this problem?

To address the first research question, we presented a defense strategy, in part inspired by mussel self-organization, to address a security concern in cloud systems. The strategy obfuscated public to private IP mapping by having account proxies perform $1-to-n$ random mapping of public to private IP addresses. This decreased the risk of adversary targeting and significantly reduced the amount of public IP addresses needed for users to access their VM instances. By having clusters periodically dissolve, our strategy decreased

the chances of directed attacks towards random users belonging to a particular account proxy.

To address the second research question, we used the scale-free properties of social networks to develop a greedy heuristic for community detection. We hypothesized that highly connected nodes, or hubs, formed the basic building blocks of communities; and assumed that each community had one global hub, and that nodes with lower degrees preferentially attached to hubs in their vicinity. We developed an algorithm based on this notion and tested it on commonly used real network data sets. In most cases, it classified nodes into communities which coincided with their respective known structures. Unlike other implementations, it did not required apriori information and detected communities in a computationally inexpensive and deterministic manner.

As for future directions, we look to: integrate the mussel algorithm as a functional part of the cloud and observe how well it performs in practice; conduct more extensive tests and use the community detection algorithm on large network data sets; and find other problem domains where the mussel algorithm will prove useful.

BIBLIOGRAPHY

[1] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, Nov. 2009, pp. 199–212. [Online]. Available: http://dx.doi.org/10.1145/1653662.1653687.

[2] K. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. Wasserman, and N. Wright, "Performance analysis of high performance computing applications on the amazon web services cloud," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. Indianapolis, Indiana, USA: Dec. 2010, pp. 159 –168.

[3] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A performance analysis of ec2 cloud computing services for scientific computing," in *Cloud Computing*. Springer Berlin Heidelberg, 2010, vol. 34, pp. 115–131. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12636-9_9.

[4] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn, "Case study for running hpc applications in public clouds," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 395–401. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851535.

[5] Hadoop. http://hadoop.apache.org/ [Last accessed: 2011-05-10].

[6] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, January 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492.

[7] Pegasus. http://pegasus.isi.edu/ [Last accessed: 2011-05-10].

[8] Swift. http://www.ci.uchicago.edu/swift/main/ [Last accessed: 2011-05-10].

[9] Saga. http://saga.cct.lsu.edu/ [Last accessed: 2011-05-10].

[10] Condor. http://www.cs.wisc.edu/condor/ [Last accessed: 2011-05-10].

[11] Wcps. http://www.opengeospatial.org/standards/wcps/ [Last accessed: 2011-05-10].

[12] Amazon elastic mapreduce. http://aws.amazon.com/elasticmapreduce/ [Last accessed: 2011-05-10].

[13] C. Zhang and H. D. Sterck, "Cloudbatch: A batch job queuing system on clouds with hadoop and hbase," in *2nd IEEE International Conference on Cloud Computing Technology and Science*. Indianapolis, Indiana, USA: Dec. 2010, pp. 368–375.

[14] R. B. Wei Lu, Jared Jackson, "Azureblast: A case study of developing science applications on the cloud," in *Proceedings of the 1st Workshop on Scientific Cloud Computing (Science Cloud 2010)*, Chicago, Illinois, Jun. 2010.

[15] D. Wall, V. F. Parul Kudtarkar, R. Pivovarov, P. Patil1, and P. J. Tonellato, "Cloud computing for comparative genomics," *BMC Bioinformatics*, vol. 11, no. 259, 2010.

[16] C. Zhang, H. D. Sterck, A. Aboulnaga, H. Djambazian, and R. Sladek, "Case study of scientific data processing on a cloud using hadoop," *Springer-Verlag Berlin Heidelberg*, 2010.

[17] C. Vecchiola, S. Pandey, and R. Buyya, "High-performance cloud computing: A view of scientific applications," in *Proceedings of the 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, ser. ISPAN '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 4–16. [Online]. Available: http://dx.doi.org/10.1109/I-SPAN.2009.150.

[18] T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox, "Cloud computing paradigms for pleasingly parallel biomedical applications," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 460–469. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851544.

[19] R. Grossman and Y. Gu, "Data mining using high performance data clouds: experimental studies using sector and sphere," in *Proceeding of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '08. New York, NY, USA: ACM, 2008, pp. 920–927. [Online]. Available: http://doi.acm.org/10.1145/1401890.1402000.

[20] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. Berman, and P. Maechling, "Scientific workflow applications on amazon ec2," in *Workshop on Cloud-based Services and Applications in conjunction with 5th IEEE International Conference on e-Science (e-Science 2009)*. Oxford, UK: Dec. 2009, pp. 59–66.

[21] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good, "On the use of cloud computing for scientific workflows," in *eScience, 2008. eScience '08. IEEE Fourth International Conference on.* Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2008, pp. 640–645.

[22] Haproxy. http://haproxy.1wt.eu/ [Last accessed: 2011-05-10].

[23] Monit. http://mmonit.com/monit/ [Last accessed: 2011-05-10].

[24] Node.js. http://nodejs.org/ [Last accessed: 2011-05-10].

[25] Apachebench. http://httpd.apache.org/docs/2.0/programs/ab.html [Last accessed: 2011-05-10].

[26] Heartbeat. http://www.linux-ha.org/wiki/Heartbeat [Last accessed: 2011-05-10].

[27] V8 javascript engine. http://code.google.com/p/v8/ [Last accessed: 2011-05-10].

[28] Eucalyptus. http://open.eucalyptus.com [Last accessed: 2011-05-10].

[29] Centos. http://www.centos.org/ [Last accessed: 2011-05-10].

[30] Screen. http://www.gnu.org/software/screen/ [Last accessed: 2011-05-10].

[31] Secure shell. http://www.openssh.com/ [Last accessed: 2011-05-10].

[32] About.com. Using rack. http://ruby.about.com/od/rack/a/Using-Rack.htm [Last accessed: 2011-05-10].

[33] P. Teixeira. Asynchronous iteration patterns. http://nodetuts.com/tutorials/19-asynchronous-iteration-patterns.html#video [Last accessed: 2011-05-10].

[34] M. C. Schatz, "Cloudburst: highly sensitive read mapping with mapreduce," *Bioinformatics*, vol. 25, no. 11, pp. 1363–1369, 2009.

[35] B. Langmead, M. Schatz, J. Lin, M. Pop, and S. Salzberg, "Searching for SNPs with cloud computing," *Genome Biology*, vol. 10, no. 11, pp. R134+, Nov. 2009. [Online]. Available: http://dx.doi.org/10.1186/gb-2009-10-11-r134.

[36] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud." in *EuroSys*, C. M. Kirsch and G. Heiser, Eds. ACM, 2011, pp. 301–314.

[37] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672.

[38] H. Takabi, J. B. D. Joshi, and G.-J. Ahn, "Security and privacy challenges in cloud computing environments," *IEEE Security and Privacy*, vol. 8, no. 6, pp. 24–31, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1109/MSP.2010.186.

[39] B. Grobauer, T. Walloschek, and E. Stocker, "Understanding cloud computing vulnerabilities," *IEEE Security and Privacy*, vol. 9, pp. 50–57, 2011.

[40] S. Paquette, P. T. Jaeger, and S. C. Wilson, "Identifying the security risks associated with governmental use of cloud computing," *Government Information Quarterly*, vol. 27, no. 3, pp. 245–253, 2010. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0740624X10000225.

[41] K. Popovic and Z. Hocenski, "Cloud computing security issues and challenges," *Computer*, no. 3, pp. 344–349, 2010. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5533317.

[42] S. Pearson and A. Benameur, "Privacy, security and trust issues arising from cloud computing," *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. Indianapolis, Indiana, USA: Dec. 2010, pp. 693–702.

[43] A. Colorni, M. Dorigo, and V. Maniezzo, "Distributed Optimization by Ant Colonies," in *Proceedings of ECAL91 - European Conference on Artificial Life*. Paris, France: Dec. 1991, pp. 134–142.

[44] A. E. Hirsh and D. M. Gordon, "Distributed problem solving in social insects," *Annals of Mathematics and Artificial Intelligence*, vol. 31, no. 1, pp. 199–221, 2001. [Online]. Available: http://www.springerlink.com/index/T648V87668512N27.pdf.

[45] C. M. Drea and A. N. Carter, "Cooperative problem solving in a social carnivore," *Animal Behaviour*, vol. 78, no. 4, pp. 967–977, 2009. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S000334720900339X.

[46] D. Werdenich and L. Huber, "A case of quick problem solving in birds: string pulling in keas, nestor notabilis," *Animal Behaviour*, vol. 71, no. 4, pp. 855–863, 2006. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0003347206000364.

[47] A. Dussutour, J.-L. Deneubourg, S. Beshers, and V. Fourcassi, "Individual and collective problem-solving in a foraging context in the leaf-cutting ant atta colombica." *Animal Cognition*, vol. 12, no. 1, pp. 21–30, 2009. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/18560906.

[48] E. Jelnikar and I. M. Cote, "Predator-induced clumping behaviour in mussels (mytilus edulis linnaeus)," *Journal of Experimental Marine Biology and Ecology*, vol. 235, no. 2, pp. 201–211, 1999. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0022098198001555.

[49] M. M. Casey and D. Chattopadhyay, "Clumping behavior as a strategy against drilling predation: Implications for the fossil record," *Journal of Experimental Marine Biology and Ecology*, vol. 367, no. 2, pp. 174–179, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/B6T8F-4TW9WHW-1/2/00a8aaa5177c0ceea5863d60b0591330.

[50] J. Kobak, T. Kakareko, and M. Poznanska, "Changes in attachment strength and aggregation of zebra mussel, dreissena polymorpha in the presence of potential fish predators of various species and size," *Hydrobiologia*, vol. 644, no. 1, pp. 195–206, 2010. [Online]. Available: http://dx.doi.org/10.1007/s10750-010-0113-2.

[51] B. Okamura, "Group living and the effects of spatial position in aggregations of mytilus edulis," *Oecologia*, vol. 69, no. 3, pp. 341–347, 1986. [Online]. Available: http://www.springerlink.com/index/10.1007/BF00377054.

[52] J. Lin, "Predator-prey interactions crabs and ribbed mussels clumps between blue living in clumps," *Estuarine, Coastal and Shelf Science*, vol. 32, pp. 61–69, 1991.

[53] C. Shields and B. N. Levine, "A protocol for anonymous communication over the internet," in *Proceedings of the 7th ACM Conference on Computer and Communications Security*, ser. CCS '00. New York, NY, USA: ACM, 2000, pp. 33–42.

[54] M. K. Reiter and A. D. Rubin, "Crowds: anonymity for Web transactions," *ACM Transactions on Information and System Security*, vol. 1, no. 1, pp. 66–92, 1998. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.2499.

[55] A. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. S. Wallach, "AP3: Cooperative, decentralized anonymous communication," in *ACM SIGOPS European Workshop*, M. Castro, Ed. New York, NY, USA: ACM, 2004, p. 30.

[56] M. Rennhard and B. Plattner, "Introducing morphmix: peer-to-peer based anonymous internet usage with collusion detection," in *Proceedings of the 2002 ACM Workshop on Privacy in the Electronic Society*, ser. WPES '02. New York, NY, USA: ACM, 2002, pp. 91–102. [Online]. Available: http://doi.acm.org/10.1145/644527.644537.

[57] Amazon Web Services LLC. Amazon ec2. http://aws.amazon.com/ec2/ [Last accessed: 2013-18-01].

[58] M. Rietkerk and J. Van De Koppel, "Regular pattern formation in real ecosystems," *Trends in Ecology & Evolution*, vol. 23, no. 3, pp. 169–75, 2008. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/18255188.

[59] M. de Jager, F. J. Weissing, P. M. J. Herman, B. A. Nolet, and J. van de Koppel, "Levy walks evolve through interaction between movement and environmental complexity," *Science*, vol. 332, no. 6037, pp. 1551–1553, 2011. [Online]. Available: http://www.sciencemag.org/content/332/6037/1551.abstract.

[60] A. James, M. Plank, and A. Edwards, "Assessing levy walks as models of animal foraging." *J R Soc Interface*, vol. 8, no. 62, pp. 1233–47, 2011.

[61] A. M. Edwards, "Using likelihood to test for Lévy flight search patterns and for general power-law distributions in nature," *Journal of Animal Ecology*, vol. 77, no. 6, pp. 1212–1222, 2008. [Online]. Available: http://dx.doi.org/10.1111/j.1365-2656.2008.01428.x.

[62] B. Kaliski and S. O. T. Memo, "Public-key cryptography standards (pkcs) #1: Rsa cryptography specifications version 2.1", rfc 3447," 2003.

[63] "Specification for the advanced encryption standard (aes)," Federal Information Processing Standards Publication 197, 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[64] T. Dierks and E. Rescorla, "The transport layer security (tls) protocol," in *IETF RFC 4346*, 2006.

[65] "Amazon aws security best practices," Amazon Web Services LLC, Jan 2011. [Online]. Available: http://media.amazonwebservices.com/Whitepaper_Security_Best_Practices_2010.pdf.

[66] G. Brunette and R. Mogull, "Security Guidance for critical areas of focus in Cloud Computing V2. 1," *CSA (Cloud Security Alliance), USA. Online: http://www. cloudsecurityalliance. org/guidance/csaguide. v2*, vol. 1, 2009.

[67] M. Jensen, J. Schwenk, N. Gruschka, and L. Iacono, "On technical security issues in cloud computing," in *Cloud Computing (CLOUD '09). IEEE International Conference on.* Bangalore, India: Sept. 2009, pp. 109 –116.

[68] J. Wayne and T. Grance, "Guidelines on security and privacy in public cloud computing," NIST Special Publication, 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-144/SP800-144.pdf.

[69] M. E. J. Newman, "Finding community structure in networks using the eigenvectors of matrices," *Physical Review E*, vol. 74, no. 3, pp. 036 104+, Jul. 2006. [Online]. Available: http://dx.doi.org/10.1103/PhysRevE.74.036104.

[70] M. Tasgin and H. Bingol, "Community detection in complex networks using genetic algorithm," in *ECCS '06: Proc. of the European Conference on Complex Systems.* Oxford, UK: Apr. 2006. [Online]. Available: http://arxiv.org/abs/cond-mat/0604419.

[71] Z. Liu, P. Li, Y. Zheng, and M. Sun, "Community detection by affinity propagation," Tsinghua University, Tech. Rep. 001, 2008.

[72] J. M. Pujol, J. Béjar, and J. Delgado, "Clustering algorithm for determining community structure in large networks," *Phys. Rev. E*, vol. 74, p. 016107, Jul 2006. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.74.016107.

[73] N. Du, B. Wu, X. Pei, B. Wang, and L. Xu, "Community detection in large-scale social networks," in *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 Workshop on Web Mining and Social Network Analysis*. New York, NY, USA: ACM, 2007, pp. 16–25.

[74] I. Gunes and H. Bingol, "Community detection in complex networks using agents," *CoRR*, 2006. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr0610.html#abs-cs-0610129.

[75] A.-L. L. Barabási and E. Bonabeau, "Scale-free networks." *Scientific American*, vol. 288, no. 5, pp. 60–69, May 2003. [Online]. Available: http://view.ncbi.nlm.nih.gov/pubmed/12701331.

[76] A.-L. L. Barabási, "Scale-free networks: A decade and beyond," *Science*, vol. 325, no. 5939, pp. 412–413, 2009.

[77] A.-L. Barabási and A. Réka, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[78] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, pp. 026 113+, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1103/PhysRevE.69.026113.

[79] W. Zachary, "An information flow model for conflict and fission in small groups," *Journal of Anthropological Research*, vol. 33, pp. 452–473, 1977.

[80] D. Lusseau, K. Schneider, O. J. Boisseau, P. Haase, E. Slooten, and S. M. Dawson, "The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations," *Behavioral Ecology and Sociobiology*, vol. 54, no. 4, pp. 396–405, 2003. [Online]. Available: http://dx.doi.org/10.1007/s00265-003-0651-y.

[81] V. Krebs, "Books about us politics," http://networkdata.ics.uci.edu/data.php?d=polbooks [Last accessed: 2012-16-09].

[82] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi, "Defining and identifying communities in networks," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 101, no. 9, pp. 2658–2663, Mar. 2004. [Online]. Available: http://dx.doi.org/10.1073/pnas.0400054101.

[83] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Phys. Rev. E*, vol. 76, p. 036106, Sept. 2007. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.76.036106.

[84] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E*, vol. 70, no. 6, pp. 066 111+, Dec. 2004. [Online]. Available: http://dx.doi.org/10.1103/PhysRevE.70.066111.

[85] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, Jun. 2002. [Online]. Available: http://dx.doi.org/10.1073/pnas.122653799.

[86] X. Liu and T. Murata, "Advanced modularity-specialized label propagation algorithm for detecting communities in networks," *Physica A: Statistical Mechanics and its Applications*, Apr 2009. [Online]. Available: http://arxiv.org/abs/0910.1154.

[87] G. Agarwal and D. Kempe, "Modularity-maximizing graph communities via mathematical programming," *The European Physical Journal B - Condensed Matter and Complex Systems*, Dec 2008. [Online]. Available: http://dx.doi.org/10.1140/epjb/e2008-00425-1.

[88] C. Pizzuti, "A multi-objective genetic algorithm for community detection in networks," in *Proceedings of the 2009 21st IEEE International Conference on Tools with Artificial Intelligence*, ser. ICTAI '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 379–386. [Online]. Available: http://dx.doi.org/10.1109/ICTAI.2009.58.

[89] R. Agrawal, "Bi-objective community detection (bocd) in networks using genetic algorithm," *CoRR*, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1109.html#abs-1109-3650.

[90] A. D. M. C O Dorso, "Community detection in networks," *International Journal of Bifurcation and Chaos*, vol. 20, no. 2, pp. 361–367, 2010. [Online]. Available: http://www.worldscinet.com/ijbc/20/2002/S0218127410025818.html.

[91] Amazon.com. Amazon. http://www.amazon.com [Last accessed: 2013-18-01].

[92] K. Steinhaeuser and N. V. Chawla, "Identifying and evaluating community structure in complex networks," *Pattern Recognition Letters*, vol. 31, no. 5, pp. 413–421, 2010.

[93] M. E. J. Newman, "Fast algorithm for detecting community structure in networks," *Physical Review E*, vol. 69, no. 6, pp. 066 133+, Jun. 2004. [Online]. Available: http://dx.doi.org/10.1103/PhysRevE.69.066133.

[94] J. Duch and A. Arenas, "Community detection in complex networks using extremal optimization," *Phys. Rev. E*, vol. 72, p. 027104, Aug 2005. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.72.027104.

[95] M. E. J. Newman, "Modularity and community structure in networks," *Proceedings of the National Academy of Sciences*, vol. 103, no. 23, pp. 8577–8582, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1073/pnas.0601602103.