A COLLECTIVE APPROACH TO HARNESS IDLE RESOURCES OF END NODES

by

Sachin Goyal

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computing

School of Computing

The University of Utah

August 2011

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Sachin Goyal**

has been approved by the following supervisory committee members:

John B. Carter , Chair **6/15/2010**
Date Approved

Sneha Kumar Kasera , Member **6/15/2010**
Date Approved

John Regehr , Member **6/15/2010**
Date Approved

Gabriel Lozada , Member **6/15/2010**
Date Approved

Ben Zhao , Member **6/15/2010**
Date Approved

and by **Martin Berzins** , Chair of
the Department of **School of Computing**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

We propose a collective approach for harnessing the idle resources (cpu, storage, and bandwidth) of nodes (e.g., home desktops) distributed across the Internet. Instead of a purely peer-to-peer (P2P) approach, we organize participating nodes to act collectively using *collective managers* (CMs). Participating nodes provide idle resources to CMs, which unify these resources to run meaningful distributed services for external clients. We do not assume altruistic users or employ a barter-based incentive model; instead, participating nodes provide resources to CMs for long durations and are compensated in proportion to their contribution.

In this dissertation we discuss the challenges faced by collective systems, present a design that addresses these challenges, and study the effect of selfish nodes. We believe that the collective service model is a useful alternative to the dominant pure P2P and centralized work queue models. It provides more effective utilization of idle resources, has a more meaningful economic model, and is better suited for building legal and commercial distributed services.

We demonstrate the value of our work by building two distributed services using the collective approach. These services are a collective content distribution service and a collective data backup service.

CONTENTS

ABSTRACT . . . . **iii**

LIST OF FIGURES . . . . **viii**

LIST OF TABLES . . . . **x**

CHAPTERS

1. INTRODUCTION **1**

1.1 Design Space 2
1.2 Thesis Statement 5
1.3 Scope 7
1.4 Contributions 8
1.5 Outline 9

2. BACKGROUND AND MOTIVATION **10**

2.1 Problem Context 10
2.1.1 Peer to Peer Services 11
2.1.2 Distributed Compute Intensive Services 11
2.1.3 Utility Computing Systems 12
2.1.4 Grid Computing Systems 13
2.2 Clusters and Cloud Computing 13
2.2.1 Cost Analysis 14
2.2.1.1 Assumptions 14
2.2.1.2 Collective 14
2.2.1.3 Self-Managed Cluster 15
2.2.1.4 Utility Computing Services From Amazon 16
2.2.1.5 Opportunity 16
2.3 Summary 17

3. SYSTEM DESIGN **18**

3.1 Participating Nodes 19
3.2 Collective Manager 21
3.2.1 Service Managers 21
3.2.2 Node Manager 21
3.2.3 Information Aware Scheduler 23
3.2.3.1 History Aware Scheduling 23
3.2.3.2 Reactive Scheduling 24
3.2.3.3 Network Aware Scheduling 24

3.3 Security 25
3.3.1 Identity and Authentication 25
3.3.2 Trust and Access Control at PNs 25
3.3.3 Malicious Behaviors 26
3.4 Incentive Model and Deterring Cheating Behaviors 26
3.4.1 Contribution Accounting and Accountability 27
3.4.1.1 Credits Earned $\propto$ Work Performed 27
3.4.1.2 Accountability 28
3.4.1.3 Offline Cheater Detection 28
3.4.1.4 Collusion 28
3.4.2 Variable Pay Rates (Raises and Cuts) 28
3.4.3 Incentive Model Summary 29
3.5 Summary 30

4. COLLECTIVE CONTENT DISTRIBUTION SERVICE 31

4.1 Design 31
4.1.1 Usage Overview 32
4.1.2 Control Flow 32
4.1.3 Content Caching 33
4.1.4 Authorization 34
4.1.4.1 Nonrepudiation 34
4.1.4.2 Connection flow 35
4.1.5 Digital Rights Mechanisms and CCDS 35
4.2 CCDS Incentive Model 36
4.2.1 Accounting System 36
4.2.1.1 PN's Report 36
4.2.1.2 Content Distributor Report 36
4.2.1.3 CCDS Manager Information 36
4.2.2 Deterring Cheating Behaviors 37
4.2.2.1 Single Client Cheating Behavior 37
4.2.2.2 Single Node Cheating Behavior 37
4.2.2.3 Collusion 37
4.3 Related Work 38
4.4 Summary 39

5. COLLECTIVE BACKUP SERVICE 41

5.1 CBS Design 41
5.1.1 Usage Overview 41
5.1.2 Content Caching 43
5.1.3 Durability 43
5.2 CBS Incentive System 44
5.2.1 Reports and Accounting 45
5.2.1.1 PN's Report 45
5.2.1.2 CBS Manager Report 45
5.2.2 Deterring Cheating Behaviors 45

5.2.3 Cheating Analysis 46
5.2.3.1 Single Node Cheating Behavior 46
5.2.3.2 Single Client Cheating Behavior 47
5.2.3.3 Collusion 47

6. ECONOMIC ANALYSIS 49

6.1 Analytical Model of the Incentive System 49
6.1.1 Perfect Monitoring 51
6.1.1.1 Cost of Sharing 52
6.1.2 Imperfect Monitoring 52
6.1.3 Variable Pay Rates 54
6.2 Evaluating the Incentive Model 56
6.2.1 Short-Lived vs Long-Lived Nodes 56
6.2.2 Deterring Cheating Behavior 56
6.2.2.1 Worst Case Analysis 61
6.2.2.2 System Tuning 64
6.3 Related Work 64
6.4 Summary 65

7. INFORMATION AWARE SCHEDULING 66

7.1 Overview 67
7.2 Simulation Environment 69
7.2.1 Performance metrics 69
7.2.1.1 Availability 70
7.2.1.2 Available Bandwidth 70
7.3 History Aware Scheduling 70
7.3.1 Availability Aware Scheduling 72
7.3.2 Bandwidth Aware Scheduling 75
7.3.2.1 Multiple Tasks on a Node 76
7.3.3 Shift Work Scheduling 82
7.4 Reactive Scheduling 85
7.4.1 Failure-Aware Scheduling 85
7.4.2 Demand-aware Scheduling 86
7.5 Network-Aware Scheduling 89
7.6 Effective Utilization of PN Resources 90
7.6.1 Effective Scheduling of Services 90
7.6.2 VM Control for Effective Utilization 92
7.7 Summary 92

8. IMPLEMENTATION 94

8.1 Players in a Collective 94
8.2 Authentication and Interplayer Communication 95
8.3 Partners and Clients 96
8.4 Collective Manager 97
8.4.1 Data Layer and Intercomponent Communication 98

8.4.2 Resource Scheduler 99
8.4.3 Service Managers 99
8.4.4 Support Components 100
8.5 Participating Nodes 101
8.6 Validation on Emulab 104
8.7 Scalability 107
8.8 Summary 111

9. OTHER SERVICES 112

9.1 Collective Compute Service 112
9.2 Collective Network Probing Service 113
9.2.1 Service Design 114
9.2.1.1 Deterring Cheating Behaviors 114
9.3 Collective WiFi Service 115
9.3.1 Service Design 115
9.3.1.1 Startup and Session Establishment 115
9.3.1.2 Security and Authentication 116
9.3.1.3 Accounting 116
9.3.1.4 Sharing Implications 117
9.4 Collective Surrogate Service for Resource-Constrained Devices 117
9.4.1 Design 120
9.4.1.1 Session Management 121
9.4.1.2 Programming Interface 121
9.4.2 Evaluation 122
9.4.2.1 Experimental Setup 122
9.4.2.2 Sphinx Speech Recognition 123
9.4.2.3 Web Services (Data Mining) 126
9.4.3 Summary 127

10. CONCLUSION 129

REFERENCES 131

LIST OF FIGURES

1.1 Collective Content Distribution Service 3

3.1 Main Players in Collective 19

3.2 System Architecture 22

4.1 Collective Content Distribution Service 32

5.1 Collective Backup Service 42

6.1 Expected Pay vs Probability of Being Caught 54

6.2 Short-Lived vs Long-Lived Player 57

6.3 Expected Value of D for Different Number of Offenses 60

7.1 Online Nodes in the System vs Time (Microsoft Trace) 71

7.2 Online Nodes in the System vs Time (Skype Trace) 72

7.3 Data Availability: Random vs Smart Selection (Microsoft Trace) 74

7.4 Data Availability: Random vs Informed Selection (Skype Trace) 75

7.5 Data Availability: Random vs Smart BW Selection (Microsoft Trace) . 77

7.6 Average Available bandwidth: Random vs Smart BW Selection (Microsoft Trace) 78

7.7 Skype Trace: Random vs Smart BW Selection 79

7.8 Multiple Objects on a Node: Average Bandwidth per Object 80

7.9 Multiple Objects on a Node: Performance with Different Shares 81

7.10 Shift Work Strategy: Availability of a 4-way Replicated Chunk (Microsoft Trace) 83

7.11 Availability with Additional Shift Work strategy (Microsoft Trace) 84

7.12 Availability Impact of Failure-Aware Scheduling (Microsoft Trace) 86

7.13 Availability Impact of Failure-Aware Scheduling (Skype Trace) 87

7.14 Demand-Aware Scheduling: Average Available Bandwidth w.r.t. Time 88

7.15 Resource Utilization: Multiple Objects vs Single Object on a Node . . . 91

8.1 Players in a Collective System 94

8.2 Collective Manager Components 97

8.3 Participating Node Architecture 102

8.4 Cumulative Data Download vs Time for a Client 105

8.5 Request Completion Time vs Request Number 106

8.6 Demand-Aware Scheduling: Request Completion Time vs Request Number 108

9.1 A Surrogate Computing Scenario: Remote Speech Recognition 118

9.2 Synthetic Surrogate Usage Benchmark: Client Current Consumption . . 128

LIST OF TABLES

6.1 Payoffs for perfect monitoring case 51

6.2 Payoffs for perfect monitoring case with cost of sharing included 52

6.3 Payoffs for imperfect monitoring case 52

6.4 Payoffs for imperfect monitoring with penalties 53

6.5 Glossary of mathematical symbols used 62

9.1 Average response time for instantiating `sphinx2` speech recognition engine. Standard deviations are in parentheses. 125

9.2 Sphinx2 speech recognition on the Zaurus. Standard deviations are in parentheses. 125

9.3 Synthetic benchmark results. Standard deviations are in parentheses. . . 127

CHAPTER 1

INTRODUCTION

Modern computers are becoming progressively more powerful with ever improving processing, storage, and networking capabilities. Typical desktop systems have more computing/communication resources than most users need and are underutilized most of the time. These underutilized resources provide an interesting platform for new distributed applications and services.

We envision a future where the idle compute, storage, and networking resources of cheap network-connected computers distributed around the world are harnessed to build meaningful distributed services. Many others have espoused a similar vision. A variety of popular peer-to-peer (P2P) services exploit the resources of their peers to implement specific functionality, e.g., Kaaza [51], BitTorrent [20], and Skype [39]. The large body of work on distributed hash tables (DHTs) exploits peer resources to support DHTs (e.g., Chord [47], Pastry [76], Tapestry [96], and CAN [72]), on top of which a variety of services have been built. SETI@home [78] and Entropia [15] farm out compute-intensive tasks from a central server to participating nodes.

We propose a new way to harness idle resources as managed "collectives". Rather than a purely P2P solution, we introduce the notion of *collective managers* (CMs) that manage the resources of large pools of untrusted, selfish, and unreliable *participating nodes* (PN). PNs contact CMs to make their resources available, in return for which they expect to receive compensation. After registering with a CM, each PN runs a virtual machine (VM) image provided by the CM. CMs remotely control these VMs and use their processing, storage, and network resources to build distributed services. Unlike projects like Xenoservers [73], *a CM does not provide raw access to end node's resources to external customers.* A CM is an application service provider, aggregating idle resources to provide services like content distribution and remote backup. Figure 1.1 illustrates a possible use of a collective to implement a commercial content distribution service that sells large content files (e.g., movies, music, or software updates) to thousands of clients in a cost-effective way.

The collective uses an economic model based on currency. A collective manager earns money in return for providing services and then shares its profits with PNs in proportion to their contribution towards different services. The basic unit of compensation is a CM-specific credit that acts as a kind of currency. Users can convert credits to cash or use them to buy services from the CM or associated partners.

Since collectives may include selfish nodes, it is important to mitigate the negative effects of selfishness. Selfish nodes can resort to cheating for earning more than their fair share of compensation. Cheating behavior has been observed extensively in distributed systems, e.g., free riding in Gnutella [1] and software modifications to get more credit than earned in SETI@home [48]. To mitigate negative impact of selfishness, we propose to employ economic deterrents comprised of an offline data-analysis-based accounting system and a consistency-based incentive model. Services are designed specifically to facilitate these economic deterrents and use security mechanisms to ensure accountability across different transactions. While we cannot prevent users from cheating, our mechanisms mitigate cheating behavior by making it economically unattractive.

The collective system as a whole is a collection of distributed services built by aggregating the idle resources provided by willing end-nodes in return for compensation. It provides a simple but efficient way to harness idle resources distributed across the Internet.

1.1 Design Space

Our target environment consists of potentially millions of end-nodes distributed all across the Internet. They are untrusted, and unreliable - i.e., they suffer from frequent node churning as well as failures. Each node can exhibit selfish behaviors. Our goal is to use unutilized resources of these end-nodes to build meaningful *commercial services.* Our design is similar to a firm in traditional economic systems, where multiple people

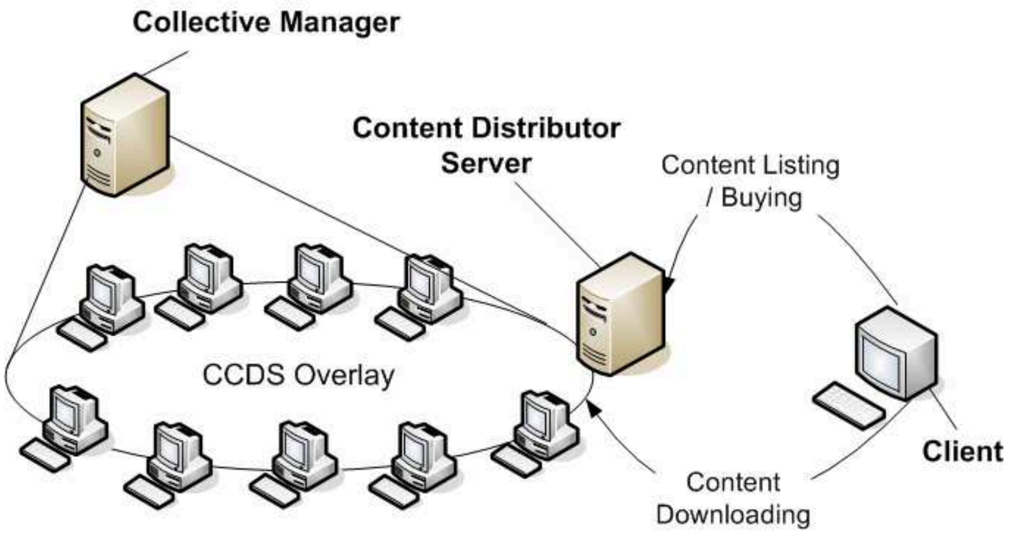

Figure 1.1: Collective Content Distribution Service

come together to work for a common goal. Our incentive and interaction model is neither based on altruism, nor on the bartering.

A system based on altruism relies on *altruistic* users that provide services without any incentive, or provide more than the required service to others. For example, in the context of file sharing P2P systems, altruistic users share the files on their computer even though they do not get anything in return. They remain in the system after downloading a file and upload the data to other users even when they are not required to do so. Systems like gnutella and Kazaa were based on altruism. These systems eventually suffer from freeloading, which degrades the quality of the network substantially. More importantly, these systems do not have an incentive model and thus are more useful for free or illegal content distribution instead of a commercial system.

Bartering is the exchange of goods or services between two people in a mutually beneficial way. One can use bartering based mechanisms to provide incentives for interaction. We categorize bartering systems into two categories - lazy bartering and active (currency-based) bartering. We use the term *"lazy bartering"* for systems where people participate in the system only long enough to perform a particular transaction such as downloading a song. As stated above, this leads to the underutilization of a node's resources unless the node's administrator is altruistic. BitTorrent is based on *lazy bartering*. To overcome this underutilization problem, one may try *active bartering*, that is using bartering to maximally utilize available resources (not just when a node needs some service in return). Using currencies as an intermediary in bartering provides the necessary tool for achieving that. It is still difficult, however, to find eligible users with whom to barter. One is limited by the goods (e.g., music/movies) that one has for sharing and by the availability of other users interested in paying currency for the objects one owns. The inherent untrusted and unreliable nature of end nodes makes currency-based bartering risky. For example, it is easy for a hacker to use a currency-based bartering mechanism to acquire resources at multiple nodes, and then use them for network attacks. Also, individual nodes can provide only a limited amount of resources, so anyone interested in building a service like a content distribution system will have to find and make bartering agreements with many end nodes. This process has to be repeated whenever a new service is built.

The collective model provides a useful approach to handle these problems in a simple but realistic manner. In a collective, a user provide his/her resources to a collective manager and gets paid in return. It is the responsibility of a collective manager to use those resources to build distributed services and to deal with customers and clients. One way to think about this is by comparing it to a person having $10000 of savings. He/she can either deposit their savings in a bank and get paid interest (a la collective), or he/she can lend it to other people, potentially getting a higher rate of return than provided by banks. Like bartering in P2P, personal lending suffers from a trust problem; what if borrower does not return your money? Additionally one has to search for potential borrowers and the savings remain unutilized during the search period.

In a collective, a participating node shares its resources using a VM instance and provides root access to that instance to the CM. Only the trusted collective manager has direct access to the VM running on a participating node, so we do not need to worry about an unknown party using the nodes for nefarious purposes, e.g., to launch network attacks. Second, the collective system uses the long-term association with nodes and the presence of a CM to counteract the untrusted and unreliable nature of PNs. A collective manager provides a simple service model to potential partners/customers, who do not have to worry about inherent chaos of a system built out of end nodes.

We can compare a collective to the formation of organizations in real life. As human civilization has progressed, there has been a clear move towards forming organizations - whether it is universities, banks, manufacturing plants, or other commercial organizations. We still have freelancers, but the majority of productive work is performed by well-defined organizations.

In some ways, a collective resembles the collections of zombie machines often used by malicious users to launch distributed denial of service attacks. We differ from "zombie nets" in that we only use the resources of willing participants and allow PNs to limit how their resources are used (e.g., no more than X kilobytes-per-second of network bandwidth).

1.2 Thesis Statement

In this dissertation we will demonstrate that:

Using the collective approach, the idle compute, storage and networking resources of myriad untrusted and unreliable computers distributed across the Internet can be harnessed effectively to build a set of useful distributed services with predictable performance, and with a practical incentive model even in presence of selfish behaviors.

By "*practical incentive model*" we mean that the collective system does not rely on altruism to sustain the system. The system provides incentives for nodes to participate in the system for extended durations. Also, all services built on collective are designed to provide incentives for the actual work performed and to handle selfish behavior by the participating nodes.

By "*harnessing effectively*" we mean that the collective system provides good utilization of a node's perishable idle resources. For example, in an incentive model that employs bartering, e.g., BitTorrent, nodes typically participate in the system only long enough to perform a particular transaction such as downloading a song. At other times, that node's idle resources are not utilized. In contrast, the collective system tries to consume as much resources as possible by scheduling diverse pools of work on a node. Using idle resources to run arbitrary services, rather than only services that the local user uses, improves resource utilization.

By "*predictable performance*" we mean that the collective manager can probabilistically understand the demand as well as current capabilities of the collective system. That is, a collective manager can dynamically take actions to minimize the impact of varying client demand, as well as inherent unreliability of a system built out of end-nodes. To achieve this, the collective system tracks all sort of past as well as live information, e.g., participating nodes' resource profiles, their past performance, service demand rates, dishonest behaviors, etc. These diverse information are then used to make informed scheduling decisions, i.e., the system is tuned according to its own inherent characteristics.

To support this hypothesis, we built a collective system and developed two distributed services: a collective content distribution service and a collective backup service.

The first service, collective content distribution service (CCDS), is a way to distribute big content files (e.g., movie videos, music, software updates, etc.) to thousands of clients in a cost-effective way. Instead of typical web-server style downloading of paid content, content distributors can hire the CCDS service to distribute content on their behalf. It is quite different from peer to peer, as clients directly download from the overlay without sharing anything. Recently there has been an explosion in legal content distribution over the Internet, e.g., the highly successful iPod and iTunes services. Many prognosticators predict that the Internet will soon become the predominant means for audio and video distribution. The typical infrastructure required to support such distribution (e.g., Apple's iTune music server) is a large dedicated clustered web service with immense external bandwidth. A system that can handle millions of multi-gigabyte (video) downloads per day requires tremendous infrastructure and administrative costs. Demand for these kinds of services is clearly increasing, and our collective model can support them effectively.

Our second service is a collective backup service (CBS), which is a paid backup system that allows clients to store and retrieve data files over the collective system. All data are encrypted at the client computer before being submitted to the CBS.

As in CCDS, from the client perspective it is a client-server solution. Clients do not have to be PNs to avail themselves of the CBS service. Rather, clients use a small application to store and retrieve data and are charged based on usage. Alternatively, clients could "pay" using credits they receive for becoming a PN in the collective system. Online backup is becoming an important service for millions of home and small business communities that do not want to spend money on professional in-house backup services.

Apart from these two services, we envision many more services that can be built on top of the collective overlay. We discuss some of these services, e.g, a collective compute service (CCS) and a collective network probing service (CNPS) in Chapter 9.

1.3 Scope

As part of this dissertation, we design, build and evaluate an infrastructure that harnesses the idle resources of end nodes distributed across the Internet. We are not proposing a generic library that can be used to compose multiple services.

Not all services can be easily supported on top of a collective system. To map a service effectively to a collective, the service should be decomposable into multiple components; it should have a mechanism to handle the failure of individual components; it should be possible to check the correctness of the service delivered by components; and it should be possible to design a selfish behavior protection model for the service.

Some of the basic mechanisms to compensate for the unreliable and untrusted nature of participating nodes can be utilized across services, and thus can be developed as a library. This, however, is not the main focus of our work.

Both the clients and participating nodes in our system are untrusted. They are assumed to be selfish nodes that acts to maximize their utility in a rational manner. Selfish nodes try to get credits without performing the required work or try to get more than their fair share of credits. Multiple selfish nodes (both clients and PNs) can collude with each other to increase their utility even further. We assume that a single colluding group consists of only a small percentage of the overall nodes in the system.

We assume all communication channels between nodes and collective manager are asynchronous, but resilient, i.e., they deliver correct data after a finite amount of time.

1.4 Contributions

Our first contribution is a practical incentive system that motivates participating nodes to participate consistently and stay in the system for long durations. We use an incentive model based on currency that rewards actual work performed, as well as the consistency of the work. That is, a node's long-term associations and continued performance lead to improved incentives for the node. We examine the impact of decisions made by selfish nodes and analyze the gain vs loss possibilities for participating nodes as we vary the likelihood of cheaters being caught. We show that while we cannot prevent users from cheating, our mechanisms mitigate cheating behavior by making it economically unattractive. We show that a small probability of catching cheaters (under 4%) is sufficient for creating a successful deterrence against cheating. We further show that our incentive system can be used successfully to motivate nodes to remain in the system for prolonged durations.

Our second contribution is a collection of security and economic mechanisms designed to overcome cheating in a collective content distribution service (CCDS) and in a collective backup service (CBS). These security and economic mechanisms along with a offline accounting system provide successful deterrents against cheating. We not only handle individual cheating behaviors, but also consider collusion among participating nodes and even clients.

Our third contribution is the use of information-aware scheduling to handle the inherent unreliability and heterogeneity of the end-nodes. That is, a collective manager tracks significant historic and live information, e.g., participating nodes' resource profiles, their past performance, service demand rates etc. This diverse information is then used to make informed scheduling decisions, e.g., creating additional replicas for a content dynamically in response to an increased demand or deciding the replication degree needed for the remote backup service based on past records of node failures.

This information-aware and adaptive scheduling provides us with the tools necessary to overcome the chaotic nature of end-nodes' ability and availability.

1.5 Outline

This dissertation is organized into 10 chapters. In Chapter 2, we discuss the problem context and provide a simple cost analysis of a collective system. In Chapter 3, we describe the architecture of a collective system, its security infrastructure, and its incentive model. In Chapter 4, we discuss the design of the collective content distribution service (CCDS) and its incentive model. We consider different type of cheating behaviors and show how our mechanisms mitigate cheating behavior by making it economically unattractive. In Chapter 5, we discuss the design of the collective backup service (CBS) and its incentive model. In Chapter 6, we evaluate our incentive model from an economic standpoint. We demonstrate how our incentive system motivates nodes to stay in the system for prolonged duration and deters cheating behaviors. Next we describe and analyze our information aware scheduling system in Chapter 7. We show that the underlying diversity of end-nodes can be exploited in a simple but useful manner to improve service performances. We then discuss implementation specific details and scalability issues in Chapter 8. In Chapter 9, we discuss other possible services that can be built on top of a collective overlay. Finally we summarize this dissertation in Chapter 10.

CHAPTER 2

BACKGROUND AND MOTIVATION

Modern computers have become quite powerful over the years, and typically have more processing, storage, and communication resources than most user need, and remain underutilized. A verification of this can be seen from the success of systems such as seti@home, Gnutella, Kazaa, and BitTorrent. Based on data from Jan 2004 to June 2004, CacheLogic reported that peer-to-peer systems (P2P) consume 80% or more of the traffic for last mile service providers [14]. Another study from CacheLogic put the P2P percentage of Internet traffic at about 60% at the end of 2004 [13]. The same study reports that BitTorrent was responsible for as much as 30% of all Internet traffic at the end of 2004 [13].

Our collective system intends to harness these unutilized resources of already deployed computers. Our target nodes can be either home desktops or computers deployed in communities or enterprise environments. These nodes are bought and deployed to serve some specific purpose, but their resources are not utilized 100% all the time. The goal of the collective system is to harness these unutilized resources to build commercial services, and distribute the profits back to participating nodes.

2.1 Problem Context

Our collective model, while similar in certain aspects to previous work, differs in a number of important ways. There are four main domains of related projects that also deal with utilizing the resources of computers distributed across the Internet. The first is peer-to-peer systems such as bittorrent [20], gnutella [31], etc. The second is compute-only services like seti@home [78], entropia [15], etc. The third is utility computing systems like Xenoservers [73]. The fourth is grid computing systems [28].

2.1.1 Peer to Peer Services

Unlike typical P2P systems, we do not assume that PNs are altruistic (e.g., Kazaa [51] or Gnutella [31]) or willing to "barter" their resources in return for access to the end service (e.g., BitTorrent [20]). Rather, PNs make their resources available to CMs to build distributed services, in return for which they are compensated by CMs.

Using idle resources to run *arbitrary* services, rather than only services that the local user uses, improves resource utilization. A node's compute and network resources are perishable — if they are not used, their potential value is lost. In an incentive model that employs bartering, e.g., BitTorrent, nodes typically participate in the system only long enough to perform a particular transaction such as downloading a song. At other times, that node's idle resources are not utilized unless the node's administrator is altruistic. In the collective, a CM will have much larger and more diverse pools of work than personal needs of individual participants; thus a CM will be better able to consume the perishable resources of PNs. PNs, in turn, will accumulate credits for their work, which they can use in the future however they wish (e.g., for cash or for access to services provided by the CM). In a sense, we are moving the incentive model from a primitive barter model to a more modern currency model.

Additionally in a collective the CM has direct control over the VMs running in participating nodes. This control can be utilized to provide a predictable service to the customers, e.g., the CM can dynamically change the caching patterns in response to sudden demand.

2.1.2 Distributed Compute Intensive Services

Unlike seti@home [78] and Entropia [15], the idle storage and networking resources of PNs can be harnessed, in addition to idle processing resources. As a result, collectives can be used to implement distributed services (e.g., content distribution or remote backup) in addition to compute-intensive services. These services have different design challenges than compute intensive services.

First, seti@home or other compute-only services are embarrassingly parallel and does not require any interaction between different nodes. Services such as content distribution or backup services require cooperation from multiple nodes to successfully cache/replicate a piece of content, and to provide service to the customers. Handling these interactions (e.g., multiple node collusion) while still being able to manage selfish behaviors is a much different and tougher problem than handling embarrassingly parallel applications.

Second, applications like content distribution or remote backup require timely delivery of service to customers – thus adding a real time response component. There are no similar real-time requirements in seti@home-like applications.

Third, applications like content distribution or remote backup require different mechanisms and design to detect selfish behaviors by participating nodes.

2.1.3 Utility Computing Systems

Utility computing systems like Xenoservers [73], Planetlab [70], and SHARP [29] deal with similar problems, but these systems handle resource sharing at the granularity of VMs, and are not bothered about the design and challenges of building a service using those resources.

For example, unlike collectives, these systems do not provide solutions for service level selfish behaviors by the participating nodes (or sites). Many of these assume trusted nodes. Projects like SHARP [29] assume that the service managers have some external means to verify that each site provides contracted resources and that they function correctly (assumption 7 in their paper [29]).

The focus of these projects are dedicated powerful servers of high reliability. While in collective, our main focus is to harness the idle resources of unreliable end-nodes.

Similar to these projects, PNs in a collective exploit VM technology for safely running arbitrary untrusted code provided by CMs. Unlike utility computing projects such as Xenoservers or SHARP, however, we do not provide raw VMs to external clients. We allocate only one VM on a node, and run multiple services inside that one VM. In other systems any untrusted third party can acquire VMs and potentially use them for nefarious activities like network attacks. In contrast, our one VM per participating node is only controlled by the trusted collective manager.

2.1.4 Grid Computing Systems

Systems like Condor [60] manage the idle resources of collections of nodes, but typically manage the resources of *trusted* nodes that remain in the "pool" for substantial periods of time. In contrast, we assume that PNs are unreliable (frequently fail or leave the collective) and are nonaltruistic (will do as little productive work as possible while maximizing their compensation).

Systems like computational grids [28] also deal with distributed resources at multiple sites, though again their main focus is on trusted and dedicated servers.

2.2 Clusters and Cloud Computing

Instead of using unutilized resources of end-nodes distributed across the Internet, one can potentially use a cluster of PCs to provide similar services. Cloud Computing Systems like Amazon Web Services [3] (AWS) provide another alternative where users can rent resources on a cluster and pay based on actual usage of resources.

In contrast to dedicated resources of a cluster, the focus of a collective is the unutilized resources of nodes distributed across the Internet. That is, it is trying to monetize resources (cycles, storage, and network) that have already been paid for to support something else (e.g., I already own a PC so I can surf the net; business already have machines that they use to run their business).

Secondly, a collective is a self-upgradable system. End-nodes are upgraded in due time by their users, thus it gets the advantage of technology advancements for free. It does not require huge amount of initial investment that is needed for a setting up a big cluster.

Thirdly, nodes in a collective are geographically dispersed across the Internet. This geographic dispersion is inherently useful for some apps, e.g., one can utilize the geographic dispersion of nodes to build a distributed network probing service. Such a service can be helpful in debugging of network routing and DNS problems.

From research perspective, a collective has different set of challenges due to (i) unreliable end nodes, (ii) untrusted and selfish end nodes, (iii) geographically distributed end-nodes instead of a local network in a cluster, and (iv) limited resources per node. To understand the business implications though, we do a quick cost analysis to compare a collective against a cluster and a cloud computing approach.

2.2.1 Cost Analysis

In this subsection, we do a quick quantitative analysis to understand the opportunity cost of a collective in comparison to the cluster and cloud computing approach. We first estimate the resource capabilities of a collective system consisting of one million nodes. We then use those estimates to calculate the potential cost for building an equivalent cluster using a self-managed and a cloud computing approach.

2.2.1.1 Assumptions

In this analysis, we assume that 10% resources of a given node are available when the end-user is actively using the node; 100% resources are available otherwise. We assume that on average a node is used actively for 10 hours per day by the end-user. We further assume an average upload bandwidth of 50 KBps (i.e., what Comcast cable Internet provides currently to non-commercial customers), which is quite conservative considering that other broadband options like DSL, VVD Communication [89] or Utopia [86] provide better bandwidth, and countries like South Korea and Japan have broadband connections providing Mbps of upload bandwidth. We assume that each node in the collective contributes on average 5GB of storage, which is a quite conservative estimate considering the sizes of modern hard disks. For processing capabilities, we assume a 2GHz processor with 1GB of RAM.

2.2.1.2 Collective

A million nodes with 50 KBps of upload bandwidth means an available aggregate upload bandwidth of 50 GBps. Since we assume that on average only 10% of each node's capacity is available for 10 hours each day, we can probabilistically estimate the available upload bandwidth as $(5*10+50*14)/24$, i.e., 31.25 GBps at any instant, although not constant. Similarly we can estimate that at any instant this collective will have processing capabilities worth $(0.2*10+2*14)/24*10^6 = 1.25 * 10^6$ GHz. For storage, disk space remains available all the time irrespective of the node's use by the end-user or not. Thus we estimate $5*10^6$GB of available storage.

Typically there will be an overhead in terms of bandwidth and storage to maintain service level properties. For example, bandwidth will be used to maintain caching in a content distribution system. Similarly there will be storage overhead to maintain durability (using either straightforward replication or erasure coding). Assuming 10% overhead for bandwidth, we are left with 28.125 GBps of bandwidth. Assuming 80% overhead for storage (required for four extra replicas), we are left with 10^6 GB of storage available.

2.2.1.3 Self-Managed Cluster

Assuming dual core 2*3GHz machines, we can build a comparable computing cluster using $(1.25 * 10^6)/6) = 200$K such machines. So let us use 100K machines as a conservative estimate for our analysis here. Assuming each machine costs around $1000 including storage, the initial investments for such a cluster will be around $100 million. Plus we will need to upgrade such cluster periodically to keep up with technology advancements. Assuming a 5-year life cycle, we need to depreciate it at $40M per year, i.e, $1.66M per month.

To estimate data center and bandwidth costs, we use the colocation costs advertised on websites like *creativedata.net*, *colocation.com*, and *apvio.net* as a guide. For bandwidth, the typical costs advertised on were around $45 per Mbps per month. Using that estimate, for 28.125 GBps we require $28.125 * 8 * 1000 * 45 = \$10.125$ million per month. Typical cabinet prices for these datacenters start at $650 for 40 units. Using this as an estimate, we require $50 * 1000 * 650/40 = \$0.8125$ million per month. Adding these two costs and ignoring setup fees and related costs, we require $10.93 million per month to operate.

As reflected in some of the TCO (total cost of ownership) studies available on Internet, administrative personal costs are one of the biggest part of overall cost of managing a cluster. For example, a TCO study by the hostbasket web hosting company [84] puts the labor cost at 54% of total cost, while Aruba.it [83] puts the labor cost at 41% of the total cost. We do not have any good way to quantify our costs here, so we use a conservatively estimate of $3 million per month for labor costs.

Thus overall we will need around $15.5 M per month to build a cluster equivalent to a million node collective.

2.2.1.4 Utility Computing Services From Amazon

Amazon web services provides cloud computing services for computing, storage, and bandwidth through Simple Storage Service (S3) and Edge Computing Service (EC2). EC2 provides an instance of 1.7Ghz x86 processor, 1.75GB of RAM, 160GB of local disk, and 250Mb/s of network bandwidth. It has the following prices: $0.10 per instance-hour consumed, $0.20 per GB of data transferred into/out of Amazon (i.e., Internet traffic). S3 has the following pricing: $0.15 per GB-Month of storage used and $0.20 per GB of data transferred.

Considering the computing resources of 1.25 * 10^6 GHz, we will require around 735K instances. Let us take a conservative estimate of 300K instances for 24x30 hours; it will cost $(300 * 1000 * 0.1 * 24 * 30) = 21.6$ million dollars per month. These machines will have enough storage space available for matching $5 * 10^6$ GB storage of collective. If we use S3 for storage, we will require 0.15 million dollars per month. For bandwidth, costs are the same for both EC2 and S3. Based on 28.125GB/s, we will require 14.58 million dollars per month for bandwidth. This is a very conservative estimate, as Amazon does not promise a bandwidth of 28.125 GBps with that pricing.

Thus overall we will require around $36M per month to have a system comparable to a million node collective.

2.2.1.5 Opportunity

Now that we have a quantitative idea of the cost of a collective-equivalent cluster, we can use that to get an estimate of potential rewards possible for participating nodes. So from the raw resource point of view, a collective can provide around $15 to $30 per month to participating nodes having resources as defined in the above assumptions. From a service point of view, services built on collective overlay may be worth much more than raw resources, and hence it may be possible to give even better returns. Additionally a collective can provide return services to the PNs in addition to direct cash payout. This can increase the profit margins even more.

2.3 Summary

In this chapter we have discussed the usefulness of a collective system in comparison to existing systems. The collective model is a better alternative to the P2P (e.g., Kazaa, BitTorrent) and centralized work queue models (e.g, Seti@Home) for utilizing the idle resources of end-nodes. The collective system has a meaningful economic model, can effectively utilize idle resources of end-nodes, and is better suited for building legal and commercial distributed services.

On the business side, clusters and cloud computing systems are the main competitors for building commercial services similar to the collective. We have used a simple cost analysis to show that the collective remains a viable alternative from a price point of view. The value of collective will improve further as end-nodes capabilities (especially bandwidth) will increase substantially over the years. Additionally the collective system has inherent advantages due to geographic distribution and self-upgradability of participating nodes.

CHAPTER 3

SYSTEM DESIGN

The collective system as a whole is a collection of distributed services built by aggregating the idle resources provided by willing end-nodes in return for compensation. Figure 3.1 illustrates the main players in a collective. These are:

1. **Participating Nodes (PNs)** are end nodes that have idle compute, storage, or network resources. They are typically connected to the Internet through some sort of broadband service. These nodes have different compute/communication capabilities, go up and down in unpredictable ways, and are inherently untrusted.

2. **Collective Managers (CMs)** are service providers to whom individual nodes provide their idle resources. A CM uses these resources to provide a set of meaningful services to clients and then compensates the PNs. Multiple competing CMs can coexist, each providing different services and/or pricing models.

3. **Partners** are commercial collaborators of a CM that use the collective overlay to offload part of their tasks, e.g., an online movie distribution company can utilize the collective service for movie distribution, while managing the content acquisition and sales itself.

4. **Clients** are individuals that wish to utilize the services offered by CMs, e.g., using a collective remote backup service or downloading a video from the collective content distribution service.

As an example, let us consider the collective content distribution service (CCDS). Here a content distribution partner collaborates with the CM to provide a content distribution service. The content distributor interfaces with the collective manager to

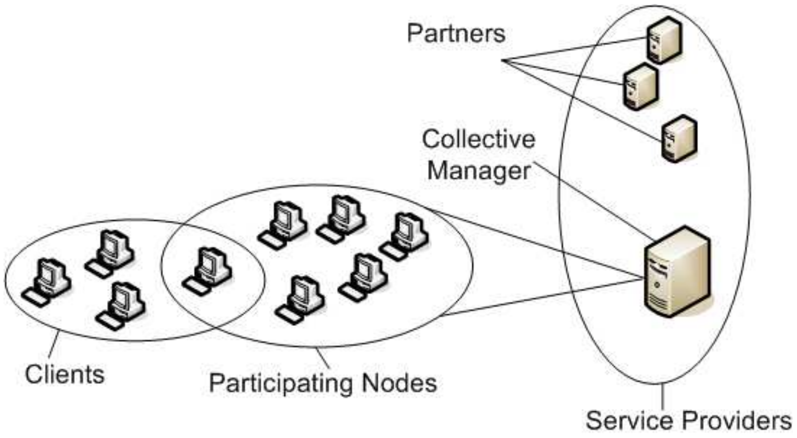

Figure 3.1: Main Players in Collective

distribute desired content. The collective manager divides the content into multiple chunks and proactively caches them across multiple PNs.

Clients run an application, e.g., an iTunes content download application, that interacts directly with the content distributor for browsing and sales. After a client purchases a particular song or video, the content distributor sends it a signed certificate that gives the client the right to download the song/video from the CCDS overlay network and a contact list of PNs. The client then downloads the content directly from PNs, with different chunks coming from different nodes.

This chapter presents the overall design of a collective system. We start by discussing the participating node architecture in Section 3.1 and then describe the design of the collective manager in Section 3.2. We then present the security related details in Section 3.3. Next we describe the collective incentive model in Section 3.4 and then summarize in Section 3.5.

3.1 Participating Nodes

To provide resources to a CM, a PN runs a VM instance and provides root access to that instance to the CM. The decision to use a virtual machine instance as the unit of resource allocation has several important advantages over alternative approaches. Virtual machine technology provides greater *isolation*, *flexibility*, and *resource control* than simply running application processes directly on top of a standard Unix or Windows box.

In terms of isolation, applications running on a virtual machine instance cannot directly interfere with applications running on the host machine, nor can they access resources (e.g., the file system) reserved for the host machine. VM technology makes it effectively impossible for rogue client software to access resources to which it does not have access rights, install viruses, or crack the root.

The virtual machine monitor can enforce resource controls (e.g., disk quota, cpu share, and physical memory allocation) on a per-VM basis. This design allows normal work to proceed on the host machine without undue impact by applications running on a VM. The protection and resource controls provided by VMs will make people more willing to make their machines accessible to a collective overlay, without fear that they will be misused or infected.

Using VM technology also provides advantages to a collective manager. A collective manager has `root` access and can install and execute arbitrary software on participating nodes. This design provides tremendous flexibility – what a collective manager can do is not limited by what some middleware layer supports. Programmers can use different middleware layers such as CORBA, RPC, or SOAP based on their needs, which enables our system to support a wide variety of distributed services and applications. For our prototype, we use the free VMware Player [88] and Xen [6] for the VM layer.

In addition to the VM, each PN runs a small *node-agent*. The *node-agent* handles the basic interaction between the user and CM, e.g., to determine when the node has sufficient idle resources to warrant joining the CM's collective or to start/stop the VM. The *node-agent* provides a simple UI through which the user can set limits on the resources provided to the collective (e.g., limits on disk/memory space or limits on network bandwidth that may vary depending on the time of day). The *node-agent* also provides a way for the host to temporarily suspend/resume the collective's VM.

3.2 Collective Manager

A collective manager is the core hub of a collective and is overall responsible for smooth functioning of the system. Figure 3.2 shows the high level architecture of a collective. There are three main components of a collective manager - *service managers*, *node manager*, and a *scheduler*. *Service managers* are per service agents that manage service-specific activities in the system. The *node manager* tracks the set of PNs currently registered with the CM and is also responsible for uploading of required executables and libraries to PNs. The *scheduler* helps schedule the resources on individual PNs with active collaboration with service managers.

3.2.1 Service Managers

A service built on a collective consists of a central component that runs collocated on the CM, called *service manager*. A service manager is responsible for overall performance of a a service in the collective and is designed based on service-specific requirements.

A service manager regularly interacts with its corresponding service partner (e.g., content distributor for the CCDS service) to get the list of tasks that need to be offloaded to the collective overlay. It then converts these tasks into small components and distributes these components to a set of PNs with the help of the scheduler. It also continuously monitors the service performance, validates it against service goals, and takes actions to add or remove more resources as and whenever needed.

3.2.2 Node Manager

The node manager tracks the set of PNs currently registered with the CM. It is also responsible for shipping of necessary executables and data to the PNs based on service manager requests.

An important part of node manager is liveness server that keeps track of nodes' liveness in the overlay and tracks node failures. It uses multiple methods to detect node failures (or churn) in a timely fashion. These methods can be divided into two main categories:

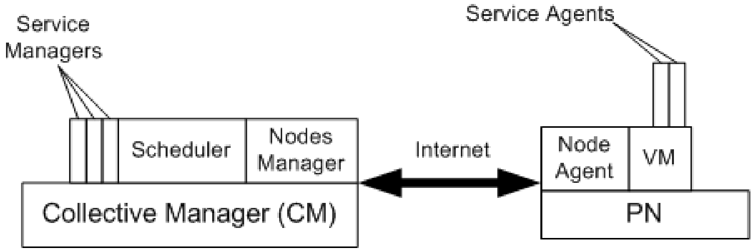

Figure 3.2: System Architecture

- **Centralized tracking mechanism**: Every node sends a join/leave message to a centralized liveness server whenever it joins the collective or shuts down gracefully.

- **Application-level alertness mechanism**: The CM, other PNs, and clients regularly contact other nodes as part of normal service operations. If they are unable to contact a node, they inform the liveness server.

In particular, whenever a participating node joins/rejoins the collective, it pings the *node manager* with an 'I-came-online' message. On receiving that, the node manager adds that node to the active node list. That node remains there until the node-manager receives a direct or indirect indication of node's not being online. A direct indication is sent by a node-agent if a user temporarily disables the node's participation in collective or when a node shutdowns gracefully (e.g., as part of node's shutdown). Indirect indications are reported either by service manager, clients or other PNs, whenever they happen to contact the node for certain data, and does not get any response. Indirect indications may be a genuine shutdown or failure, but can also result from network partitioning, or due to wrong reporting by a malicious client/PN. On direct indication, CM removes the node from the active list. On indirect indication, CM adds the list to a check-alive list. CM pings the nodes on check-alive list few times (mostly when CM is free from other work) before it removes the node from active list.

3.2.3 Information Aware Scheduler

One of the biggest challenge of a CM is to effectively utilize the resources of participating nodes towards meaningful activities. A CM needs to understand the importance, and resourcefulness of individual nodes, and should try to maximize the utilization of its idle resources. At the same time, appropriate resources should be made available to the different services to maintain acceptable performance.

The core of the scheduler is based on a mix of information aware and adaptive strategies. It tracks all sort of past as well as live information, e.g., participating nodes' resource profiles, their past performance, service demand rates etc. These diverse information are then used to make informed scheduling decisions. We can divide the information aware scheduling process into three main categories: history aware scheduling, reactive scheduling, and network aware scheduling.

3.2.3.1 History Aware Scheduling

The collective incentive model rewards consistency of participating nodes and hence it leads to nodes staying longer in the system. This provides an opportunity to learn about participating nodes' available resources and performance over an extended period of time. This information is used strategically during scheduling of services.

A collective manager (CM) periodically collects historical data about all participating nodes in the system. The historical data are collected with the help of a small agent called *node-agent* that runs on every node participating in a collective. A *node-agent* collects useful information, e.g., observed upload bandwidth, node's boot up timings, etc., and sends these information to the collective manager periodically (e.g., after every 3 days). Based on this information, the CM has an idea about each participating node resources and its past history about active/nonactive timings in the collective. Typically the CM creates different clusters (information lists) based on different desired behaviors - e.g., nodes having the longest active time during the last 5 days, during the last month, or the node having the highest upload bandwidth, etc. These information lists are then used to make informed decisions for various activities, e.g., to decide the caching pattern of a content. Nodes can be grouped based on a node's up/down timings, a node computing or storage or networking capabilities or a node network location.

3.2.3.2 Reactive Scheduling

As part of reactive scheduling, a collective system can reactively take actions based on observed dynamic behaviors. These actions lead to adaptive scheduling of resources to fit changing circumstances. The main methods for reactive scheduling are as follows:

- **Service Demand**: A CM can monitor service demand rate with the help of partners (e.g., content distributor in CCDS) and then use that information to achieve better scheduling of resources. For example, if there is a rise in a particular content demand (e.g., from a sale of 10 per day to 1000 per day), the scheduler can increase replication to handle the increased load.

- **Churning Detection**: Another technique is to detect failures/churning and then take actions to mask those failures by creating more replicas when a previous replica fails.

- **Client Reports**: Another approach to get feedback is based on performance reports sent by client application. For example, if a client application downloading a content does not get sufficient bandwidth, it can send a report to the CM. The CM can then take actions to fix the problem by creating more replicas or moving replicas to better nodes.

3.2.3.3 Network Aware Scheduling

Network aware scheduling utilizes network characteristics of participating nodes to take informed scheduling decisions. For example, a university network or an ISP such as utopia network [86] has significant intradomain bandwidth that is way higher than its Internet bandwidth. In such a scenario, selecting a node within the same network can provide much higher bandwidth than a node anywhere else in the Internet.

Additionally, some services may have specific network requirements. For example, we discuss a collective network probing service in Section 9.2 where probing tasks are assigned based on a node's ISP or other specific network characteristics.

3.3 Security

The basic security problems that must be addressed in a collective infrastructure are (i) how to uniquely identify and authenticate each entity, (ii) how to ensure that a PN is not misused or corrupted as a side-effect of participating in a collective, and (iii) how to handle selfish or malicious behaviors.

We discuss the high level security architecture in this section. Other service level security mechanisms (e.g., nonrepudiation protocol used in CCDS and CBS) are discussed along with service design in Chapters 4 and 5.

3.3.1 Identity and Authentication

Each PN and each client is identified by a unique machine-generated ID and a unique public/private key pair. The CM acts as the root of the public key infrastructure (PKI) employed by its collective. Each PN and client is issued a certificate signed by the CM that associates the public key of the PN or client with their unique IDs. Similarly each partner also is identified by a unique public/private key pair. These keys and certificates are used to create secure communication channels and to digitally sign the reports sent to the CM.

3.3.2 Trust and Access Control at PNs

The collective uses a VM sandboxing environment where a PN runs a VM instance to provide resources to a CM. This ensures that the collective software is isolated from the host PN. That is, applications running inside the VM can neither directly interfere nor access resources belonging to the host PN. Additionally, the host PN can enforce resource controls such as disk quota, cpu share, and physical memory allocation for the VM. This allows normal work to proceed on the host PN without undue impact. The protection and resource controls provided by VM technology will make people more willing to make their machines accessible to the collective, without fear that they will be misused or infected.

Even though VMs provide good isolation, a malicious user can still misuse the virtual machine to launch network attacks [35]. This problem is handled through access control, i.e., a VM instance on a PN can be directly controlled only by the CM. We do not allow external entities to run arbitrary code on VMs. All entities other than the CM interact with VMs only through a well-defined application-level protocol.

On the flip side, a host PN can get complete access to the VM running on it. A selfish PN administrator can potentially see or even modify files and data loaded on the virtual machine. Selfish behaviors and prevention mechanisms are discussed in Section 3.4 along with the incentive model used in collective.

3.3.3 Malicious Behaviors

Malicious nodes, clients, or external entities are interested in disrupting a collective without any benefit to them (in contrast to selfish entities that try to maximize their personal gain). We do not focus on these issues as part of this dissertation, and they remain part of the future work.

3.4 Incentive Model and Deterring Cheating Behaviors

Since collectives may include cheating nodes, it is important to mitigate the negative effects of cheating behavior. Cheating nodes strive to earn more than their fair share of compensation. Selfish behavior has been observed extensively in distributed systems, e.g., free riding in Gnutella [1] and software modifications to get more credit than earned in SETI@home [48].

For a collective system to work, the system must discourage dishonest behaviors (e.g., cheating users who lie about how many resources they have provided) and encourage nodes to stay in the collective for extended periods of time.

To address these challenges, we have designed an incentive system based on game theory and the economic theory behind law enforcement that motivates just these behaviors. In 1968, Becker [7] presented an economic model of criminal behavior where actors compared the expected costs and expected benefits of offending, and only committed crimes when the expected gains exceed the expected costs. Since then there has been significant research extending the work of Becker. Polinsky et al. [71] provide a comprehensive overview of the research dealing with deterrents in law enforcement. In this section we describe our incentive system and our mechanisms to discourage dishonesty.

In a collective system, a PN's compensation is based on how much its resources contribute to the success of services running on the collective. A CM shares its profits with PNs in proportion to their contribution towards different services. For example, in the CCDS example, PNs will receive a fraction of the money paid by the content distributor roughly proportional to the fraction of the total content that they deliver. The basic unit of compensation is a CM-specific credit that acts as a kind of currency. Users can convert credits to cash or use them to buy services from the CM or associated partners.

For the incentive system to work, the CM needs an accurate accounting of each PN's contribution. The CM cannot simply trust the contribution reported by each node, since selfish nodes can exaggerate their contributions. In this section we discuss how we create deterrents against cheating behavior.

3.4.1 Contribution Accounting and Accountability

Contribution accounting is mostly done at the service level and depends on the design of the service involved. The basic idea is to collect information from multiple sources (e.g., PNs, partners, clients, and the CM) and do offline data analysis to decide the individual node's contribution.

3.4.1.1 Credits Earned $\propto$ Work Performed

The work performed to support a service invocation, e.g., downloading a movie, should be credited to the appropriate PNs. Each PN sends a detailed daily report of its activities to the CM. In the absence of cheating PNs, each service activity can be credited to unique contributing PNs. If one or more nodes are cheating, more than one node will request credit for the same work. To resolve such conflicts, the accounting system needs additional information.

3.4.1.2 Accountability

Each PN and each client is identified by a unique public/private key pair. The CM acts as the root of the public key infrastructure (PKI) employed by its collective. Each PN and client is issued a certificate signed by the CM that associates the public key of the PN or client with their unique IDs. These keys and certificates are used to create secure communication channels and to digitally sign the reports sent to the CM.

3.4.1.3 Offline Cheater Detection

To identify cheating nodes, the system collects data from PNs, CM scheduling records, service scheduling records, partners' sales records, and even completion reports by client applications (if available). These data are used to resolve conflicts by comparing what work nodes claim they did against what other entities claim was done. Conflict resolution is done offline periodically (e.g., daily). With multiple information sources, it is possible to detect cheating behaviors by PNs. However, we do not assume that CMs will be able to detect all instances of cheating behaviors. In Chapter 6 we show that our incentive model works even when we can detect only 4%-5% of cheating behaviors.

3.4.1.4 Collusion

Of course, PNs can collude with each other and with clients. Collusion allows cheaters to confuse the CMs by providing incorrect reports from multiple channels. We counteract this by using service-specific mechanisms to make it economically unattractive to collude.

3.4.2 Variable Pay Rates (Raises and Cuts)

To provide an incentive for nodes to provide stable resource levels and to penalize cheating, the amount of credits received by a node in return for work depends on the node's long-term *consistency*. A node that remains in the CM's pool for long periods of time and that provides continuous predictable performance receives more credit for a unit of work than a node that flits in and out of the CM's pool.

Credit-per-unit-work (pay) rates are divided into levels. PNs enter the system at the lowest pay rate; a node's pay rate increases as it demonstrates stable consistent contributions to the collective. The number of levels and the behavior required to get a "pay raise" are configurable parameters for any given service.

To discourage cheating behavior, the system can apply a pay cut when it identifies a node mis-reporting the amount of work it performs. The size of the pay cut can be configured on a per-service basis. Selfish behavior in one service leads to pay cuts in other services run on that node. As an alternative, we could ban PNs from the system when they are caught cheating, but doing so eliminates nodes that might "learn their lesson" after finding that cheating does not pay in the long run. If a node continues to cheat, its pay rate becomes negative (i.e., it accumulates debt that must be worked off before being paid), which has the same effect as simply banning them.

Other factors can be applied to determine a particular node's pay rate. For example, nodes that are particularly important to a given service due to their location or unique resources (e.g., a fat network pipe or extremely high availability) may receive a bonus pay rate to encourage them to remain part of the CM's pool.

3.4.3 Incentive Model Summary

Our incentive model employs a currency-based system that rewards work performed, as well as the consistency of the work. Further, it is a well-known phenomenon in game theory that repeated interactions give rise to incentives that differ fundamentally from isolated interactions [62]. Thus, the collective manager employs offline analysis of data provided by participating nodes, partners, clients, and collective managers to determine future pay rates for each node. Consistently desirable behavior leads to increased rewards, e.g., the pay rate of nodes increases in response to predictable long-term availability. Undesirable behavior results in decreased rewards, e.g., the pay rate of nodes decreases in response to being caught lying about work done in an attempt to receive undeserved compensation.

In Chapter 6, we analyze the impact of our incentive model from an economic standpoint to derive key properties of our incentive system. We examine the impact of decisions made by selfish nodes and analyze the gain vs loss possibilities for participating nodes as we vary the likelihood of bad actors being caught. We show that while we cannot prevent users from being dishonest, our mechanisms mitigate dishonest behavior by making it economically unattractive. We show that a small probability of catching cheaters (under 4%) is sufficient for creating a successful deterrence against cheating. We further show that our incentive system can be used successfully to motivate nodes to remain in the system for prolonged durations.

3.5 Summary

This chapter has presented the overall design of a collective system. Especially it has described the the architecture of the collective manager and the overall incentive system that makes the core of a collective.

Many of the topics covered here are revisited in upcoming chapters in more detail. We discuss the detailed design of two services - a collective content distribution service (CCDS) in Chapter 4 and a collective backup service (CBS) in Chapter 5. Similarly the incentive model is revisited in Chapter 6 and information aware scheduling is discussed in detail in Chapter 7. Many of the implementation specific details are presented in Chapter 8.

CHAPTER 4

COLLECTIVE CONTENT DISTRIBUTION SERVICE

Recently there has been an explosion in legal content distribution over the Internet, e.g., the highly successful iPod and iTunes services. Many prognosticators predict that the Internet will become the predominant means for audio and video distribution in the foreseeable future. The typical infrastructure required to support such distribution (e.g., Apple's iTune music server) is a large dedicated cluster of web servers with tremendous external bandwidth. A web-system that can support millions of downloads per day, especially of multi-gigabyte videos, is an expensive operation in terms of both infrastructure and administrative costs.

We have designed and implemented a collective content distribution service (CCDS), where the idle resources of myriad cheap network-connected computers distributed around the world are harnessed to achieve that. CCDS distributes large content files (e.g., movies, music, or software updates) to thousands of clients on behalf of its service partners. Typical partners are content distributors, e.g., an online movie distribution company that utilize the collective service for movie distribution, while managing the content acquisition and sales itself. Figure 4.1 illustrates a typical collective content distribution network where clients buy content from a content distributor and download it from a CCDS overlay.

4.1 Design

CCDS is managed by a service manager running on the CM, called the CCDS manager. The CCDS manager builds a content overlay network using the storage and bandwidth resources of participating nodes to cache and serve the data to the clients. Each participating node runs a CCDS component called the *CCDS agent* to

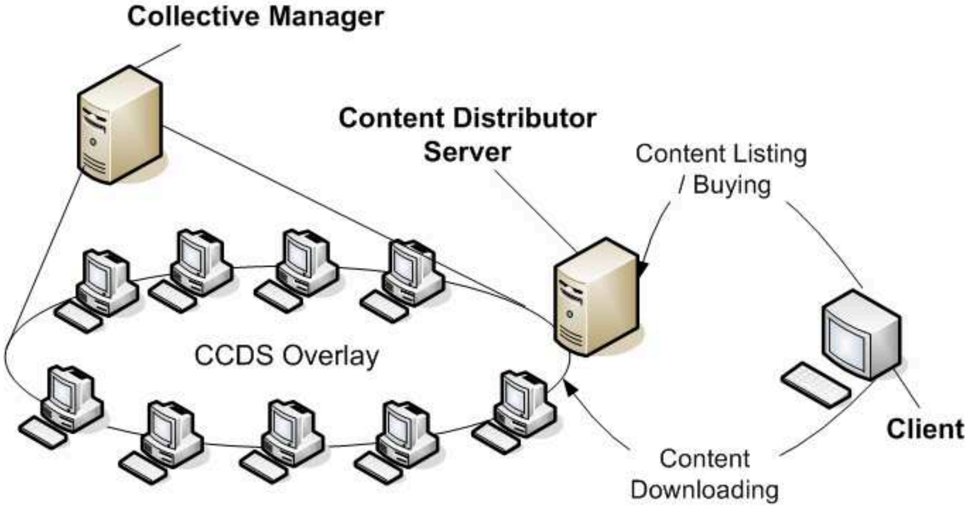

Figure 4.1: Collective Content Distribution Service

provide this service. *CCDS agents* are run inside the VM allocated to the CM on the PNs.

4.1.1 Usage Overview

Clients have an application interface, e.g., something like the iTunes content download application, that interacts directly with the content distributor for content browsing and purchase, but uses CCDS for actual content download. To initiate a request the client software contacts the content distributor, and after proper authentication/payment obtains a small signed certificate with rights to download the content from the overlay, along with a list of related participating nodes. The client then downloads the content directly from the participating nodes, with different chunks coming from different nodes.

4.1.2 Control Flow

CCDS is a push-based service, where the content distributor pushes content to the CCDS overlay, and then has clients download the content directly from the overlay. We anticipate that a content distributor will typically push its most popular content to the CCDS overlay, while serving the requests for odd content itself.

A content distributor interfaces with the CCDS manager, and can issue dynamic requests to it to cache particular content on its overlay system. Content consists of a file or a set of files usually in an encrypted form. The content distributor assigns a unique *content ID* to each content and creates a metainfo file that can be used to digitally verify the correctness of the content. Each content is logically divided into chunks of fixed size (we use 512KB in our prototype) and md5sum of each chunk is calculated to prepare the content's *metainfo*.

To push a content into the CCDS, the content distributor passes the associated metainfo to the CCDS manager. Upon receiving the metainfo, the CCDS manager generates an initial distribution pattern and informs the appropriate PNs to cache the relevant data. The CCDS manager then provides the content distributor with the caching map for the content. This map is used by the content distributor to provide a list of PNs that clients should contact to complete the download.

The CCDS manager ensures that the caching map used by the content distributor is kept updated by pushing a revised map periodically, or when major reshuffling.

4.1.3 Content Caching

Once the CCDS manager gets a request from the content distributor to distribute a particular content, it must decide a caching strategy for it. As part of this, the CCDS manager selects a set of participating nodes, and distributes chunks of the content to them. A given chunk of the content is cached on multiple nodes to ensure availability in case of node departures/failures, and to improve aggregate available to the clients.

The CCDS manager uses a feedback-driven system where it adapts to changing demand of the content as well as node departures or failures. To help in this, the content distributor keeps track of recent purchases for a particular content, and periodically informs the CCDS manager about the demand statistics for that content - e.g., 10 copies sold in the last hour. The CCDS manager uses this information to predict future demand, and ensures that data are cached at enough places to meet the demand. Similarly, the node manager informs the CCDS manager about any observed failures or departures among PNs used by the CCDS service. The CCDS manager processes this information and initiates new replicas if needed.

4.1.4 Authorization

The goal of our authorization system is two fold: 1) to ensure that the only authorized clients/nodes are able to download the content from CCDS and 2) to have a trail that enables us to determine who downloaded from whom for resource accounting purposes.

After purchasing a song, a client gets a signed certificate (called a *cauth*) from the content distributor. A *cauth* acts as an authorization to download the data from CCDS. A *cauth* consists of a 4-tuple data signed using the private key of the content distributor. The *cauth* data consist of $\{contentID$, $ckey$, $authID\}$. Here $contentID$ represents the content that was bought by the client. $cKey$ is the client public key that identifies the client that was authorized to download the content. $authID$ is the transaction ID for resource accounting purposes. A client must provide the *cauth* certificate as part of the handshake protocol to the PNs from which it is requesting a part of the data. PNs verify the authorization and provide the data only if the client has the valid authorization.

4.1.4.1 Nonrepudiation

To guard against certain cheating behaviors, we use a nonrepudiation protocol for any data transfer between PNs and clients. This ensures that when a PN sends data to the client, the client cannot deny receiving the data. That is, a PN receives a verifiable proof whenever it provides certain data to the client.

Our protocol is directly based on offline TTP (trusted third party) based protocol described in [55]. A trusted third party (TTP) is said to be offline if it is involved in the protocol only in case of a incorrect behavior of a dishonest entity or in case of a network error. As per this protocol, a sender sends ciphered data (D_k, i.e., D encrypted with cipher k) to a receiver instead of the data D. The cipher key k is sent separately, only after the receiver sends a signed proof of receipt of the data to the sender. To prevent a sender from getting a proof of receipt and then not delivering the key k to the receiver, the sender has to send the key k encrypted with public key of the TTP to the receiver along with the data. So if the receiver does not get the key k, it contacts TTP with the key k encrypted with the public key of TTP and a signed proof of the receipt of the data. In response, the TTP decrypts the key k and sends it back to the receiver. In addition, the TTP sends the proof of the receipt of the data to the sender. This is necessary, as the receiver can contact the TTP directly without sending the signed proof of receipt of the data to the sender. Exact details of the protocol can be seen in [55]. In our system, the CCDS service manager acts as the offline TTP.

4.1.4.2 Connection flow

We use transport layer security protocol (TLS) [23, 74] layered on top of TCP as the basic communication channel between clients and nodes. As already mentioned in the Section 3.3.1, each client and node in the system has a unique PKI certificate associated with its identity. This certificate acts as the authentication proof during the TLS handshake.

Once authenticated, the requester needs to provide the necessary authorization certificate (*cauth*) and only then the actual data transmission happens.

4.1.5 Digital Rights Mechanisms and CCDS

The CCDS service is a system to distribute/download content and as such does not suggest any Digital Rights Mechanisms (DRM). However, it is possible to implement a content distribution system with a DRM mechanism that uses CCDS for content distribution. For example, it is possible to use the CCDS system to distribute content using the two most prevalent DRM systems today, fairplay used by Apple's Itune and Microsoft's DRM system [65].

In both the cases, a client interested in a particular content will download the data in an encrypted format from the CCDS overlay, but will need to acquire the licenses from the content distributor to decrypt the content before consuming it.

In the fairplay system, the encrypted content and licensing information are represented as a single file. The license keys are present in the starting block of the file. To achieve this functionality using the CCDS system, the initial block of the file would be delivered directly from the content distributor server. The rest of the file containing the encrypted content will be downloaded from the CCDS overlay.

In the case of Microsoft's DRM technology [65], the encrypted content and license keys are already handled separately. A client will download the file from the CCDS overlay and will then request the license information from the content distributor server.

4.2 CCDS Incentive Model

In this section we start by describe the contribution accounting system and then we present how we deter cheating behaviors by participating nodes.

4.2.1 Accounting System

The accounting system is responsible for determining how much a PN should be paid based on its contribution to the useful work. The core of the CCDS's accounting system is an offline processing system that acts on the reports collected from the PNs and content distributor, and on information available to the CCDS manager.

4.2.1.1 PN's Report

PNs periodically (e.g., every day) send reports to the CCDS manager that detail the content they have distributed and to which clients. This report contains the content ID, transaction ID, and specific chunk numbers that were delivered. It also includes the nonrepudiation of receipts (NORs) that they have accumulated as the proof of their contributions. PNs send this at a random time of the day to prevent contacting CMs all at the same time.

4.2.1.2 Content Distributor Report

The content distributor sends a log on what content was sold to whom. The log includes content ID, the client ID, and timestamp of the transactions. This log also includes the list of the PNs that were specified to the client from which to download the data.

4.2.1.3 CCDS Manager Information

The CCDS service manager maintains the metainfo file for content ID which can be used to determine the size of content, the number of chunks into which it was divided, and corresponding chunk sizes.

The accounting system uses reports provided by PNs and the content distributor to determine individual PN contributions. It organizes information based on transaction ID. For each content sold it prepares the list of nodes involved and their declared contribution. It verifies the NORs sent by the PNs in their reports and removes any unverified contribution records from the list (labeling nodes that provide such reports as cheater). Thus it prepares a list of contributors for every content sold and compensates PNs accordingly.

4.2.2 Deterring Cheating Behaviors

To create a successful deterrent, any cheating behavior should be economically unattractive to the participating nodes or should lead to detection and penalty for the cheating. Our use of nonrepudiation protocol provides crucial help for this deterrence. As already mentioned, we employ a nonrepudiation protocol to transfer the data to clients. This ensures that a PN can provide verifiable proof (i.e., nonrepudiation of receipt, NOR) for all data delivered by it.

We can divide the possible cheating behaviors in three main categories: (i) single node cheating behavior, (ii) single client cheating behavior, and (iii) collusion based cheating behavior.

4.2.2.1 Single Client Cheating Behavior

A cheating client cannot affect the accounting system, because as part of the protocol it must follow the nonrepudiation protocol and deliver the right NORs.

4.2.2.2 Single Node Cheating Behavior

A cheating PN can try to increase its profit by mis-reporting its actual contribution. However, it cannot generate the NORs necessary to verify the contribution and thus will be caught easily by the CCDS manager.

4.2.2.3 Collusion

PNs can collude with each other and with clients. Collusion allows cheaters to confuse the CMs by providing incorrect reports from multiple channels. We counteract this by using service-specific information to make it economically unattractive to collude. There are three main possibilities for collusion: 1) clients colluding among themselves, 2) PNs colluding among themselves, and 3) clients and PNs colluding with each other.

If only clients collude with each other, they cannot cause incorrect accounting because they still need to issue NORs whenever they successfully receive content from any PN. Similarly, if only PNs collude with each other, they cannot generate any false NORs and thus cannot fool the accounting system. Thus, a set of colluding cheating clients or a set of colluding cheating PNs cannot lead to any incorrect accounting. If they try, they will be detected and punished.

The only interesting case arises when clients and PNs collude with one another. A cheating client or clients can provide unlimited false NORs to their fellow colluding PNs. For example, a client can issue a false NOR for a chunk to colluding node A, even though it downloaded that chunk from another node B.

Two mechanisms limit the value of such collusions: client download cost and random scheduling. Since our target service is a paid download service, colluding PNs can receive “extra” credit only for content purchased by a colluding client. The economic value of the credit per download to the participating nodes is always less than the cost paid by their buyer, as only a fraction of the profit trickles back to the participating nodes.

Random scheduling limits the value of collusion even further. The CCDS service manager divides content into multiple chunks and distributes these chunks to a random set of PNs. Even for content bought by a colluding client, the colluding PNs can get credit only for those parts of the content that they are scheduled to provide, which will likely be a small fraction of the content. For collusion to be useful, it must include a very large percentage of the nodes involved in the collective, including clients.

4.3 Related Work

CCDS is similar to content distribution networks like Akamai’s [2]. Like Akamai, the CM proactively caches content on participating end nodes. Akamai uses dedicated trusted computers, while CCDS is built on untrusted, unreliable, and resource-limited end nodes.

Bullet [52] and Bullet2 [53] are systems for high bandwidth data dissemination using an overlay mesh. They are peer-to-peer systems, although they assume the presence of scalable mechanisms for efficiently building and managing an underlying overlay tree (whose overhead is not clear). These systems also assume altruistic users who are willing to cooperate both during and after the file download without any incentives. Thus it is fairly trivial to construct free-riding strategies that can vastly degrade these systems performance.

Slurpie [79] is a P2P protocol to reduce server overhead while downloading a popular file by sharing file pieces between the clients. It requires a centralized topology server that has to be contacted by each client to register itself, and to retrieve a list of peer nodes. It does not provide any incentive scheme and does not provide any guard against clients modifying the protocol to their benefit. It basically assumes altruistic clients. One can easily download files without sharing anything. Also it is built on top of the http protocol and requires all clients to use the slurpie protocol; otherwise nonslurpie clients will go directly to the server, thus making it unfair to slurpie users. They do not give any way to achieve this, however.

Codeen [91] is a network of caching web proxy servers that help in reducing the load on a web server. Unlike CCDS, their main focus is on latency-sensitive web access. They do not have any incentive model for the participating nodes and all the participating nodes are trusted. Unlike CCDS, their node set consists of hundreds of nodes. They use active monitoring to monitor the health and status of all web proxy servers. Such monitoring is not possible in CCDS where we propose to deal with thousands to millions of participating nodes.

4.4 Summary

This chapter describes the design of a collective content distribution service. Given the growing need of legal content distribution (e.g., paid songs, movies, and tv shows) over the Internet, this service is a great fit for utilizing the idle resources of end-nodes for a commercially useful purpose.

The most important contribution of this chapter is the mix of security and economic mechanisms designed as a core part of the service. This ensures the correct accounting of individual nodes' efforts and creates a successful deterrence against cheating.

CHAPTER 5

COLLECTIVE BACKUP SERVICE

Collective backup system (CBS) is a distributed backup service build on top of our collective system. Online backup is becoming an important service for millions of home and small business communities that do not want to spend money on professional in-house backup services.

CBS is a paid backup service that allows clients to store and retrieve data files over the collective. All stored data are encrypted at the client computer before being submitted to CBS. As in CCDS, from the client's perspective, CBS appears to be a traditional client-server solution. Clients do not have to be a participating node to use the service. Clients use a small application to store and retrieve data and are charged based on their usage. Figure 5.1 illustrates a typical collective backup service where clients interact with a CBS server as a front-end to service, while actual data are stored and retrieved from the collective overlay in the background.

5.1 CBS Design

The main entities in CBS are the CBS server, the CM, participating nodes, and clients. The CBS server provides the backup services to clients and manages their login and accounting details. The CBS is managed by a service manager running on the CM, called the CBS manager. The CBS manager builds a storage network using the storage and bandwidth resources of participating nodes to cache and serve data to the clients. Each participating node runs a CBS component called the CBS agent to provide this service. CBS agents are run inside the VM running on PNs.

5.1.1 Usage Overview

A client uses an application that allows it to easily back up a file on the CBS or to restore files from the CBS. The CBS client app works like a typical file manager

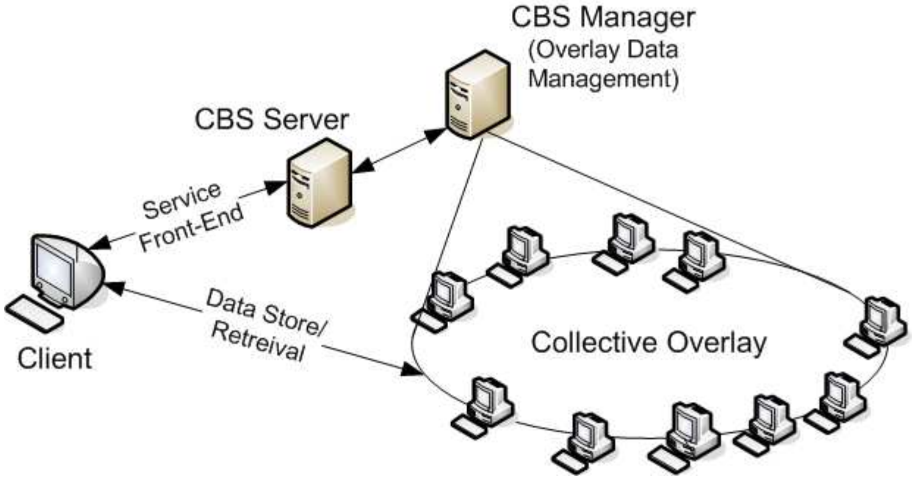

Figure 5.1: Collective Backup Service

application and displays a list of all files and directories stored by the client in the CBS. To back up a file, a user starts by adding the file or directory to the CBS client app's list of files being managed, e.g., through a drag-and-drop user interface. The client app then contacts the CBS manager and uploads the data to appropriate nodes on the overlay. To restore/retrieve a file or directory, the client selects the particular file in the client app and a destination folder on the local disk. The client app then downloads the requested file from the overlay and saves it in the appropriate folder.

Clients are charged based on their actual usage of the service. There are service costs for data storage as well as for data transfer. Data storage is measured as a function of data size and the actual time that data were stored in the CBS. For example, storage of 3 GB of data for a month on the overlay leads to charge of 3GB-Month. Data transfer costs are measured based on actual amount of data transfered to/from the overlay. There can be different rates for data-in (store) and data-out (retrieval). PNs using this service can "pay" using credits they receive for becoming a PN in the collective system (as an alternative to actual money paid by normal clients).

One can also layer different cost models over this basic service. For example, one can package a backup service providing 5GB of storage with a fixed monthly cost.

5.1.2 Content Caching

Given a data store request, the client app starts by creating a metainfo that will be used to digitally verify the correctness of the data. Each file is logically divided into chunks of fixed size (we use 512KB in our prototype). The md5sum of each chunk is calculated and added to the data's *metainfo*. The client app then contacts the the CBS manager to register the data and passes the metainfo to the CBS manager. The CBS manager then assigns a unique *data ID* for the data, and provides a list of PNs to which the client should upload the data. Different data chunks are assigned to different PNs with different data chunks going to different PNs. The client app then uploads the data to those nodes.

The CBS manager maintains a list of data files saved by a client and corresponding metainfo. The client app can retrieve this information from the CBS manager by providing appropriate authentication information. This authentication information is based on unique PKI certificate associated with each client as described the Section 3.3.1. The CBS client app typically keeps a locally cached copy of the user's data's metainfo and provides a GUI that lets the user browse the list of files stored for the user in the CBS.

To restore a file, the client selects the desired file in the CBS client app. The CBS client app then contacts the CBS manager to retrieve a list of nodes storing copies of the data and downloads the data directly from those nodes.

Each transfer employs a nonrepudiation protocol similar to the one employed in CCDS, so that both the sender and receiver can provide verifiable proof of the transfer. We use the offline trusted third party based nonrepudiation scheme described in [55].

It is the responsibility of the CBS manager to ensure that data remain available even in presence of node churnings, failures, and selfish behaviors.

5.1.3 Durability

Durability is the most important concern when building a remote backup service out of unreliable end-nodes. The CBS must ensure data durability even in the presence of node failures and churning.

These problems have been studied extensively in projects like Ocenastore [75, 56], Total Recall [8], Glacier [41], and Carbonite [18]. We use lessons learned in those systems in the design of CBS. Similar to Oceanstore and Glacier, we employ both erasure coding and replication to maintain durability [32]. That is, data chunks created through erasure coding are further replicated on multiple nodes. We use the Cauchy Reed-Solomon code [10] for erasure coding.

We use the PN historic availability data collected by the CM to decide the necessary degree of replication and the erasure code ratio to use. These values are dependent on typical node availability, node churning, and rate of permanent failures (e.g., due to disk failures) observed by the system.

The CBS manager keeps track of permanently failed replicas, i.e., replicas no longer available either due to disk failures or because of a PN permanently leaving the collective. The CBS manager creates additional replicas at a rate roughly equal to the permanent failure rate to ensure the durability of the data. There are two main ways that the CBS uses to detect failures — active mode detection and passive mode detection. Passive mode detection is achieved whenever the system notices failure as part of some other activity, e.g., a client trying to retrieve certain data. Active detection is based on periodic data checks (e.g, once a month) to ensure that data stored on a PN is not lost. Data check involves the act of successfully transferring data and verifying its correctness using a hash to ensure that a node still possess a particular piece of data. Typically a node will be asked to transfer only a randomly selected small portion of the data stored by it. These checks also serve the additional purpose of ensuring safety against cheating behaviors (more details on cheating detection are provided in Section 5.2.2).

5.2 CBS Incentive System

The goal of the CBS incentive system is to reward correct behavior by PNs supporting the CBS service and to create a successful deterrent against cheating behaviors. In this section we describe the mechanisms employed by the CBS to implement the incentive system. We start by describing the contribution accounting system and the role of different reports used for this purpose. We then follow it up with our mechanisms to deter cheating behaviors by participating nodes.

5.2.1 Reports and Accounting

5.2.1.1 PN's Report

Each participating node sends a periodic report to the CBS manager listing the data it is storing, the time-frame of the storage, and a list of transfers in which it has participated since the last report. It also appends the nonrepudiation certificates for each transfer.

5.2.1.2 CBS Manager Report

The accounting system gets a detailed history of the clients' requests to the CBS manager for storage or recovery, and the corresponding list of the PNs sent by the CBS manager.

With these reports, the CBS accounting system does offline analysis to tabulate the contributions of individual PNs. A participating node participates in two main activities: storing data on its disk and transferring data to other nodes/clients for replication or recovery. Thus, the CBS manager rewards each participating node in proportion to the data stored over time and for the amount of data transferred.

In addition to the above reports, the CBS manager periodically does data checks to ensure protection against cheating behaviors. These checks and their role in deterring cheating are discussed in next subsection.

5.2.2 Deterring Cheating Behaviors

To discourage cheating behaviors, CBS must offer incentives to PNs to behave according to system-specified rules. Any cheating behavior should be economically unattractive to the participating nodes, or should lead to detection and penalty for the cheating.

As a first layer of protection, all data transferred between nodes, or between nodes and clients, employ a nonrepudiation protocol, thus creating a verifiable trail of who did what. This accountability is crucial to ensure correct higher level functionality.

As a second layer of protection, each participating node must send periodic reports to the CBS manager, as mentioned above. Each participating node is assigned a set of data chunks to store. If a PN is found to have successfully received some particular data, but then does not report it as part of its periodic report of data being stored, the CBS manager penalizes the PN for wasting system bandwidth.

A third mechanism ensures correct data storage and detects any data deletion by cheating PNs. To achieve this, the CBS manager performs periodic data checks (e.g., once in a month). The data check involves the act of successfully transferring data from the PN to verify that the PN still possesses a correct replica of the data of concern.

A node storing GBs of data does not have to transfer all of its data to prove its possession. A node is asked to transfer only a randomly selected small portion of the data stored by it. At first glance these periodic data transfers may seem to be a big overhead, but we observe that the CBS system must create additional replicas of the stored data continuously to survive disk failures or data deletions. The transfer needed to create such replicas can act as the data possession checks for the nodes involved.

This work is managed by a component of CBS manager called verification server. Verification server ensures that a periodic data check (e.g., once a month) is performed either directly by it or indirectly with the help of other participating nodes. Depending on past behavior, the verification server can also turn to auditing mode where only a randomly selected small percentage of PNs are verified every month.

5.2.3 Cheating Analysis

5.2.3.1 Single Node Cheating Behavior

A PN earns credit for data storage or data transfer. Thus, possible cheating behaviors include misreporting the amount or duration of data stored or misreporting a data transfer. A particularly cheating node may even delete the stored data and still try to get credit for its storage.

Our nonrepudiation protocol ensures that a PN cannot misreport a data transfer. If a PN does so, it will be caught and penalized. Additionally, a trail of valid data transfers created by the nonrepudiation protocol allows the CBS manager to correctly determine the actual amount of data stored on a PN.

Also our nonrepudiation protocol and periodic data checks ensure that a node retains the data that have been transfered to it for storage or penalized for misreporting.

5.2.3.2 Single Client Cheating Behavior

A client can potentially initiate a data retrieval request even when there is no requirement of that particular data. This can lead to unnecessary use of system resources, especially bandwidth. There is no mechanism to differentiate a valid retrieval request from a invalid request made only for wasting system resources.

However, from the CBS perspective, this is not a problem. Clients are charged based on actual usage, which includes both storage and data transfer costs. So clients are charged for invalid requests as well and thus have a negative incentive to not waste CBS resources.

5.2.3.3 Collusion

PNs can either collude among themselves and/or with clients to increase their payments beyond their fair share based on the actual work performed.

A colluding set of PNs can try to avoid storing data at a node if another node in the set is also storing the same data. They can achieve this by using a reflection mechanism to pass periodic data checks. That is, given a data check request, a colluding PN can pass the query to another colluding node with the data and supply the returned reply to the checker. Though given a small set of colluding nodes, it should be rare for colluding PNs to share replicas, which mitigates the storage savings achieved by colluders.

A second case involves a mix of colluding clients and PNs. Here, colluding clients can try to invoke data recovery operations again and again, with the goal of getting extra transfer credits for the colluding PNs. However, this is not a useful strategy, because clients have to pay for each data transfer from the system. The CBS credit system is designed in such a way that eventual credits rewarded to PNs for a work is less than or equal to the profit made by the system for that work. Additionally as in the above case, the possibility of colluding PNs having the backup data owned by the colluding client is very low. The CBS manager ensures that any give data storage request is fulfilled by randomly chosen nodes, minimizing interstorage of data by a small set of colluding nodes.

CHAPTER 6

ECONOMIC ANALYSIS

In this chapter we evaluate our incentive model from an economic standpoint to derive key properties of our incentive system. In particular, we use game theory and probabilistic analysis to gain better insight into the implications of our design choices. We examine the impact of decisions made by selfish nodes and analyze the gain vs loss possibilities for cheaters as we vary the likelihood of cheaters being caught. We show that while we cannot prevent users from cheating, our mechanisms mitigate cheating behavior by making it economically unattractive. We show that a small probability of catching cheaters (under 4%) is sufficient for creating a successful deterrence against cheating. We further show that our incentive system can be used successfully to motivate nodes to remain in the system for prolonged durations.

6.1 Analytical Model of the Incentive System

Our economic analysis focuses on the two main entities in our system, collective managers and participating nodes. Participating nodes are assumed to be self-interested, rational parties, which from a game theory standpoint means that they act in ways that maximize their long-term financial gain even if this involves *cheating*. A collective manager is a trusted party that manages the resources of participating nodes to support commercial services. Its goal is to build a successful business providing services to external customers using its PNs' resources.

In game theory, systems are modeled as *games* played between *players*. Players are faced with a series of options from which they must choose. The outcome of each game (choice) depends on the player's choice and the choice(s) made by their opponent(s). The most famous example of game theory is the Prisoner's Dilemma [42], where two prisoners who are both accused of a crime are separated and individually given the option of either “cooperating” (staying silent) or “defecting” (confessing to the crime and testifying against the other prisoner). If both prisoners stay silent, they receive a 6-month sentence. If both prisoners defect, they both receive a 5-year sentence. If one prisoner cooperates and the other defects, the one who defects is set free, while the one who cooperates is given 10-year sentence. In a variant of the game where the players play the game repeatedly, researchers have found that they tend to learn to cooperate with one another and thus receive light sentences [42]. We exploit this phenomenon in our incentive model.

We model the interaction between participating nodes and the collective manager using a basic game theoretic utility model. At any given time, we present PNs with two orthogonal choices: (i) should they remain in the collective or not and (ii) should they report the correct amount of work for the last reporting period or attempt to claim they did more work than they did to receive a higher (undeserved) payment from the CM. In this game at any time slot s, a rational PN node can either choose to share or not share its resources based on the expected reward of each choice. We can represent the choices available to PNs and the collective manager using a simple table like Table 6.1. Each column represents the options available to the collective manager and each row represents the options available to a PN. Entries in table take the form a/b where a is the payoff (reward) for the row player (i.e., PN) and b is the payoff for the column player (CM). In a typical game theory situation, the two players make simultaneous decisions, but in our scenario PNs make their decision (share, no share) and then CMs make their decision (reward or not reward the PN).

The “games” played as a part of collective are non-zero-sum games, meaning that one player’s gain is not necessarily another player’s loss (and vice versa). PNs are not assumed to be altruistic, but rather we want to derive an incentive model where it is in each node’s rational self-interest to cooperate. In other words, it is our goal to design rules for the “game” such that rational actors will find cheating economically unattractive. In the remainder of this section, we consider different scenarios and determine whether the outcome realized achieves this goal.

Table 6.1: Payoffs for perfect monitoring case

	Reward	No Reward
Share	G_S/G_S	-
Not Share	-	0/0

6.1.1 Perfect Monitoring

We start by assuming a perfect monitoring scenario, i.e., the collective manager has perfect information about the contributions made by PNs it manages. In this case PNs cannot successfully lie to a CM about how much work it performs, because if they lie, they are guaranteed to be caught.

Table 6.1 shows the payoff structure for this scenario. A dash means that a particular case is not possible in this scenario, e.g., it is not possible for a PN to choose "Not Share" and have the CM choose to "Reward" it. Assuming a CM shares its income 50%-50% between itself and the PN concerned, we get the value of G_S/G_S for the PN share case. This means that if a PN share its resources, the PN and the CM both receive G_S benefit. Here G_S is a positive number, which denotes the gain (payoff) received for sharing.

If we apply standard game theory analysis to this utility table, both the states of (share/reward) and (no-share/no-reward) are *Nash equilibrium* [42]. Informally, a strategy is a Nash equilibrium if no player can do better by unilaterally changing his or her strategy. Even though (share/reward) is Pareto optimal, meaning that it leads to both players receiving their highest reward, both cases are equally possible from a Nash equilibrium point of view.

This analysis assumes that both players choose their actions independently, which as we mentioned above is not the case in our design. In our case, a CM makes its choice only after analyzing the action of the PN concerned, which is why the two dashed states are not possible. Here, the CM will always choose *reward* in response to *share*. Such a sequential game is known as Stackelberg (or "leader-follower") game. To find the equilibrium in such games, the technique of backward induction is used. That is, if the PN has played Share, the PN knows the CM (the follower) will respond by playing Reward, and the PN will get G_S credits; whereas if the PN has played Not

Share, the PN knows the CM will respond by playing No Reward, and the PN will get 0 credits. So the Stackelberg equilibrium for this case will be leader (PN) playing Share followed by CM playing Reward. Hence, a rational PN will always play Share.

6.1.1.1 Cost of Sharing

Table 6.1 does not model the fact that there is a cost associated with performing a job (e.g., power charges). Let c be the cost of performing a job, which typically will be small but positive since we are exploiting idle resources. Table 6.2 shows a modified reward structure that accounts for this cost.

6.1.2 Imperfect Monitoring

The previous analysis assumes that the CM has perfect knowledge regarding whether a PN is accurately reporting how much work it performs. Table 6.3 shows a payoff table if we assume that a CM can not always detect PN lies. Here c continues to represent the cost for a PN to perform a unit of work. Rational PNs now have an additional choice available to them; they can chose to *lie* to the CM, claiming to do work that they have not done. G_{cheat} is the expected reward that a PN will receive if it lies, and L is the loss incurred by the CM due to incorrect awarding of credits. If the system cannot detect lies, then G_{cheat} is equal to G_S, in which case a rational PN will always lie, since this lets it receive a reward without doing any work. Thus, if CMs cannot detect lying PNs, the system will destabilize since cheating PNs will always claim to do work, but not do it.

Table 6.2: Payoffs for perfect monitoring case with cost of sharing included

	Reward	No Reward
Share	$(G_S - c)/G_S$	-
Not Share	-	0/0

Table 6.3: Payoffs for imperfect monitoring case

	Reward	No Reward
Share	$(G_S - c)/G_S$	-
Cheat	$G_{cheat}/-L$	-
Not Share	-	0/0

Our collective service is designed to make it nearly impossible for PNs to successfully lie about their contributions. However, it is impractical to track enough information to catch all instances of a PN lying. If we assume that only a fraction of all lies will be detected, we can analyze the impact of undetected lies to determine what probability of lie detection is necessary to motivate selfish PNs to report the truth. Assume that the probability of detecting a lie (offense) is p_o. In that case, the *expected* payoff for lying (G_{cheat}) is:

$$G_{cheat} = (1 - p_o) * G_S$$

We can create a deterrent that punishes PNs when they are caught cheating, i.e., when they provide incorrect accounting information. If F is the amount we penalize PNs when we catch them lying, the *expected* payoff for lying (G_{cheat}) becomes:

$$G_{cheat} = (1 - p_o) * G_S + p_o * -F$$

We can represent F as a certain fraction of G_S, i.e., a PN is penalized a fraction (defined as b) of pay for each unit of work it falsely claims to have done. Adding this penalty results in an expected reward for lying (G_{cheat}) as:

$$\begin{aligned} F &= b * G_S \text{ where } b > 0 \\ G_{cheat} &= (1 - p_o) * G_S - p_o * b * G_S \\ G_{cheat} &= G_S * (1 - p_o - p_o * b) \end{aligned}$$

This results in the payoff table shown in Table 6.4.

Figure 6.1 plots possible payoff for a single unit of reward ($G_S = 1$) as a function of p_o, the probability of being caught. Different curves in the graph represent the payoff for different values of b, i.e., different sized penalties relative to the standard reward. We observe that the potential payoff of lying drops below zero when the

Table 6.4: Payoffs for imperfect monitoring with penalties

	Reward	No Reward
Share	$(G_S - c)/G_S$	-
Lie	$G_S * (1 - p_o - p_o * b)/ - L$	-

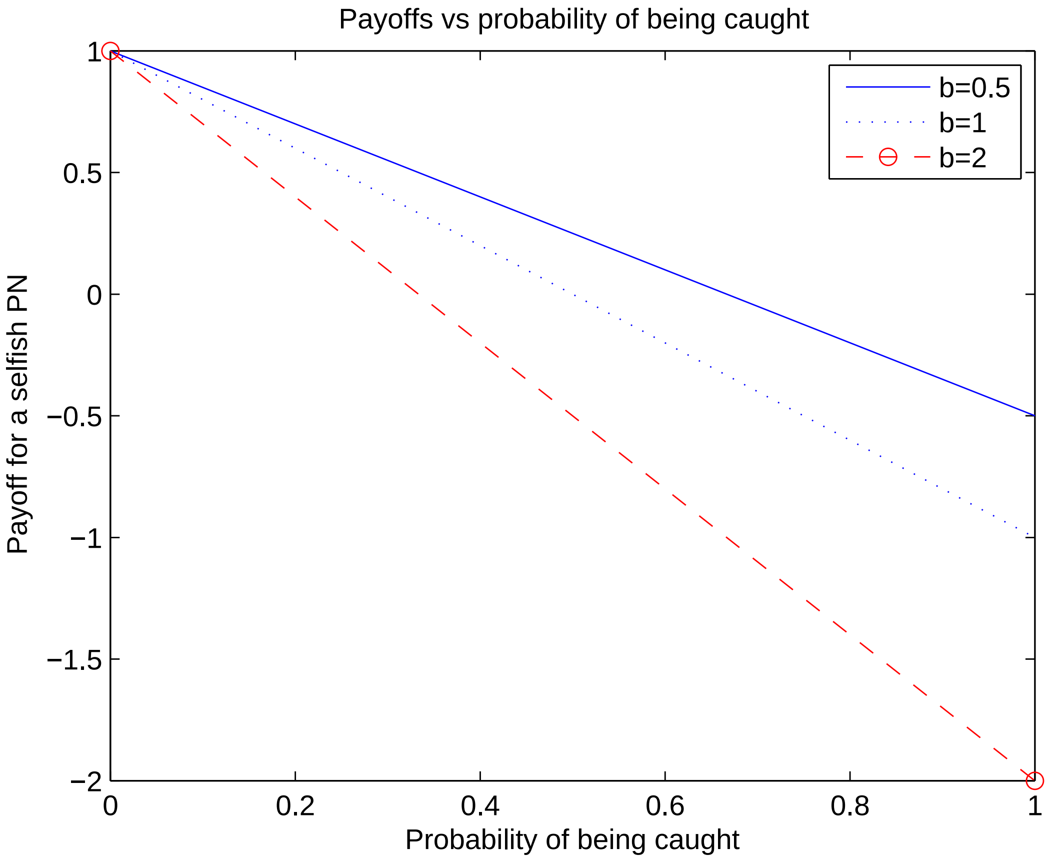

Figure 6.1: Expected Pay vs Probability of Being Caught

probability of being caught crosses a threshold that depends on b. Specifically, we can derive $G_{cheat} < 0$ as follows:

$$G_S * (1 - p_o - p_o * b) < 0 \implies p_o > \frac{1}{1+b}$$

So for $b = 1$, a probability of 0.5 or more is required to make fake sharing economically uninteresting to a user. A collective manager can effectively use different values of b to create different degrees of deterrence.

6.1.3 Variable Pay Rates

In the previous analysis, we considered only single-round games. However, in our system, PNs typical participate in a series of games, which lets us employ the game theory of repeated interactions [42] to analyze the impact of repeated interactions on the behavior of PNs.

A simple solution treats repeated interactions as independent, using the rules presented in earlier sections. In this case, we can use the sum of the individual round payoffs to understand the dynamics of repeated interactions. However, this approach does not exploit our ability to employ a variable pay rate mechanism that responds to observed PN behavior to motivate rational PNs to cooperate. We use pay variability to achieve two types of positive behaviors from PNs: (i) to encourage nodes to remain in the collective for extended, predictable periods and (ii) to punish cheaters.

To address our first goal, that of encouraging nodes to remain in the collective for extended periods, the amount of payment that a node receives in return for work is varied depending on its long-term consistency. A node that remains in the CM's pool for long periods of time and that provides continuous predictable performance receives more credit for a unit of work than a node that flits in and out of the CM's pool.

In our design, pay rates(R) are divided into l levels, $(R_1, R_2, ...R_l)$, whereby each pay rate is a fixed constant above/below the level below/above it, as follows:

$$R_n = R_1 + I * (n-1) \qquad \text{for } n = 1, 2, \ldots, l. \tag{6.1}$$

PNs enter the system at the lowest pay rate (R_1); a node's pay rate increases as it demonstrates stable consistent contributions to the collective. If a node contributes successfully to collective for T_{raise} consecutive time periods, its pay rate is increased. Periods during which no work is scheduled on a node are not counted for this calculation. The number of levels (l), initial pay rate (R_1), pay rate increment (I), and effort needed to warrant a raise (T_{raise}) are configurable parameters for a given service, and are dependent on the profit margins of the service.

To discourage cheating, the system can apply a pay cut when it identifies a node mis-reporting the amount of work it performs. When such an offense is detected, the PN's pay rate is reduced by the amount of pay increases that would normally accrue for T_{cut} steps (periods) of useful work. Typically T_{cut} is a multiple of T_{raise} (i.e., $T_{cut} = o * T_{raise}$ where $o \geq 1$), so pay is dropped by some configurable number of pay levels. The size of the pay cut (T_{cut}) can be configured on a per-service basis, depending upon the criticality of the offense committed.

We can represent a PN's pay rate at any time t as $R(t)$:

$$R(t) = R_1 + I * \frac{t1}{T_{raise}} - N_{detected} * I * \frac{T_{cut}}{T_{raise}}$$

Here $t1$ represents the number of timeslots where some useful work was performed or claimed to have been performed and the lie went undetected. After time $t1$, a node will receive $t1/T_{raise}$ pay increases, each worth I. $N_{detected}$ represents the number of detected offenses; each such offense leads to a decrease in pay rate equivalent to T_{cut} steps.

6.2 Evaluating the Incentive Model

Let us use this model to analyze the accumulated payoffs for different node profiles to understand how our mechanisms affect node behavior.

6.2.1 Short-Lived vs Long-Lived Nodes

To analyze the difference between short-lived and long-lived players, we plot the average pay rate received by different honest nodes of similar capabilities with different active life times in the system. We assume $R_1 = 1$, a sample increment of 0.2 with 10 levels and $T_{raise} = 7$ (e.g., 7 days). Figure 6.2 plots the average payrate vs the life of a player in the system. This graph clearly shows that patient long-lived players gain clear advantage over short-lived players.

6.2.2 Deterring Cheating Behavior

A rational node will cheat only if the gain from cheating is more than that of honest behavior. Earlier we discussed the expected gain for a single unit of the work. Here we discuss the expected gain for a series of interactions.

A PN that performs work on behalf of a collective can complete only a limited amount of work per time unit given its available resources. In comparison, a cheating PN can fake the completion of an almost unbounded amount of work, irrespective of its resource capabilities. In this section we analyze the expected accumulated gain of a node over a period of time of n time periods, e.g., n days (*we use summation of*

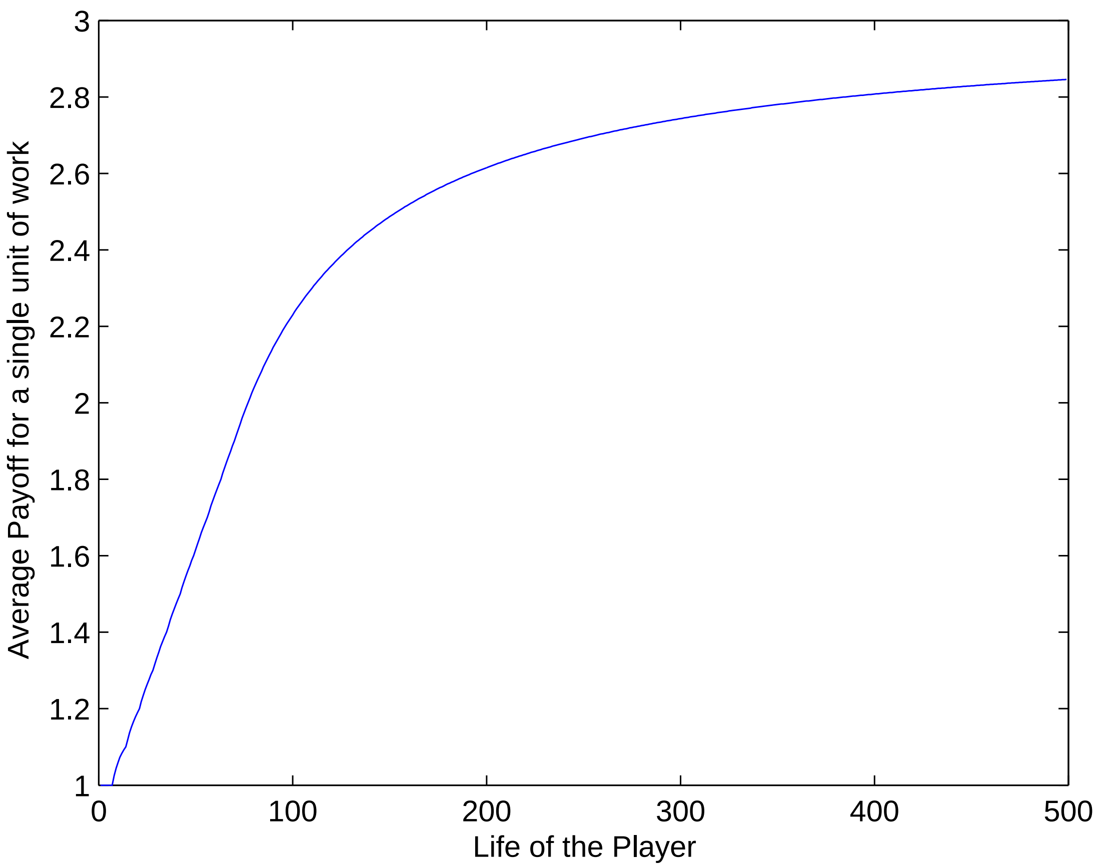

Figure 6.2: Short-Lived vs Long-Lived Player

gain over t from 1 to n to show this). We consider two scenarios. In the first scenario nodes behave honestly, while in the second scenario nodes claim to complete more work than they really performed (i.e., they cheat).

Let G_{honest} be the expected gain of behaving honestly and G_{cheat} be the expected gain of cheating. If G_{cheat} is more than G_{honest}, then a rational node will always take the cheating route to maximize its gain. We can represent the difference between G_{cheat} and G_{honest} by D:

$$D = G_{cheat} - G_{honest}$$

To remain effective in the face of cheating nodes, D should be less than zero in our system.

We can divide a node's offenses[1] (lies about work done) into two categories, detected offenses and undetected offenses. As explained in previous sections, a detected offense not only leads to a fine but also impacts a node's pay rate. Here we analyze the accumulated reward of a cheating node over a period of time to understand the long-term impact of cheating.

We first consider the case of perfect monitoring where every offense is successfully detected by the CM. Since every offense is detected, a cheater will suffer a penalty for every offense.

$$\begin{aligned} G_{honest-perfect} &= \sum_{t=1}^{n} N_{actual} * R(t) \\ G_{cheat-perfect} &= \sum_{t=1}^{n} N_{actual} * R(t) - N_{off} * F \end{aligned}$$

Here N_{actual} represents the number of units of work per unit of time that the node can perform given its available resources, and N_{off} represents the number of units of work faked by a cheating node.

If the cheating node commits one (detected) offense in every time slot, it will always be paid at or below the base pay rate, R_1. Effectively,

$$G_{cheat-perfect} \leq \sum_{t=1}^{n} N_{actual} * R_1 - N_{off} * F,$$

where F is the fine levied by a CM upon detecting an offense. In this case, $G_{cheat-perfect} < G_{honest-perfect}$, so cheating is not economically attractive. Even when nodes only cheat once in a while, the fine and lower pay rate lead to less net income than honest nodes, which is unsurprising given the assumption of perfect monitoring.

In case of *imperfect monitoring*, the system does not detect all offenses. Let p_o be the probability that a offense is successfully detected by the CM. In this case, the accumulated gain over a period of time depends upon the distribution over time of offenses performed by the node.

[1] We use the term *offense* to denote instances when a node attempts to cheat the system. This choice of terms is motivated by the fact that the following analysis is derived from the game theory associated with criminal law, where offenses refer to crimes [7, 71].

We first consider a case where a node performs N_{off} offenses during every time period (e.g., every day). Given N_{off} offenses in a time period, each having a probability of detection of p_o, we can represent the probability of all offenses going undetected by p_{ndo}. p_{ndo} is the cumulative probability that none of N_{off} offenses is detected, which is $(1-p_o)^{N_{off}}$. Given p_{ndo}, we can estimate the pay rate at any time interval using the following equation:

$$\begin{aligned} p_{ndo} &= (1-p_o)^{N_{off}} \\ R(t) &= R(t-1) - p_o * N_{off} * o * I + p_{ndo} * \frac{I}{T_{raise}} \end{aligned}$$

Here $p_o * N_{off}$ represents the expected number of detected offenses. Each detected offense leads to pay rate cut equivalent to $o * I$. Note that $R(t)$ is capped at R_l.

We can represent the accumulated gain of a cheating node, G_{cheat}, as follows:

$$\begin{aligned} SU &= (1-p_o) * N_{off} + N_{actual} \\ G_{cheat} &= \sum_{t=1}^{n} SU * R(t) - p_o * N_{off} * F \end{aligned}$$

Here SU represents the number of units of work successfully billed by a PN, which includes both real work and undetected falsely claimed work. F represents the fine for a detected offense, which we represent as a multiple of the equivalent reward for performing a unit of work, i.e., $F = b * R(t)$ where $b > 0$

To visualize the implication of these equations, let us consider the case of accumulated gain over a period of 25 days, where each time slot is one day. We use $R_1 = 0.1$ (initial pay rate), a sample increment of 0.02 (pay increase per day of sustained honest operation) with 10 levels, $T_{raise} = 7$, $T_{cut} = 7$, $N_{actual} = 50$, and $F = 1 * R(t)$. In Figure 6.3 we plot D, the difference between the accumulated gain of cheating and genuine node as we vary N_{off} from 1 to 25.

The results make clear that increasing the probability of detecting offenses leads to a very sharp decrease in the value of D. If p_o is 0 meaning no offenses are ever detected, D is positive and increases with each offense, so nodes are motivated to cheat. However, D quickly becomes negative for p_o values of 0.05, 0.1 and 0.15. Thus rational nodes will determine that it is in their own best interest to not cheat even

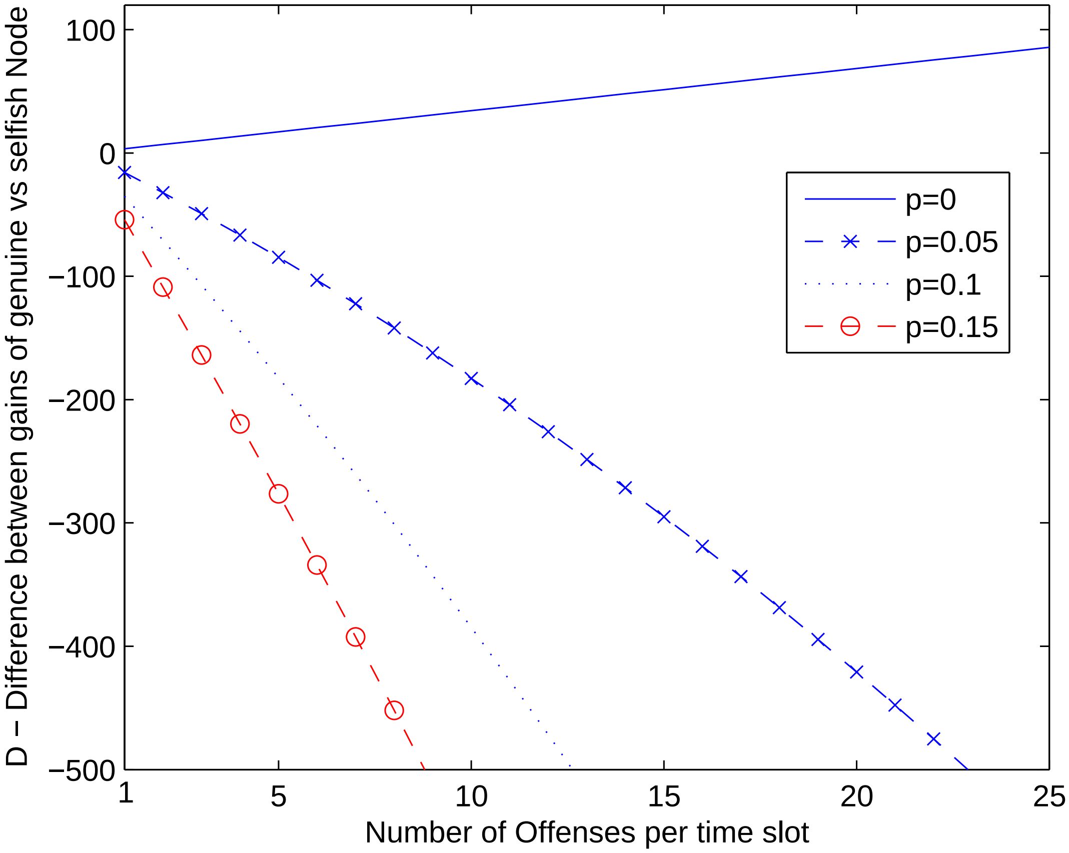

Figure 6.3: Expected Value of D for Different Number of Offenses

when the chance of being caught is small, and the disincentive to cheat increases as the number of offenses increases.

As an alternative to fining and reducing the pay rate of PNs when the CM catches them lying about work performed, we could simply ban users found to commit an offense. Our approach warns misbehaving nodes to mend their ways, and rational nodes will realize that there is no benefit from cheating and cooperate. If individual nodes persist in misbehaving, their pay rate will soon turn negative (due to cuts), which for practical purposes is as effective as banning the node.

6.2.2.1 Worst Case Analysis

In this section we investigate the maximum expected benefit (D_{max}) that a dishonest node can gain from cheating. We show that the maximum expected benefit (D_{max}) will be negative if the probability of catching offenses (p_o) is greater than the value given below.

$$p_o > \frac{R_l}{R_l(b+1) + \frac{o(o+1)}{2}T_{raise} * N_{actual} * I}$$

Here o represents the ratio of pay cut rate to pay raise rate. That is, $T_{cut} = o * T_{raise}$. In other words, being caught cheating reduces a PN's pay rate by the equivalent of o pay raises.

Going ahead, we first provide a proof for the above equation and then analyze the significance of this result. Table 6.5 shows a glossary of mathematical symbols used in this section.

We can calculate the expected gain from cheating by finding three values: (i) the expected payoff from undetected cheating (G_{ucheat}), (ii) the expected loss due to fines for detected cheating (L_{fines}), and (iii) the expected losses accrued from receiving a pay cut due to detected cheating (L_{paycut}).

$$G_{cheat} = G_{ucheat} - L_{fines} - L_{paycut}$$

Now for $D = G_{cheat} - G_{honest}$ and $G_{honest} \geq 0$, we can say that:

$$D_{max} \quad = \quad G_{cheat} = G_{ucheat} - L_{fines} - L_{paycut}$$

Given a particular fine, $F = b * R(t)$, we can estimate G_{ucheat} and L_{fines} using the following equations:

$$\begin{aligned} G_{ucheat} &= \sum_{t=1}^{n} N_{off}(t) * (1 - p_o) * R(t) \\ L_{fines} &= \sum_{t=1}^{n} N_{off}(t) * p_o * b * R(t) \\ G_{ucheat} - L_{fines} &= \sum_{t=1}^{n} R(t) N_{off}(t) * (1 - (b+1)p_o) \end{aligned}$$

Table 6.5: Glossary of mathematical symbols used

p_o	Probability of detecting offenses
$R(t)$	Pay Rate at time period t
b	fine ratio, $Fine = b * R(t)$
N_{off}	Number of offenses per time period
N_{actual}	Number of work units that can be completed by an honest node per time period
T_{raise}	Time periods required for a pay raise
T_{cut}	Time period equivalent to a pay rate cut for an offense
o	Ratio of pay cut rate to pay raise rate ($T_{cut} = o * T_{raise}$)
I	Pay raise increment
N_{total}	Total number of offenses committed = $\sum_{t=1}^{n} N_{\text{off}}(t)$
l	Number of Levels (max pay rate = R_l)

Here there are two cases - (i) $(1-(b+1)p_o) < 0$ and (ii) $(1-(b+1)p_o) \geq 0$. If the first case is true, then D_{max} is bound to be negative. That is given $p_o > 1/(b+1)$, the maximum expected benefit for a cheater is bound to be negative. This is exactly similar to the fixed pay rate model derived in Section 6.1.2.

For the second case, i.e., $p_o \leq 1/(b+1)$, we need to analyze further to see if we can find a better bound on the p_o.

Given the second case and the prior knowledge that the maximum value of R(t) is R_l, we can refine our estimate as follows:

$$\begin{aligned} G_{ucheat} - L_{fines} &\leq \sum_{t=1}^{n} R_l N_{off}(t) * (1-(b+1)p_o) \\ G_{ucheat} - L_{fines} &\leq R_l * (1-(b+1)p_o) * \sum_{t=1}^{n} N_{off}(t) \\ G_{ucheat} - L_{fines} &\leq R_l * (1-(b+1)p_o) N_{total} \end{aligned}$$

Here N_{total} is the total number of offenses over the time period and $N_{off}(t)$ represents the number of units of work faked by a cheating node for time slot t .

When a PN is caught cheating, its pay rate is decreased in addition to it receiving a fine, which decreases how much it receives for work it actually performs. Since the max pay rate is capped at R_l, the impact of a pay cut persists only until a PN's pay rate recovers to R_l, which occurs if it is honest or not caught cheating for a period of time. Thus, the impact of pay cuts is minimized when pay raises are frequent. If we assume that all cheating occurs when a PN's pay rate is R_l, we can calculate the minimum loss induced by being caught cheating.

Assume that cheaters receive a pay rate cut of $T_{cut} = o * T_{raise}$. In other words, being caught cheating reduces a PN's pay rate by the equivalent of o pay raises. In this case, we can calculate the loss a PN suffers due to the decreased pay rate from a single detected cheating event ($L_{s-paycut}$) as follows:

$$\begin{aligned}
L_{s-paycut} &\geq \sum_{k=1}^{o} N_{actual} * PayRateCut(k) \\
L_{s-paycut} &\geq \sum_{k=1}^{o} N_{actual} * k * T_{raise} * I \\
L_{s-paycut} &\geq \frac{o(o+1)}{2} * T_{raise} * N_{actual} * I
\end{aligned}$$

The total number of expected detected offenses can be calculated as $p_o * N_{total}$. Using this, we can refine the previous equation to find L_{paycut}:

$$L_{paycut} \geq \frac{o(o+1)}{2} * T_{raise} * N_{actual} * I * p_o * N_{total}$$

This lets us calculate D_{max} as follows:

$$\begin{aligned}
D_{max} &= G_{ucheat} - L_{fines} - L_{paycut} \\
D_{max} &\leq R_l * (1 - (b+1)p_o)N_{total} \\
&\quad - \tfrac{o(o+1)}{2} T_{raise} * N_{actual} * I * p_o * N_{total}
\end{aligned}$$

A rational node is motivated to cheat only if the gain from cheating is more than the gain from behaving honestly. For our variable pay system to deter cheating, we should select system parameters to ensure that D_{max} is negative. Using the above equation, we can determine what conditions are necessary for D_{max} to be negative as follows:

$$p_o > \frac{R_l}{R_l(b+1) + \frac{o(o+1)}{2} T_{raise} * N_{actual} * I}$$

At first glance, this formula might appear complicated, but we can gain some intuition by solving it for a sample case. If we use the same parameters that were used for Figure 6.3 (1-day time slots, a pay scale with 10 levels that increases 20% per

$T_{raise} = 7$ days, a pay decrease when caught cheating equal to $T_{cut} = 7$ days worth of raises, $N_{actual} = 50$, $R_1 = 0.1$, $I = 0.02$, and $b = 1$), we need only detect cheaters with a probability p_o greater than $\frac{3}{76} = 0.0395$ (roughly 4.0%). This probability remains unchanged for different values of R_1 as long as pay raise increment (I) is 20% of R_1. In contrast, if we assess fines, but not pay decreases, when a PN is caught cheating (the fixed pay rate model derived in Section 6.1.2), the probability p_o of catching a cheater must be greater than 0.5 to build an effective deterrent. Thus, varying pay based on longevity and honesty is an important feature for our incentive model.

6.2.2.2 System Tuning

Even if we are unable to identify a cheating PN, the CM can obtain an estimate of the frequency of cheating in the system using service-level information. For example, in a collective content distribution system, clients will retry unsuccessful downloads using a different PN, which will lead to multiple PNs requesting credit for same work if the first failure was due to a cheating PN. If a CM observes a particular frequency of undetected cheating, it can tune the parameters used to calculate pay rates and fines (e.g., the fine ratio b, the pay cut ratio o, the rate of pay increases T_{raise}, and the pay rate increment I) to maintain an acceptable profit margin.

6.3 Related Work

We have designed our incentive system based on game theory and the economic theory behind law enforcement that motivates just these behaviors. In 1968, Becker [7] presented an economic model of criminal behavior where actors compare the expected costs and expected benefits of offending, and only commit crimes when the expected gains exceed the expected costs. Since then there has been significant research extending the work of Becker. Polinsky et al. [71] provide a comprehensive overview of the research dealing with deterrents in law enforcement.

Many recent projects [57, 80] have applied game theory techniques to build incentives models based on bartering. These projects model nodes as rational self-interested parties similar to us.

Currencies have been used extensively in the systems community in various contexts [90, 12]. Recent projects [87, 81] have used currencies to handle the problem of free riding in peer to peer systems. None of these incentives techniques, however, address the issue of motivating nodes to stay in the system for extended durations. Also these projects do not provide any mechanisms for deterring selfishness in presence of undetected offenses in the system.

Numerous research and industry projects have used reputation based mechanisms to deal with the untrusted users [40, 49, 21]. With our variable pay rate system, we do have something similar to a reputation in our system, although our reputation is built on offline data analysis of past transactions instead of potentially unreliable peer feedback as evident in reputation based systems.

We can also compare a collective to the formation of organizations/firms in real life [19]. Similar to employees in firms, PNs in a collective need to be motivated to do better work and demotivated from shirking away from work.

The collective credit system is similar to Google adsense program [33] that allows webpage publishers to display ads on their sites and earn money.

6.4 Summary

Our analysis focused on two important challenges: (a) ensuring prolonged participation by nodes in the collective and (b) discouraging dishonest behavior. An analysis of the economic underpinnings of the system allowed us to gain important insights into the likely behavior of different players in the system, which we used to derive an incentive model that achieves our goals.

The most important contribution of this chapter is to demonstrate how a mix of rewards and punishments can be used to successfully motivate PNs to behave in ways that benefit the collective. We have also shown that a real system can sustain profitability even in presence of undetected offenses or deviations from the desired behavior, as long as we are able to detect as few as 4%-5% of dishonest behaviors.

CHAPTER 7

INFORMATION AWARE SCHEDULING

End-nodes go up and down in unpredictable ways and have different compute and communication capabilities. Due to this unreliability and the heterogeneity of end-nodes' resources, it is difficult to provide consistent service to clients. However, this problem can be made tractable by understanding and exploiting the underlying diversity to our advantage.

In this chapter, we focus on three important challenges: 1) unreliability inherent in the system due to node churn, 2) resource heterogeneity among participating nodes, and 3) changing resource demands of a service. We describe and analyze how a collective system addresses these challenges using a mix of information aware and adaptive strategies. We use a discrete event based simulator to investigate different options and present our results in this chapter.

Our analysis in this chapter leads us to the following insights regarding how to design an efficient scheduling system for a collective. First, analysis of nodes' past behavior can provide interesting clues to the scheduler that it can use to improve overall performance. Nodes' historic availability, bandwidth, and network location can be used as an important aid for scheduling decisions. Second, live observation of system behavior can be used to adapt services dynamically, which is important to ensure good service performance.

We do not focus on detailed scheduling algorithms in this chapter as that is not the focus of this thesis. Instead we focus on the unique opportunities made possible by collective model and their contribution in improving service performance and in ensuring effective resource utilization. These contributions need to be viewed in the context of resource scheduling strategies used in existing systems built for exploiting idle resources of end-nodes. For example, systems like bitTorrent or Kazaa do not even have the capability to schedule job at will on participating nodes. In these P2P systems, the resource scheduling is done as a side-effect of participating nodes' users personal choices, e.g., user A is interested in downloading movie 'foo' and so on. In contrast, a collective system provides a platform where scheduling can be based on service requirements, participating nodes' resource capabilities, and an informed understanding of historical and live system behavior.

This chapter is organized as follows. We describe a information aware scheduling system for collectives in Section 7.1. We then describe our trace-based evaluation environment in Section 7.2. We follow that up with simulation-based experiments that analyze the impact of history-aware scheduling strategies in Section 7.3. We then analyze different reactive scheduling strategies and their impact on performance in Section 7.4. We discuss network-aware scheduling strategies in Section 7.5 and effective utilization of end-nodes' resources in Section 7.6. Finally in Section 7.7 we summarize.

7.1 Overview

The goal of information aware scheduling is to utilize the diverse information available in the system for efficient scheduling of resources. Given a large enough set of nodes, different nodes bring different capabilities with them. Some have high bandwidth pipes, whereas others have significant available storage. Similarly different nodes stay online or offline in different ways. An intelligent understanding of each node's behavior and capabilities can be used strategically to mitigate the problems that arises from the unreliability and heterogeneity of end-nodes.

Our first level of defense against node churning is the collective incentive model. As described in Chapter 6, the collective incentive model employs a variable pay rate system to reward consistency of nodes' contribution. This motivates nodes to stay in the system for prolonged durations. Thus, compared to traditional p2p systems, nodes in a collective system have clear incentive to stay online for long durations and infrequent churning. In a typical p2p system, most node churning occurs due to short-lived participants who are active only to download a particular song and then go offline. Thus we expect a more stable environment in a collective system than is typical for a traditional p2p system.

Information aware scheduling provides the next level of defense against the unreliability in the system. Two core concepts here are *information and adaptation*. That is, a collective manager tracks significant historic and live information, e.g., participating nodes' resource profiles, their past performance, service demand rates, dishonest behaviors, etc. This diverse information is then used to make informed scheduling decisions, i.e., the system is tuned according to its own inherent characteristics. We believe that such an approach is crucial in exploiting the inherent diversity in a collective to its advantage. The core of information aware scheduling is built on top of following ideas:

- **Division and Replication:** The basic scheduling approach of a collective manager is based on *division and replication*. A service is divided into multiple components and then these components are replicated on multiple participating nodes in the system to achieve the desired behavior. For example, in CCDS a given content is divided into multiple chunks and these chunks are then replicated on one or more nodes.

- **History-Aware Scheduling**: The collective scheduler keeps track of each participating node's past performance records (e.g., upload bandwidth and online-offline patterns). This information is used to make informed scheduling decisions. For example, a given degree of required availability can be achieved with fewer of nodes if they stay online consistently.

- **Reactive Scheduling:** Reactive scheduling focuses on taking immediate actions based on observed dynamic behaviors. These actions lead to adaptive scheduling of resources to fit changing circumstances. For example, a reactive scheduler tracks client demand rate of a cached content and increases or decreases the degree of caching of the content accordingly to fit the service needs.

- **Network-Aware Scheduling:** Network-aware scheduling focuses on utilizing knowledge about the network characteristics of participating nodes to improve scheduling. For example, given a client request for an object, we can fulfill it by selecting a random replica or we can select a replica based on the network location of replicas with respect to the client.

- **Multiple Tasks on a Node:** Assigning only a single task on a node leads to underutilization of node's resources if that task is not active all the time or if the task require only a subset of available resources. Thus in a collective, we assign multiple tasks on a participating node. The collective manager provides a high level guidance to a PN which in turn does local scheduling to share available resources between multiple competing services.

We explore above methodologies in details in the rest of this chapter.

7.2 Simulation Environment

We use a custom discrete event simulator to analyze the behavior of a collective system under different node churning patterns. We primarily use two different traces of real-life systems as a representative environment for evaluation.

Our first trace is based on a Microsoft PC trace taken by Bolosky et al. [11]. For this trace, 51662 desktop PCs within Microsoft corporation were pinged every hour for 35 days beginning July 6, 1999. We use the first 10000 nodes from this data set as our first trace. Our second trace is based on a Skype superPeers trace provided by Guha et al. [39]. For this trace, a set of 2182 nodes participating in the Skype superpeer network were pinged (at the application-level) every 30 minutes for one month beginning September 12, 2005.

Apart from above traces, we also use synthetic traces of node churn to help make some underlying trends more visible.

7.2.1 Performance metrics

We use different performance metrics for different experiments, but there are two metrics that are used repeatedly: availability and available bandwidth.

7.2.1.1 Availability

Availability is a measure of how many replicas of a particular data object are active at any instant of time and whether it is possible to retrieve the complete data object from the collective overlay or not.

In our system, a typical piece of content is divided into multiple chunks that are distributed across different nodes. Because individual nodes exhibit different churn and failure patterns, we get different availability (i.e., number of accessible replicas) for different chunks at any instant of time. The availability of a piece of content is calculated as the number of accessible replicas for its least available chunk. For example, if a piece of content is divided into four chunks with the first chunk having three accessible replicas, the second and third chunk having five accessible replicas, and the fourth chunk having two replicas, the content availability is two. If the availability drops below one, then at that instant of time it is not possible to download a complete copy of the content from the overlay.

7.2.1.2 Available Bandwidth

Available bandwidth measures the total achievable bandwidth for the delivery of a particular piece of content at any instant of time.

7.3 History Aware Scheduling

History-aware scheduling utilizes historical data available from diverse sources in the system to understand the underlying system characteristics. For example, an analysis of participating nodes' past availability and resource profiles (e.g., network bandwidth records) provides important clues to the scheduler that can be used for making informed scheduling decisions.

As an example, Figure 7.1 shows the total number of active nodes in the system over the entire 35 days of the Microsoft PC trace. One can easily make a few important observations from this graph. First, the number of active nodes at a time never drops below 6466. Second, one can observe a periodic dip in the number of available nodes. For example, the large dips occur every 7 days and persist for around 2 days and then rise for around 5 days. This weekly cycle seems to coincide with weekends when many

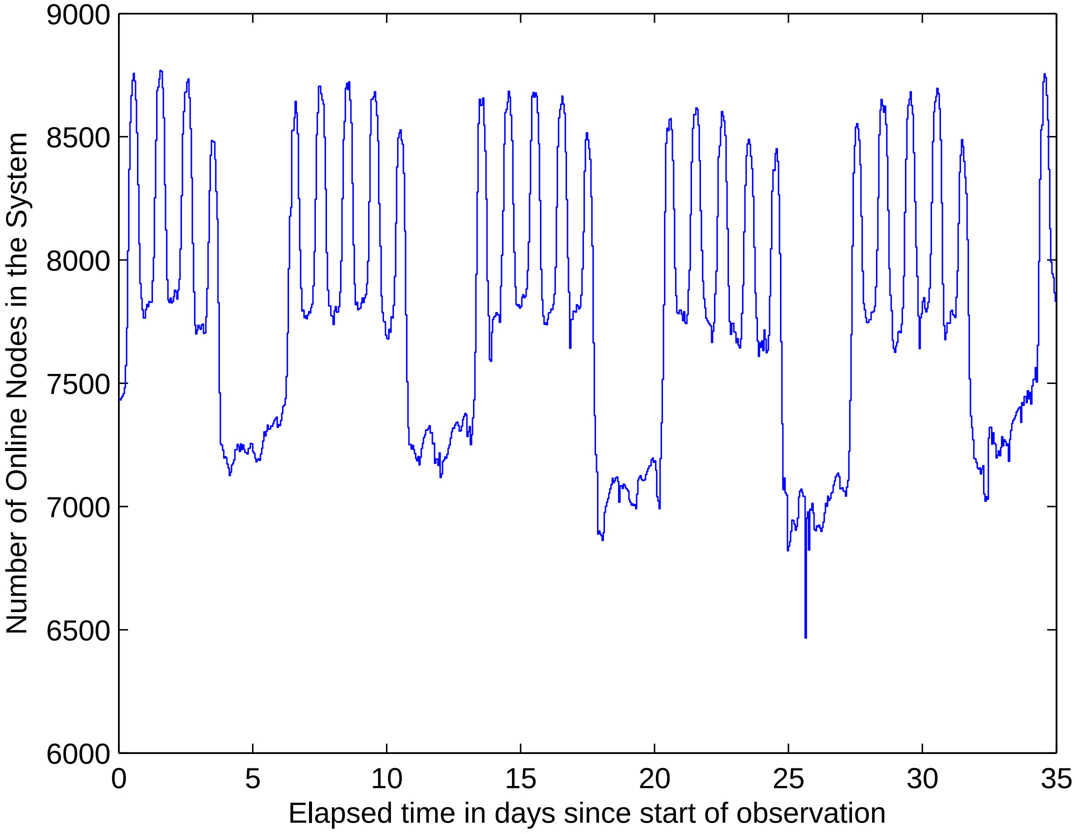

Figure 7.1: Online Nodes in the System vs Time (Microsoft Trace)

employees may be switching off their computers. Similarly, Figure 7.2 shows a similar graph of number of active nodes for the 2081-node Skype Superpeer Network [39]. Here the number of active nodes at any moment in time is always less than 935 (out of a total of 2081) and remains more than 530 for most of the time. Understanding such phenomena is crucial for the CM as it leads to a realistic understanding of aggregate system capabilities and can be utilized to make informed scheduling decisions.

In this section, we experimentally evaluate different scenarios that depict the opportunity to make better scheduling decisions by analyzing historical data. We focus on three categories of scheduling strategies that utilize history information. We start with availability aware scheduling in subsection 7.3.1. We then present bandwidth

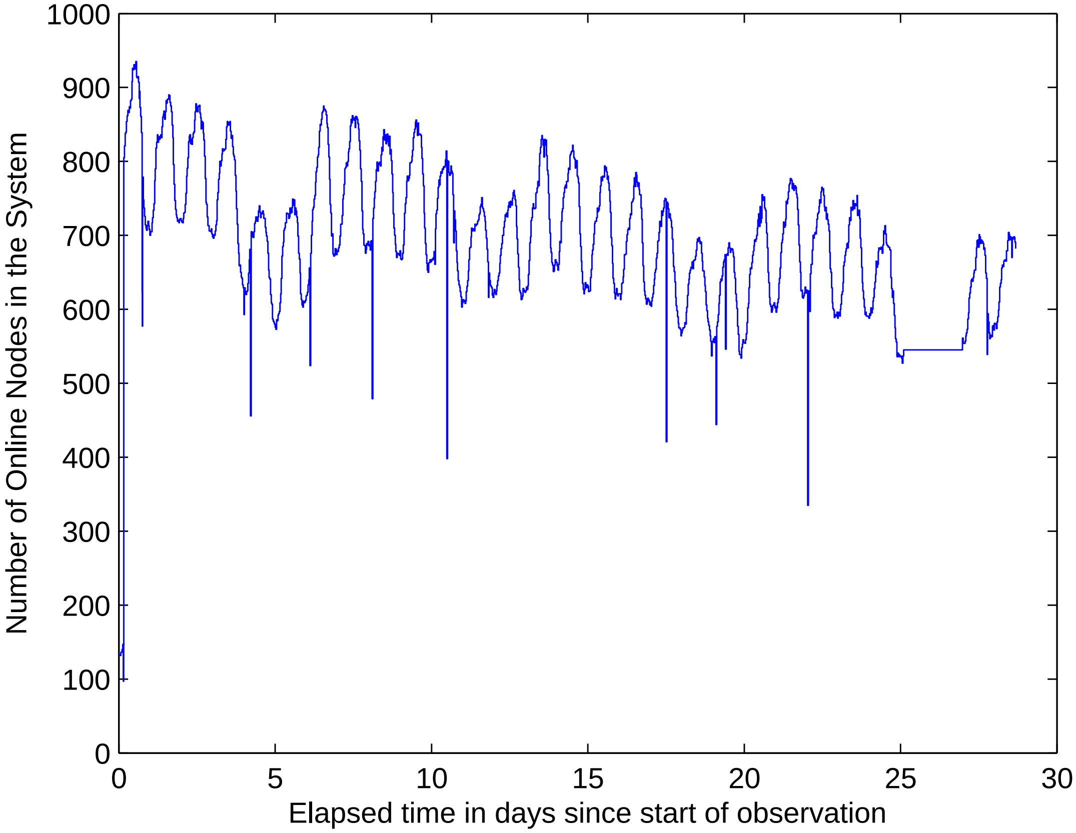

Figure 7.2: Online Nodes in the System vs Time (Skype Trace)

aware scheduling in subsection 7.3.2. Finally, we discuss shift work scheduling in subsection 7.3.3.

7.3.1 Availability Aware Scheduling

A system made up of end-nodes distributed across the Internet experiences node churn that affects the performance of distributed services built on top of it. An *Availability aware scheduling* strategy provides a mechanism to manage this problem.

Given a task, the collective manager divides the task into multiple components and then schedules these components on a set of participating nodes. An easy way to select this set is to choose nodes randomly from the available pool. However, a better outcome can be achieved if node selection is based on historic availability information. For example, a given level of availability can be achieved with fewer nodes if the selected set of nodes remains online for most of the time.

To explore the impact of node churn, we simulate the life-cycle of a data object cached on nodes selected from Microsoft and Skype Traces. For this experiment, the data object is of size 250 MB, divided into 10 MB chunks (i.e., total 25 chunks), and each chunk is replicated on four nodes. Thus we distribute our data object across 100 different nodes with each node having a chunk of size 10MB.

Ideally if there is no node churn, the availability of each chunk remains at 4. However, in the presence of the node churn, we observe different behaviors depending on the nodes selected and their churning behavior. For our first experiment we evaluate two node selection strategies. In the first strategy, we select nodes randomly from the set of all participating nodes. In the second strategy, we use history information to select nodes. Specifically, we monitor participating nodes' 5 day availability and use that as a metric to do node selection. Nodes with higher availability are preferred over nodes with lower availability (each node though can get only a limited set of tasks and thus only the nodes having empty task slot are considered for the selection).

Figure 7.3 shows the results of this experiment on a 10000-node overlay based on the Microsoft Trace. Here we tabulate the impact of node churn on object availability for three cases – two based on random selection and a third (called *Smart AV*) based on an informed selection of nodes using availability over the past 5 days. In each case, 100 nodes are selected for caching the data object at the start of the sixth day (as the past 5 days worth of availability data is used as history as described above). For the two random selection cases, we use two seeds to select nodes based on a uniform random distribution.

Figure 7.3 shows the cumulative time spent by the data object with a given availability over a period of 30 days. Whenever a node goes down, the availability of chunks stored on it decreases by one. Similarly when a node reboots or rejoins the collective, it resumes serving the previously cached chunk, thus increasing the availability of its cached chunk by one. We define the availability of the object based on the availability of the least available chunk.

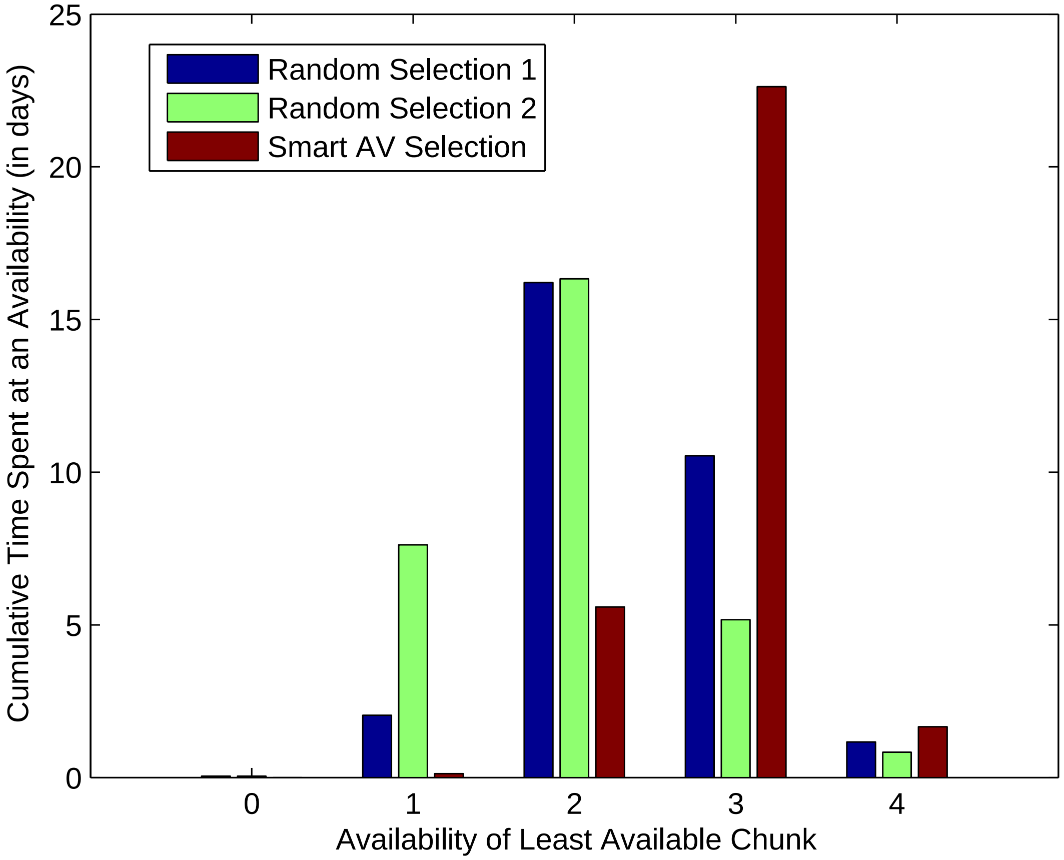

Figure 7.3: Data Availability: Random vs Smart Selection (Microsoft Trace)

Here we can observe that the *Smart AV* selection strategy improves overall availability. With random selection 1, the data object spends around 16 days with an availability of 2 and 10.5 days at an availability of 3. With random selection 2, the data object spends around 16 days with an availability of 2 followed by around 7.6 days with an availability of 1. In comparison with the *Smart AV selection*, the data objects spends around 22.6 days with an availability of 3 followed by around 5.5 days with an availability of 2.

Figure 7.4 shows a similar graph for the Skype trace. Here the results are not as good as for the Microsoft trace. Here also *Smart AV* selection strategy improves

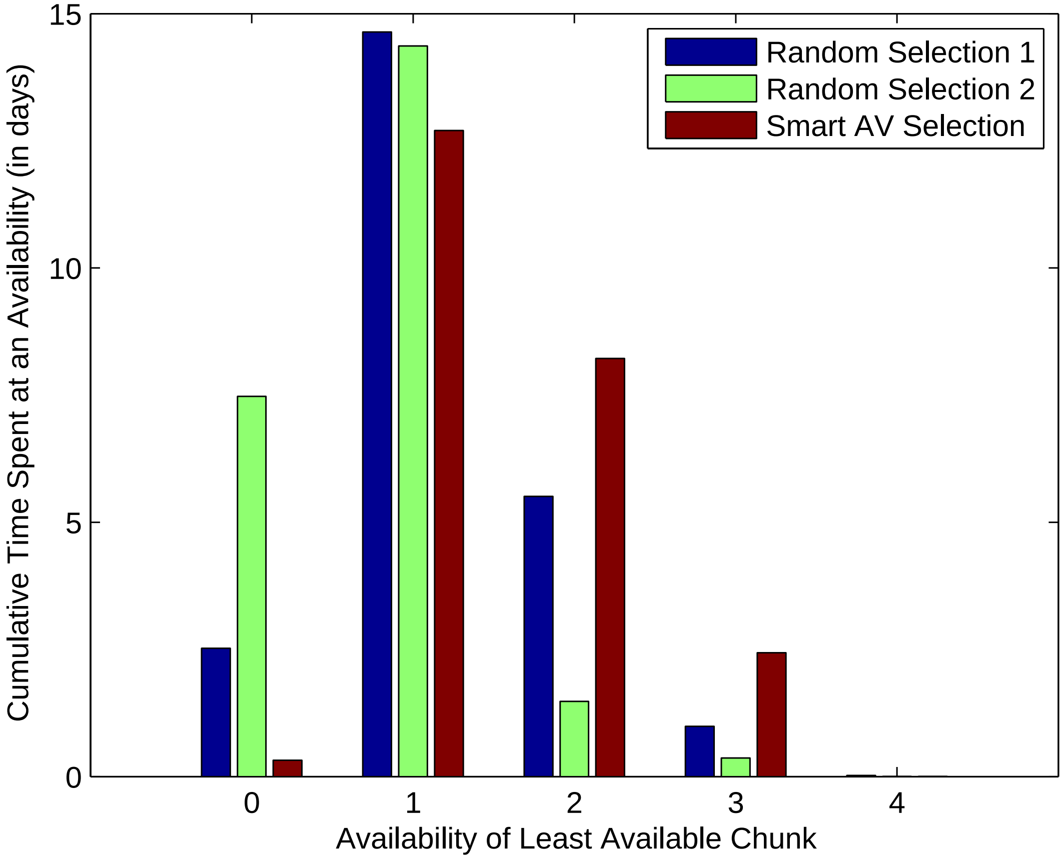

Figure 7.4: Data Availability: Random vs Informed Selection (Skype Trace)

overall availability by decreasing the time spent with availability of 0 or 1 and by increasing the time spent with availability of 2 or 3.

Thus a collective system can handle churn better by using historical data as a guide to select nodes strategically. In practice, we use this strategy in conjunction with other strategies discussed further in this chapter.

7.3.2 Bandwidth Aware Scheduling

Another important metric for node selection is the upload bandwidth rate of different nodes. Individual participating nodes have different upload bandwidth and thus a scheduling strategy that can utilize this knowledge can provide huge benefits.

For example, available bandwidth for a content can be improved by creating a replica on a node with large upload bandwidth.

To better understand the potential benefit of bandwidth aware scheduling, we experiment with Microsoft and Skype traces enhanced with bandwidth data. We assign upload bandwidth for each participating node between 10KBps and 200KBps based on a uniform random distribution. Similar to the experiment described in the previous subsection, we experiment with 250 MB data object that is divided into 25 chunks of 10MB each and each chunk is distributed over four nodes. Thus the content is spread over 100 nodes. Here we compare the performance of three strategies – two random selection strategies and an informed selection strategy called *Smart BW*. In the *Smart BW* selection strategy, nodes with higher upload bandwidth are preferred over nodes with lower upload bandwidth. Figure 7.5 shows the availability of the least available chunk for these three strategies. Here we observe that nodes selected using the *smartBW* selection strategy provides more or less similar availability pattern as that of random selection strategies. However, the available bandwidth differs substantially among the different strategies. Figure 7.6 shows the available bandwidth for the least available chunk averaged over the experiment duration. The least available chunk is the chunk with the minimum available bandwidth among all chunks at any instant of time. Here the *Smart BW* strategy is the clear winner in terms of providing more bandwidth. *SmartBW* achieves more than a 2x improvement in average bandwidth available in comparison with both random selection strategies. Figure 7.7 shows the similar graph for the Skype trace. Here also the *Smart BW* strategy shows good improvement over random strategies.

In practice, however, we will have multiple objects cached on a node and the available bandwidth of a node will get divided into multiple objects cached on it.

7.3.2.1 Multiple Tasks on a Node

In a collective, we store multiple data objects on a node to efficiently utilize available resources.

In a simple approach, we can cache a single object per node, thus utilizing the complete bandwidth of a node for a single object. However, this leads to wasted

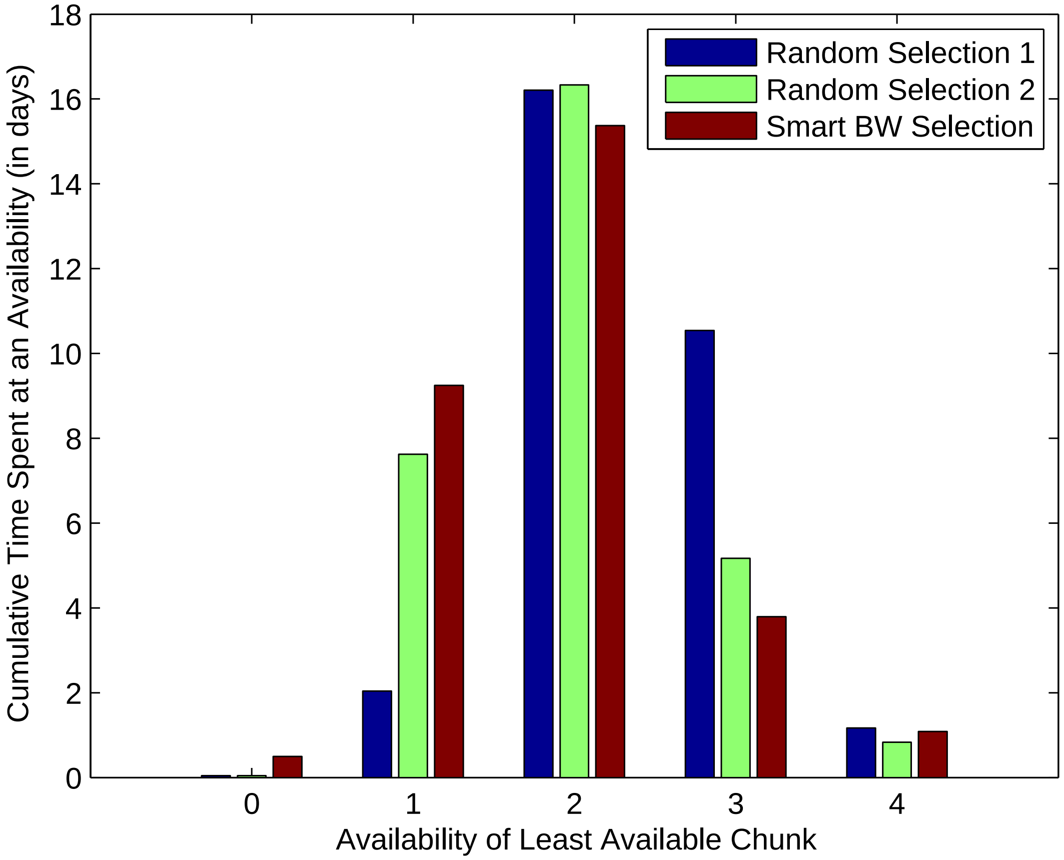

Figure 7.5: Data Availability: Random vs Smart BW Selection (Microsoft Trace)

bandwidth as there are likely to be times when there are no active requests for that object. Also, even if there is an active request, it may not completely utilize the available bandwidth. Putting multiple data objects per node can fix this. Whenever there is no demand for one object, bandwidth can be used for other objects.

Given multiple data objects on a node, the sharing of bandwidth is dependent on actual distribution of requests for different objects cached on a node. When there are active requests for only a single object, it can use 100% available bandwidth for it. When there are requests for more than one object at a time, the available bandwidth is shared among various objects in proportion to their *BW Share*. *BW Share* is assigned per object by the collective manager.

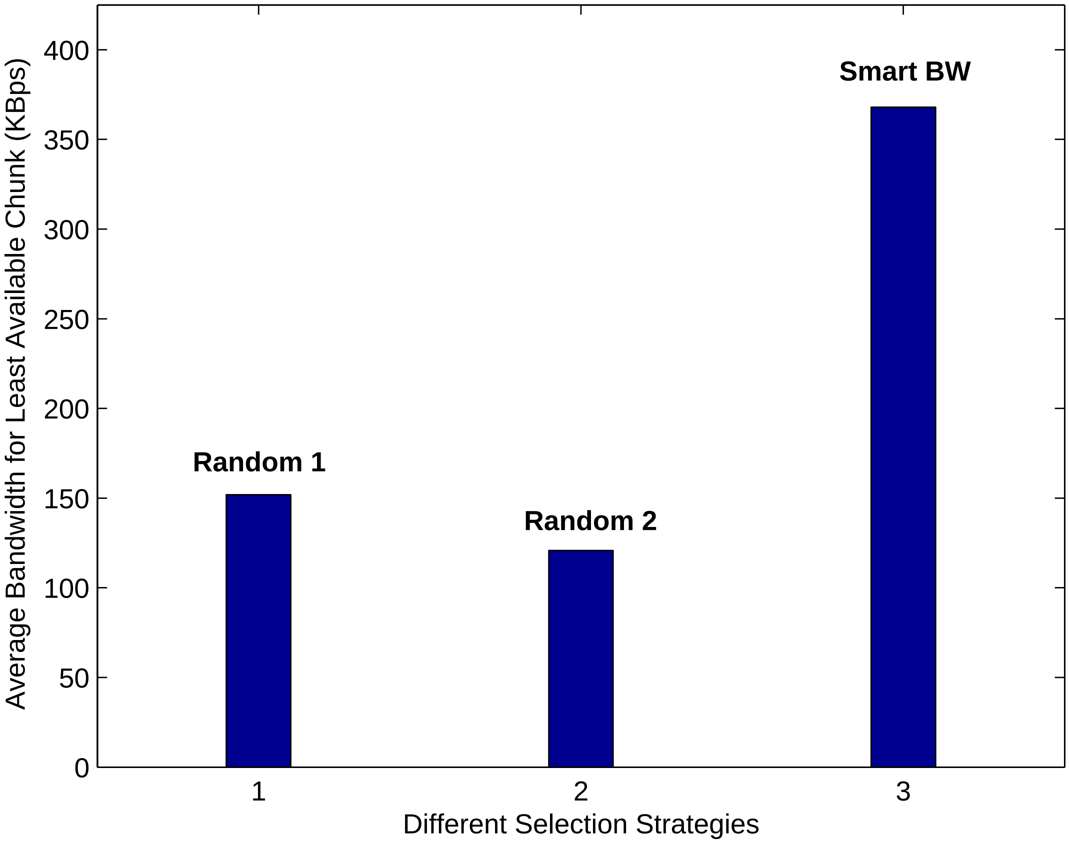

Figure 7.6: Average Available bandwidth: Random vs Smart BW Selection (Microsoft Trace)

To analyze this phenomenon, we explore the impact of multiple data objects cached together on a node. This node has an upload bandwidth of 100KBps and has three data objects cached together called D1, D2, and D3 of size 5MB each. For these experiments, we simulate a 12-hour period. For our first set of experiments, each data object is assigned an equal share of bandwidth (represented as *BW share* of 1 each). We run three experiments – first with client requests at the rate of 15 requests per hour for each object, second with 25 requests per hour, and third with 35 requests per hour for each object. These requests are simulated based on different traces generated according to a Poisson process for each object.

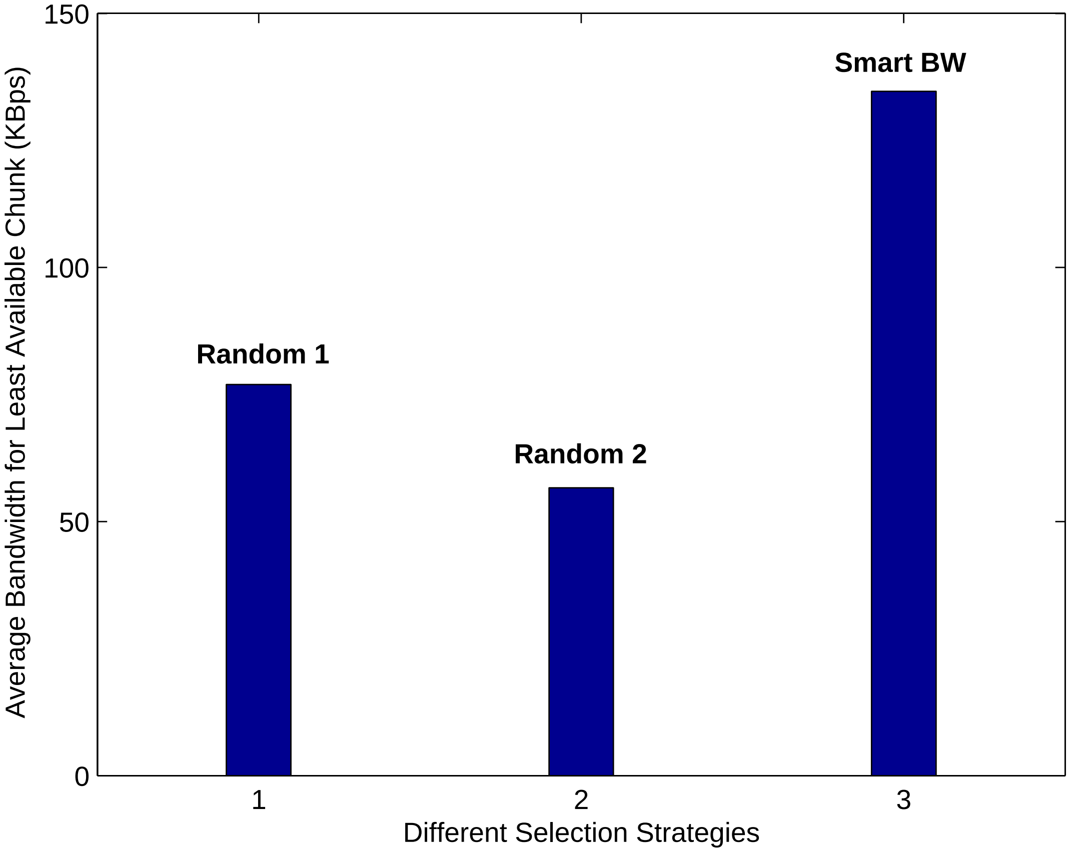

Figure 7.7: Skype Trace: Random vs Smart BW Selection

Figure 7.8 summarizes the results of this experiment. Here we plot the average bandwidth available for each data object over the 12 hour period. To calculate this, we keep track of time periods when there is one or more active requests for a data object and corresponding bandwidth and then average them to calculate average bandwidth. For 15 requests per hour we get an average bandwidth of 61.19 KBps, 61.32 KBps, and 60.56 KBps for D1, D2, and D3, respectively. These numbers are significantly better than the one third share of 100KBps, i.e., 33.33 KBps for each data object. This is because each object is able to utilize a much higher share of the available bandwidth during time-periods when there are active requests only for only one or two data objects. In the ideal case where the active periods of D1, D2, and D3 are disjoint, each object will get the full available bandwidth for itself, i.e., an average

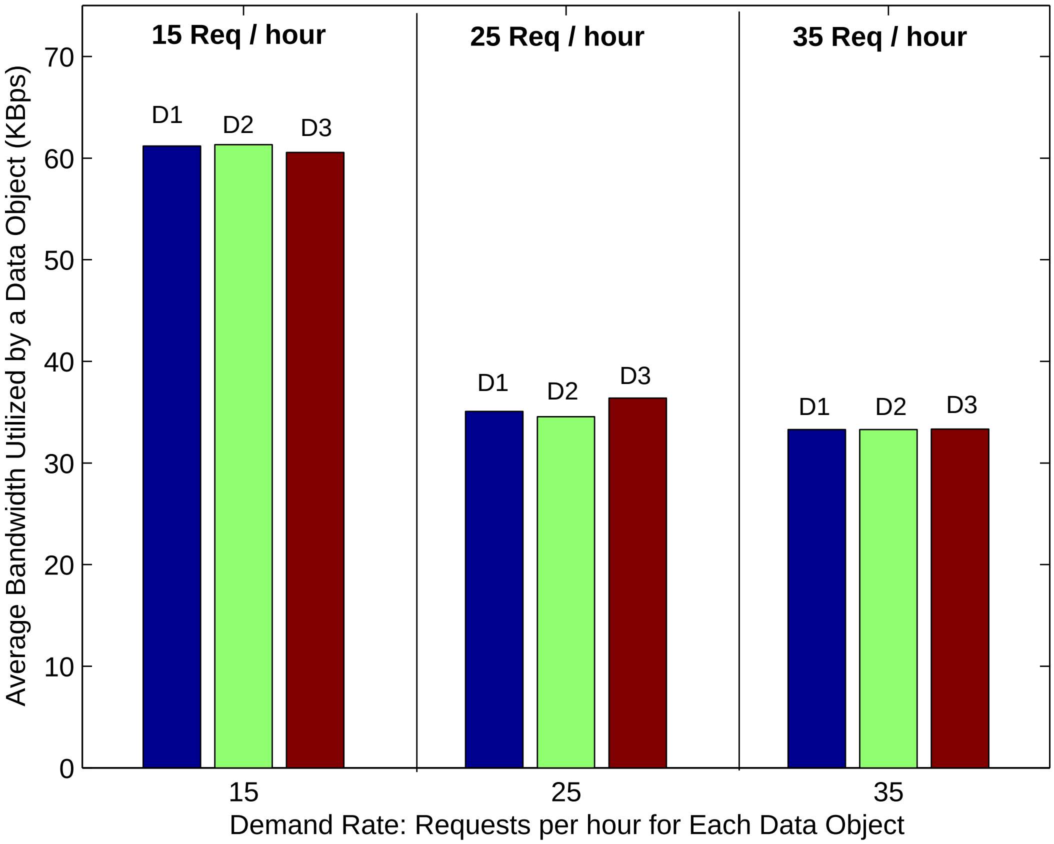

Figure 7.8: Multiple Objects on a Node: Average Bandwidth per Object

bandwidth of 100 KBps each. This shows the value of scheduling multiple data objects on a node. In contrast, for 35 requests per hour we get an average bandwidth of 33.29 KBps, 33.28 KBps, and 33.33 KBps for D1, D2, and D3, respectively. Here the node has active requests for all there objects most of the time; thus bandwidth is shared evenly among the three objects. Figure 7.8 also shows numbers for 25 requests per hour for the comparison.

For our next figure, we repeated the experiment described above with different bandwidth share (represented by *BW Share*) per object. Here D1 has a share of 2, while D2 and D3 have shares of 1 each; i.e., D1 is having a double share of the bandwidth compared to D2 and D3. Figure 7.9 shows the average bandwidth available for each data object over the 12 hour period for this experiment. For 35

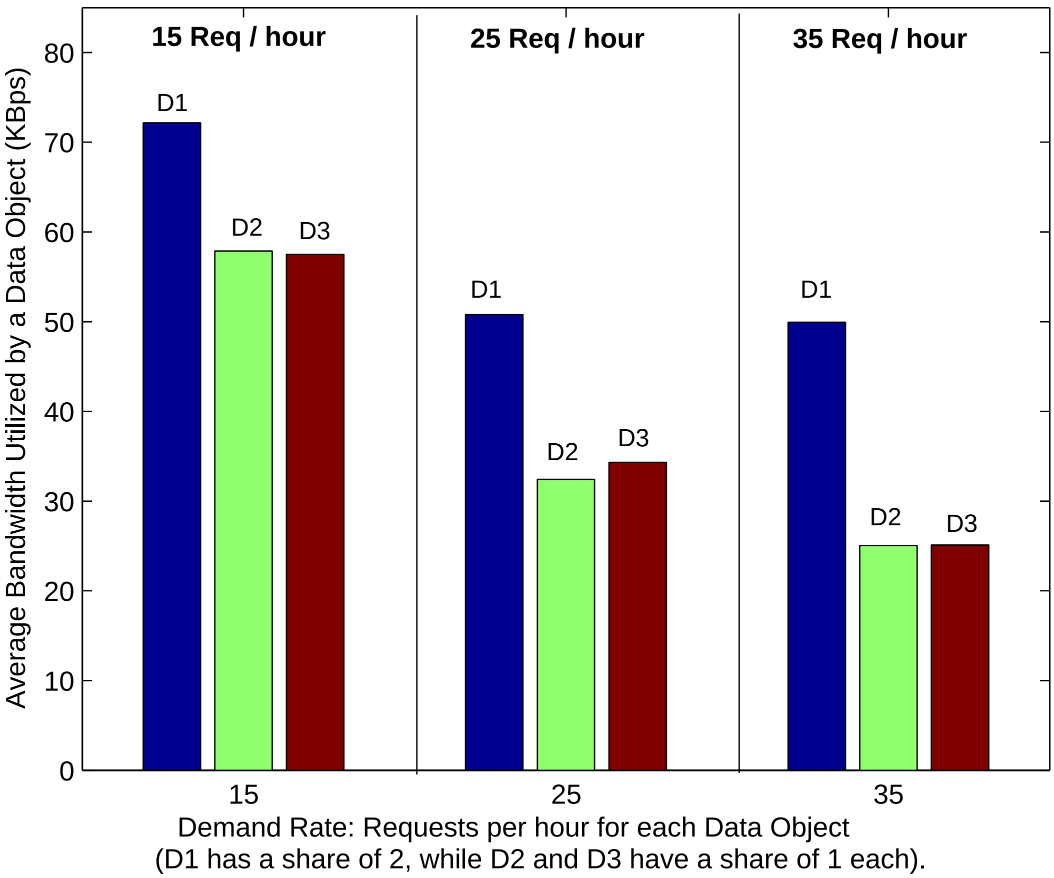

Figure 7.9: Multiple Objects on a Node: Performance with Different Shares

requests per hour, we get an average bandwidth of 49.94 KBps, 25.06 KBps, and 25.11 KBps for D1, D2, and D3 respectively. Here there are active requests for all three objects for most of the time and the available bandwidth is shared in proportion to relative shares of each object correctly. Similarly for 15 requests per hour, we get an average bandwidth of 72.16 KBps, 57.88 KBps, and 57.50 KBps for D1, D2, and D3 respectively. Similar to our earlier experiment, each data object is able to achieve higher average bandwidth due to getting a much higher share of the available bandwidth during less busy time-periods.

Overall, scheduling of multiple tasks provide an opportunity to efficiently utilize the available resources of a node. At the same time, *resource shares* provide finer control that can be used by the CM to allocate more resources to a selected task.

The result shown in the start of the *bandwidth aware selection* subsection (i.e., Figure 7.6 and 7.7) have to be considered in complement with above results. Given an object, the CM can achieve required results by assigning appropriate *BW Share* on selected nodes.

7.3.3 Shift Work Scheduling

Shift Work is a popular employment practice in industry to utilize resources for 24 hours per work day instead of standard working hours of 8-9 hours from morning till evening. As a part of this practice, a day is divided into multiple shifts (typically three shifts of 8 hours each) and each employee works just one of the shift every day.

Our *shift work scheduling* strategy is very similar to this industrial practice. As part of this strategy, a given job is performed by a set of two or more nodes that rotate the responsibility.

This sort of strategy is useful in cases where end nodes might be online only for limited periods of time. For example, some office desktops might be online only from morning to evening while some home PCs might be online only from evening to mid-night. At a glance, such nodes do not appear to be a good fit for providing services as they are offline very often. However, a smart collective manager can identify patterns in nodes' availability and use this knowledge to exploit nodes' availability in a shift work style.

As a case study, we have applied the shift work selection strategy to nodes selected from the Microsoft trace. As discussed earlier, there is a periodic dip in the number of active nodes in the Microsoft trace, as shown in Figure 7.1. These periodic dips occur every 7 days, where the number of available nodes drops drastically for around 2 days during each cycle. One explanation for this phenomenon is that some users may be switching off their desktop PCs over the weekend. This sort of periodicity can be handled effectively by applying *shift work* strategy. As an example, we selected four nodes from the Microsoft trace that were offline in tune with the above pattern during first 7 days of trace (i.e., during the first periodic dip). We replicated a data chunk on these nodes and then measured its availability during the 35 days of trace. Figure 7.10 shows the availability of this data object. We can observe that availability of the data object goes to zero every seven days except during the third 7-day cycle around the 18th-19th day. To handle this availability of zero, we created an additional replica of object on a node that served the object over the 64-hour window every seven days in a time shift manner (the 64-hour period corresponds to 5pm on Friday to 9am on Monday). Figure 7.11 shows the availability of the object after applying this shift work pattern. We can observe that now availability never drops below zero, i.e., 7-day cycle is fixed with the help of a shift work node.

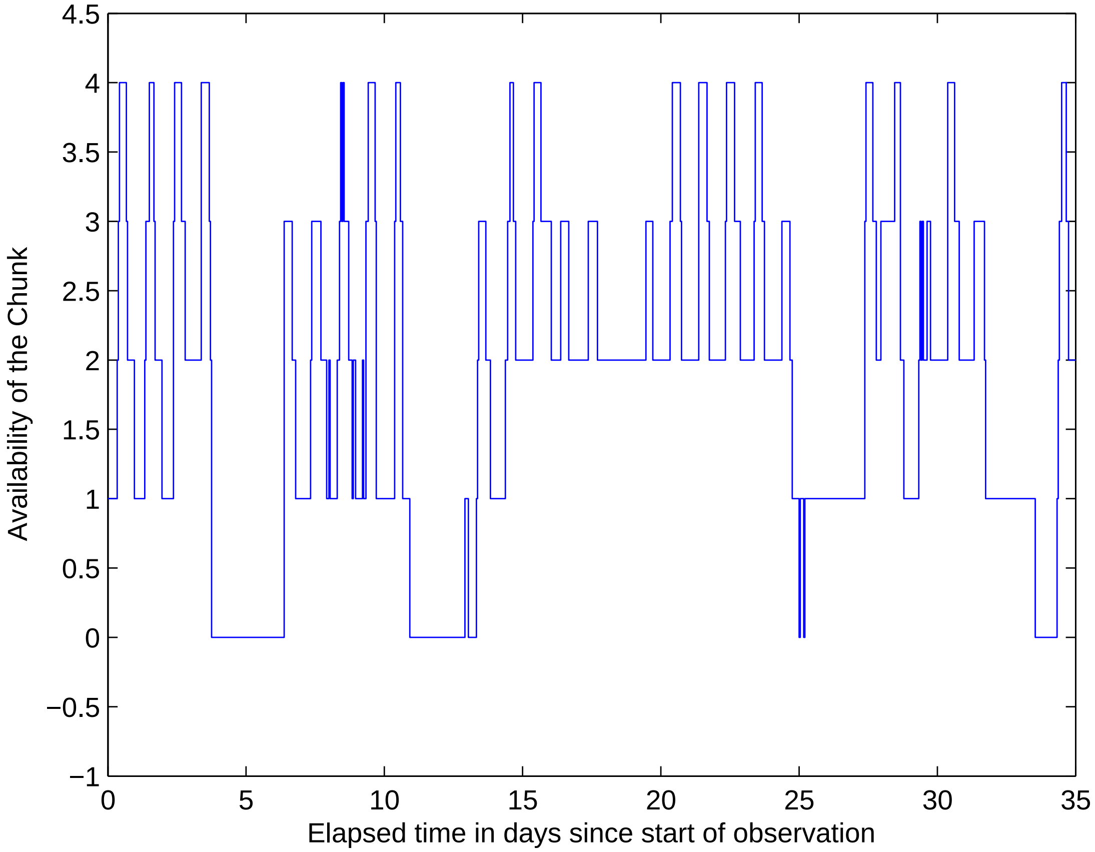

Figure 7.10: Shift Work Strategy: Availability of a 4-way Replicated Chunk (Microsoft Trace)

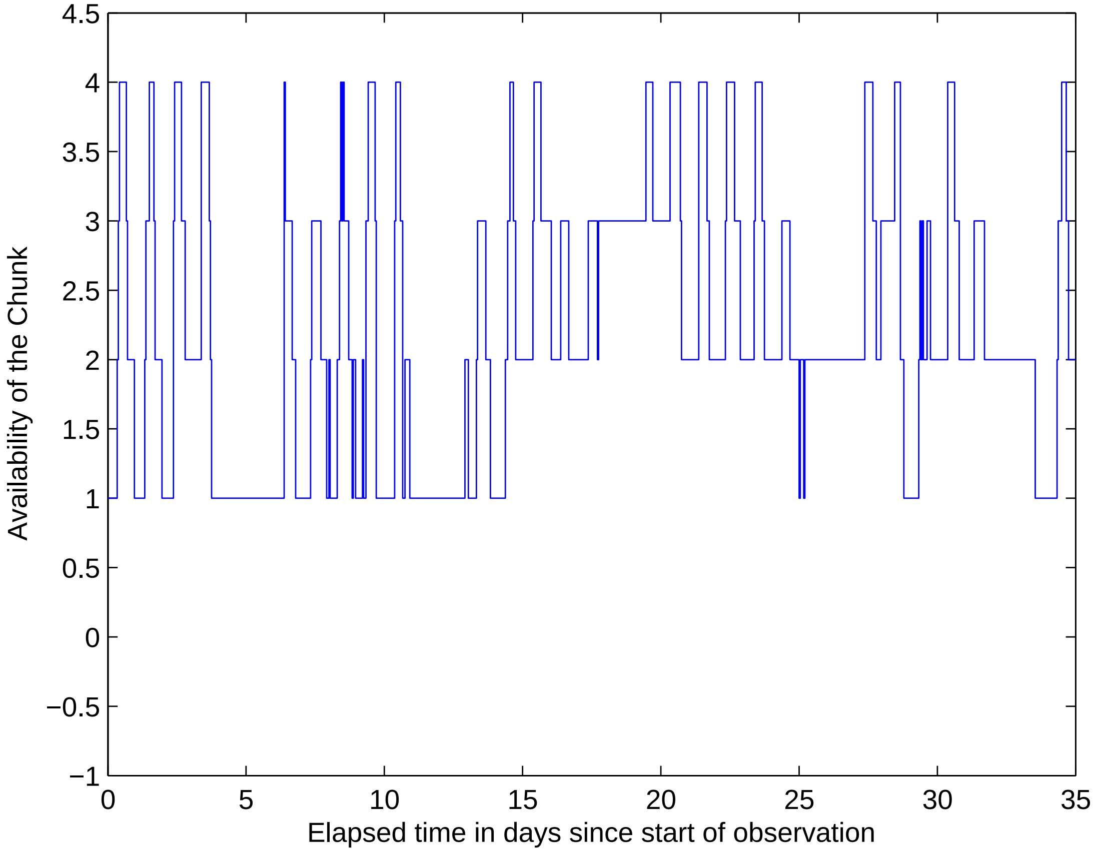

Figure 7.11: Availability with Additional Shift Work strategy (Microsoft Trace)

In summary, the shift work strategy is an excellent way to utilize the diversity of nodes available in a collective. This strategy can exploit the variance in node availability patterns and service demand patterns caused by time differences among nodes in different geographical locations. The success of this approach lies in identification of useful patterns among participating nodes. Luckily, there is a large body of research work in related domains of statistical learning, data mining, and machine learning [9, 58, 93] that can be utilized for this purpose.

7.4 Reactive Scheduling

Reactive scheduling involves taking immediate actions based on observed dynamic behaviors to improve service performance. To achieve this, the scheduling system tracks different performance metrics and triggers changes whenever a tracked metric goes above or below a particular threshold. Two main categories of reactive scheduling are failure-aware scheduling and demand-aware scheduling.

7.4.1 Failure-Aware Scheduling

Failure aware scheduling works by actively tracking node failures in the system and mitigating those failures through corrective actions, e.g., by initiating additional replicas on detecting a node failure. A collective system can use variety of methods to detect failures and node churn as discussed in Section 3.2.2.

Here we analyze the value of reacting to failures in improving the system performance. Our experiment is similar to the one performed in Section 7.3.1 - i.e., we divide a 250 MB data object into 25 chunks of 10MB each and replicate each chunk over four nodes. Our experiment compares the performance of two strategies – *Smart AV* strategy, as defined in Section 7.3.1, and a failure-aware strategy built on top of *Smart AV* strategy.

In this failure-aware strategy we select nodes based on their past 5 day availability. In addition, we detect node churn and initiate corrective action by creating alternative replicas whenever we detect a node failure. To simulate failure detection time, we introduce a delay of 1 hour before we take corrective action after a node goes offline. Figure 7.12 shows the availability of the least available chunk based on these two strategies for the Microsoft trace. We observe that the *failure aware* strategy leads to significant improvement compared to the basic *smart AV* strategy, which itself was a great improvement over random strategies.

Figure 7.13 shows the similar plot for Skype trace. We again observe that failure-aware scheduling leads to significant improvement in availability in comparison to *Smart AV* strategy.

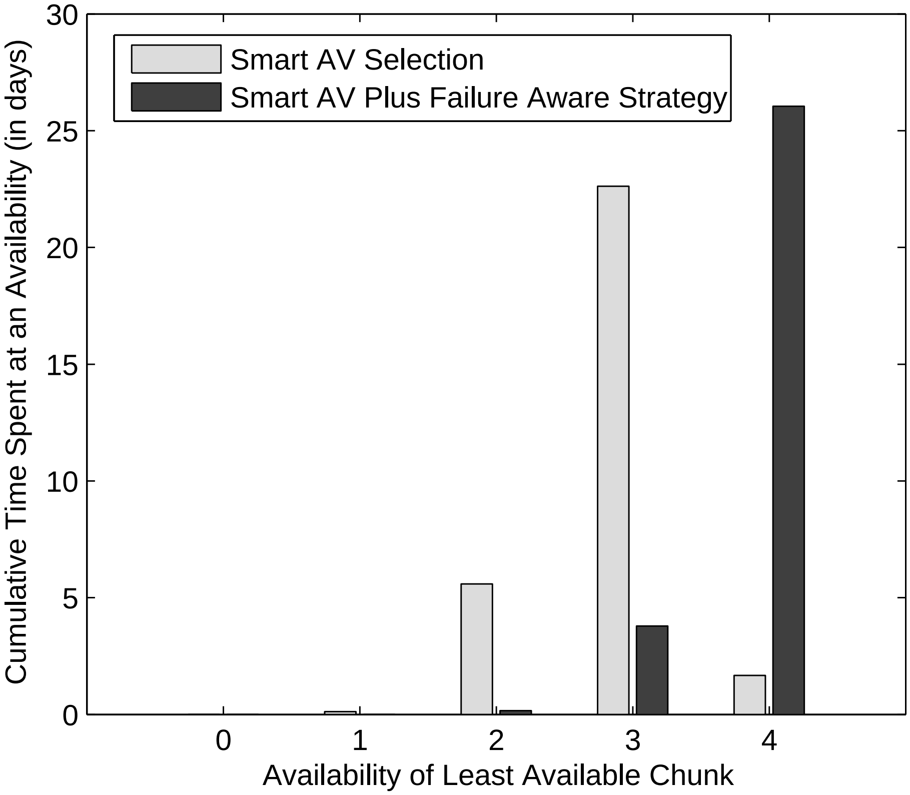

Figure 7.12: Availability Impact of Failure-Aware Scheduling (Microsoft Trace)

Overall, failure-aware scheduling is an important tool in the hand of the scheduler, and along with history-aware scheduling, it provides an efficient mechanism to mitigate the effect of underlying unreliability in the system.

7.4.2 Demand-aware Scheduling

Another important reactive strategy is based on detecting and responding to changes in a service demand rate. For example, in a collective content distribution service (CCDS), whenever the demand for a particular content (e.g., a popular movie) increases, there needs to be a corresponding increase in its degree of replication to meet the increasing demand.

Our demand-aware scheduling algorithm is built on top of a trigger invocation system in the collective manager. Different agents monitor different phenomena.

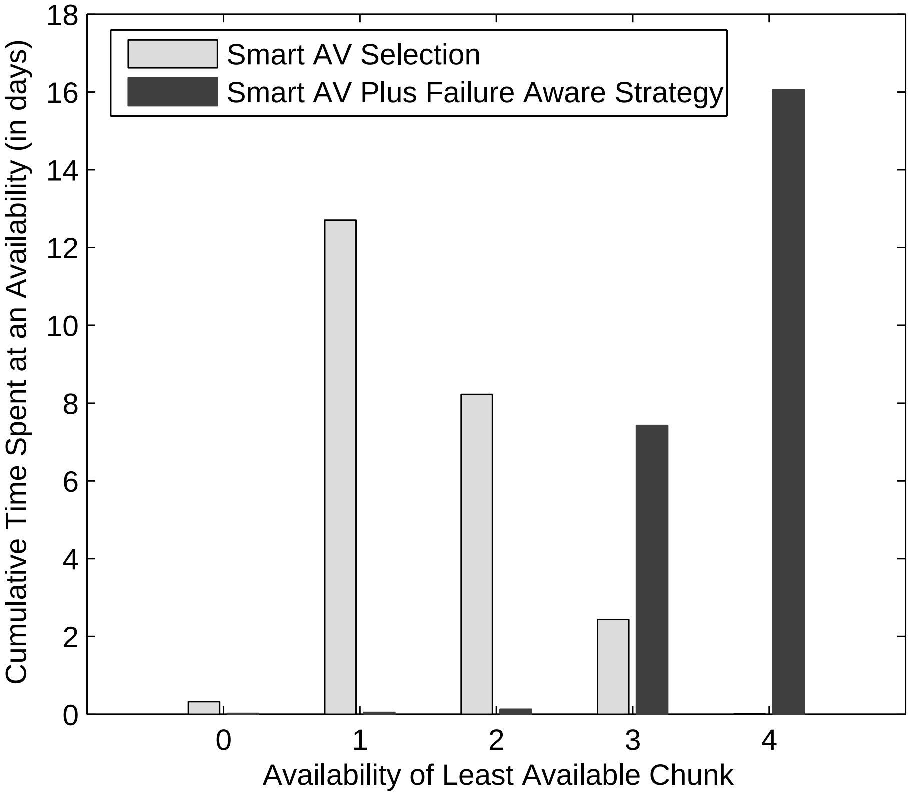

Figure 7.13: Availability Impact of Failure-Aware Scheduling (Skype Trace)

When the agents detect a change greater than a threshold, they trigger an event for the scheduler to act upon. As an example, content distributors in the CCDS service have an accurate understanding of client demand as it is directly related to the content sale rate. Whenever the demand of a content changes above or below a predefined threshold, it triggers an event for the scheduler. The scheduler then acts upon this trigger by increasing or decreasing the caching of the content over the collective overlay accordingly.

To analyze the value of demand-aware scheduling, we experiment with a content caching scenario using the Microsoft trace over a 10-day period. For this experiment, we assign each participating node with an upload bandwidth of 40KBps. Similar to earlier experiments, we divide a 250 MB data object into 25 chunks of 10MB each.

Initially the observed demand for the content is 50 requests per hour, which requires each chunk to be replicated on four nodes; thereby distributing the content over 100 nodes. We then slowly increase the sale rate of the content and at the start of 6th day it crosses 75 requests per hour. At this moment the content distributor triggers an *increased-demand* event to the scheduler. The scheduler receives the event and reacts by increasing the replication of each chunk from 4 to 6 to handle the 50% increase in demand rate. Figure 7.14 compares the average bandwidth available for a chunk for this demand-aware strategy in comparison to based fixed replication strategy. We can observe that the demand-aware strategy leads to an increase in available bandwidth to handle increased load.

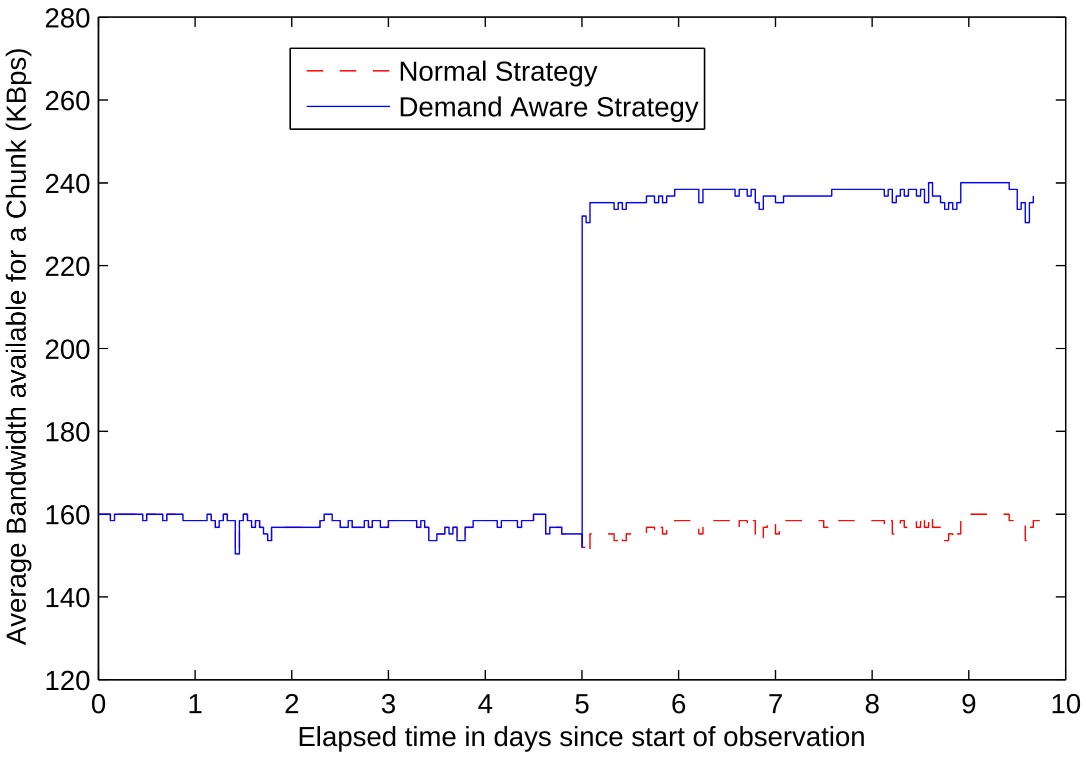

Figure 7.14: Demand-Aware Scheduling: Average Available Bandwidth w.r.t. Time

We perform another experiment for demand-aware scheduling where we emulate multiple clients requesting a data object using emulab network testbed. Details and results of this experiment are discussed in the next chapter (section 8.6).

7.5 Network-Aware Scheduling

Network-aware scheduling focuses on collecting information about the network location of participating nodes and utilizing this information to improve scheduling. For example, given a client request for an object, we can fulfill it by selecting a random replica or we can select a replica based on network topology information (e.g., select a replica near the requester).

Depending upon circumstances and service involved, different aspects of a node's network topology can be important. For a content distribution service, the bandwidth between a client and PNs is more important than latency between them. In contrast, a collective network probing service (discussed in Section 9.2) needs nodes from a specific ISP or at a specific physical location.

A clear picture of network topology can unearth many interesting opportunities. For example, university networks or ISPs like utopia network [86] have intradomain bandwidth that is much higher than their Internet bandwidth. In these situations, selecting a node within the same network can provide much higher bandwidth than selecting a node anywhere else in the Internet. Many research papers [50, 61, 95, 38] have shown that using network-aware approaches can improve application performance substantially.

However, getting an accurate picture of network topology is not straightforward and requires otherwise unnecessary probing traffic. The *P4P project* [95] addresses this problem in an interesting way. In P4P, application developers work in cooperation with Internet service providers (ISPs) to obtain useful network topology and traffic information. In return, they help ISPs ensure efficient network usage, for example, by not overloading a high cost inter-AS link but rather using an alternate low cost path exposed by the ISP. Even though this project is quite new, there is already a consortium of ISPs (e.g., Pando network, Verizon Communications, and Comcast, among others) in its working group [68] and an internet draft [38] was released in October 2008 describing the experiences from a P4P trial on many ISPs.

Overall, network aware scheduling is an important tool in the hands of a collective scheduler. With decent network topology information (either with cooperation from ISP or based on network probing), a network aware scheduler can improve service performance substantially.

7.6 Effective Utilization of PN Resources

Our collective model provides a practical way to efficiently utilize idle resources of end-nodes. There are two main reasons for this: (i) our information-aware scheduling that understand each node's resource capabilities and (ii) the controls made available by a virtual machine centric approach that allows unutilized resources to be used without affecting normal work.

7.6.1 Effective Scheduling of Services

A node's idle resources are perishable — if they are not used, their potential value is lost. In an incentive model that employs bartering, e.g., BitTorrent, nodes typically participate in the system only long enough to perform a particular transaction such as downloading a song. At other times, that node's idle resources are not utilized unless the node's administrator is altruistic. In contrast, in a collective the CM has much larger and more diverse pools of work than personal needs of individual participants; thus a CM is better able to consume the perishable resources of PNs.

In a collective, the CM schedules multiple services with different resource requirements on a node to efficiently utilize its resources. To analyze this phenomenon, let us reconsider the scenario used in Section 7.3.2. Here a node has an upload bandwidth of 100 KBps and can cache one or more objects on it. We compare two scenarios; in the first scenario only a single 5 MB data object is cached on the node, whereas in the second scenario three 5 MB data objects are cached on the node. Each data object receives client requests at a rate of 15 requests per hour These requests are simulated based on traces generated according to a Poisson process for each object for a 12-hour simulation period. Figure 7.15 summarizes the results of this experiment.

In the case where we cache only a single object, only 21.33% of upload bandwidth is utilized . That is, the upload bandwidth of the node is used for only 2.56 hours out of the 12 hours. In comparison, caching three objects on the node at the same time leads to 61.15% utilization of the upload bandwidth. Thus using idle resources to run multiple services improves resource utilization on the node.

Another point to consider is the ability of a CM to understand the unique resource capabilities of individual PNs and utilize them for the best returns. For example, a node may have limited bandwidth capabilities but is always online, while another node may not be consistent in its availability, but have a fat bandwidth pipe. As our collective manager accumulates historical records of every PN, it can easily detect these patterns and utilize them strategically. This sort of smart scheduling not only improves the overall performance of a collective system, but also leads to higher

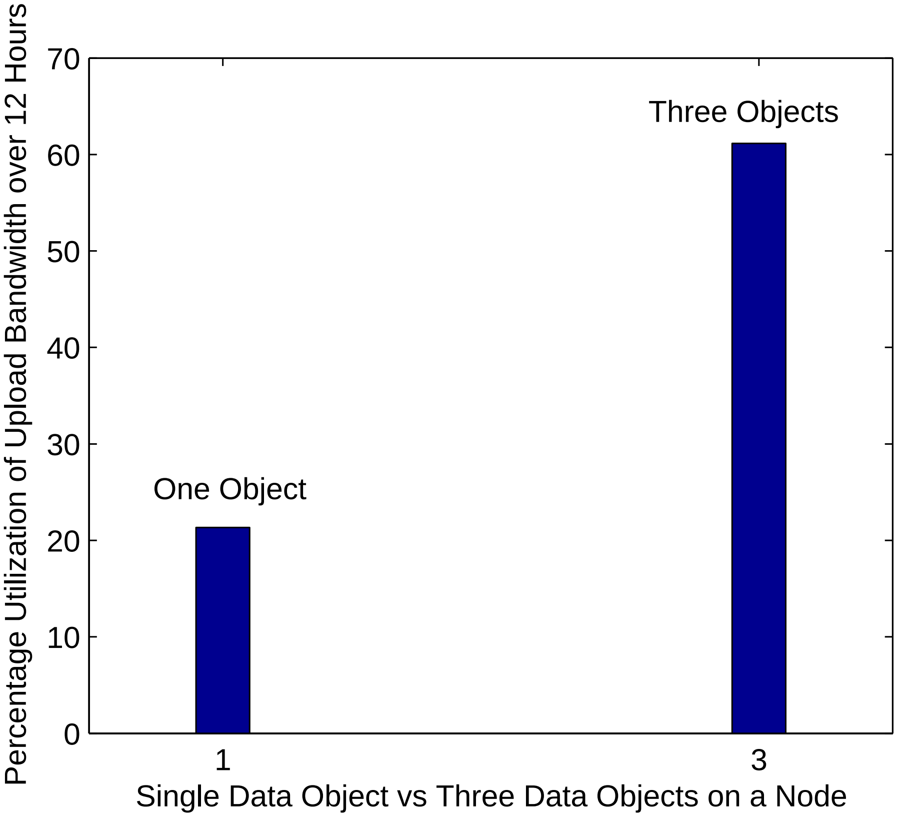

Figure 7.15: Resource Utilization: Multiple Objects vs Single Object on a Node

returns to participating nodes. A scheduling system that allocates all participating nodes same way cannot exploit unique features of each node.

7.6.2 VM Control for Effective Utilization

An easy approach for idle resource utilization is based on a binary decision of a node being used or not by a user. That is, if the software detects any active user logged on the node, it does not run the application. This is a simple approach, but suffers from under-utilization of resources. With modern computers, there are many unutilized resources even when a user is actively using the machine. What we want is a control system that gives preference to local user's resource needs, but allows unutilized resources to be used by the collective.

We achieve this goal with the help of virtual machine technology. The decision to use virtual machine as the unit of resource allocation has several important advantages over alternative approaches. Virtual machine technology allows greater *isolation*, *flexibility*, and *resource control* than running application processes directly on top of a standard Windows or Linux box.

Most importantly, the virtual machine monitor can enforce preferential treatment to local host processes. Dynamic priorities can be set for both cpu cycles and network bandwidth, while static limits can be set for disk quotas and physical memory allocation. Moreover, if the host user wants, he/she can enforce more stronger resource controls for cpu and bandwidth, e.g., predetermined cpu share or upper caps on bandwidth utilization. These controls provide powerful tools in the hands of PN administrators to ensure that normal work can proceed on the PN without undue impact from collective. The protection and resource controls provided by VMs to participating nodes can make people more willing to make their machines accessible to the collective even when they are actively using it.

7.7 Summary

In this chapter, we have shown how the underlying diversity of PNs' resources and availability can be exploited in a simple but useful manner to improve service performance and availability. Idle resources of end-nodes suffer from unreliability and heterogeneity, but amid this chaos lies some order that can be exploited by intelligent resource tracking and scheduling.

The collective's scheduler approach is based on information collection and adaptation. The scheduler collects a wide variety of information from diverse sources in the environment and uses this information intelligently to adapt the system behavior to best suit the underlying environment and service needs. Existing systems built for exploiting end-node's resources (e.g., bitTorrent, kazaa, etc) do not have the capability to schedule tasks at will on participating nodes and hence cannot optimally utilize the unique abilities of each participating node.

We have presented three classes of intelligent scheduling strategies in this chapter: history aware scheduling, reactive scheduling, and network aware scheduling. These scheduling strategies help in improving performance of services built out of end-nodes and can make collectives an important platform for commercial services.

CHAPTER 8

IMPLEMENTATION

In this chapter we present the implementation details of a prototype collective system. We start with a quick overview of the different players in a collective system in Section 8.1. We then describe the underlying authentication framework and interplayer communication mechanism in Section 8.2. We follow this discussion with details of partners and clients in Section 8.3. The implementation of the collective manager and participating nodes are discussed in Sections 8.4 and 8.5, respectively. We then present the results of validation experiments performed at Emulab network emulation testbed [25] in Section 8.6. We then discuss scalability-related issues in Section 8.7 and finally summarize in Section 8.8.

8.1 Players in a Collective

Figure 8.1 shows the high level architecture of a collective. As shown in the figure, a collective system has four main players: a collective manager, participating nodes, partners, and clients.

A *Collective Manager (CM)* is the central hub for a collective system. There is only one collective manager per collective. The CM is deployed on multiple server nodes at a single or potentially multiple sites across the Internet. *Participating nodes*

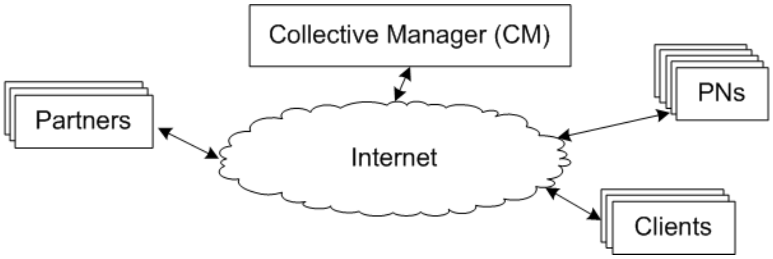

Figure 8.1: Players in a Collective System

are end nodes with idle compute, storage, or network resources that can be exploited by the collective. There can be thousands or millions of independent participating nodes all connected to the Internet through any variety of networking. *Partners* are service collaborators that use the collective overlay to offload all or part of their tasks associated with the service. Partners could be considered to be a front-end for services. For example, in CCDS a content distributor manages content listing and selling itself, but uses the collective overlay to distribute content on its behalf. Partners run their own managed software and interact with the collective manager periodically to offload their tasks. *Clients* are individuals that use a service managed by the collective. Clients use a service-specific application to interact with a service, e.g., a collective backup service (CBS) client uses an application to save files and to view and restore already saved files. Different players can have completely different system environments as long as interplayer interfaces are well defined. For example, the CM, some partners and some clients may have Linux-based system environment, while other PNs and clients may run a Windows-based environment.

For our prototype, we have implemented these players in a Linux-based environment. We use a mixture of C and C++ for most of the implementation and perl for experimentation scripts.

8.2 Authentication and Interplayer Communication

We need a simple but secure mechanism for mutual authentication as well secure communication between players.

The *Authentication* system used in our prototype is based on public key infrastructure (PKI) [66]. PKI is a mechanism that binds the public key of a player with its identity with the help of a certificate authority. This certificate authority role is performed by the CM in our collective, i.e., the CM acts as the root of the PKI employed by a collective. Each player in a collective is identified by a unique machine-generated ID and a unique public-private key pair. These identities are bound together with the help of a certificate generated by the CM that associates the public key of a player with its unique ID. The unique ID, public key and the signed certificate together provide the necessary mechanism for identification and authentication of any player in the system.

Communication between different players in a collective (e.g, between a PN and the CM, or a partner and the CM, or a client and PN) happens over TCP/IP secured with the help of the Transport Layer Security Protocol (TLS) [23, 74]. TLS is a sister protocol of popular SSL protocol and provides the mechanism to build a secure communication channel over TCP/IP. It can provide both privacy and data integrity between two communicating ends. TLS uses PKI system described above for authentication of both communicating ends.

For our implementation, we use the OpenSSL library [67], which provides an open source implementation of the TLS protocol. OpenSSL is written in C and is a widely used TLS/SSL library in the Unix and Linux worlds.

8.3 Partners and Clients

Partners are service collaborators that use the collective overlay to offload part of their tasks. From a client perspective, the service is offered by a partner and they may not even notice the presence of the collective. A partner can be an external entity, e.g., a content distributor that uses the collective for content distribution, or a partner can be an internal entity whose front-end is managed by the collective itself, e.g., the CBS server in the collective backup service (Figure 5.1).

Partners run their own managed software and can develop it in any way they want. They interact with the service manager running as part of the CM for offloading their tasks to the collective overlay. Service managers provide a well-defined API for partners to interact with. For example, a content distributor interacts with corresponding service manager to cache particular content and to pass on information like current demand rate to help in scheduling.

Clients are individuals who wish to utilize the services offered by partners, e.g., downloading a video. Clients use an application that they download from a partner's website to use a service. This application interacts with the corresponding partner for service listing and purchase, but uses the collective overlay for the actual service. Partners need to develop client applications in collaboration with the collective.

8.4 Collective Manager

A collective manager is a resource manager; it aggregates computing resources of thousand or millions of participating nodes and uses them to build useful services. The work of the collective manager can be divided into multiple components. Figure 8.2 shows these components and how they interact. There are six main components. *Service managers* are per-service agents that are responsible for managing all aspects of a service in the system. The *resource scheduler* is the central component that helps allocation of resources between different services. The *node manager*, *report collection engine*, *offline analysis engine*, and *offline TTP system* are support components that help help with the tracking of resource utilization across the collective.

The rest of this section is organized as follows. We start with the data layer and intercomponent communication system in Section 8.4.1. We then describe the resource scheduler in Section 8.4.2 followed by service managers in Section 8.4.3. Finally we discuss different support components in Section 8.4.4.

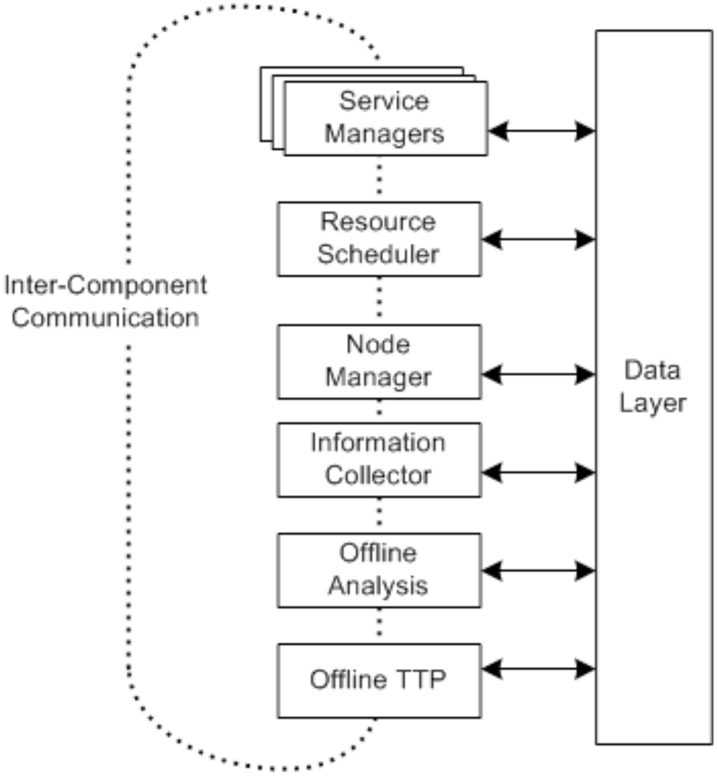

Figure 8.2: Collective Manager Components

8.4.1 Data Layer and Intercomponent Communication

In a collective manager, different components are glued together with the help of a *data layer* and an *intercomponent communication system.* Each component is a separate program and can be instantiated on independent machines or can be co-located with other components.

The data layer is responsible for handling all data needs of different components in a collective manager (CM). A CM needs persistent data storage to store records of participating nodes, past history records, service performance records, offline analysis records, and payment-related records. Our prototype uses a *MySql* database as the back-end for the data layer.

While the data layer provides a mechanism to access data as and when needed by components, a separate system is used to deliver messages to components dynamically. This dynamic message delivery system is based on the *Spread toolkit. Spread* [82] is a group communication toolkit that supports application-level multicast, group communication, and point to point communication. It is resilient to faults across local and wide area networks. Spread has been used for Zope replication service [97] and for the backend of the popular *wordpress.com* website, among others.

We use Spread in the CM as a loosely coupled way to communicate between different components. This loosely coupled communication mechanism is important as the CM has been structured as a collection of replicated components for load management. That is, there can be multiple replicated copies of a single component on multiple server nodes each handling part of the load. A new replica of a component can be initiated dynamically either to replace a failed server node or to handle increased load. In such a system, *Spread* ensures that a message is received by the right receiver without the sender worrying about the location of receiving component.

Every server configured as a part of the CM runs a *Spread* daemon on it. All local components running on a server register with the local *spread* daemon so they can send and receive messages to/from different groups. Different spread daemons interact with each other to manage group memberships and to ensure correct message delivery.

8.4.2 Resource Scheduler

The resource scheduler is responsible for keeping track of all available resources in the system and their allocation across different services. As described in Chapter 7, the collective scheduler is an information-aware system that collects information from all available channels (e.g, the past performance of individual PN, resource profiles of each PN, service demands, etc.) and uses this information to make informed scheduling decisions.

The bulk of this work is performed with the help of an *information collection engine* and an *offline analysis engine* that collect and process information, respectively, for use by the resource scheduler. Processed information is stored in the data layer from where the scheduler can get it whenever it needs. The scheduler also uses the data layer for keeping track of current resource allocation patterns and other state associated with scheduling.

The control flow of a resource scheduler can be seen as a request-response cycle with *service managers*. Service managers are agents responsible for ensuring that a service is running smoothly. They issue requests to the resource scheduler for resource allocation according to their needs and get appropriate resources in return.

8.4.3 Service Managers

Service managers are agents running on the CM that are responsible for the performance of individual services. Service managers are the central hub for a service and need to be designed especially to handle the requirements of a particular service.

A service manager's work starts with a request from a partner. A partner is a front end for a service and interacts with the service manager to offload tasks over the collective overlay. Service manager converts these tasks into small components, and with the help of resource scheduler, allocates the required resources across different PNs. Next, the service manager ensures that the necessary executables (service agent) and data are copied to the selected PNs with help from the *node manager*. If needed, the service manager also sends dynamic instructions to *service agents* to achieve desired results.

Service managers continuously monitor their service's performance, validate it against service goals, and take corrective actions whenever performance degradation happens. In this work, they are aided by the resource scheduler (who helps them to add or remove resources to/from the task whenever needed) and performance monitors.

Performance monitors are agents used to observe different dynamic phenomenon related to collective performance. Many of these monitors are part of existing components, though there can be separate components for the monitoring purpose itself. For example, the *node manager* keeps track of node churning, *partners* keep track of service demand rates, etc. Whenever an observed phenomenon goes above/below a certain threshold, its corresponding performance monitor invokes an event trigger with the help of the *Spread messaging system*. Different service managers register with Spread to listen to these triggers and act upon them when they receive one.

8.4.4 Support Components

These support components help record and track the diverse information available in a collective.

- **Node manager:** A node manager is responsible for shipping necessary executables and data to the PNs based on service manager requests. It achieves this by connecting to the *VM controller* running on participating nodes and providing them necessary links to data that need to be downloaded.

  Another important component of the node manager is the liveness server that keeps track of nodes' liveness and tracks node failures. As described in Section 3.2.2, the liveness server actively listens to graceful join/leave messages sent by PNs as well as indirect messages sent by other players in the system. Based on this information, it updates each node's liveness status stored in the data layer and sends messages to service managers about observed failures using the *spread* messaging system.

- The **information collection engine** is responsible for collection of periodic data sent by different components, partners, and PNs. These data play a crucial part in information aware scheduling as well as in the calculation of credit rewards for different PNs. CM also uses these reports to investigate different dishonest behaviors in the system.

Each set of data received by the information collection engine is accompanied with a message digest digitally signed with the sender private key. The information collection engine authenticates the sender and verifes the data integrity and accompanied digital signature before storing the data into the data layer for future analysis.

- The **Offline data analysis engine** focuses on two important tasks: (1) cross-verification of task completion reports sent by PNs and (2) analysis of past performance records and node resource profiles to help resource scheduler. Offline analysis works on the data available in the data layer.
- The **Offline trusted third party (TTP)** is used whenever a service needs to ensure nonrepudiation for a data transfer, e.g., as described in Section 4.1.4. Our protocol is directly based on offline TTP (trusted third party) protocol described in [55]. Our TTP component is a very simple system that actively listens to any mediator requests, answers them based on the protocol, and saves a record of answered queries in the data layer.

8.5 Participating Nodes

Participating nodes (PNs) are end nodes that provide resources to the collective manager. Figure 8.3 shows the high level architecture of a PN. There are four main components within a PN - a node agent, a virtual machine (VM), a VM controller, and service agents.

- The **node agent** is a small program running on a PN. It is responsible for starting the virtual machine. When a participating node enters a collective, its administrator starts by downloading and installing this *node agent.* The node agent in turn downloads the required VM image and starts the VM. The node agent also provides a convenient user interface to PN administrators.

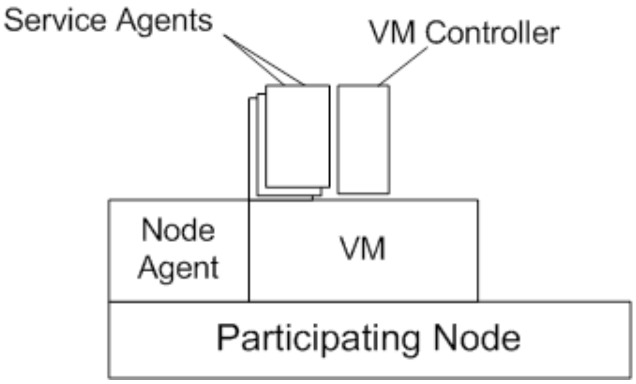

Figure 8.3: Participating Node Architecture

PN administrators can use this interface to pause or stop the virtual machine and can configure the resources available to the virtual machine. The UI also provides a summary of the credits awarded to the PN for completed work.

- **Virtual machine:** A PN provides resources to a CM in the form of a virtual machine (VM). We use Xen as well as the free VMware player to create VMs for our prototype. Our VM uses a customized Linux configuration based on the Damn Small Linux distribution [24] and is less than 50 MB in size. The virtual machine monitor can enforce resource controls (e.g., disk quota, cpu share, and physical memory limit) on the VM. This design allows normal, i.e., noncollective related, work to proceed on a participating node without undue impact by programs running in the VM.

  The VM contains a *VM controller* for managing the VM and *service agents* for managing different services running on the VM.

  The CM has root access to the VM. VM is configured with a password-less *ssh* key based access control mechanism that allows the CM to directly login to the VM. This public key access system is established through the VM disk image shipped to PN when it first becomes part of a collective. Only the CM can login into the VM using the above mentioned method. All communication between a PN and the CM is either layered on *ssh* or uses *TLS* [23] connections to the VM controller.

- **Service agents** are responsible for completing service-specific jobs assigned to a node. They are invoked by the VM controller. They get their commands from *service managers* running in the CM and accordingly perform their tasks.

- The **VM controller** is the main component responsible for overall management of the VM running on a participating node. Its work includes instantiation of different service managers, VM resource management and control, and the VM's interaction with the collective manager.

  To start service agents, the VM controller interacts with the *node manager* running as part of the CM. Based on instructions provided by the *node manager*, the VM controller downloads the programs necessary to run different service agents in the VM. It also ensures that different service agents are instantiated properly and running smoothly.

  The VM controller is also responsible for scheduling available computing resources among competing services. It makes sure that different services get resources according to the high level guidelines provided by the CM. For disk storage this is simple as the CM's guidelines specify the storage allocated for each service. However, for bandwidth, the CM provides guidelines in the form of *BW shares* that VM controller has to enforce dynamically all the time. The VM controller achieves this with the help of the Linux traffic control system [45]. It creates different virtual interfaces for each service and then uses hierarchical token bucket queues [43] to divide the bandwidth between them.

  In addition, the VM controller dynamically monitors the resource usage of the node, e.g., the VM online/offline timings and history of available bandwidth as observed by passive monitoring of network communication. The VM controller also collects the service completion records and any other important data from service agents running on the VM and makes sure that these records and node resource profile records are sent to the CM periodically.

8.6 Validation on Emulab

To provide basic validation of our prototype, we perform a set of experiments using the Emulab network testbed [25]. The goal of these experiments is to demonstrate a few sample scenarios of how a collective system performs on a set of representational nodes.

We model the Internet as a cloud (an opaque network) with the help of the *lan* modeling option of *emulab*. Each node in our experiment is connected to this cloud, with its last hop shaped according to its specific characteristics (described below).

Our first experiment explores a collective content distribution service (aka 'CCDS') that replicates data over a collective overlay. Here we emulate a collective system with 20 participating nodes, each with a download and upload bandwidth of 250KBps and 50 KBps, respectively. These bandwidth numbers depict observed capacity of a typical end user connected to the Internet via a basic Comcast cable broadband service. Each participating node runs the necessary service agent to support this service. When the experiment starts, a content distributor contacts the collective manager and initiates caching of a 50 MB data object over the overlay. Upon getting this request, the collective manager decides the cache distribution pattern for the data object. It divides the object into chunks of 5 MB each (10 chunks total) and then schedules two replicas for each chunk; thus allocating 5MB of data on each of 20 nodes. The collective manager then informs each PN of its specific allocation. PNs then start caching of data from the content distributor. Afterwards when a client requests this data object, its request is fulfilled through the collective overlay instead of the content distributor.

Figure 8.4 shows the download performance as seen by a client. Similar to other nodes in the experiment, the client has a download bandwidth of 250 KBps and upload bandwidth of 50KBps. Here the X-axis represents time and the Y-axis represents the cumulative data downloaded by the client. The client downloads the content simultaneously from all 20 participating nodes with different chunks coming from different nodes. The client takes 291 seconds to download the object, achieving an average download rate of 175.95 KBps.

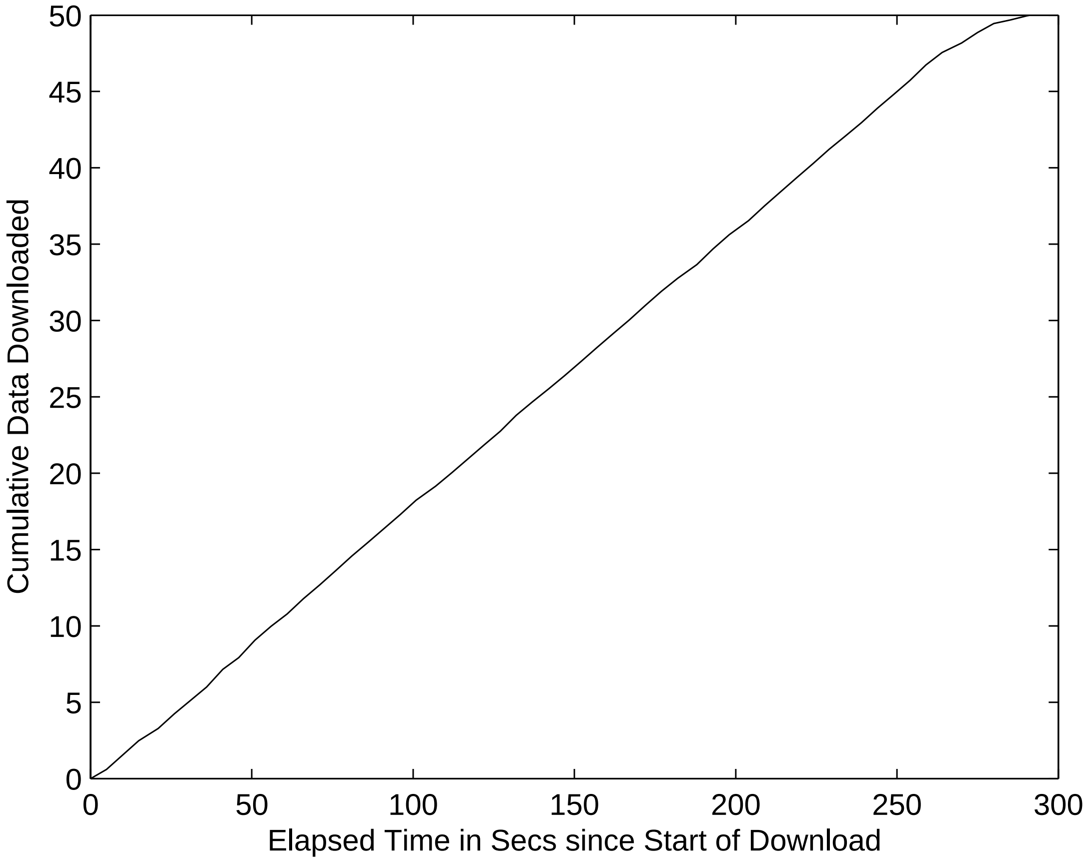

Figure 8.4: Cumulative Data Download vs Time for a Client

For our second experiment, we emulate a continuous stream of client requests over a 30 minutes period. These client requests follow a Poisson distribution with a request rate of 60 requests per hour. Each of our client requests is made from a different client, each having a download bandwidth of 250KBps. Figure 8.5 shows the download performance of all client requests. On the Y-axis we show the download completion time in seconds, while on the X-axis we show the client request number sorted based on request start time. Thirty one client requests are made during the 30 minutes period, though the experiment runs until each of the requests is completed. As time increases, the number of active downloads in the system increases and hence an increase in the completion time in the middle of the graph. After 30 minutes, there

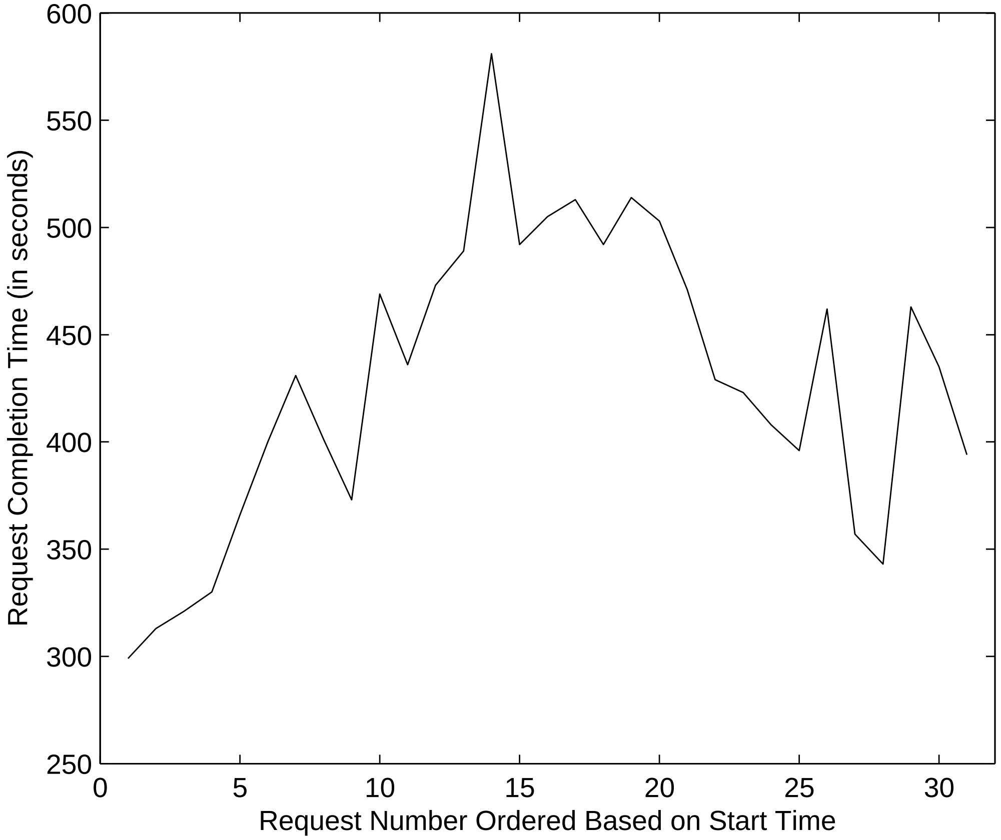

Figure 8.5: Request Completion Time vs Request Number

are no more new requests coming and hence the number of active requests decreases and thus the last bunch of requests finishes in less time.

Our third experiment shows a case of demand-aware scheduling where the system acts on a trigger. Here, during the course of the experiment, the client request rate changes, which triggers a corresponding change in scheduling strategy. During the first half an hour of the experiment, 60 requests per hour are made, similar to the previous experiment. After half an hour, the client request rate is changed to 90 requests per hour. In a production environment, this change in demand will lead to an hint message from the content distributor to the collective manager. To emulate this, we inject a message using the *spread* to inform the CM about the change in demand at 32 minutes into the experiment. The collective manager then adapts by increasing the replication degree to 3, which leads to the addition of 10 more nodes, each caching 5 MB of content.

Figure 8.6 shows the request completion time for this scenario. In this figure we compare the performance to two different scheduling – first where the system adapts dynamically by adding 10 additional nodes and second where no additional nodes are added. Here, similar to Figure 8.5, the Y-axis represents the request completion time while the X-axis represent request number sorted according to request starting time. These requests cover 1-hour period; in the first half hour clients requests are issued at the rate of 60 per hour and and in the second half hour at the rate 90 per hour. Requests are emulated using Poisson distribution. We observe that *demand-aware* case performs better than baseline case (represented as normal scheduling in the plot) due to additional bandwidth becoming available.

8.7 Scalability

Using centralized CMs can impact scalability, but we believe that this potential scalability problem is manageable. The presence of a centralized CM is an important feature of our collective architecture. Centralized control makes building meaningful commercial distributed services feasible, because the CM has a clear idea of the collective system's resources and can use information-aware scheduling to provide good service to clients. It also greatly simplifies the design of services. Additionally, we believe that a trusted CM is necessary for attracting external partners and a large number of participating nodes. Our collective model is not alone in its use of centralization. Many important commercial systems use centralization yet scale to very large systems, e.g., SETI@home has 5 million users, BitTorrent sessions have a centralized tracker, and websites such as Google and Yahoo! handle huge amounts of traffic every day.

To achieve acceptable scalability, the CM is designed as replicated distributed services run on dedicated nodes, not on a single computer or on unreliable end nodes. There has been extensive research and commercial work in this that can be utilized to build a scalable collective manager [17, 30, 37, 59, 69].

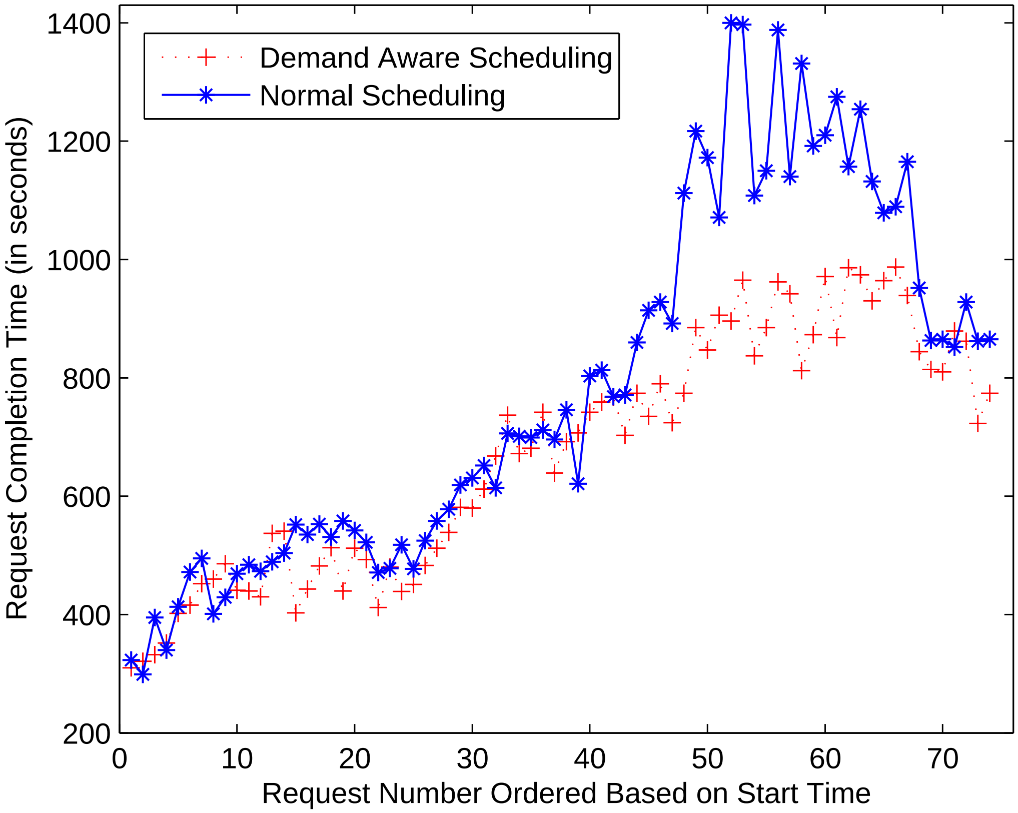

Figure 8.6: Demand-Aware Scheduling: Request Completion Time vs Request Number

From an implementation standpoint, the collective manager is a collection of multiple components. These components are instantiated as individual programs and can be distributed on different server nodes for load management. Use of a *data layer* and *intercomponent communication* layer ease the difficulty of distributing CM components across multiple nodes. Next, most of these components can be replicated across multiple nodes to further improve load distribution.

Here we have a look at the different components of a collective manager and consider how they can impact the CM's scalability. Our main goal is to determine whether each component can be distributed across multiple servers without major complications (i.e., is it easily parallelizable or not).

- **Data Layer**: A well-defined data layer separates the data from the collective manager front-end. This helps a system designer distribute components across different server nodes as long as each component can access the data in an efficient manner.

  In terms of the scalability of data layer itself, a database based backend (especially commercial variants like Oracle) can handle a huge load safely. Also the system can use a distributed in-memory cache system like memcached [63] for alleviating database load. As a separate approach, a very big collective can build a system like Bigtable [17, 46] to manage massive data storage needs (there are opensource implementations like hypertable [46] and Cassandra [16] that provide functionalities similar to bigtable).

- **Node Manager and Liveness Server**: The node manager is responsible for node registration and for shipping the necessary executables and data to PNs. The liveness server is responsible for keeping track of node aliveness by receiving direct pings from PNs during graceful join and leave or by acting on indirect reports sent by other players based on application level pings. Based on these, it updates node status in the data backend and sends triggers to service managers if needed. All of these activities are easily parallelizable as their is no dependency between multiple tasks.

- **Information Collection Engine**: The information collection engine collects reports sent by various players in the system, verifies the correctness of senders to protect forgery and then saves these reports to the data back end. There is no dependency between different report collection activities and hence this work is easily parallelizable and can be distributed across multiple servers based on the load.

- **Offline Data Analysis Engine**: The offline analysis engine focuses on two important tasks: (1) cross-verification of task completion reports sent by PNs and (2) analysis of past performance records and node resource profiles to help resource scheduler.

Cross-verification of task completion reports can be easily parallelized. A single analysis focuses on a single transaction, e.g., a content download by a specific client during a specific time interval). It only requires data records submitted by participating nodes involved in that transaction. Different transactions do not have any cross-dependency between them for verification purposes. So this analysis work can be easily distributed among separate servers with different sets of transactions assigned to different servers.

Analysis of past performance records and node resource profiles can become resource intensive in the case of a very large collective from millions of nodes. In such a case, a technique like MapReduce [22] should be deployed to distribute the work across a number of server nodes.

- **Offline Trusted Third Party**: The trusted third-party-based protocol is used to achieve nonrepudiation during data exchange whenever needed. Scalability-wise, a nonrepudiation task does not have any dependency with other nonrepudiation tasks and hence can be easily distributed across different servers.

  Moreover, the protocol used in the collective [55] is an *offline protocol*, i.e., the trusted third party does not intervene in the protocol when no problem occurs. Only during a problem does the trusted third party play a role in resolving the issue, which should be an uncommon event.

- **Service Managers**: Service managers are per-service agents that are responsible for ensuring smooth running of services over the collective overlay. The core of a service manager consists of (1) listening to partner's requirements and service performance triggers, (2) making decisions regarding the required distribution of the service over the collective, and (3) interacting with the resource scheduler for resource allocation.

  Scalability-wise, different service managers can be run on different servers. For a single service manager direct replication is not possible as the major work is involved only in decision making (step 2 above) and needs to be done by a single master. However, this work can be distributed in a divide-and-conquer style where subproblems are distributed over a cluster of nodes and then results merged on a master node (similar to MapReduce [22]).

- **Resource Scheduler**: Due to the presence of the data layer, a resource scheduler is a stateless system. It acts upon requests made by service managers, decides resource allocation, and then updates the status in the data backend. Hence, it is easily parallelizable.

  To improve performance, requests related to a particularly high demand task (e.g., requests for a highly popular content in CCDS) can be routed to the same instance of the resource scheduler. This instance can cache processed data related to this task distribution in its memory to speed decision making. It still needs to ensure data freshness and update the data layer after an allocation change so that other instances of resource scheduler can perform their work correctly.

In summary, a collective manager can be made scalable using techniques developed for replicated distributed services and for distributing work over a cluster. With the advent of highly available web-services (e.g. Google, yahoo, Amazon etc.), there has been extensive research work in this direction [17, 30, 37, 59, 69] that can be utilized to build a scalable CM.

8.8 Summary

In this chapter we have presented the implementation specific details of a collective system. However, methodologies discussed in this chapter present only one out of many possible ways of implementing a collective system.

CHAPTER 9

OTHER SERVICES

A collective infrastructure is a great platform to build a variety of distributed services using idle resources of end-nodes. In this dissertation, we have primarily focused only on a Collective content distribution service (CCDS) and a collective backup service (CBS). However, there can be many more such services that can exploit the underlying platform offered by a collective approach.

In this section, we describe four such possible services - a collective compute service, a collective network probing service, a collective WiFi service, and a collective surrogate service.

9.1 Collective Compute Service

A node's idle computing resources can be used productively to run compute-intensive jobs, especially high-end scientific computing experiments requiring a large amount of computing power. Projects that harness the idle computing resources (e.g., seti@home [78]) have been quite successful.

A collective compute service (CCS) exploits idle computing resources to run compute intensive tasks. Applications of this service are typically embarrassingly parallel applications, i.e., applications that can be divided into a large number of independent parallel tasks. Our design for CCS is similar to the seti@home design. There are two main components of the CCS that run on the CM and PNs, respectively.

To provide this service, each participating node runs a CCS component inside the VM called the CCS agent. Compared to directly running an application on the node, as in the seti@home [78], our VM based model allows us to safely run arbitrary programs on a PN without introducing any security issues. For example, in 2003 many security vulnerabilities were discovered in the seti@home software [36]. One vulnerability involved a buffer overflow bug that could allow an attacker to take control of the host computer by sending specially formatted web requests. Using a VM-based approach, CCS provides a much more secure environment; so more potential end nodes can be more willing to participate in such a system. Also in a VM based solution, the application need only be developed for a single operating system, because the VM runs an OS image provided by the CM and hence provides a consistent execution environment for applications.

From the CM side, CCS is managed by a service manager running on the CM called the CCS manager. CCS agents periodically contact the CCS manager to get a list of tasks and to download corresponding data and executables The CCS agent then runs the task to completion and sends back the result to the CCS manager.

The CCS manager awards credits to PNs based on actual work units completed. Similar to other services, a selfish client can cheat by providing bogus results without actually running the task. To create a deterrent against cheating, a CCS manager sends the same task to multiple PNs similar to what seti@home already does, and then compares the results to identify cheating. Cheating nodes are penalized heavily.

9.2 Collective Network Probing Service

Over the last few years, there has been a growing interest in Internet measurement. Many projects have been working on understanding the behavior of the Internet. A measurement system based on measurements taken only from a limited number of academic sites (e.g., Planetlab) does not provide a representative model of the Internet. One of the best approaches for fixing this problem is to measure Internet properties from end-nodes distributed across the Internet. Many projects like DipZoom [92] have been working on such an approach.

We can use our collective infrastructure to implement a collective network probing service (CNPS). The CNPS can take network measurements from end-nodes distributed across the Internet and send results back to the interested party. For example, using such a service one can easily diagnose DNS resolution issues across multiple ISPs.

9.2.1 Service Design

The CNPS service is provided with the help of a CNPS service manager running as part of the collective manager. The CNPS service manager interacts with clients interested in Internet measurements and assigns measurement jobs to a set of PNs. These PNs are selected based on the type of measurements needed. For example, if a client needs a set of measurements from all autonomous systems (As) in the USA, the CNPS manager needs to select few nodes from each AS. Or if a client wants to debug a DNS problem from a particular ISP, then the CNPS manager needs to select a set of nodes from that particular ISP.

On the participating node side, the CNPS service is managed by a CNPS agent running inside the VM. Based on the measurement job assigned by the CNPS service manager, the CNPS agent downloads the required measurement probe executable from the collective manager. Each measurement job consists of a probe and corresponding timing information (e.g., run the probe once every 3 hours for 2 days). The CNPS agent schedules different measurements assigned to the node at the requested times and sends back the results of the measurements to the CNPS service manager. The CNPS service manager collects measurements results from different PNs and submits the complete report back to the clients.

9.2.1.1 Deterring Cheating Behaviors

A cheating node may try to fool the system by reporting results but not performing the actual measurements. There can be many approaches to deter cheating. For example, a given measurement can be submitted to multiple nodes in the same ISP and their results can be compared to ensure that the results look similar. Another approach might be based on auditing, where a PN can be issued measurement requests that are tracked to ensure if the work is actually performed or not. For example, if the measurement includes making a http-get request to a trusted web server, the collective manager can use the website logs of the trusted web server to check if the measurement was actually made or not.

Additionally a rational node would not be very interested to fake results if the cost of making measurements is negligible in terms of resources consumed. For CNPS, the real value of a network measurement comes from a node's unique location on the Internet. Typical network measurement takes minimal system resources and these resources are insignificant compared to the usual work performed on a computer. Thus if CNPS can avoid any measurements based on bulk data transfer, it may not suffer from cheating behaviors.

9.3 Collective WiFi Service

The basic idea behind a collective WiFi Service (CWS) is to exploit unutilized bandwidth available through existing WiFi installations in private administrative domains, e.g., home, shopping complexes, or offices. Our proposed CWS service uses these existing WiFi installations to provide Internet service to external clients.

The CWS service is provided with the help of a CWS manager running on the collective manager. This CWS manager acts as a global authentication and accounting service. Anyone interested in sharing his Internet connection registers his or her WiFi hub with the CWS manager. The registered hub can then share its Internet connection with external clients and get compensated in return.

9.3.1 Service Design

9.3.1.1 Startup and Session Establishment

Interested clients need to first register with the CWS manager and buy service credits to use the CWS service. They can then connect to any CWS-enabled WiFi hub in their environment and get charged according to their usage.

In detail, a registered client needs to download and install a software on his or her mobile device (e.g., laptop) that aids in authentication with CWS-enabled WiFi hubs. A typical client session starts with a client using the standard WiFi hub discovery mechanism based on SSID broadcast to list the WiFi hubs in his or her immediate environment. Once a CWS-enabled WiFi hub is selected, the user's mobile device can connect and authenticate itself automatically with the help of the installed CWS software.

9.3.1.2 Security and Authentication

For the easiest configuration, a CWS-enabled WiFi-hub can be set to not use encryption. Doing so will allow external devices to easily connect to the hub and get a temporary IP address. Authentication with the CWS manager and subsequent authorization to route the packets to the Internet can be layered on top of this. However, wireless networks without any encryption are inherently unsafe, as intruders can snoop without any restrictions. This can create security problems for both the external as well as internal nodes connected to the WiFi hub and thus is not recommended.

WiFi Protected Access (WPA) based on the 802.1x authentication is the currently preferred security/encryption standard for WiFi. Our proposed solution is to set a master 802.1x authentication server at the CWS manager and a local 802.1x authentication server (e.g., tinyPEAP [85]) on each WiFi-hub. Any incoming connection request is first authenticated against the local 802.1x server and if that fails, the request is authenticated using the master authentication server running on the CWS. Authentication against the local 802.1x server provides internal/local users (i.e., the users who own that WiFi hub) an uninterrupted access to the Internet. Thus, both internal users and external clients use the standard WPA authentication and encryption mechanism to access the Internet.

9.3.1.3 Accounting

Whenever a client accesses the Internet through a CWS-enabled hub, the client is charged by the CM for the service. In turn, the administrator of the hub gets credits for sharing his or her connection. This charging and awarding of credits requires an accurate accounting of the service usage.

Both external clients and administrators of hubs may cheat by providing wrong information to the CM. To deter this cheating, we propose a solution based on signed certificates. In our proposed solution, each registered CWS client is associated with a public-private key pair that is used for digital signatures.

CWS Internet service is charged based on actual connection time measured as a multiple of a fixed time-unit called *CWSU* (e.g., 5 minutes). A user using the service for a time smaller than *CWSU* is charged for a complete *CWSU*. To access the Internet, a client has to provide a signed authorization certificate to the hub after every CWSU minutes to continue the service. A typical authorization certificate will contain the IDs of the client, WiFi hub, and a signed authorization by the client to charge for access to a single unit of CWS service. These authorization certificates can be generated easily using the CWS software running on the mobile device.

9.3.1.4 Sharing Implications

Sharing a private WiFi network with external clients creates a few problems. First, external clients should not be able to access any Windows file shares or shared printers in the local environment (and vice versa). Thus the WiFi hub must logically separate local and CWS network by dropping the packets destined across the boundary.

Second, providing too much Internet bandwidth to external clients can affect the Internet bandwidth available to the internal/local users of a WiFi hub. To solve this, the CWS-enabled WiFi hub can limit the bandwidth available to the external users. It can set max upload and download bandwidth limits for external clients that they can exceed only if the local users are not utilizing their share of bandwidth. Additionally, a simple admission control can be used to limit the number of concurrent external users.

All these features can be easily implemented in Linux-based WiFi hubs using standard routing and traffic control system available in Linux [45].

9.4 Collective Surrogate Service for Resource-Constrained Devices

Today's mobile devices are becoming small computers, complete with integrated cameras, GPS and MP3 support, and a multitude of other functions. With their nearly ubiquitous connectivity and increasingly powerful capabilities, mobile devices are being used increasingly for critical applications in the social, defense, financial, and health sectors.

The same portability that makes mobile devices immensely useful creates many limitations, e.g., limited battery capacity, limited CPU power, and limited storage. The resource limitations pose challenging problems, but they can be masked by utilizing the resources of less resource-constrained computers found in the environment, a concept called *surrogate computing* or *cyber foraging* [4, 77, 34].

A collective infrastructure can provide such a service in a simple but efficient manner. Based on a request made by a resource-constrained device, a collective manager can instantiate required application on a participating node's VM. This way a participating node can act as a surrogate for a client device and perform tasks on its behalf. Figure 9.1 illustrates a simple surrogate usage scenario in which a resource-constrained PDA offloads the compute-intensive portions of a speech recognition service to a VM running on a participating node.

This section focuses on two ways that a surrogate service can enhance a conventional mobile computing environment: by (i) reducing the energy consumption of mobile clients and by (ii) enabling more powerful applications than can be realistically run on resource-constrained mobile devices. We present the design of a surrogate service for mobile devices that allows clients to dynamically invoke compute jobs on participating nodes and show how it enables interesting mobile applications such as:

- Offloading compute-intensive jobs to a surrogate to enhance the basic capabilities of a mobile device, e.g., a speech recognition server;
- Offloading network-intensive jobs to a surrogate to reduce the number of battery-draining network operations that a mobile client needs to perform to accomplish

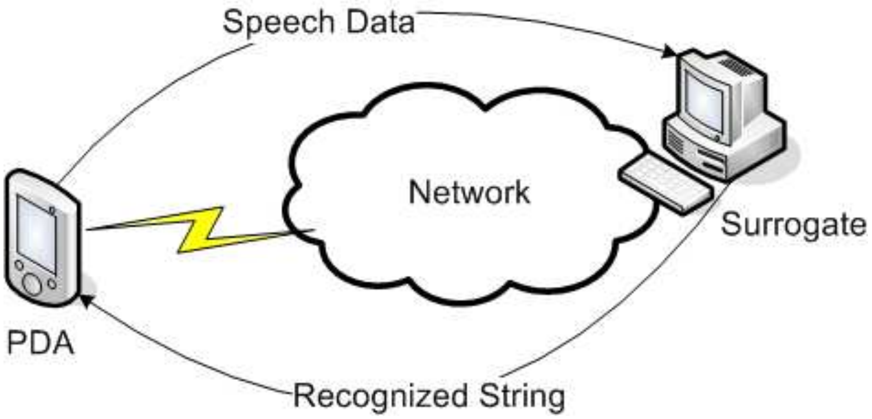

Figure 9.1: A Surrogate Computing Scenario: Remote Speech Recognition

some task, e.g., performing an Internet query that involves downloading data from multiple source and mining it to get a result; and

- Offloading client personalization operations to a nearby surrogate, e.g., invoking a device-specific proxy server that modifies web content based on the device's I/O characteristics.

In addition to enabling new applications, a surrogate infrastructure can decrease the storage, battery, and computation requirements of surrogate-aware mobile devices, thereby enabling the use of smaller, less complex, and less expensive mobile clients. Surrogate-aware mobile devices would need only enough local resources to perform their common tasks, and could use surrogate resources to perform more complex or less common tasks.

For this service, we assume a *trust relationship* between client and the collective manager as we allow client devices to run an arbitrary application on a surrogate provided by the collective infrastructure. Such a trust can be established based on the verification of the real-world identity of clients, e.g., a verification based on a credit card transaction. A better approach can be to allow clients to instantiate only from a list of trusted application supported by a collective system. Also for this section, we assume a trust relationship between client and potential surrogates (i.e., selected participating node) and hence do not discuss cheating issues. This service was actually designed as a precursor to the collective system with a different trust structure in mind.

Previous research [5, 26, 27, 64] has demonstrated some of the potential benefits of using surrogates to enhance the mobile computing experience. However, surrogate computing has not received widespread attention or been deployed outside of a handful of academic research laboratories. We believe that one of the main reasons for this lack of impact to date is due to the requirement that application developers write surrogate-capable applications in a specific language and/or restricted environment.

Our surrogate service is based on virtual machine technology that addresses these concerns. Our design enables the safe and effective sharing of surrogate resources, while allowing clients to install and run arbitrary untrusted code on surrogates. We do not require clients who use our surrogate infrastructure to use any particular communication framework, development environment, programming language, or middleware/runtime system. Because we impose no restrictions on the programming language or runtime system, existing mobile applications can be ported to use our surrogate service with little effort by partitioning them into a front end that runs on the mobile client and a back end that runs on the surrogate.

In this section, we first describe the architecture and implementation of a surrogate service built on a collective infrastructure. Then we demonstrate the effectiveness of the surrogate service by showing how it enhances existing mobile computing applications by reducing their response time and/or the amount of energy that they consume.

To evaluate the performance and energy benefits of offloading compute- or network-intensive operations from a mobile client to a surrogate, we have implemented surrogate-enabled versions of the `sphinx2` speech recognizer and a synthetic web services application. We found that using a surrogate improves the response time of speech recognition by a factor of 120 while reducing the energy drain on the client device by a factor of 60. For the synthetic web services operation, response time improves by a factor of 21 and client energy drain drops by a factor of 48. In short, using surrogates can lead to dramatic performance and energy savings.

9.4.1 Design

To set the context of our surrogate service, consider the following usage scenario. A PDA user clicks on the icon associated with an application configured to run, at least partially, on a surrogate machine. If the OS has not already obtained access to a surrogate, it invokes a surrogate acquisition module to contact a collective manager for access to a surrogate service. Assuming the client's access is approved, the collective manager allocates a participating node for this request and allows the client a direct access to the virtual server/machine running on the PN.

To invoke an application on the virtual server, the client ships a small program to a daemon listening on a well-known port on the virtual server and the daemon runs the program on behalf of the client. Typically, this program is a shell script that downloads the real application over the Internet, installs it, and runs it. Once the surrogate portion of the application is installed on the surrogate, the client portion of the application running on the PDA launches a client interface (if any), transparently ships input data to the surrogate portion of the application, collects responses, and outputs them through the client user interface.

9.4.1.1 Session Management

A client wishing to locate a surrogate server contacts a collective manager with its specific requirement and is allocated a surrogate by the collective manager.

Each virtual server is configured to run a *surrogate service agent* (SSA) that handles requests for surrogate operations from its client. To invoke an operation on a surrogate, the client sends a request to the surrogate service manager, which listens on a well-known port on the virtual server. Client requests consist of a URL that points to a program that the client wants the SSA to run on its behalf. Typically, the program is a shell script that downloads the necessary software/packages, installs them, and then invokes them. For example, in Section 9.4.2 we describe an experiment that involves installing a Sphinx2 voice recognition server on the surrogate and having it accept client voice recognition requests on a well-known port.

Our current protocol utilizes SSL/TLS for client communication with the virtual server because they are well understood and widely available. For authentication, we use public key certificates signed by the collective manager acting as PKI root. After allocating a PN to be used for surrogate, the collective manager informs the PN about the client certificate. This certificate is used by the surrogate service agent to determine which clients are authorized to use it.

9.4.1.2 Programming Interface

We provide a simple client library that supports the following operations:

- **get_surrogate()**: Takes as input the IP address of a collective manager and a service description string. Returns a success code. If successful, the IP address of the virtual server is also returned.

- **subtask_conf_request()**: Takes as input the URL where a program that the client wishes to run can be found, which is sent as a subtask configuration request to the appropriate surrogate service agent. Returns a success code.
- **get_virtual_server_ip()**: Returns the IP address of the active surrogate virtual server, if any.

To enable client-side scripting, these functions are also provided as programs and the configuration information (e.g., IP address of current surrogate server) is available via environment variables.

9.4.2 Evaluation

In this section we evaluate our surrogate service using two applications, the `sphinx2` speech recognition system and a transcoding web proxy. We chose these two applications to investigate the value of offloading work to surrogates for both compute-intensive applications (`sphinx2`) and network-intensive applications (the synthetic web services application).

9.4.2.1 Experimental Setup

For our experiments, we have two surrogate platforms. The first is a Dell Dimension 4550 Series Computer with a 2.40-GHz P4 processor and 512MB of 266 MHz DDR RAM (*university surrogate*). The second is a Dell Dimension 2400 Series Computer with a 2.80-GHz P4 processor and 512MB of 333 MHz DDR RAM (*home surrogate*). The client is a Sharp Zaurus SL-5500 PDA running Linux 2.4.18-rmk7-pxa3-embedix (OpenZaurus distribution 3.5.1). The Zaurus has a 206-MHz SA1110 processor, 16MB of FlashRAM, and 64MB of DRAM, 24MB of which is dedicated to file storage. The Zaurus's network connectivity is provided by a Linksys WCF12 compact flash 802.11b wireless card.

For all experiments, the client device is connected to the CS department network via a Wi-Fi access point. To test the importance of the physical proximity of the client and surrogate, we first evaluate the surrogate service with a surrogate co-located on the CS department LAN (*university surrogate*) and again with the surrogate at the first author's home (*home surrogate*). The home surrogate is connected to the Internet via a cable modem. The RTT between the client and the university surrogate varies from 2-7ms, whereas the RTT between the client and the home surrogate varies from 98-114ms.

For each of our applications, we run two sets of experiments, one in which the application is run in its entirety on the PDA and one in which the resource-intensive portion of the application is dynamically instantiated on a surrogate node. We first report the time required to initialize the surrogate service, including the time required to perform service discovery, instantiate a new virtual server, and install and start the test application. We then report the run times and energy consumed by the PDA to run the application both locally and with the help of a surrogate. For experiments where we run the application entirely on the PDA, we disable the network card and LCD backlight to minimize energy drain. In contrast, when we execute the resource-intensive portions of the application on the surrogate, we leave the network card enabled, which results in a conservative estimate of the energy requirement. In both situations, the PDA is idle other than the test application and an instance of `top`, a system management tool that we use to extract CPU and memory utilization.

To determine the amount of energy consumed by the PDA, we monitor the actual current drawn by the device during the experiment. We use the Tektronix TDS784D digital oscilloscope, using a TCP202 current probe to sample current and a TEK P6139A voltage probe to sample voltage at the rate of 2500 samples per second. Actual energy consumption is calculated by the average instantaneous power consumption over a time interval t, multiplied by time t. For statistical accuracy, each experiment is repeated five times and we report the average and standard deviation over those five runs.

9.4.2.2 Sphinx Speech Recognition

`Sphinx2` [44] is a real-time, large-vocabulary, speaker-independent speech recognition system developed at CMU. It uses pre-made models for American English, including an acoustic model, a pronunciation dictionary, and a language model. These models are stored in several files that in aggregate are roughly 23MB in size. When `sphinx2` is loaded and run completely on the Zaurus, storing 23MB worth of model files requires deleting most other applications and user files from the Zaurus. In contrast, the client stub required to run `sphinx2` on the surrogate server is only 12KB in size.

To perform this experiment, we created two versions of `sphinx2`, one that ran entirely on the Zaurus and another that split the functionality between the Zaurus and the surrogate. For both, we used the `sphinx-2.0.4` source code from sourceforge. Porting `sphinx2` to run on the Zaurus required a nontrivial effort because it uses nonstandard sound driver settings and the Zaurus sound driver did not support many of the `ioctl` calls used by `sphinx2`. Once ported, it was relatively straightforward to divide the application into two components for use in our surrogate service. The client portion uses the system calls described in Section 9.4.1.2 to allocate a virtual server and instantiate an instance of the `sphinx2` server. It then records what the user says and sends the raw digitized sound data to the surrogate for analysis. We believe this development experience will be typical for surrogates – porting an application to the small device entails nontrivial work, but splitting the application to exploit surrogate computing is straightforward.

For our first set of experiments, we measure how long it takes to instantiate the `sphinx2` application on a participating node's VM.

Table 9.1 shows the time to download and instantiate a `sphinx2` surrogate, which entails downloading the `sphinx2` server software (6.3MB) and starting the server. For this experiment, the server software was stored on a webserver in the same LAN as the university surrogate computer. It takes 3.84 seconds to download, install, and run the application on the university surrogate. The same operations take between 22 and 23 on the home surrogate. The slower startup on the home surrogate is due to the overhead of downloading 6.3MB of software across the Internet. If the surrogate is configured to cache previously downloaded packages, repeat invocations of `sphinx2` require only 4.42 seconds to initiate.

Table 9.2 compares the results of running the speech recognizer on the Zaurus and on the surrogate machines. The performance differences are striking. When run on the Zaurus, it takes around 45 seconds to recognize a short four word phrase, far slower than real time. In contrast, the surrogate version can recognize the phrase in

Table 9.1: Average response time for instantiating `sphinx2` speech recognition engine. Standard deviations are in parentheses.

Surrogate Location	Response Time
Univ	3.843 (0.008)s
Home	22.581 (0.167)s
Home (Caching)	4.417 (0.095)s

under a half second when the client and surrogate share the same LAN, or in just over 1 second when the surrogate is accessed over the Internet. In this case, the slower response time for the home surrogate is due to the higher latency between the client and surrogate and the low upstream bandwidth of the cable modem.

In addition to dramatically reducing the recognition time, using a surrogate also dramatically reduces resource consumption on the Zaurus. As mentioned above, simply storing the `sphinx2` application and model files on the Zaurus requires deleting most other applications and data from the Zaurus. Also, when run locally `sphinx2` utilizes more than 95% of the Zaurus' CPU throughout the experiment, spiking as high as 98.8%, and occupies more than 50% of its memory. This left the PDA unusable for any other activity, and resulted in a popup warning, "*The Memory is very low. Please end this application immediately!*". In contrast, using a surrogate reduced the CPU and memory overheads to 0.4% and 1.1%, respectively, leaving the Zaurus free to perform other tasks.

When we compare the energy cost of invoking the speech recognizer locally versus on the surrogate, the results again strongly favor using a surrogate. Performing speech recognition on the university surrogate consumes roughly 60 times less PDA

Table 9.2: Sphinx2 speech recognition on the Zaurus. Standard deviations are in parentheses.

Type	Response Time	CPU Util	Memory Util	App Size	Energy Consumption
local	44.72(0.12)s	>95%	51.6-55.9%	23MB	32.62(1.36)J
surrogate univ	0.37(0.02)s	.3-.5%	1.1%	12KB	0.53(0.03)J
surrogate home	0.95(0.03)s				1.27(0.04)J

energy than running it on the Zaurus, while performing speech recognition at on the home surrogate requires roughly 25 times less energy. These results come despite the nontrivial energy consumed transferring data between the client and surrogate. The PDA consumes roughly 0.271J of energy when idle for one second, so a small portion of the 32.61J consumed by the local speech recognition experiment is a fixed cost, but the net savings are dramatic.

Overall, these results clearly show that using a surrogate can lead to dramatic improvements in both response time and energy consumption for compute- and storage-intensive applications like `sphinx2`.

9.4.2.3 Web Services (Data Mining)

Typical web services entail sifting through potentially large amounts of data acquired from one or more storage servers. As we shall see, they are also well suited to surrogate computing. In contrast with speech recognition, web service (data mining) applications tend to be bandwidth-intensive rather than compute-intensive. The amount of computation done per-byte of data is much lower than for speech recognition. Thus, the potential benefits of using a surrogate to perform the data filtering include both the reduced processing time and the reduced amount of data that needs to be transferred over the energy-hungry wireless LAN.

To evaluate the impact of using a surrogate on this class of applications, we designed the following synthetic benchmark. The benchmark entails downloading three 6.3 MB files, performing an MD5 message digest operation on each file, and outputting the resulting three checksum values. For this experiment, we consider only the university surrogate. When the surrogate is used, the client sends it URLs for the three files, the surrogate downloads the three files and computes their MD5 checksums, and then the surrogate returns the resulting values to the client. In both the cases, we put the Zaurus's network card into power saving mode [94] if there is no network activity. While in power saving mode, the network card wakes up every 100ms to listen for a beacon message from the access point to determine if there are any queued messages awaiting delivery.

Table 9.3 presents the results of these experiments. The web site hosting the three files resides on the university LAN. Downloading the files to the Zaurus and performing the MD5 checksums locally takes 21 times as long and consumes 48 times more battery energy than using a local surrogate.

Figure 9.2 shows the current consumption pattern of the Zaurus when it uses a surrogate for this benchmark. We can see that the wireless network card remains in sleep mode most of the time. Small current spikes occur every 100ms, which is the frequency with which the network card leaves sleep mode to listen for any waiting packets. There are two clear sets of higher peaks, which are the times when the client sends the query to the surrogate and subsequently receives a response from the surrogate. In contrast, if the client downloads the files and performs the checksums locally, it cannot exploit the power saving sleep mode during the duration of download. We anticipate that applications like transcoding proxies will have similar energy benefits because of how they reduce the amount of data that is sent to/from the mobile client.

We conclude by noting that the power saving mode that we employ is fairly conservative. Even greater power savings are possible should the mobile client employ a more aggressive power saving mode, e.g., bounded slowdown mode [54].

9.4.3 Summary

In this section we have discussed the design of a surrogate service using the collective infrastructure. We have shown that surrogates enable powerful mobile applications and reduce the energy consumption of mobile clients. In particular, we demonstrated experimentally that the use of a surrogate improved the response time of speech recognition by a factor of 120 while reducing the energy drain on the

Table 9.3: Synthetic benchmark results. Standard deviations are in parentheses.

Type	Response Time	Energy Consumption
local	45.888 (1.721)s	67.394 (0.577)J
surrogate	2.170 (0.055)s	1.400 (0.067)J

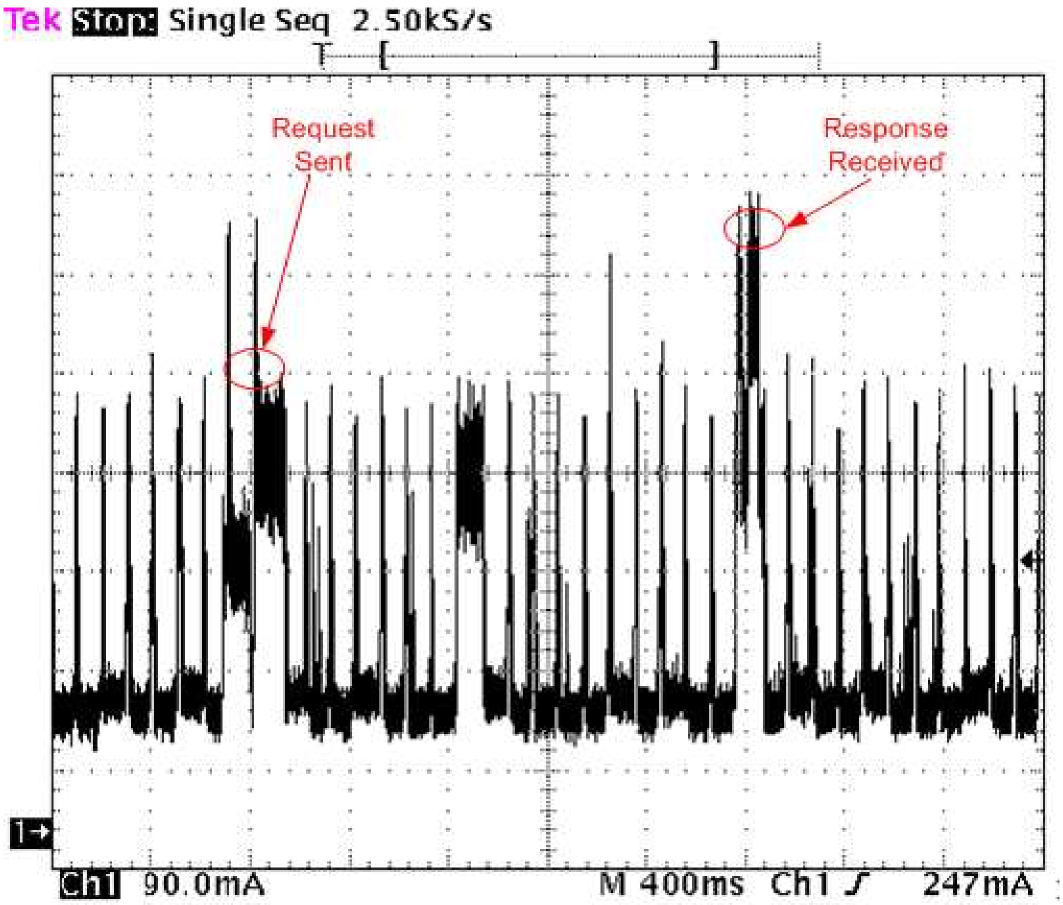

Figure 9.2: Synthetic Surrogate Usage Benchmark: Client Current Consumption

client device by a factor of 60. Using a surrogate to perform a synthetic web services operation improves response time by a factor of 21 and reduces mobile client energy drain by a factor of 48.

CHAPTER 10

CONCLUSION

In this dissertation, we have presented a system that exploits the idle compute, storage, and networking resources of myriad network-connected computers distributed around the world.

Unlike previous efforts to harness idle resources of end-nodes across the Internet, our incentive and interaction model is based neither on altruism nor on bartering. Instead, we use a currency based incentive model. Our system is based on a *collective manager* (CM) that manages the resources of large pools of untrusted, selfish, and unreliable *participating nodes* (PN). Participating nodes provide idle resources to the CM, which unifies these resources to run meaningful distributed services for external clients, and pays back participating nodes in proportion to their contributions. Our approach provides a useful alternative to the typical P2P approach; it provides a more effective utilization of idle resources, has a more meaningful economic model, and is better suited to build commercially distributed services.

We have designed and implemented two services on top of a collective model - a collective content distribution service (CCDS) and a collective backup service (CBS). Especially, we have designed a collection of security and economic mechanisms as part of these services that ensures accountability across different transactions. This accountability is then tied up with an incentive model and an offline accounting system to correctly compensate participating nodes and to create deterrents against cheating.

Our incentive model provides explicit credits to participating nodes for each work performed for successful delivery of services. Additionally, to provide an incentive for nodes to provide stable resource levels and to penalize cheating, the amount of credits received by a node in return for work depends on the node's long-term *consistency*.

Thus in a collective, consistently correct behavior leads to better incentives over the long-term, and similarly cheating behavior has long-term negative implications. We show that while we cannot prevent users from being dishonest, our mechanisms mitigate cheating behavior by making it economically unattractive. We show that a small probability of catching cheaters (under 4%) is sufficient for creating a successful deterrence against cheating. We further show that our incentive system can be used successfully to motivate nodes to remain in the system for a prolonged time.

We use an information aware scheduling system that exploits the underlying diversity in end-nodes' resources to mitigate the affect of unreliability and heterogeneity inherent in the system.

Overall we show that a collective system is an excellent approach for efficiently exploiting idle resources to build commercial services. We envision that collectives centered around competing CMs can grow to millions of nodes and act as an excellent infrastructure for exploiting idle resources and for building new and interesting services.

REFERENCES

[1] E. Adar and B. Huberman, *Free riding on gnutella*, First Monday, 5 (2000).

[2] Akamai. `http://www.akamai.com/`.

[3] Amazon Web Services. `http://aws.amazon.com/`.

[4] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, *The case for cyber foraging*, in 10th ACM SIGOPS European Workshop, 2002.

[5] R. Balan, M. Satyanarayanan, S.-Y. Park, and T. Okoshi, *Tactics-based remote execution for mobile computing*, in 1st USENIX International Conference on Mobile Systems, Applications, and Services (MobiSys), 2003.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, *Xen and the art of virtualization*, in Proceedings of the 19th Symposium on Operating Systems Principles (SOSP), October, 2003.

[7] G. S. Becker, *Crime and punishment: An economic approach*, The Journal of Political Economy, 76 (1968), pp. 169–217.

[8] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker, *Total recall: System support for automated availability management*, in Proceedings of Symposium on Networked Systems Design and Implementation (NSDI), Mar. 2004.

[9] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2007.

[10] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, *An xor-based erasure-resilient coding scheme*, tech. rep., International Computer Science Institute Berkeley, 1995.

[11] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, *Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs*, in ACM SIGMETRICS international conference on Measurement and modeling of computer systems, 2000.

[12] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger, *Economic models for resource management and scheduling in grid computing*, The Journal of Concurrency and Computation: Practice and Experience, 14 (2002).

[13] Cachelogic, *P2p in 2005*. `http://www.cachelogic.com/home/pages/research/p2p2005.php`.

[14] ——, *True picture of file sharing*, 2004. http://www.cachelogic.com/home/pages/research/p2p2004.php.

[15] B. Calder, A. Chien, J. Wang, and D. Yang, *The entropia virtual machine for desktop grids*, in International Conference on Virtual Execution Environment, 2005.

[16] Cassandra: Distributed Storage system. http://code.google.com/p/the-cassandra-project/.

[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, *Bigtable: a distributed storage system for structured data*, in OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation, 2006, pp. 205–218.

[18] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, F. Kaashoek, J. Kubiatowicz, and R. Morris, *Efficient replica maintenance for distributed storage systems*, in Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06), San Jose, CA, May 2006.

[19] R. H. Coase, *The nature of the firm*, Economica New Series, 4 (1937), pp. 386–405.

[20] B. Cohen, *Incentives build robustness in bittorrent*, in Proceedings of the Workshop on Economics of Peer-to-Peer Systems, 2003.

[21] E. Damiani, D. C. di Vimercati, S. Paraboschi, P. Samarati, and F. Violante, *A reputation-based approach for choosing reliable resources in peer-to-peer networks*, in CCS '02: Proceedings of the 9th ACM conference on Computer and communications security, 2002.

[22] J. Dean and S. Ghemawat, *Mapreduce: Simplified data processing on large clusters*, in OSDI'04: Sixth Symposium on Operating System Design and Implementation, 2004.

[23] T. Dierks and E. Rescorla, *The transport layer security (tls) protocol version 1.1*, 2006.

[24] Damn Small Linux Distribution. http://damnsmalllinux.org/.

[25] Emulab - Network Emulation Testbed. http://www.emulab.net.

[26] J. Flinn, D. Narayanan, and M. Satyanarayanan, *Self-tuned remote execution for pervasive computing*, in HotOS-VIII, 2001, pp. 61–66.

[27] J. Flinn, S. Park, and M. Satyanarayanan, *Balancing performance, energy, and quality in pervasive computing*, in Proceedings of the 22nd International Conference on Distributed Computing Systems, July 2002.

[28] I. Foster, C. Kesselman, and S. Tuecke, *The anatomy of the Grid - enabling scalable virtual organization*, Internation Journal of Supercomputer Applications, 15 (2001).

[29] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat, *Sharp: An architecture for secure resource peering*, in Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003.

[30] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The google file system*, in SOSP, 2003, pp. 29–43.

[31] Gnutella. `http://www.gnutella.com`.

[32] A. V. Goldberg and P. N. Yianilos, *Towards and archival intermemory*, in Proc. IEEE International Forum on Research and Technology Advances in Digital Libraries (ADL'98), IEEE Computer Society, April 1998, pp. 147–156.

[33] Google Adsense. `http://www.google.com/adsense`.

[34] S. Goyal and J. Carter, *A lightweight secure cyber foraging infrastructure for resource-constrained devices*, in Proceedings of the Sixth IEEE Workshop on Mobile Computing Systems and Applications, Dec. 2004.

[35] ——, *Safely harnessing wide area surrogate computing -or- how to avoid building the perfect platform for network attacks*, in Proceedings of the First Workshop on Real Large Distributed Systems, Dec. 2004.

[36] P. Gray and R. Lemos, *Security flaw hits seti@home.* http://news.zdnet.co.uk/itmanagement/0,1000000308,2133025,00.htm.

[37] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler, *Scalable, distributed data structures for internet service construction*, in Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000), 2000.

[38] C. Griffiths, J. Livingood, and R. Woundy. Comcast's ISP Experiences In a Recent P4P Technical Trial. `http://www.ietf.org/internet-drafts/draft-livingood-woundy-p4p-experiences-02.txt`.

[39] S. Guha, N. Daswani, and R. Jain, *An experimental study of the skype peer-to-peer voip system*, in 5th International Workshop on Peer-to-Peer Systems, Feb. 2006.

[40] M. Gupta, P. Judge, and M. Ammar, *A reputation system for peer-to-peer networks*, in NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video, 2003.

[41] A. Haeberlen, A. Mislove, and P. Druschel, *Glacier: Highly durable, decentralized storage despite massive correlated failures*, in Proceedings of the 2ndt USENIX Symposium on Networked Systems Design and Implementation (NSDI '05), Boston, Massachusetts, May 2005.

[42] S. P. Hargreaves-Heap and Y. Varoufakis, *Game Theory: A Critical Introduction*, Routledge, 2004.

[43] HTB Linux queuing discipline manual. `http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm`.

[44] X. Huang, F. Alleva, H.-W. Hon, M.-Y. Hwang, K.-F. Lee, and R. Rosenfeld, *The SPHINX-II speech recognition system: an overview*, Computer Speech and Language, 7(2) (1993), pp. 137–148.

[45] B. Hubert, *Linux advanced routing and traffic control howto*, http://lartc.org.

[46] Hypertable: Distributed Storage System. `http://www.hypertable.org/`.

[47] Ion Stoica *et al*, *Chord: A scalable peer-to-peer lookup service for internet applications*, in Proceedings of the ACM SIGCOMM '01 Conference, Aug. 2001, pp. 149–160.

[48] L. Kahney, *Cheaters bow to peer pressure*, Wired, (2001).

[49] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina, *The eigentrust algorithm for reputation management in p2p networks*, in WWW '03: Proceedings of the 12th international conference on World Wide Web, 2003.

[50] T. Karagiannis, P. Rodriguez, and K. Papagiannaki, *Should internet service providers fear peer-assisted content distribution?*, in IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement, 2005.

[51] Kazaa. `http://www.kazaa.com`.

[52] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat, *Bullet: High bandwidth data dissemination using an overlay mesh*, in Proceedings of the 19th ACM Symposium on Operating System Principles, 2003.

[53] D. Kosti, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat, *Maintaining high-bandwidth under dynamic network conditions*, in Proceedings of the USENIX 2005 Annual Technical Conference, 2005.

[54] R. Krashinsky and H. Balakrishnan, *Minimizing energy for wireless web access with bounded slowdown*, in The Eighth Annual International Conference on Mobile Computing and Networking, 2002.

[55] S. Kremer, O. Markowitch, and J. Zhou, *An intensive survey of non-repudiation protocols*, Computer Communications Journal, 25 (2002), pp. 1606–1621.

[56] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, *OceanStore: An architecture for global-scale persistent storage*, in Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems, Nov. 2000.

[57] K. Lai, M. Feldman, I. Stoica, and J. Chuang, *Incentives for cooperation in peer-to-peer networks*, in Workshop on Economics of Peer-toPeer Systems, 2003.

[58] T. W. Liao, *Clustering of time series data–a survey*, Pattern Recognition, 38 (2005), pp. 1857–1874.

[59] B. C. Ling, E. Kiciman, and A. Fox, *Session state: beyond soft state*, in NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation, 2004, pp. 22–22.

[60] M. Litzkow, M. Livny, and M. Mutka, *Condor — a hunter of idle workstations*, in Proceedings of the 8th International Conference on Distributed Computing Systems, June 1988, pp. 104–111.

[61] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani, *iplane: an information plane for distributed services*, in OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation, 2006.

[62] G. J. Mailath and L. Samuelson, *Repeated Games and Reputations*, Oxford University Press, 2006.

[63] Memcached: Distributed memory object caching system. `http://www.danga.com/memcached/`.

[64] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, and T. Giuli, *Towards a distributed platform for resource-constrained devices*, in Proceedings of the 22^{nd} International Conference on Distributed Computing Systems, 2002, pp. 43–51.

[65] Microsoft Corporation, *Architecture of windows media rights manager.* `http://www.microsoft.com/windows/windowsmedia/howto/articles/drmarchitecture.aspx`.

[66] A. Nash, W. Duane, and C. Joseph, *PKI: Implementing and Managing E-Security*, McGraw-Hill, Inc., 2001.

[67] OpenSSL: The Open Source toolkit for SSL/TLS. `http://www.openssl.org`.

[68] P4P Working Group. `http://www.openp4p.net/front/p4pwg`.

[69] D. A. Patterson, D. E. Culler, and T. E. Anderson, *A case for now (networks of workstation)*, in PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, 1995, p. 17.

[70] PlanetLab. `http://www.planet-lab.org`.

[71] A. M. Polinsky and S. Shavell, *The Theory of Public Enforcement of Law*, vol. 1 of Handbook of Law and Economics, North Holland, Nov 2007.

[72] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content addressable network*, in Proceedings of the ACM SIGCOMM Conference, Aug. 2001.

[73] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford, *Xenoservers: Accounted execution of untrusted code*, in IEEE Hot Topics in Operating Systems (HotOS) VII, Mar. 1999.

[74] E. Rescorla, *SSL and TLS - Designing and Building Secure Systems*, Addison-Wesley, 2001.

[75] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, *Pond: the OceanStore prototype*, in Proceedings of the USENIX File and Storage Technologies Conference (FAST), 2003.

[76] A. Rowstron and P. Druschel, *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*, in International Conference on Distributed Systems Platforms, Nov. 2001.

[77] M. Satyanarayanan, *Pervasive computing: Vision and challenges*, IEEE Personal Communications, (August 2001).

[78] SETI@home. http://setiathome.ssl.berkeley.edu.

[79] R. Sherwood, R. Braud, and B. Bhattacharjee, *Slurpie: A cooperative bulk data transfer protocol*, in In Proceedings of IEEE INFOCOM, 2004.

[80] J. Shneidman and D. Parkes, *Rationality and self-interest in peer to peer networks*, in 2nd Int. Workshop on Peer-to-Peer Systems (IPTPS'03), 2003.

[81] M. Sirivianos, X. Yang, and S. Jarecki, *Dandelion: Cooperative content distribution with robust incentives*, in NetEcon, 2006.

[82] Spread Toolkit: high performance messaging system. http://www.spread.org.

[83] M. S. P. S. C. Study, *Aruba.it*. http://download.microsoft.com/download/6/b/e/6be5466b-51a5-4eaf-a7fc-590f32bc9cb3/Aruba.it%20Case%20Study.doc.

[84] ——, *Hostbasket*. http://download.microsoft.com/download/b/f/3/bf34b7be-81e9-46a8-a5e3-ccb648a98547/Hostbasket%20Final.doc.

[85] tinyPEAP. http://www.tinypeap.com.

[86] Utopia. http://www.utopianet.org/.

[87] V. Vishnumurthy, S. Chandrakumar, and E. Sirer, *Karma: A secure economic framework for peer-to-peer resource sharing*, in P2P Econ, 2003.

[88] VMware Player. http://www.vmware.com/player.

[89] VVD Communications. http://www.vvdcommunications.com.

[90] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta, *Spawn: A distributed computational economy*, Software Engineering, 18 (1992), pp. 103–117.

[91] L. Wang, K. Park, R. Pang, V. S. Pai, and L. Peterson, *Reliability and security in the CoDeeN content distribution network*, in Proceedings of the USENIX 2004 Annual Technical Conference, Boston, MA, June 2004.

[92] Z. Wen, S. Triukose, and M. Rabinovich, *Facilitating focused internet measurements*, in SIGMETRICS, 2007.

[93] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann, 2005.

[94] H. Woesner, J.-P. Ebert, M. Schlager, and A. Wolisz, *Power saving mechanisms in emerging standards for wireless lans: The mac level perspecitve*, IEEE Personal Communications, 5 (1998), pp. 40–48.

[95] H. Xie, R. Y. Yang, A. Krishnamurthy, Y. G. Liu, and A. Silberschatz, *P4p: provider portal for applications*, in SIGCOMM 2008, 2008.

[96] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, *Tapestry: A resilient global-scale overlay for service deployment*, IEEE Journal on Selected Areas in Communications, (January 2004).

[97] Zope Replication Service.
`http://www.zope.com/products/zope_replication_services.html`.