A Collective Approach to Harness Idle Resources of End Nodes

Sachin Goyal
sgoyal@cs.utah.edu

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 thesis 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 three distributed services using the collective approach. These services are: a collective content distrbution service, a collective data backup service, and a high-performance computing service on top of the collective overlay.

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 [24], BitTorrent [11], and Skype [20]. The large body of work on distributed hash tables (DHTs) exploit peer resources to support DHTs (e.g., Chord [22], Pastry [36], Tapestry [46], and CAN [32]), on top of which a variety of services have been built. SETI@home [37] and Entropia [9] 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 [33], *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 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 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 effect of selfishness. Selfish nodes will 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 [2] and software modifications to get more credit than earned in SETI@home [23]. To mitigate 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 being selfish, our mechanisms mitigate selfish behavior by making it economically unattractive.

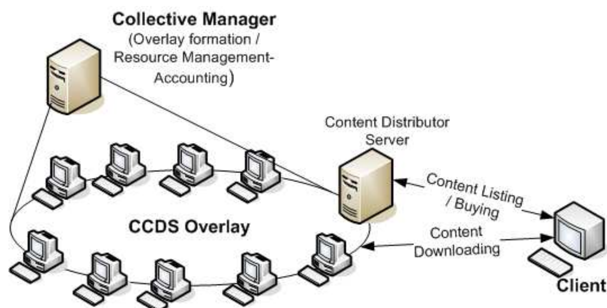

Figure 1: Collective Content Distribution Service

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 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 like bittorrent [11], gnutella [15] etc. The second is compute-only services like seti@home [37], entropia [9] etc. The third is utility computing systems like Xenoservers [33]. The fourth is grid computing systems [13].

Peer to Peer Services:

Unlike typical P2P systems, we do not assume that PNs are altruistic (e.g., Kazaa [24] or Gnutella [15]) or willing to "barter" their resources in return for access to the end service (e.g., BitTorrent [11]). 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.

Distributed Compute Intensive Services:

Unlike seti@home [37] and Entropia [9], 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 chanllenges 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 like 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.

Utility Computing Systems:

Utility computing systems like Xenoservers [33], Planetlab [31], and SHARP [14] 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 collective they do not provide solutions for service level selfish behaviors by the participating nodes (or sites). Many of these assume trusted nodes. Projects like SHARP [14] 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 [14]).

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. But unlike utility computing projects like Xenoservers or SHARP, 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.

Grid Computing Systems:

Systems like Condor [29] 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 non-altruistic (will do as little productive work as possible while maximizing their compensation).

Systems like computational grids [13] also deal with distributed resources at multiple sites, though again their main focus is on trusted and dedicated servers.

1.2 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 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. Others 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. But it is still tough 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. Additionally inherent untrusted and unreliable nature of end nodes makes currency-based bartering risky For example, it is easy for a cracker 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 a like content distribution system will have to find and make bartering agreements with many end nodes. And this 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. First, in a collective, only the trusted collective manager has direct access to the VMs running on the participating nodes, so we do not need to worry about an unknown party using the nodes for nefarious purposes, e.g., to launch network attacks. 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.

Another way to think about a collective 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, 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 himself and the savings remain unutilized during the search period.

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. While we still have freelancers, the majority of productive work is performed by well defined organizations.

Centralized control in a collective makes building meaningful commercial distributed services feasible, because the CM can have a reasonable understanding of the collective system's resources and can provide service level guarantees to customers. The presence of the CM also makes the design of services much simpler. Additionally, a trusted well known CM brand makes it easier to attracting external customers and large numbers of participating nodes.

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).

1.3 Thesis Statement

We hypothesize 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 will not rely on altruism to sustain the system. The system will provide incentives for nodes to participate in the system for extended durations. Also, all services built on collective will be 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 will provide 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 will try 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, will improve 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, collective system relies on proactive understanding of the system based on history, and adaptive response to the observed behavior.

To support this hypothesis, we propose to build a collective system and develop three distributed services: a collective content distribution service, a collective backup service, and a collective compute service.

The first service we propose to build and evaluate is a collective content distribution service (CCDS). 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 service of CCDS 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 server 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 is 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 or a fixed monthly fee. 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.

Our third service is a collective compute service (CCS). CCS work is to execute compute intensive applications, similar to projects like seti@home and entropia. The focus of CCS is embarrassingly parallel applications for which it is possible to divide the application into a large number of parallel tasks easily without much dependencies. For example, a typical application may involve applying the same mathematical function over a huge amount of data set. Many applications like protein folding, genetic algorithms, genome data processing etc, fall into this space.

1.4 Opportunity Cost Analysis

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 like 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 [8]. Another study from CacheLogic put the P2P percentage of Internet traffic at about 60% at the end of 2004 [7]. The same study reports that BitTorrent was responsible for as much as 30% of all Internet traffic at the end of 2004 [7].

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.

Instead of using unutilized resources of end-nodes, one can potentially use a cluster of PCs to provide similar resources. In this subsection, we do a quick quantitative analysis to understand the opportunity cost of a collective in comparison to the cluster 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 utility computing approach.

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 [43] or Utopia [41] 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.

1.4.1 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 4 extra replicas), we are left with 10^6 GB of storage available.

1.4.2 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 [40] puts the labor cost at 54% of total cost, while Aruba.it [39] 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.

1.4.3 Utility Computing Services from Amazon

Amazon web services provides utility 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 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's take an 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 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.

1.4.4 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 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.

Overall we are 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). Thus a collective can provide equivalent services at a more competitive rate than a cluster based approach, while still providing decent return back to its participating nodes. Additionally a collective does not require huge amount of initial investment that is needed for a setting up a big cluster. It does not require the costly maintenance or power costs. End-nodes are upgraded in due time by their users, thus it gets the advantage of technology advancements for free.

1.5 Scope

We are proposing to 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. A service developed on top of collective system only receives basic functionality like node management and the ability to start a program on a node.

Not all services can be easily supported on top of a collective system. To map effectively to a collective, a 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. But this is not the main focus of our work.

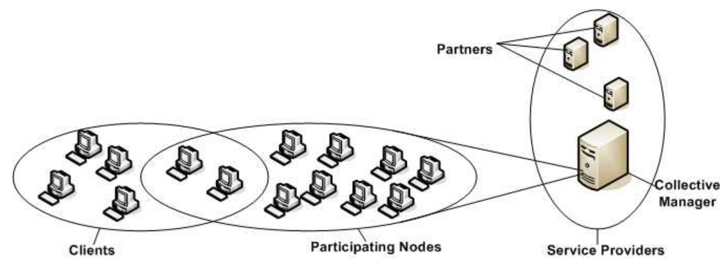

Figure 2: Main Players in Collective

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 (say less than 0.1%, i.e. 1000 nodes in a collective system build out of 1 million nodes). Sybil attacks, i.e., a single user using multiple identities are possible. One can also see them as a form of collusion between those multiple identities.

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.

1.6 Contributions

The intended contributions of this thesis are as follows:

We propose to design, implement and experiment with an infrastructure to build commercially feasible services that run on the idle resources of untrusted and unreliable end nodes. Our goal is to provide predictable service, and we take proactive steps to maintain the QoS required for building commercially viable services.

Our collective system will use a practical incentive system 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 will lead to improved incentives for the node. We will use it to keep a tab on selfish behaviors. It is well known phenomenon in game theory that repeated interactions give rise to incentives that differ fundamentally from isolate interactions [30]. Thus in a collective, a consistently correct behavior will lead to better incentive over long term, and similarly selfish behavior will have long term negative implications.

We will also use long term associations as an opportunity to collect historical information about nodes up/down timings, resource capabilities etc, and use them as a guide to better handle the inherent chaotic nature of end-nodes' ability and availability.

We will design and build three actual services: a collective content distribution service, a collective backup service, and a collective compute service. These services will use security mechanisms in novel ways to ensure accountability across different transactions. We will show how an offline accounting system can use the data collected from such services to provide accurate incentives to the participating nods.

Apart from commercial services, we will look into how we can share free content (e.g., linux distributions) over the collective overlay. We propose to use virtual currency to deal with such services. These virtual credits can not be converted to money, but give PNs the right to download free content in the future.

2 System Overview

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. There are four main players:

1. **Participating Nodes (PNs)** are end nodes that have idle compute, storage, or network resources. They are typically connected to the Internet though 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 co-exist, each providing different services and/or pricing models.

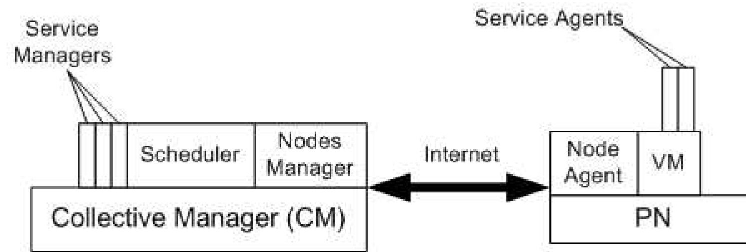

Figure 3: System Architecture

3. **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.

4. **Partners** are commercial users of a CM, e.g., an online movie distribution company can utilize the collective service for movie distribution, while managing the content acquisition and sales itself.

A service may not necessarily have both clients and partners. For example, a compute service may have partners but no clients, while a remote backup service may have clients but no partners. The Collective Content Distribution Service (CCDS) has both clients and partners. Figure 2 shows main players in a collective.

2.1 Participating Nodes

To provide resources to a CM, a PN runs a VM instance and provides root access to that instance to the CM. By using VMs as the unit of resource sharing, we provide strong protection to PNs against buggy or malicious CM software. Also, to the extent allowed by the VM system, using VMs lets CMs run arbitrary OS and application images without worrying about the underlying PN's OS. For our evaluation, we will use the free VMware Player [42] or Xen [4] 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 .

2.2 Collective Manager

A typical distributed service built on a collective consists of components that run colocated on the CM (called *service managers*) and other components that run on the PNs. A service manager is responsible for converting service requirements into small components and distributing these components to a set of PNs. Typically each service component will be replicated to provide availability and scalability.

As an example, Figure 1 shows how we might provide a 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 service manager to distribute (probably encrypted) content. The service 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.

Figure 3 shows the high level architecture of a collective. Apart from the service managers, the CM consists of a *node manager* and a *scheduler*. The node manager tracks the set of PNs currently registered with the CM, including their available resources and resource restrictions. The scheduler helps schedule the resources on individual PNs. A given PN can run multiple services.

2.2.1 Node Manager

Node manager keeps track of all the registered nodes, as well as currently active (online) nodes in the collective. Whenever a participating node joins/rejoins the collective (e.g after every power cycle), it pings the 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. On direct indication, CM removes the node from the active list. Indirect indications may be a genuine shutdown or failure, but it can also result from network partitioning, or due to wrong reporting by a malicious client/PN. So 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.

Apart from the active/non-active status, node manager keeps an historical record of each node. Basically the node-agents running on the PNs collect lot of useful information - e.g. observed upload bandwidth, node's boot up timings etc., and send these information to the node-manager periodically (e.g. after every 3 days).

2.2.2 Scheduler

The scheduler helps schedule the available resources on individual PNs to different services running on the collective. The scheduler uses this historical data available at the node manager to group PNs according to their typical availability (when they typically enter/leave the collective) and resources (how much processing, storage, and network resources are available). These grouping are then used to decide on which node a given service component should be scheduled.

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 an affect 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., non-repudiation protocol used in CCDS and CBS) are discussed along with service design in section 5 and 6.

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.

3.2 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 isolation of the collective software from the host PN. That is, applications running inside the VM cannot directly interfere or nor access resources belonging to the host PN. Additionally the host PN can enforce resource controls (e.g., disk quota, cpu share, and physical memory allocation) for the VM. This allows normal work to proceed on the host PN without undue impact. This protection and resource controls provided by the 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 provides good isolation, a malicious user can still misuse the virtual machine to launch network attacks [19]. This problem is handled through access control, i.e., VM instance on a PN can only be directly controlled by the CM. We don't give direct access to external entities to run any 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 has 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 4 along with the incentive model used in collective.

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 don't focus on these issues as part of this thesis, and they remain part of the future work.

Though as part of our service design, we will make sure that any malicious activity should not lead to incorrect service behavior or wrong crediting of rewards. At best malicious activities can lead to disruption of services and wastage of resources similar to traditional denial of service attacks.

4 Incentive Model and Selfish Behaviors

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. 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 discourage selfish behavior economically.

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. We employ the following mechanisms:

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 selfish/malicious PNs, each service activity can be credited to unique contributing PNs. If nodes are selfish, more than one node will request credit for the same work. To resolve conflicts, the accounting system needs additional information.

Offline Cheater Detection: To identify selfish nodes, the system collects data from PNs, CM scheduling records, service scheduling records, and partners' sales records. This data is 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 become possible to detect selfish behaviors by PNs.

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.

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 un-attractive to collude.

4.2 Variable Pay Rates (Raises and Cuts)

To provide an incentive for nodes to provide stable resource levels and to penalize node churn, 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 selfish 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 also lead to pay cuts in other services run on that node.

Other factors can also 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.

5 Collective Content Distrbution 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 clustered 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 propose 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.

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

5.1.1 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, alongwith a list of related participating nodes. The client then downloads the content directly from the participating nodes, with different chunks coming from different nodes.

5.1.2 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, 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 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.

5.1.3 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 the caching, 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 maintains a list of all contents served by it and information about the distribution. Content caching is dynamic and should adapt to changing demand of the content as well as PNs arrivals, departures, and failures. The goal of the CCDS is to provide good download performance to clients.

The CCDS manager utilizes a feedback-driven engine to adaptively optimize the required goals. Caching spread of a particular content is dictated by its demand, i.e., the number of recent purchases. 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 last hour. The CCDS manager uses this information to predict future demand, and ensures that data is cached at enough places to meet the demand.

5.1.4 Authorizations

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, 2) to have a paper trail that let us 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 consists 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* as part of the handshake protocol to the PNs from which it's requesting a part of the data. PNs verify the authorization, and only provide the data if the client has the valid authorization.

Similar to client authorization, node authorization certificates (*nauth*) are issued by the CCDS manager to control the flow of data movement within the overlay. With every job request, the CCDS manager also sends a *nauth* to the node. A *nauth* contains the node ID, content ID, and the list of allocation units allocated to the node. It is signed by the private key of the CM. PNs need to provide the *nauth* to download its copy of content from other participating nodes for caching.

Non-Repudiation:

To guard against certain malicious behaviors, we use a non-repudiation protocol for any data transfer between PNs and clients. This ensures that when a PN sends data to the client, the client can not 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 [27]. A TTP is said to be offline if it does not intervene in the protocol when no problem occurs. In our system, the CCDS service manager acts as the TTP. The protocol basically involves sending a ciphered data (D_k, i.e. D encrypted with cipher k) instead of data D. Cipher key k is sent separately, only after the receiver sends a signed proof of receipt of the data. To prevent a PN getting the proof of receipt and then not delivering the k, PN has to be 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 encrypted k, and can retrieve the k. Exact details of the protocol can be seen in [27].

5.2 CCDS Incentive Model

The core of CCDS's accounting system is an offline processing system that acts on the reports collected from the PNs, content distributor, and on the information available to the CCDS manager. Both the clients and PN are untrusted and are assumed to be selfish (or rational) nodes that act to maximize their utility. We assume all communication channels are asynchronous, but resilient, i.e., they deliver correct data after a finite amount of time.

We start by describing the different reports and their value to the contribution accounting system. Then we describe our solution to handle selfish behaviors by participating nodes.

5.2.1 Reports

PN's Report: PNs periodically (typically every day) send reports to the CCDS manager that detail the content they have distributed and to which clients. This report contains the contentID, transaction ID and specific chunk numbers that were delivered. It also include the non-repudiation of receipts (NORs) 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. If the data is late by more than a day, they try to contact CM at a higher frequency (e.g., every hour).

Content Distributor Report: The content distributor sends a log on what content was sold to whom. It 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.

CCDS Manager Information: The CCDS service manager have the metainfo file for contentID. It can be used to determine the size of content, the number of chunks into which it was divided, and corresponding chunk sizes.

5.2.2 Accounting

The accounting system starts using the 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 selfish). If the reported contributions match the amount of content sold, the PNs are compensated accordingly.

If the reported contribution exceeds the aggregate work reported by the content seller, the accounting manager investigates further to understand the underlying problem.

No Collusion: If a client is behaving selfish, it can not affect the contribution system, because as part of the protocol it must follow the non-repudiation protocol and deliver the right NORs.

If a PN is selfish, it can try to increase its profit by mis-reporting its actual contribution. Though it can't generate the necessary NORs and thus will be caught easily by the CCDS manager.

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 un-attractive to collude. There are three main possibilities for collusion: 1) clients collude among themselves, 2) PNs collude among themselves, and 3) clients and PN collude with each other.

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

So the only interesting case arises when clients and PNs collude with each other. Basically a selfish 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 the colluder 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 economical value of the credit per download to the participating nodes is always less than the actual cost paid by their buyer, as only a small part 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.

6 Collective Backup Service

Collective backup system (CBS) is an 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 is encrypted at the client computer before being submitted to CBS. As in CCDS, from the clients 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 can be charged either based on a usage or a fixed monthly price.

6.1 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 manage their login and accounting details. From the CM side, CBS is managed by a service manager running on the CM, called the CBS manager. The CBS manager effectively builds a storage network using the storage and bandwidth resources of participating nodes to cache and serve the data to the clients. Each participating node runs a CBS component called CBS agent to provide this service. CBS agents are run inside the VM running on PNs.

A client has an application interface that allows clients to easily backup a file on the CBS or to restore files from the CBS. To backup a file, a user starts by adding the file or directory to the CBS client app. The client app starts by creating a metainfo that is used to digitally verify the correctness of the data later. Each file is logically divided into chunks of fixed size (we use 512KB in our prototype) and the md5sum of each chunk is calculated to prepare the data's *metainfo*. The client app then contacts the the CBS manager to register the data, and passes the metainfo to it. The CBS manager assigns a unique *data ID*, and provides a list of PNs to which the client should upload the data, with different data chunks going to different PNs. The client app then uploads the data to those nodes.

CBS manager keeps 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 the right authentication info. The CBS client app typically keeps a locally cached copy of the user's data metainfo, and shows the list of files stored by the client on CBS and their total size etc.

To restore, the client starts by selecting the particular file in the client app. The client app then contacts the CBS manager to retrieve a list of related nodes, and then downloads the data directly from those node, with different chunks coming from different nodes.

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

Durability and Availability

Durability and availability are the two most important concerns when building a remote backup service out of unreliable end-nodes. One should be able to ensure data durability and availability in the presence of nodes failures and churning.

These problems have been studied extensively in previous projects like Ocenastore [28,34], Total Recall [5], Glacier [21], and Carbonite [10]. We will use lessons observed in those systems in the design of CBS. Similar to Oceanstore and Glacier, we will use erasure coding to maintain durability [16] and replication to ensure availability. Our goal is to have 4-nines (0.9999) of durability and a reasonable availability within few hours (depending upon the data size). We will use Cauchy Reed-Solomon code [6] for erasure coding.

We will use the historical data collected by the CM to decide the replication degree and the exact erasure code ratio to use. It will be dependent on typical node active time in the collective system, node churning, and complete data failures (e.g., due to disk failures) observed by the system. Apart from just erasure coding and replication, we will also use techniques like creating a replica on one of the most consistently performing PN to increase availability.

6.2 CBS Incentive System

A participating node participates in two main activities: storing data on its disk and second 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 transfer of the data.

Each transfer employs the non-repudiation protocol, so that both the sender and receiver can provide a verifiable proof of the transfer. We propose to use the offline trusted third party based non-repudiation scheme described in [27] (similar to CCDS). Each participating node sends a periodic report to the CBS manager listing the data it is storing, time-frame of the storage, and a list of transfers in which it has participated since the last report. It also appends the non-repudiation certificates for each transfer.

The CBS accounting system does offline analysis to tabulate the contributions of individual PNs. Apart from PNs reports, 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.

Handling Selfish Behaviors

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

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

As a first layer of protection, all data transferred between nodes, or between nodes and clients employ a non-repudiation protocol, thus creating a trail of who did what. This accountability is crucial in ensuring higher level functions.

As second level of protection, each participating node must send periodic report to the CBS manager as mentioned above. Each participating node is assigned a set of data chunks to be stored and transfer them to other nodes/clients on demand. If a PN is found to have successfully received some particular data, but then does not report it as part of its periodic report, the CBS manager gives negative penalty to the PN for wasting the system bandwidth without actually storing it.

A third layer mechanism ensures correct data storage and detects any data deletion by the PNs. To achieve this, we rely on a periodic data check (e.g., once in a month). Data check involves the act of successful data transfer (correctness verified by hash) as a means to verify that a node still possess a particular piece of data. 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.

Collusion: PNs can either collude among themselves, or can also include clients in their collusion to increase their payments beyond their fair share based on the actual work performed.

A colluding set of PNs can try to dodge storing data at a node if some other node is also storing the same data. Colluding nodes can then implement a reflection mechanism to pass periodic data checks. Given a request it can pass the query to the colluding node having the data, and can supply the returned reply to the checker. Given a small set of colluding nodes, it's quite rare to have any common data between them - thus reducing the effective storage savings made by colluders. Second, common data is not possible in an erasure coding based scheme as there is no exact replication of chunk involved.

A second case involves a mix of clients and PNs colluding. Here, colluding clients can try to invoke data recovery operations again and again, with a goal of getting extra transfer credits for the colluding PNs. 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 inter-storage of data by a small set of colluding nodes.

7 Collective Compute Service

We can use the collective to provide compute intensive services (CCS) similar to earlier projects like seti@home. Applications of this service are typically embarrassingly parallel, i.e., it is possible to divide the application into very large number of independent parallel tasks.

Our design for CCS is same as that seti@home. There are two main components that run on the CM and PNs respectively. From the CM side, CCS is managed by a service manager running on the CM called the CCS manager. Each participating node runs a CCS component inside the VM called CCS agent to provide this service. CCS agents contacts the CCS manager to get the code and corresponding data to run, and then send backs the result to the CCS manager.

8 Validation

8.1 Success Criteria

The success of this thesis lies in successfully demonstrating the viability of a collective system, and in successfully showing the validity of the claims made in thesis statement. These include the following main properties:

- Building meaningful distributed services out of untrusted, and unreliable end nodes: we propose to demonstrate this by building three services - a collective content distribution service, a collective backup service, and a collective compute service.
- Developing practical incentive model that works: A practical incentive model must be able to provide meaningful returns to the participating nodes according to their contributions even in the presence of selfish nodes.
- Effective harnessing of the Idle resources of end-nodes:
- Predictable performance of distributed services:

8.2 Distributed Services and Predictable Performance

A validation of predictable performance requires experimental evaluation of how a collective system behaves in the presence of events like node churning, node failures, link failures, and varying workloads.

We focus primarily on the collective content distribution service (CCDS) and collective backup service (CBS). We don't focus much on collective compute service (CCS) as its performance analysis is trivial. Embarrassingly parallel applications are only concerned with eventual completion of jobs without requiring any stricter notions of quality of service. Additionally, CCS is just like seti@home and other compute-intensive service that are already running successfully in production environments.

Environmental Assumptions: We use the following assumptions regarding the quantity and the quality of end-nodes participating in the collective.

We plan to experiment with a maximum of 500 emulated nodes running real code (using multiplexing on a limited number of physical nodes), and a simulated system on max 10000 nodes. Even though we expect millions of nodes active in a single collective, it is not possible to acquire and experiment with such a large number of nodes. This limitation will only have a limited impact on a service's performance analysis, as a single service instance like distribution of a 5 GB content to multiple

clients involves only a small subset of total nodes. However, we can not realistically evaluate the CM load in this environment, so we will do experiments to characterize the load on CM in our limited emulation environment, and use that as a guide.

Regarding the resource characteristics of the participating nodes, we assume a mixture of end-nodes including typical home desktops connected through broadband cable or DSL service, computers in universities or other communities with plentiful bandwidth, and nodes in business environments where companies are trying to monetize their idle resources. We will perform experiments based on conservative estimates of bandwidth/storage available in typical modern systems. Apart from that, we need patterns of node failures and node churning to create a meaningful experimental environment. Instead of assuming a particular model, we will experiment with a variety of different failure and churning patterns, either derived from a distribution or based on traces.

8.2.1 Basic Performance Analysis

For each service, we will design the system components to run on the CM and participating nodes, and show how they work together to provide a meaningful service to the clients.

We will implement our system on Linux, and hence all experimental work will be performed on Linux-based environments. We will not be focusing much on the VM in our experiments, as our main goal in using VMs is to provide an isolated execution environment at end-nodes. Their presence do not affect the performance of the collective services.

We will employ two main evaluation environments for our basic performance analysis: Emulab [1,45] and Planetlab [31]. Emulab will be our main evaluation platform due to its ease of experimentation and the ability to experiment with large number of nodes.

As a baseline, we will evaluate our system using the Emulab. For participating nodes, we will either emulate a single node per emulab node or multiplex multiple nodes on a single emulab node. We will generate different distributions of client load, and use different number of participating nodes in the environment to understand the base system performance.

On planetlab, we will try to evaluate our services using 100 nodes distributed across the Internet (based on availability).

8.2.2 Effect of Churning, Node Failure, and Link Failure

Node churning, node failures and link failures are three ever-present characteristics of a collective system. We will mainly rely on simulation to explore the the effects of these on the performance and availability of collective services.

We will only do few sample experiments on planetlab and emulab in this regard. Planetlab can provide us with a realistic link characteristics but very limited number of nodes, and limited churning or failure characteristics. Emulab provides relatively larger number of nodes with limited network topology setup, but allowing simple churning and failure models.

Our simulation experiments will be comprehensive in terms of evaluating the effect of churning and link/node failures, but we don't propose to simulate any network link-characteristics. Our experiments will be similar to the ones performed by Chun et.al. [10]). We will do two types of simulation runs. First set of experiments will be based on a statistical distribution of node churnings and failures that we assume to be independent. Our second set of experiments will use more realistic models derived from traces of real-life systems. We propose to use the traces used in previous work [10, 35].

8.2.3 Quantifying response to change in service demand rate

For both content distribution and backup service, we will experiment with varying service demand rate. The goal of these experiments will be to understand how well collective system can adapt to changing service demands. We will create different patterns of client request variation over the time, e.g., increasing the client request rate for a particular content in CCDS, or an increased backup/retrieval request rate in the CBS service, and analyze the impact of this variation.

8.2.4 Overhead analysis

Using Emulab, we will measure the load imposed on the centralized collective manager for different numbers of participating nodes, ranging from 100 to 500. We will do this for an experiment running CCDS, another one running CBS, and a third one running both CCDS and CBS at the same time.

8.3 Realistic Incentive Model

To validate our incentive model, we will need to show that:

1. The collective system can correctly estimate or determine each participating node's contribution towards service level objectives.

2. The collective incentive model rewards node's contribution accurately and motivates long term association of the nodes with the collective.
3. The collective system can defend against selfish behaviors by participating nodes.

The validation will be based on analytical analysis of the system design to understand different mechanisms, and their effectiveness. For each service we will describe and analyze how each node's contributions towards the realization of that service is measured, and how that translates to actual credit rewarded to the node.

8.3.1 Handling Selfish Behaviors

For each service designed on collective system, we will analyze how it performs in the presence of different selfish behaviors. We will consider the following scenarios:

- selfish behaviors exhibited by individual nodes in the system
- collusion between multiple selfish participating nodes
- collusion between multiple selfish clients nodes
- collusion between multiple selfish clients and participating nodes

For each case, we will first analyze all potential benefits available to a selfish node. We will then explore the possible ways to achieve those benefits, and whether it's practically achievable in our proposed system. If needed, we will do a cost analysis to better visualize the success or failure of economic deterrents of the system.

We might also do simulations to understand the impact of possible collusions being undetected. That is, we will assume a certain percentage of nodes colluding, and see if the collective system still remains profitable.

8.3.2 Variable Pay Rate

We propose to use variable pay rate as a mechanism to motivate nodes to remain active in the collective for long durations and to reward consistent participants for their resources.

To understand and validate the effectiveness of this mechanism, we will run simulations to calculate the pay rates for different PN profiles. We will analyze the long-term affects of this mechanism by comparing different participating nodes with similar resource availability, but different consistency levels. We will also analyze the effects of being detected as a selfish nodes, and how that will affect one's earning.

We will also compare different mechanisms to decide pay rates, e.g, only based on last month records, or based on all previous records etc.

8.3.3 Game Theoretic Models

To better understand implications of our mechanisms, we will model different mechanisms or a subset of them using game-theoretic methods. We will mathematically formulate our environment, the desired goal function, and economical mechanisms used. We will then analyze if the equilibrium state realized as an outcome of our environment and mechanisms achieves the desired goal function or not (Though it is currently not clear what all can be validated using game-theoretic analysis of reasonable complexity).

8.4 Effective Utilization of Idle Resources

We propose to demonstrate that a collective system will provide good utilization of end nodes' perishable idle resources, especially in comparison to a bartering based system like BitTorrent. Its validation is pretty much due to the design, as collective manager can schedule diverse type of work on a node as compared to a BitTorrent load motivated by a content download for user's immediate needs.

To understand this better, we will experiment with different loads of service demand, and how that affects the idle resource utilization at participating nodes.

We will also experiment with running all there, i.e., CCDS, CBS, and CCS together in the collective with a fixed number of PNs, and see how that helps in improving the utilization of the idle resources.

9 Related Work

There is a large body of related work in addition to the work discussed in Section 1. We use common distributed system techniques like replication and certificates. We use P2P mechanisms like simultaneous disjoint downloads from multiple destinations [11, 24].

Much of the work on P2P systems has used distributed hash tables (DHTs) as the building blocks for building completely decentralized peer to peer systems, e.g. Chord [22], Pastry [36], and CAN [32]. Many papers report on distributed services built on top of DHTs. We do not use DHTs in our design, but instead use the presence of a CM to help with routing and content mapping. Also, our collective platform can support multiple distributed services simultaneously that collectively use the available resources.

Several researchers have demonstrated the convenience of using virtual machines as the granularity of resource sharing, e.g., Xenoservers [33], grids [9, 12], and surrogate computing [18]. Previously we discussed the network security risks involved in providing complete access to the VM on end-nodes to external partners [19]. In our new collective model, only the trusted CM has access to the end node VMs.

Selfish behaviors have been observed extensively, e.g., free riding in gnutella [2] and software modifications to get more credits in SETI@home [23]. Our system can use a payout system similar to the one used by the Google adsense program [17] that allows webpage publishers to display ads on their sites and earn money.

SHARP [14] presents a secure distributed resource peering framework, and is a currency based bartering system as defined in Section 1. As compared to the collective, SHARP uses a completely decentralized structure and focuses on mechanisms to reserve and claim resources in a secure way. Unlike the collective, SHARP does not provide any solutions for service level selfish and malicious behaviors by participating nodes (or sites). They 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). They focus are dedicated resource infrastructures like Planetlab and utility computing systems where they allocate resources using VMs for every user (e.g., using vserver slices in Planetlab or using VMs on VMWare ESX). Our focus is on idle resources of end-nodes already in use (e.g., desktop computers); we allocate only one VM on them, and run multiple services inside that one VM. In SHARP, any untrusted third party can acquire VMs and potentially use them for nefarious activities like network attacks. In contrast, our one VM per PN is controlled by trusted collective manager. On other level, our service level mechanisms to handle selfish and malicious behaviors can be used with SHARP like infrastructure to provide services in their decentralized resource peering framework.

CCDS is similar to content distribution networks like Akamai's [3]. 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 [25] and Bullet' [26] are systems for high bandwidth data dissemination using an overlay mesh. They are peer to peer systems, although they assume presence of scalable mechanisms for efficiently building and managing an underlying overlay tree (whose overhead is not clear). Similar to many other P2P systems, Bullet and Bullet' assume altruistic users that are willing to cooperate both during and after the file download without any incentives. Their system depends on a lot of state being transferred between peers; thus any selfish users can easily bring the whole system down. In short, it is fairly trivial to construct free-riding strategies that can vastly degrade these systems performance.

Slurpie [38] 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 though, 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 require all clients to use the slurpie protocol; otherwise non-slurpie clients will go directly to the server, thus making it unfair to slurpie users. They do not give any way to achieve this though.

Codeen [44] is a network of caching web proxy servers that help in reducing the load on a web server. Unlike CCDS, their main focus is to 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 100s of the node. They use active monitoring to monitor the health and status of all the web proxy server; something like that is not possible in CCDS where we propose to deal with 1000s to millions to nodes.

10 Conclusion

In this proposal, we present a system that exploits the idle compute, storage, and networking resources of myriad cheap network-connected computers distributed around the world. Unlike previous efforts to harness idle distributed resources, we propose a system based on CMs that provide explicit credits for work performed on behalf of services. To discourage selfish behavior, we use a combination of offline data analysis to detect selfishness and an incentive model that encourages stable,

collusion-free, unselfish behavior. Our approach provides a useful alternative to the dominant P2P approach; it provides a more effective utilization of idle resources, has a more meaningful economic model, and is better suited to building legal commercially interesting distributed services. We think that a collective system 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] Emulab - network emulation testbed. http://www.emulab.net.

[2] Eytan Adar and Bernardo Huberman. Free riding on gnutella. *First Monday*, 5(10), October 2000.

[3] Akamai. http://www.akamai.com/.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP)*, October, 2003.

[5] Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoff M. Voelker. Total recall: System support for automated availability management. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

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

[7] Cachelogic. P2p in 2005. http://www.cachelogic.com/home/pages/research/p2p2005.php.

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

[9] Brad Calder, Andrew Chien, Ju Wang, and Don Yang. The entropia virtual machine for desktop grids. In *International Conference on Virtual Execution Environment*, 2005.

[10] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, Frans Kaashoek, John Kubiatowicz, and Robert 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.

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

[12] R. Figueiredo, P. Dinda, and J. Fortes. A case for grid computing on virtual machines. In *Proceedings of the 23rd Internatinal Conference on Distributed Computing Systems*, 2003.

[13] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid - enabling scalable virtual organization. *Internation Journal of Supercomputer Applications*, 15(3), 2001.

[14] Yun Fu, Jeffery Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. Sharp: An architecture for secure resource peering. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[15] Gnutella. http://www.gnutella.com.

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

[17] Google Adsense. http://www.google.com.

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

[19] S. Goyal and J.B. Carter. 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*, December 2004.

[20] Saikat Guha, Neil Daswani, and Ravi Jain. An experimental study of the skype peer-to-peer voip system. In *5th International Workshop on Peer-to-Peer Systems*, February 2006.

[21] Andreas Haeberlen, Alan Mislove, and Peter 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.

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

[23] Leander Kahney. Cheaters bow to peer pressure. *Wired*, 2001.

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

[25] 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.

[26] Dejan Kosti, Ryan Braud, Charles Killian, Erik Vandekieft, James W. Anderson, Alex C. Snoeren, and Amin Vahdat. Maintaining high-bandwidth under dynamic network conditions. In *Proceedings of the USENIX 2005 Annual Technical Conference*, 2005.

[27] Steve Kremer, Olivier Markowitch, and Jianying Zhou. An intensive survey of non-repudiation protocols. *Computer Communications Journal*, 25(17):1606–1621, November 2002.

[28] 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*, November 2000.

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

[30] George J. Mailath and Larry Samuelson. *Repeated Games and Reputations*. Oxford University Press, 2006.

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

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

[33] 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*, March 1999.

[34] 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.

[35] Rodrigo Rodrigues and Barbara Liskov. High availability in dhts: Erasure coding vs. replication. In *4th International Workshop on Peer-To-Peer Systems*, 2005.

[36] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms*, November 2001.

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

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

[39] Microsoft Service Provider Solutions Case Study. Aruba.it. `http://download.microsoft.com/download/6/b/e/6be5466b-51a5-4eaf-a7fc-59%0f32bc9cb3/Aruba.it%20Case%20Study.doc`.

[40] Microsoft Service Provider Solutions Case Study. Hostbasket. `http://download.microsoft.com/download/b/f/3/bf34b7be-81e9-46a8-a5e3-cc%b648a98547/Hostbasket%20Final.doc`.

[41] Utopia. `http://www.utopianet.org/`.

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

[43] VVD Communications. `http://www.vvdcommunications.com`.

[44] Limin Wang, KyoungSoo Park, Ruoming Pang, Vivek S. Pai, and Larry Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the USENIX 2004 Annual Technical Conference*, Boston, MA, June 2004.

[45] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[46] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, January 2004.