# An Efficient Disk Resource Management Mechanism for Scalable Disconnected Access to Web Services*

Nabeel Ahmed, Mike Dahlin
vtigtti@cs.utexas.edu, dahlin@cs.utexas.edu
Department of Computer Sciences
The University of Texas at Austin

**ABSTRACT**
This paper examines the importance of requiring a disk resource management mechanism for disconnected services and presents a robust system that embodies several features that are required of such a disconnected framework.

Disconnected services in which clients access services in *degraded* mode (i.e without relying on network connectivity) are important for providing greater service availability to potential clients. Because much of the content that requires connectivity is not cacheable, there is a trend towards downloading code that is "un-trusted", one that may place limitless demands on the resources available to the client.

Although such resource management has a broad area of application, this paper takes a look at disk resource management for write buffering, where data are stored for disconnected services with the intent that they will be evacuated at a later time. In this paper we explore a per-service popularity algorithm to address the write buffering problem effectively. In doing so, we present a system that implements an 'automatic' disk resource management policy and examine how it performs relative to the more rudimentary techniques already in use. As a result, we discuss how our system provides greater service availability by allowing flexibility in the introduction of new services while also providing greater disk access to existing ones.

**KEYWORDS**
Resource Management, Un-trusted Code, Degraded Mode, Mobile Extensions, Evacuation Bandwidth, Write Buffer, Soft Limit

## 1. INTRODUCTION

This paper presents a disk resource management mechanism for scalable disconnected access to web services and explains the importance of requiring such a resource management system for services that execute in disconnected mode. We focus on environments that utilize write buffering to support disconnected operation by allowing services to write data to disk with the intent that the data shall be evacuated sometime in the future [8,10].

There is an increasing demand for services clients subscribe to that do not require network connectivity for their execution. *Active Names* [1] is an infrastructure that supports such services that are location independent – meaning their execution may occur anywhere in the network (client, proxy, server etc). Although extensible, such services may be potential resource hogs where their execution could place limitless demands on resources launching commonly known *denial of service* attacks. The disk is one such resource that may be a potential target of such attacks. Disk resource management presents a greater challenge due to the additional need of guaranteeing persistent storage of data where disk space is not as easily revocable as memory or network bandwidth. Greater caution is therefore required to ensure that only space consumed by malicious services is reclaimed. Since we specifically address the issue of write buffering that deals with sensitive data, unlike a disk cache, we therefore must guarantee data persistence for services. The key challenge is therefore to create a balance between disk limits and our desired abstraction of infinite persistence.

In this paper, we propose a disk resource/write buffering management system for a resource management framework that supports disconnected access to web services. The system uses a per-service popularity algorithm combined with per-service data eviction rates to produce a system that manages disk allocations across different services. Specifically, the system monitors client trends in service usage and assigns priorities to each service based on the information gathered. In the event that the priorities of services decrease, disk space is reclaimed from the services as a function of their data evacuation rates. Using these capabilities, the system has the ability to set limits for each service and proportionately allocate space to each of them. The system described in this paper is dynamic; it continually

* This work was supported in part by the Texas Advanced Technology Program, the Texas Advanced Research Program, and a grant from Novell. Dahlin was also supported by an NSF CAREER award (CCR-9733842) and an Alfred P. Sloan Research Fellowship.

adjusts and re-calculates limits for different services and increases/decreases service allocations.

The contribution of this paper is a novel algorithm for building an effective disk resource management mechanism that may be used as the base for more comprehensive and robust systems capable of providing complete protection and security against denial of service attacks, while at the same time providing greater availability per service. However, as we discuss later in the paper, the ideal notion of infinite persistence is not attainable, forcing us to eventually relax rules on persistence, as a trade-off for greater control.

The rest of this paper is structured as follows: In the next section we discuss our motivation for pursuing research in this area. Then we briefly describe some of the traditional techniques that have been used to build a system for such disconnected environments, followed by detailed sections of the prototype system, its implementation, and experimental analysis of its performance and interaction. Then we finally discuss some of the system limitations and conclude with a section on future work.

## 2. MOTIVATION

There are a number of factors that motivate the development of a disk resource management system for write buffering. These are discussed separately in the following paragraphs.

First, current web technologies (Java Applets, Script-lets, etc) do not provide the infrastructure needed to support disconnect access to web services [2]. Although Java Applets and Script-lets allow code to be downloaded to the client, these technologies are too restrictive. They prevent access to disk completely by un-trusted code. These technologies therefore do not provide the necessary tools that are required by disconnected services/extensions, aimed at increasing service availability for potential clients.

Second, as mentioned earlier, the write buffering model is harder to attain then other techniques such as disk caching. The difficulty primarily stems from the need to provide data persistence guarantees for write buffers. Current options have a number of limitations since they a) prevent disk access as a result of enforcing limits, b) arbitrarily evacuate data, c) and do not implement controls that lead to *denial of service* attacks [2]. Such characteristics prevent construction of useful services aimed at providing greater availability.

Third, web service workloads and the large number of services that a client may access [2] introduce hundreds of un-trusted extensions that contend for resources including disk. This creates a need to provide controlled access for such environments to prevent situations leading to service starvation or *denial of service*. Mobile services/extensions that provide disconnected access for clients use techniques such as pre-fetching, hoarding [9,11], write buffering and message queues [8,10] in order to provide maximum availability for clients that access these services in *degraded* mode. These techniques may use the disk aggressively and if not controlled could completely fill the disk, preventing other services from utilizing it. This phenomenon, termed a *denial of service attack*, can render the disk completely inaccessible. Therefore, the focus of this paper is to build a system to guard against such hostile services and prevent such malicious attacks from establishing their presence.

Finally, many of the existing systems that provide disk resource management capabilities require a lot of user intervention to manage the system. Such systems are highly undesirable both for the end-user and for the extensions that utilize the system. This necessitates the design of a disk resource management mechanism that provides 'automatic' resource management without extensive hand tuning.

## 3. TRADITIONAL PROGRAMMING MODELS

Given all the challenges discussed in the previous section, our goal is to build a system that models the behavior of an ideal prototype that fulfills the need for disk resource management in a write buffering environment. But before presenting our proposal to this problem, it is worthwhile explaining some of the traditional techniques that have been used to tackle this problem, as follow-up of the characteristics discussed in the motivation section of this paper.

The two most popular policies that have been used widely to provide resource management across multiple services are static and demand based allocation. Their description and limitations are discussed further.

### 3.1 STATIC POLICY

Static allocation is an approach that gives an equal share of disk space to all services. The allocation per service is pre-determined and does not change over time. Thus, every service, regardless of popularity, is given the same amount of space on disk.

**LIMITATIONS.** First, this scheme could cause *denial of service* attacks. For example, if the fixed share of disk given to each service is too large, it could prevent introduction of new services in the system, causing denial of service. Second, this scheme yields poor disk space utilization whereby services that may not require a lot of disk space are given enormous amounts to account for their fair share of the disk. If on the other hand the fixed amount is too small, this scheme may prevent construction of useful services (e.g. Disconnected Hotmail) that require a greater amount of space on disk, even though these services may be most valuable to the client. Finally, this scheme also requires significant hand tuning on the user's part to decide what allocation size to use for a particular disconnected environment in order to

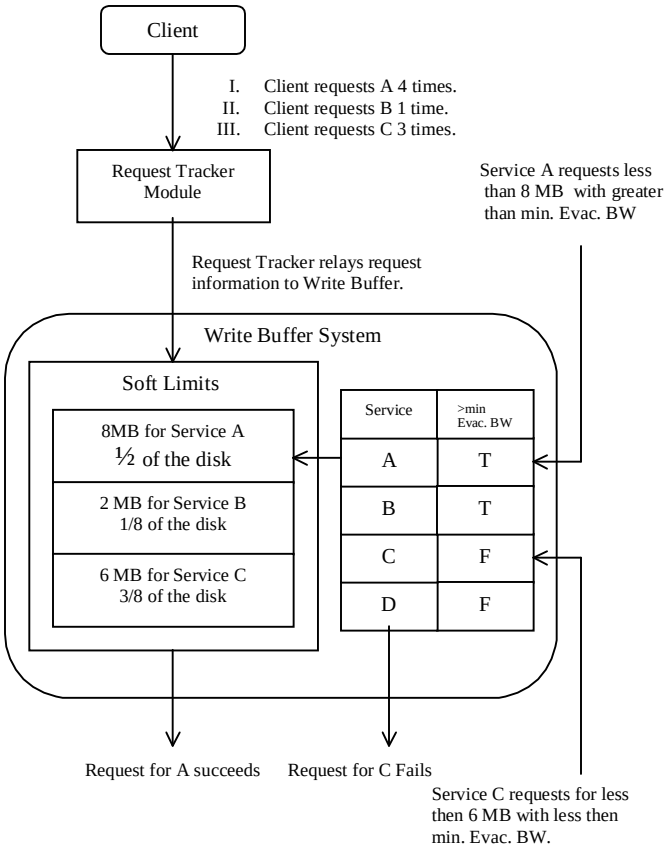


Figure 1. Shows the execution of the writer buffer system.

maximize the number of *popular* services that may be available to the user.

### 3.2 DEMAND-BASED POLICY

Demand-based allocation improves upon the static allocation scheme by allowing an open-ended policy whereby services that access disk more often are given more space. Examples are LRU and LFU for cache replacement [2]. In such cases, a program that accesses more data is given more space or increasing allocations. Such an approach also fulfills the requirements for self-tuning. However, this policy also does not fully meet the requirements for a system that may be deployed in a disconnected environment.

**LIMITATIONS.** First, this type of policy encourages denial of service attacks where deviant services increasing accesses are rewarded with increasing allocations until the disk is completely exhausted. Since the purpose of the disk management system is to protect against such malicious services, such behavior is intolerable for disk resource management systems operating in a disconnected environment.

Keeping these ideas in mind, the next section discusses the requirements for an ideal system executing in a write buffer environment, capable of dividing disk *perfectly* among competing services.

## 4. AN IDEAL SYSTEM

In this section we outline the requirements of an ideal disk resource management system that solves write buffering in a disconnected environment.

1. **Persistent Storage**. The system should never discard data until the data have been successfully written to the server.
2. **No Denial of Service**. Every service is allocated its *fairshare* of the disk (i.e. if a service *s* with priority *p(s)* attempts to write, that write should succeed unless the disk is full and the service has already consumed *P(s)/sum(p)* of the disk, where *sum(p)* is the sum of all service priorities).
3. **Automatic**. The policy should be flexible and require minimum hand tuning by the end-user.
4. **Work conserving allocations**. The system should never reject a write unless disk is full or if doing so will force later write requests to be rejected. Thus, if a service is not using and will not use its full space, then the excess space should be allocated to other services (to provide illusion of *infinite disk*).

These requirements, however, are not pragmatic because it is not entirely possible to build systems that provide such guarantees. For example, work conserving allocations require future knowledge of a service's allocation characteristics where guessing incorrectly could violate either requirements of *persistent storage* or *no denial of service*.

## 5. PROPERTIES OF THE SYSTEM

The system implemented as a result attempts to imitate the ideal system but relaxes some of the requirements mentioned above. The fundamental properties of the system are,

1. **Persistent storage.** We relax requirements by guaranteeing protection of data if the service promises to evacuate its data at a certain rate, specified by the service itself.
2. **No Denial of service.** Upper limits, (also known as *soft-limits)*, are implemented in the system to approximate *fairshare* by controlling the amount of data that each service may write to disk.
3. **Automatic.** The client is not required to manually configure the system specifically for different environments since the system is inherently environment independent.
4. **Work conserving allocations**. The system implements work conserving allocations by comparing the write

rates of services with their evacuation rates. Based on these rates, the system decides whether the excess space of one service may be utilized by another service.

The fundamental property of our system is that is uses the evacuation rate of services to solve the dilemma of persistence vs. *denial of service* and work conservation. The key idea is evacuation bandwidth. The evacuation bandwidth for a service is defined as *the rate at which a service's data are successfully written to their destination*. In such a system, a service with a particular priority promises an evacuation bandwidth that is used in combination with the priority to define an upper limit on the space that may be utilized by the service at any given point in time. Thus after, if a service fails to meet the evacuation bandwidth it *promised* or its priority decreases, subsequent requests for increased allocation by the service are rejected. Any "excess" data (greater than the *soft limit*) that is accumulated as a result is evacuated at a rate equal to the evacuation bandwidth promised by the service. This prevents the system from arbitrarily discarding data for a service due to abrupt changes in service priority. It also allocates more space to those services that promise to evacuate at a higher rate then the average incoming bandwidth of data to the buffer. The following sections describe in more detail the policies, architecture, and implementation of the system.

## 6. PROTOTYPE SYSTEM

The system we present here attempts to forge a compromise between static allocation that does not require any knowledge of services and dynamic allocation that requires unrealistic knowledge about services. The result is a system that provides approximate, albeit not perfect, allocations of disk to the various extensions executing on the client.

The proposed system framework is based on two policies that in combination provide an ideal framework for implementing disk resource management for write buffering. The next section describes these policies in greater detail.

### 6.1 POLICY

This section outlines two of the major policies implemented by the write buffering system. These policies are outlined below.

#### 6.1.1 Popularity

Our proposed system implements a per-service popularity policy to approximate the *fairshare* for each service. Although the policy does not "perfectly" predict the *fairshare*, per-service popularity predictions come very close to the actual *fairshare* values for each service. The per-service popularity policy is based on the intuition that commonly accessed services are more valuable to end-users than those less frequently accessed. Thus our system assigns a priority to each service and charges services disk space based on their priority as measured by the *request-tracking* module shown in Figure 1. This constructs a dynamic system, where changing priorities (due to request changes) translate into changing allocations, thus satisfying the goal of self-tuning. Per-service popularity therefore assigns more important services more space than those less important. But this policy alone does not fulfill the requirements of a write buffer system since it lacks the idea of data persistence because abrupt changes in priority could result in loss of crucial service data. The solution to this problem is established through the idea of evacuation bandwidth.

#### 6.1.2 Evacuation Bandwidth

Our system also applies the notion of evacuation bandwidth in deciding whether particular services may access the disk or not. Services executing in the system supply a *promised* minimum evacuation bandwidth that they must evacuate data at. The main idea here is to use promised evacuation bandwidth as a means of predicting when data will be written to the server. Since we are not able to predict the precise bandwidth by which services will write data to the server, we let the each service decide the rate at which it can evacuate its data. Thus, if a service promises a very high evacuation bandwidth, then it may lose its data and conversely, if the service promises a lower evacuation bandwidth, its corresponding allocation would be lower. This mechanism thus prevents loss of crucial data for a service that meets its promises by ensuring that the data are evacuated only after they have been written to the server. How this technique is applied in the buffer system is discussed in later sections.

Therefore, the above policies, when combined, produce the necessary ingredients to build a system that implements write buffering in a hostile environment of *un-trusted code*. The popularity-based policy lends itself to a policy discussed by *Chandra et.al* [2], that provides a generic resource management policy for a system that allocates disk in proportion to service popularity, focusing mainly on systems that utilize the disk as a cache.

## 7. SYSTEM ARCHITECTURE

This section details the techniques used in the implementation of the write buffer system. Although the system matches the characteristics mentioned in the properties section very closely, we have restricted the *promised* evacuation bandwidth to be fixed across all services in-order to simplify the system. It may however be enhanced to allow service specified evacuation bandwidths, which is discussed in detail in the future work section of this paper.

**Initial Condition.** Any service that accesses the buffer system is initially given a minimum amount of disk space.

The system also enforces a minimum acceptable evacuation bandwidth requirement at which every service must evacuate. This bandwidth is precisely the *promised* evacuation bandwidth that is identical across all services accessing the system. Every service, thus by default, promises this minimum evacuation bandwidth. The buffer also enforces a *maximum evacuation time*/*eviction timeout* that is fixed across all services that each service must evacuate its data in.

**Write Requirements/Minimum System Guarantees.** In this system, writes by service/extensions succeed if the services meet the minimum system guarantees discussed here. If a service that accesses the system wishes to write data to disk, the measured bandwidth by the system for that particular service must be greater then or equal to the minimum acceptable evacuation bandwidth specified by the buffer and provided that the service has space to write data to disk. If the evacuation bandwidth is lower than the minimum evacuation bandwidth, the write is blocked. This makes the system more robust since it is capable of filtering out those services that use the system irresponsibly or maliciously.

**Limit Calculation.** In such a system, allocations are based on the *soft limit* of a given service. This system utilizes dynamic limits that vary from service to service and access to access. In order for this system to work and prevent denial of service, the limits are assigned in a way that creates meaningful division of disk across the various services. This system calculates the limit for a particular service by considering its priority and its evacuation bandwidth. Specifically, the *soft limit* for each service's space is equal to,

$$\textit{Soft limit} = CP*PEB*MET*k$$

Where CP = Current Priority,
PEB = Promised Evacuation Bandwidth,
MET = Maximum Evacuation Time,
$k$ = Current Scaling Factor

**Allocation Increase Requirements.** The buffer system uses the *soft limit* for a particular service to decide whether extra space should be allocated to the service or not. If the service meets the minimum system guarantees mentioned above, the write succeeds. But if the service requires more space to write its data to disk, then the buffer refers to the services *soft limit*. If the *soft limit* for the service is greater than the services current allocation plus the current request size, the write succeeds and the service is granted enough space to fulfill its request. If however, the total space is larger, the write is rejected.

**Disk Scaling.** The buffer system assigns limits to the services based on their priorities and their promised evacuation bandwidth. In such an environment, at any given moment when the buffer succeeds a write request to the disk, the sum total of all the allocations may exceed the disk space present. In such a scenario, the buffer system must scale down the allocation amount for all of the services to account for the larger total allocated space on disk. In this manner, every service still receives its *fairshare* of disk space whilst not the exceeding the total space available by the buffer.

**Data Eviction Requirements.** The current system gives services a lot of flexibility in evacuating their data. Specifically, it implements a per-transmit failure warning to the service to notify the service of possible data eviction. We describe the interval between the time a service stops transmitting its data and the time at which part of its data is discarded as the *eviction timeout* for the system. Under the current implementation, this timeout is fixed across all services but may be enhanced later to allow variable timeouts across each of the services.

**Data Eviction Amount.** Under the current system implementation, if a service is punished for not evacuating its data on time, its space is reduced to its current limit calculated by the system. This is correct since at any given point we would want all services to consume less than or equal to the limit currently calculated by the system.

**Network Disconnections.** The buffer system also monitors network connectivity and verifies if the buffer is connected to the network. If the buffer at any given moment disconnects from the network, the time period for which it remains disconnected is not charged towards the service data's time on disk. If however, for any particular service or group of services, the data destinations/target remote servers are unreachable, the time for the data on disk is charged towards the service data's time on disk and if it exceeds the *eviction timeout*, the space allocation for the particular service or group of services is reduced as mentioned above. We anticipate improvements to this part of the system as well. These improvements are discussed in later sections of this paper.

## 8. SYSTEM IMPLEMENTATION

Using the above mentioned system logistics; our goal is to build an effective write buffer management system that fulfills all the requirements already discussed.

As mentioned earlier, write buffering is achieved in this system by applying the evacuation bandwidth and per-service popularity policies. Using these policies, we have built a system that meets most if not all of the requirements stated in section 5 of this paper.

The Buffer System uses a hierarchical approach of dividing the components of the system where the services/extensions issue write requests through a single

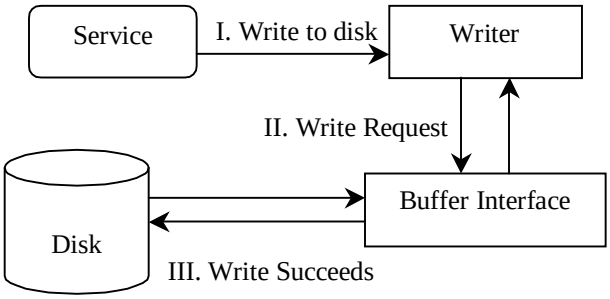


Figure 2. Steps I - III show how a successful write request propagates through the system.

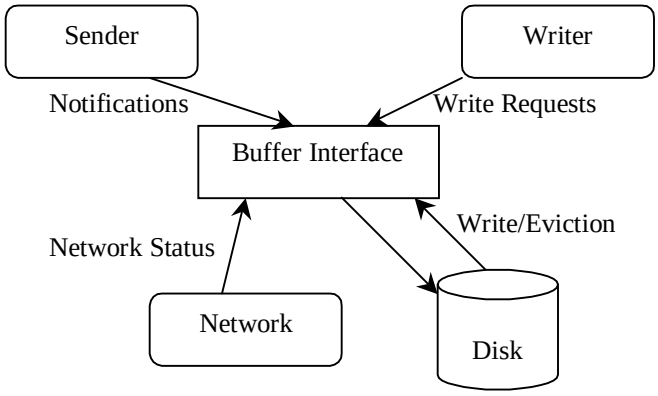


Figure 3. Shows the buffer interface interaction with the rest of the components.

interface to the buffer system. The Write Buffer system implementation consists of essentially three main components: *Write Buffer Interface Module, Writer Module, Sender Module*. These components are individually discussed in the subsections following this one.

**8.1 WRITER**

The Writer Module sends write requests by different services to the buffer system. The Writer is the only interface that the services interact act with. Requests specifically consist of *Service Name*, *Data to be written to Disk*, and *Destination to send the data*. All services communicate with the Write Buffer Interface through the Writer.

The Writer although part of the Write Buffer system, operates independently of the Buffer system and is solely used to transfer write requests to the Buffer system. The transmission is done by opening connections to the write buffer interface that waits for write requests from writers. Under this infrastructure, every write request by a service constitutes a separate Writer Module that transmits the request to the Write Buffer Interface.

**8.2 WRITE BUFFER INTERFACE**

The Write Buffer Interface module is the most important component in the whole system. This module performs the following functions. (i) Registers services/receives write requests from *Writers* (ii) Sinks data to/evicts data from disk. (iii) Receives network status notifications and (iv) handles callbacks from the *Sender*, taking appropriate actions when necessary. The Write Buffer interface stores service histories at two levels in the hierarchy, the service level and the disk level. The service level maintains a hash table database of service specific information including service *priority*, *measured evacuation bandwidth* etc. The disk level is managed by a Disk Manager that contains a database that stores service *allocations* on disk and *file* specific information for the data on disk. Both these repositories are utilized by the write buffer interface to provide the necessary functionality described earlier.

**Service Registration.** When the Write Buffer Interface receives a request from the Writer, it registers the service to record its history. This is done by inserting the service specific information into the hash table of histories and updating the allocation data structure as well. Next time a write request is received from the same service, the Buffer System refers to the service history to decide whether service writes succeed or not.

**Processing Writes.** When the Write Buffer Interface receives a write request from a service past the first time the service accessed the disk, it first checks to see if it meets the *minimum system guarantees*. If so, it writes the data for the service to disk. If the request is larger than the space allocated to the service, the buffer calculates the services current *soft limit*. If the services aggregate space does not exceed its calculated *soft limit* (section 7), the write succeeds, else it is rejected.

**Network Status Notifications.** The Write Buffer Interface system maintains a persistent pinging module that periodically checks for network connectivity and notifies the buffer system if there are network changes. These notifications allow the buffer system to decide whether to charge disk allocation time to the services or not.

**Sender Notifications.** The Write Buffer Interface also receives notifications from the sender on whether a send for a service succeeded or not. If sends are successful, the write buffer interface de-allocates the corresponding data from disk. If unsuccessful, the buffer interface charges the time to the service data's time on disk.

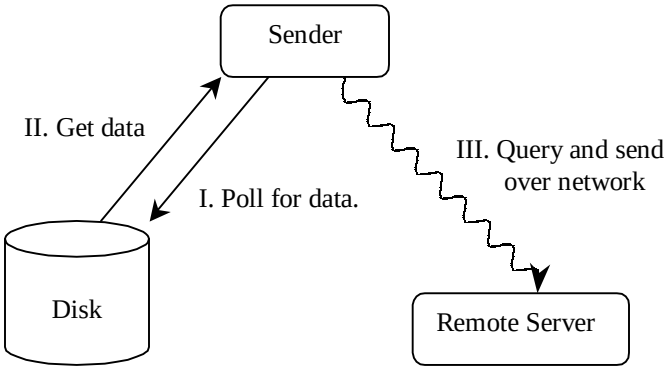


Figure 4. Steps I - III show the execution steps of the sender.

### 8.3 SENDER

Sending in this system is achieved by creating a separate thread that is dedicated to sending any data on disk to the desired destination. The Sender specifically,
(i) Queries the system to see if there are any data to be sent, (ii) Queries the network for connectivity, and (iii) Sends notifications to the write buffer interface notifying it whether attempted sends were successful or not.

**Querying Data.** The Sender queries for files by simply looking up the Disk Manager database that contains service names along with their corresponding data. If data are available to send, the sender sends the data (in the form of files) for the service in a linear fashion.

**Write Buffer Notifications.** Whenever the Sender attempts to send data for a particular service, it notifies the Write Buffer Interface on whether the send was successful or not. The Write Buffer Interface receives these notifications and takes the appropriate actions.

**Query Network.** The Sender also queries the network before attempting to send any data. This prevents the sender from trying to send data during periods of disconnection.

## 9. EXPERIMENTS

In this section we analyze the behavior of the system by conducting simulations to observe the changes in allocations of the services with respect to time. By observing these micro-benchmarks we are able to determine if the system behaves according to specification.

### 9.1 MICRO-BENCHMARKS

This section details several experiments that were performed to analyze the system behavior with respect to the expected behavior of the system. Each of the individual benchmarks is discussed in detail in the sections that follow.

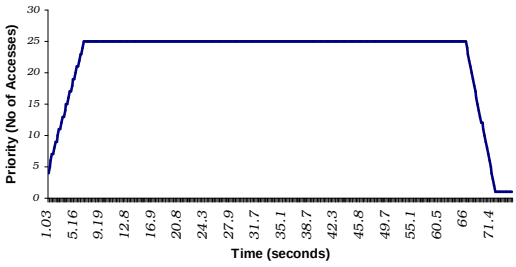


Figure 5. Shows how the priority for a program changes over time.

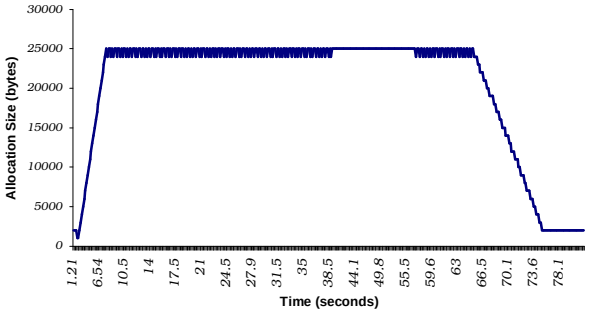


Figure 6. Shows how the allocations change for a single service. Corresponding priority illustrated in Figure 5.

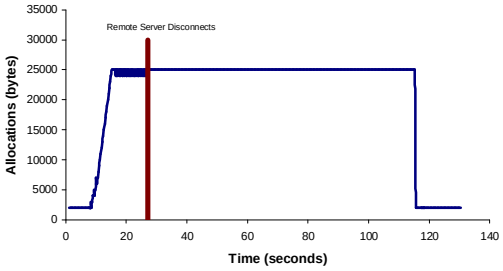


Figure 7. Illustrates system behavior against malicious services. Corresponding priority illustrated in Figure 5.

#### HIGH BANDWIDTH SERVICE

In this section we describe the most basic execution of the system where a service maintains a consistently high evacuation bandwidth as its priority changes over time. As is expected, we see that as service priority increases, allocations increase proportionately. And as the priority decreases, the service allocation also decreases. This experiment obviously does not model a situation where the priority changes are drastic. In such cases, we would

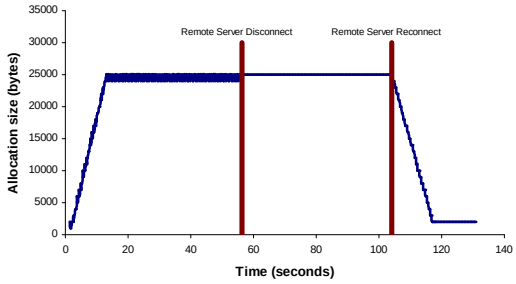


Figure 8. Illustrates an experiment testing network failures. Corresponding priority illustrated in Figure 5.

want the expected behavior seen in Figure 10. In this experiment, although the services priority has decreased from 25 to 1, it's allocation only changes at the rate proportional to the services promised evacuation bandwidth.

**LOW BANDWIDTH SERVICE**

In this section we detail a test that describes how the system behaves towards a service that maintains low evacuation rates and does not evacuate its data on time. Figure 7 illustrates such a service. Initially, the remote server is reachable and the services allocations increase as the priority for the service increases. But later the remote server disconnects and the buffer is unable to send the data to the destination. In this scenario, the buffer allots the service a time period (discussed in section 7) within which to evacuate its data. This is the *eviction timeout* period after which the services data are evicted. This experiment clearly marks how the system behaves towards hostile services that allocate huge amounts of space on disk and fail to meet their bandwidth promises. This experiment models how the write buffer system protects against *denial of service attacks*.

**NETWORK FAILURE EXPERIMENTS**

Here we model two different experiments that simulate how the system would behave in the presence of network failures or slow downs on the buffer end (Section 7). In both cases we will see that the buffer system implements a fair policy whereby services are not penalized for network failures closer to the system.

The first of these series of experiments presents what happens when the buffer experiences a disconnection from the network (Figure 8). In this experiment, during connectivity, as service priority increases, the allocations for the service increase. After the priority for the service stabilizes, the buffer abruptly disconnects from the network. In such a scenario we would expect that the service data not be evicted even though it's priority has decreased which is exactly what is seen. After the buffer rebuilds a connection to the network, the data is evacuated from the disk and the allocations for the service decrease

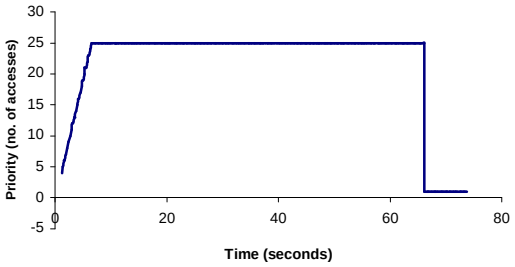


Figure 9. Illustrates abrupt priority changes for a service.

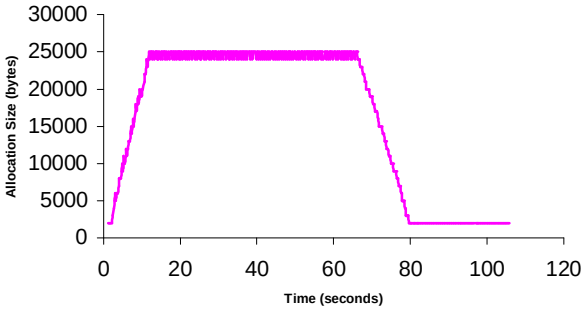


Figure 10. Illustrates allocation changes corresponding to the priority graph shown in Figure 9.

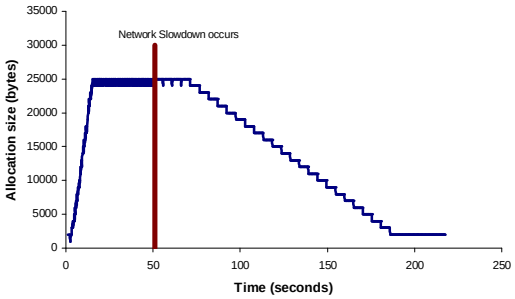


Figure 11. Illustrates system behavior during network slowdown. Corresponding priority illustrated in Figure 5.

since the priority for the service has decreased.

The second of these series of experiments models a situation where the buffer experiences a network slow down causing the sending bandwidth for the buffer to fall below the minimum evacuation bandwidth (Figure 11). Initially, when the network is providing peak bandwidth, the allocations/priorities are consistent with each other but as soon as the network slows down, the decrease in allocations for the service are much slower than the decrease in its the priority. This is exactly what is expected since we would not want to penalize services for

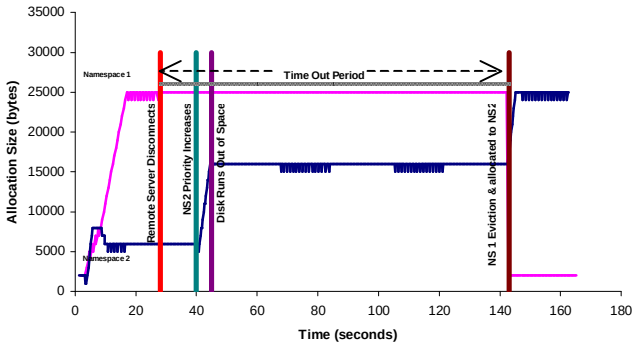


Figure 12. Figures shows a combined experiment where two services compete for disk space (max. Disk Size = 40 Kbytes).

not maintaining the promised bandwidth since the buffer itself cannot operate at the minimum acceptable bandwidth discussed in section 7.

### 9.2 A DETAILED EXPERIMENT

Lastly, we describe a combined experiment that builds upon the benchmarks we discussed in the previous sections. This experiment is modeled in Figure 12. We see that initially service 1 maintains a higher priority than service 2 and thus is allocated greater space by the buffer. However, after a while the remote server for service 1 disconnects and the buffer is unable to send service 1's data. Once again, the system allows service 1 a timeout within which to evacuate the data before they are discarded. Meanwhile service 2's priority increases and its corresponding allocation increases until the buffer runs out of space. Thus, service 2 cannot be allotted more space until space is reclaimed from service 1. After the *eviction timeout* for service 1 expires, its data are evicted and the resulting space is allotted to service 2.

The above experiment is a classical example shown where we see that service 1 which exhibits malicious behavior consumes allot of disk space but later is kicked out by the system to allow other services access to it. Compare this to the traditional techniques of static and demand-based allocation, and our proposed system performs much better and allocates disk space to services in a much more meaningful manner.

## 10. DISCUSSION

The write buffering system we have discussed in this paper is aimed at preventing *denial of service* by implementing an algorithm that provides *sensible* allocation of disk to services. However, in its implementation we came across numerous difficulties and subtleties. Of-course, enhancing the system to make it more robust is an open area of research. At any rate, we think that describing some of the limitations of the system would be helpful to anyone who may later pursue this area of research or decide to enhance the system discussed here.

## 11. LIMITATIONS

The buffer system described in this paper has been implemented in a very simplistic manner without adding additional complexities into the design. The policies implemented for the system are overly simplified. As a result the system has numerous limitations. These limitations are briefly discussed here.

Our implementation of popularity for the write buffer system uses a simplistic approach and overlooks many subtleties. Our simplistic view does not take into account the rate at which services may write data to disk or their aggressiveness. In such a system, two services with identical priorities may be allocated different amounts of space on disk based on their *agility* [13] (the frequency at which they access the buffer) and time at which they accessed the buffer. In this scenario, a more aggressive service may be allocated more space than one that is less aggressive.

Our fixed bandwidth implementation policy is also too simplistic since a fixed bandwidth may not work well across all services/extensions. Some services may not be able to *promise* even the *minimum evacuation bandwidth* enforced by the system. For such a system to work, the *min. evacuation bandwidth* would have to be chosen carefully.

Finally, under the current system, we have adopted a simplistic approach by which the system detects network connectivity. Specifically, the system does not attempt to measure network bandwidth but simply polls (pings) a set of well-defined servers to determine if the network connection is up or not. If the system is unable to reach any of the destinations, it concludes that the network is down. Such an approach disregards accurate measurements of bandwidth both on the buffer and server ends of the network.

We would also like to mention an important point here regarding any priority-based system; any system that allocates priorities based on popularity provides opportunities for other activities to *game* the system. In addition, program priorities may not necessarily be accurate indicators of program popularity. For example, a client using the system may decide that the email service is most important to him even though he only accesses the service once a day. Although priority does not capture the precise picture, we have implemented this policy to make the system as simplistic as possible as a tradeoff for simplicity over optimality.

## 12. RELATED WORK

The work that has been described in this paper is closely related to a much wider area of resource management. Much effort has been spent in building automatic and precise resource management tools for *fair* allocation and

*control* of resources at all levels of the hierarchy. These areas of resource accounting include memory management, network bandwidth measurement, CPU etc. A discussion of a generic resource management framework has already been proposed in Dahlin [2]. In their paper, the authors have proposed a resource management system applicable to all resources that services may utilize in a disconnected environment with a detailed discussion on a cache-based disk management prototype.

As previously mentioned, such resource management techniques are applicable to systems that provide location independent services that may execute anywhere in the network. Systems advertising such extensible and location independent services are directly impacted by the resource management system presented here. Many projects have been proposed to this end that include Active Names [1], Active Networks [6] and Active Services [7]. These systems are based on providing flexible location independent services (generally termed Active Content) that may be executed at the client, server, proxies, etc. Such services rely on a resource management infrastructure that provides control mechanisms for the various services to prevent resource hazards as a result of buggy or malicious code.

There has been little contribution to effective write buffering for disconnected operation but hopefully this paper will be used as a basis for a more in-depth study on this topic. Among a few that attempt to solve similar resource management problems at the disk level include Khoja [8] and Dahlin [2], that propose a similar per-service popularity algorithm for disk resource management surrounding *mobile extensions* [3].

## 13. CONCLUSIONS AND FUTURE WORK

In this paper we have described a strategy to provide disk resource management for a system exploiting write buffering for disconnected operation. As we have previously discussed, this system is very simplistic and may be enhanced to build a more robust disk management system. However, we also find that although our proposed per-service popularity and evacuation bandwidth scheme does not create *perfect* allocations, it manages disk amongst competing services much more effectively then many traditional techniques presented in this paper.

An area of future work could include enhancing the current system to make it more robust. A few enhancements that may be attributed to this system are discussed further.

First, one may use network bandwidth measurement as a scaling factor in scaling the evacuation rates of the services accessing disk. At times of slow network activity, the system may decide to scale evacuation bandwidths down for all the services to account for the decrease in the overall send bandwidth of the network. Bandwidth measurement tools that may be employed for this purpose are detailed in Baker [4] and Savage [5].

Second, we may enhance the system by allowing a service specified bandwidth to provide greater flexibility and availability to the services. A control mechanism however would still be required in such an environment to prevent services from *over-promising* and then failing to meet their promises.

Third, in-order to build a more work conserving system the priority for each service could be scaled over the average priority across all services to allow easier accommodation of newer extensions. In such a scenario, a significant tradeoff in space would have to be made to allow such a system to be implemented such that the buffer would have to set aside ½ of the disk space as free to all services and the other half controlled by it. In such a system, in the worst case, a malicious service could consume a maximum of ½ of the disk. Such an implementation may be considered tolerable considering the amount of disk space available today.

Finally, this paper is a great source of benefit for anyone trying to build a complete resource management system for the execution of commonly downloaded un-trusted code to clients operating in a disconnected environment.

## ACKNOWLEDGEMENTS

I would like to especially thank Dr. Gouda, without whose guidance and moral support this thesis would not be possible. My appreciation also goes to all the other people in this area of research. To Amol Nayate and Bharat Chandra for assisting in the clarifications regarding the Active Names System, and distributed web services in general. I owe a great deal to Usman Shuja and Mirza Omer Beg for their assistance throughout the project, which helped make this work possible. Lastly I would like to thank Dr. Dahlin: Thank you for your patience and continuous support, which helped me in more ways than I can imagine.

## REFERENCES

[1] A. Vahdat, M. Dahlin, T. Anderson and A. Aggarwal. Active Naming: Flexible Location and Transport of Wide-Area Resources. *In proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, October 1999

[2] Bharat Chandra, Mike Dahlin, Lei Gao, Amjad-Ali Khoja, Amol Nayate, Asim Razzaq, Anil Sewani. Resource Management for scalable disconnected access to web services. WWW10, May 2001.

[3] Mike Dahlin, Bharat Chandra, Lei Gao, Amjad-Ali Khoja, Amol Nayate, Asim Razzaq, Anil Sewani. Using Mobile Extensions to Support Disconnected Services. Technical Report TR-2000-20, University of Texas at Austin.

[4] K. Lai and M. Baker, Nettimer: A Tool for Measuring Bottleneck Link Bandwidth. *In proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, California, March 2001.

[5] Stefan Savage, Sting: A TCP-based Network Measurement Tool. *In proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems.* pp. 71-79, Boulder, CO, October 1999

[6] David Wetherwall, Ulana Legedza, and John Guttag. Introducing New Network Services: Why and How. Int eh IEEE Network Magazine, Special issues on Active Programmable Networks, July 1998

[7] Elan Amir, Steven McCanne, and Randy Katz. An Active Service Framework and its Application to realtime Multimedia Transcoding. *In proceedings of the SIGCOMM*, September 1998.

[8] A. Joseph, A. deLespinasse, J. Tauber, D. Gifford, and M. Kaashoek. Rover: A Toolkit for Mobile Information Access. *In proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995

[9] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. ACM Transactions on Computer Systems, 10(1):3-25, February 1992.

[10] IBM Corporation. Mqseries: An introduction to messaging and queuing. Technical Report GC33-0805-01, IBM Corporation, July 1995. ftp://ftp.software.ibm.com/software/mqseries/pdf/horaa101.pdf.

[11] G. Kuenning and G. Popek. Automated Hoarding for Mobile Computers. *In proceedings of the sixteenth ACM Symposium on Operating Systems Principles*, pages 264-275, October 1997.

[12] Active channel technology overview. http://msdn.microsoft.com/workshop/delivery/channel/overview/overview.a%sp, 1999.

[13] Brian D. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, K. Walker. Agile Application-Aware Adaptation for Mobility. *In proceedings of the sixteenth ACM Symposium on Operating Systems Principles*, October 1997