INCODA–INCREMENTAL HOARDING AND REINTEGRATION IN MOBILE ENVIRONMENTS

By

ABHINAV KHUSHRAJ

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2001

To my Mom.

ACKNOWLEDGMENTS

I express my sincere gratitude to Dr. Helal for all the encouragement, guidance and above all the motivation that he has constantly provided. He has been of real help providing me with the right direction for this thesis work and writing. Without him the thesis would not have been possible.

I also thank Dr. Hammer and Dr. Su for being on my thesis committee. I would also like to thank the people at Harris Lab for their guidance and support during this course.

I specially thank Jinsuo for helping me out at many points during this thesis especially with incremental hoarding and its evaluation.

I especially thank my friend Sovrin for the fruitful discussions that we had on this topic and for the constant support he provided. I thank Amar, Raja and Subodh for their great company.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

INCCODA – INCREMENTAL HOARDING AND REINTEGRATION
IN MOBILE ENVIRONMENTS

By

Abhinav Khushraj

August 2001

Chairman: Dr. Abdelsalam Helal
Major Department: Computer and Information Science and Engineering

Disconnection is one of the popular techniques for operating in mobile environments and is here to stay for sometime until long-range wireless connectivity becomes a reality. However, disconnection requires periodic *hoarding* and *reintegration* of data, which raises performance issues especially during weak connection. A common hoarding and reintegration mechanism involves complete transfer of contents. In order to hoard and reintegrate efficiently, an incremental approach is being introduced to do data transfers based on the delta between changes. Data objects are differentially transferred in either direction – to hoard from the server to the client, and to reintegrate changes made while disconnected from the client to the server – and are patched on the receiving side to generate full copies.

QuickConnect and MoneyConnect are two important and useful features that are also being introduced to allow mobile users to control the amount of connection time and money spent during hoarding and reintegration.

Performance evaluation of this system proves that it does efficient incremental hoarding and is beneficial on any type of connection. Analysis also shows that incremental reintegration is particularly beneficial in the weakly connected mode of operation.

The incremental hoarding and reintegration setup is built within the Coda File System of the Carnegie Mellon University to replace the full file transfer mechanism with the incremental approach, based on the Revision Control System (RCS).

CHAPTER 1
INTRODUCTION

1.1 Significance of the Problem

In the present computing age users demand constant availability of data and information which is typically stored on their workstations, corporate file servers, and other external sources such as the WWW. With the increasing popularity and prevalence of mobile computing, mobile users are demanding the same when only limited network bandwidth is available, or even when network access is not available. Moreover, given the growing popularity of portables, laptops and personal digital assistants (PDA), mobile users are requiring access to the data regardless of the form-factor or rendering capabilities of the mobile device they choose to use.

In this space, three broad challenges are imposed by mobility [HEL01]:

1. Any time, anywhere access to data, regardless of whether the user is connected, weakly connected via a high latency, low bandwidth network, or completely disconnected

2. Device-independent access to data, where the user is allowed to use and switch among different portables and PDAs

3. Support for mobile access to heterogeneous data sources such as files belonging to different file systems and/or resource managers.

This work tries to solve the first challenge – access to data anywhere, anytime, which is a step towards ubiquitous computing and facilitates management of data in different connectivity modes – strong, weak and no connection.

1.2 Goal and Approach

In today's computing world, networks provide very high performance and bandwidth and are highly transparent. The network bandwidth is no longer the rate-determining step for applications and systems. However, this argument holds true only for the fixed and wired networks. To reach the same level of network performance and efficiency over mobile networks is still a far-fetched reality and so we have to explore ways to counter the challenges posed by mobility.

Mobile devices are in general resource poor and computationally starved when compared to desktops and their counterparts. They have smaller memory and persistent storage space. Their weight, power and size are much smaller and lesser [SAT93]. The biggest challenge is, however, the network. Mobile elements have to operate under a very broad range of network conditions. The connection may vary from full range 10mbps on 802.11b wireless LANs to 56kpbs modem connection from home to 0bps in disconnected mode while traveling. The connection may also be intermittent and users may want to disconnect to save connection cost and power consumption.

To operate disconnected, two important mechanisms are required – *hoarding* and *reintegration*. *Hoarding* is the process of caching important and relevant user data onto mobile devices for use while operating disconnected. While disconnected, the cache in the mobile device might be updated due to changes made by the user and applications running on the mobile device. *Reintegration* is the mechanism by which all these updates are synchronized with the fixed network upon reconnection.

Users invariably operate under weak connection (56kbps ranges), for example –

- At home when users want to work over a weekend they have to rely on a slow modem connection that ranges from 28kbps to 56 kbps.

- When high-flying executives want to collaborate from the airplane with peers on the fixed network, they have to use the expensive phone connection that provides very little bandwidth.

- Mobile users on the 802.11b wireless LAN have very weak connection when they are in the periphery of the coverage area. Moreover, there is intermittence owing to their movement into and out of range of the LAN access points.

Under weak connection, hoarding and reintegration is time consuming owing to the minimal bandwidth available. Especially, hoarding and reintegrating large files over slow networks may not be desirable in many situations. Another important aspect is that users make very little changes while mobile. If there is a text file or a program file they are working on, they update only few lines while being mobile.

The main goal of this work is to do incremental hoarding and reintegration by exploiting the fact that users make minimal changes while they are mobile. In this approach, we do differential transfer of data objects (files, databases, etc.) instead of hoarding and reintegrating full data objects. To this end, we use a version control system to compute and maintain object differentials.

We also provide two useful tools – QuickConnect and MoneyConnect that will aid the mobile user to make conscious decisions regarding the time and money spent for reintegration. QuickConnect is a tool that gives the mobile user the ability to specify the connection time for reintegration. MoneyConnect, in the same lines, lets the user specify the amount of money he is willing to spend for reintegration. This is especially useful when the cost model is based on packets transferred (e.g., iDen packet data or GPRS).

The entire work that has been done in this thesis is based on the Coda File System of the School of Computer Science at the Carnegie Mellon University. Coda is a distributed file system that has a user level cache on the client, called *Venus* that is used

to support disconnected operation. This client cache services all the file system calls from the users and applications on the Coda namespace. Before disconnection, the cache files are *hoarded* from the Coda server, *Vice* into the user cache. The operations that are done on the cache during the disconnected period are logged persistently and are *reintegrated* into the Vice upon reconnection. During hoarding and reintegration, files are transferred in full from the server to the client and vice versa. When the network connection is strong, this is not a particular problem. However, if these transfers were to happen under weak connection, it would impose unnecessary load on the network, because we only need to send those parts of the files that were changed while disconnected and do not have to send complete files. This is the basis of incremental hoarding and reintegration

To be able to do incremental hoarding and reintegration we need to keep track of the different versions of the files on both *Vice* and *Venus*. Revision Control System (RCS) [TIC85] is a software tool originally developed by Walter. F. Tichy of Purdue University that assists with this task of version control by automating the storing, retrieval, logging and identification of revisions. We use RCS functionalities to maintain the various file versions and to compute the file differences between changes.

1.3 Organization of the Thesis

Chapter 2 of the thesis discusses related work on how to operate disconnected and how to optimize the use of weak connectivity. In Chapters 3, we set up the base for the incremental concept by discussing differing connectivity levels, Coda's states and model and Revision Control System (RCS). In Chapter 4 we will discuss in detail the how the Coda File System has been redesigned to support incremental hoarding and reintegration.

We evaluate performance results of the incremental approach in Chapter 5. Chapter 6 concludes the thesis and suggests future work.

CHAPTER 2
REVIEW OF RELATED WORK

Significant work has been done in the area of disconnected operation and managing connectivity. Researchers have long back identified the usefulness of the disconnected model and have produced many interesting results and came up with various prototypes such as Ficus and Coda. Hoarding and reintegration are two areas that have aroused considerable interest in the research community. In the industry, IntelliMirror™, built into Microsoft® Windows® 2000 operating system, supports disconnected operation. In this chapter, we will discuss some interesting research and industry efforts that have been carried out in disconnected operation, hoarding techniques and in managing varying connectivity during hoarding and reintegration.

2.1 Update Notification/Propagation and Reconciliation in Ficus

The Ficus project at UCLA is a distributed file system that provides replication facility with optimistic concurrency control. Ficus is a peer-to-peer system that allows updates under network partition. The updates are done in a non-serializable fashion called *one-copy availability* that allows updates for any copy of the data, not requiring a minimum number of available copies [GUY90]. This replication technique is used to provide high availability in an environment that has frequent communication interruptions. Any file or directory in the file system may be replicated at any set in of Ficus hosts.

Every replica in the file system has a *file-identifier* number identifying a logical file represented by a physical set, and a *replica-identifier* that identifies that replica. Also associated with each replica is a *version vector* that holds the updates history of the replica.

Updates are applied first to a single physical replica. The update makes an entry in the *version cache* for the current update. An update propagation daemon checks this cache and periodically propagates updates. For regular files, the updates are done atomically by creating a shadow copy first to allow the complete file to transfer. The shadow copy then replaces the original version. Update propagation for directories is a little more complicated and Ficus uses a *directory reconciliation* algorithm. A reconciliation algorithm gets the new operations performed at a remote host and inserts or deletes accordingly on the local copy.

As can be noted, the updates are using full file transfer causing unnecessary burden on the network, especially when Ficus aims at servicing users over large and wide-area networks.

2.2 Disconnected Operation in IntelliMirror™

IntelliMirror™ [MIC99] is a powerful tool built in Microsoft® Windows® 2000 that provides change and configuration management. The three main features that it provides are

- User data management
- Software installation and maintenance
- User settings management

It provides a mechanism by which user data automatically follows the user whether he is online, connected to the network or offline. IntelliMirror stores the selected data in specified network locations that makes it appear local to users.

Data are placed in specific network locations. When users want to disconnect they specify the option to make these folders available offline. A local copy of these files is made and then the user can move with his mobile device continuing to work on the offline folders.

When the user comes back to the network, IntelliMirror identifies if any changes have been made and if so prompts the user asking him permission to synchronize. All the folders on the network location are updated upon confirmation from the user. If a particular file has been changed both on the network and the users local computer then the user is prompted to resolve conflicts and is given the option to save the files as he wishes.

The “Make Available Offline” can be compared to hoarding and ‘Synchronize’ to reintegration in our context. However, the point to be noted here is that issues about network connection have not been addressed. IntelliMirror assumes that network connection is never a problem.

2.3 The InterMezzo File System

InterMezzo is a distributed file system that is capable of journaling updates and versions made at the client while connected and disconnected. A client module in the kernel intercepts the updates and the associated details and writes journal records about them.

When the client loses network connection either due to disconnection or a network or server failure, InterMezzo client starts operating disconnected and makes journal records for any updates that are done. Upon getting back connection all the updates are forwarded from the server into the client for those changes that have been done on the server while the client is disconnected. Following this, the client then reintegrates all of the changes made while disconnected into the server [BRAa].

This system does not provide any feature to hoard before disconnection and so if users happen to disconnect involuntarily there may be some portions of the client files that may not be available while he is operating disconnected.

2.4 SEER: Predictive File Hoarding

To operate disconnected a good quality cache has to be hoarded into the client that results in least number of cache-misses. Hand specification of the files to be hoarded is not practically viable as it would require computer expertise and understanding of which files have to be hoarded and so this mechanism would fail to reach out to common users who want transparent mechanisms. An LRU scheme, though it is very appealing, is not very effective as the cache miss penalty is heavy and very often causes all the on going work to come to a stall if an important cache file is missing.

SEER is a system that predicts and hoards user cache based on semantic information that it gathers from user file access pattern. It identifies files that are *naturally related* such as a set of C program files, and *incidentally related* files like the editor and the C compiler. It uses these relations to create *semantic distances* that are used to ensure that the necessary files will be hoarded into the cache.

Semantic distance is a measure that quantifies the relationship between any two files. The closer the relation between two files is, the smaller the semantic distance will be. The semantic distance is based on measurements of individual *file references*. A file reference is considered to be a high level operation, such as an open or status inquiry on a particular file. In the SEER system semantic distance is computed on the basis of the sequence of file accesses. It uses a simple heuristic that operates in constant time and linear space, and still discovers useful reference relationships.

The SEER system is implemented with a small modification to the UNIX kernel and a bunch of user-level processes. The kernel module logs system calls that are placed in a trace buffer that are read by the *observer* process. It then passes the log information to a *correlator* process that computes the various semantic distances and builds a database of these relationships to be used by a *cache manager*. The cache manager runs periodically and makes file clusters based on these semantic relationships. It then chooses appropriate file clusters that are most likely to be needed on the mobile device and hoards them into its cache [KUE94].

2.5 Support for Hoarding with Association Rules

This method uses data mining rules to identify the set of data items that are to be hoarded into portable computers prior to disconnection. Data mining techniques can be used to create associative rules based on the users access history. These rules would then represent the users access pattern on the cache and can be used to determine what contents need to be hoarded.

Client request history can be divided into sessions. In each session there is a pattern of client's requests. Data mining techniques are used to find the patterns and produce rules to build a rule base of associations.

Partitioning of the history into sessions is done in two ways. The *flat approach* extracts data irrespective of who requested particular data. The *user-based approach* separates the client request history on the basis of the specific users who requested the objects. A window-based approach called the *gap* is used to separate session boundaries. When more than a threshold amount of time separates two consecutive requests they are supposed to be part of different windows. In the flat-approach, however, windows are separated by a fixed time period but in the user-based approach the gap-based approach is used to separate client requests. With these sessions and times[1] standard data mining algorithms (like *Apriori*) are used to find the association rules.

Candidate sets and *hoarding set* are then obtained using the associative rules just obtained. Candidate sets are those that are eligible for being hoarded into the client for the next disconnected session and are identified using an inferencing mechanism. These are distinguished by priorities assigned to them at the time of creating associative rules. The hoarding set has those entire candidate sets identified on the basis of the priority that can fit into the client's cache [SAY00].

2.6 Operation-based Update Propagation

Operation-based update propagation as shown in Figure 2.1 is especially designed for dealing with weak and flaky network connections. It efficiently transfers changes made to large files over a weak connection. In this technique, file system changes are

[1] Windows are mapped to *sessions* and data requests are mapped to *items*

collected above the file-system level and during weak connection are passed on to a surrogate client that is strongly connected to the server. The surrogate then re-executes the changes and then transfers over the changes to the server. Changes are sent directly to the server from the client if the connection is strong or if the re-execution fails on the surrogate.

This approach takes place in four distinct steps [LEE99]:

1. *Logging of user operations:* The user operations that are logged are high level commands issued by users that can be intercepted, logged and replayed later. In application-transparent logging, in which the applications are non-interactive like compilers and linkers, the logging of operations can be done without modifying the applications. The shell executing the applications is modified such that it logs the user operations by issuing two newly implemented ioctl commands - VIOC_BEGIN_OP and VIOC_END_OP at the beginning and end of an operation. Application-aware logging requires modifications to the application and has not been addressed in this work.

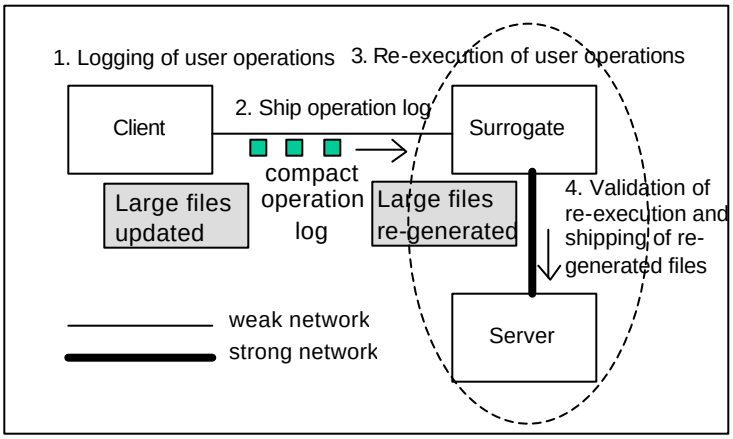

Figure 2.1: An Overview of Operation Shipping

2. *Shipping of operation log:* The reintegrator that is responsible for the normal reintegration process is modified to support this system. It does a UserOpPropagate RPC to transfer the packed operation log before trying to reintegrate directly to the server using value-shipping. If the RPC fails then the client continues to reintegrate in the regular fashion over the slow network.

3. *Re-execution of operations:* The surrogate client upon receiving the user operations puts the corresponding volumes[2] in the write-disconnected[3] state and then it re-executes the operations by pawning a new user thread called the re-executor. The operations are reintegrated at the surrogate after they are validated. If at all any of the volumes fail the reintegration process at any stage the updates are discarded and the failure of the RPC is intimated to the requesting client. Those operations that execute on the surrogate exactly as they would on the requesting client, the operation is said to be repeating re-execution. The system tries to facilitate repeating re-executions by trying to give every operation the same execution environment (shell, environment-variable, files, etc.), but it does not guarantee it.

4. *Validation of re-execution:* After re-executing the operation log, the re-executor creates a re-execution log that captures all the mutations made by

[2] In Coda, a volume is a collection of files forming a partial subtree of the Coda name space.

[3] In write-disconnected state the files can be retrieved from the servers but any changes made locally cannot be written to the servers.

it. To validate the re-execution, the original execution log of the client is compared with the re-execution log. It assigns fingerprints to each log entry and validates it if the fingerprints match in the two logs.

The contribution of operation-based propagation is noteworthy, as it particularly addresses the problem of reintegration during weak connectivity that is at the heart of this work. This system's performance tests prove that it efficiently uses the network traffic and reduces the elapsed time. However, it is not transparent and requires user intervention in many specific scenarios and would not be a feasible solution if the user does not have enough expertise.

CHAPTER 3
CODA AND RCS

This chapter discusses how varying connectivity affects portable devices, what Coda is and how it works, and describes the use of RCS in the context of this work. Special emphasis is laid on the design of Coda with respect to disconnected operation, hoarding and reintegration in weakly connected environments. This chapter concludes with the discussion of RCS that will be used for computing and maintaining file differences and versions respectively.

3.1 Types of Connectivity

In the foreseeable future, mobile clients will encounter a wide range of network characteristics in the course of their journeys. Cheap, reliable, high-performance connectivity via wired or wireless media will be limited to a few oases in a vast desert of poor connectivity. Mobile clients must therefore be able to use networks with rather unpleasant characteristics: intermittence, low bandwidth, high latency, or high expense [MUM95].

Networks that are not limited by such shortcomings are generally called *strongly connected*. We mean that the network is not responsible in any way for any performance degradation in the systems and applications that are using it. With the current technology, such high-quality networks with unlimited bandwidths are usually found only in Local Area Networks (LAN).

Mobile networks are inherently *weak* and are challenged by all the above factors. Strong connectivity for wireless networks is a distant reality and weak connection may always be a characteristic of mobile computing.

To counter the challenges posed by lack of connectivity, researchers have come up with interesting mobile computing models. One of the earliest and most impressive models is the disconnected operation model that came into importance due to constant network and server failures. Its implications were soon identified for supporting mobility by Dr. M. Satyanarayanan who pioneered the research of this model in the classic work called Coda a distributed file system that supports disconnected operation for mobile clients.

Disconnected operation is a mode of operation in which the client continues to operate by accessing data that is locally stored during temporary failure or absence of shared data repository. It is based on caching wherein important and useful data is cached locally thereby increasing availability and performance [KIS92].

Disconnections can be of two types: involuntary disconnections that are caused by network and server failures, out of range of connectivity and line-of sight constraints and voluntary disconnections that are caused when users choose to not have network access for their portable computers. Disconnections are caused by handoff blank out (> *1ms* for most cellulars), drained battery disconnection, battery recharge down time, voluntarily disconnected to preserve battery power, theft and damage. The ways these are handled are almost the same except that user expectation and co-operation are likely to be different.

However, disconnected operation has its own limitations and drawbacks [MUM96].

1. Updates are not visible to other clients
2. Cache misses may impede progress
3. Updates are at risk due to theft, loss or damage
4. Update conflicts become more likely
5. Exhaustion of cache space is a concern

Coda exploits weak connectivity to offset these problems. In the next sub-section we will discuss about Coda and how it manages the different types of connectivity.

3.2 Coda

Coda is a distributed file system that has many features that are highly desirable by network file systems. One of the most important and valuable contribution of Coda is support for disconnected operation. In the following few sub sections we will discuss the client-server model of Coda, its different operating states and how it adapts to varying connectivity.

3.2.1 Client-Server Model

Coda is based on the classic client-server computing model. It is designed for a large collection of untrusted clients and a much smaller number of trusted servers [KIS92, SAT90]. It provides high availability by server *replication* and *local file caching*. Each Coda client is called *Venus* and has a local cache on which it relies for all its file accesses. All files are grouped into *volumes*, each forming a partial subtree of the entire namespace and typically containing the files of a single user or a group. The client uses RPC2, a variant of RPC implemented at CMU, to communicate with the servers while it

is connected. However, it may be not be able to access the server in the event of network or server failure or if the mobile client has been detached from the network.

Coda consists of a group of replicated servers each called a *Vice*. The entire group of servers is collectively called the *volume storage group* (VSG). The subset of the VSG that is available to a particular client at a moment is called the *accessible volume storage group* for that client (AVSG). The client is disconnected when its AVSG becomes zero.

When the AVSG becomes zero for a client, all the file system operations are served from the its persistent cache. The Client Modification Log (CML) records those operations made while Venus is disconnected and replays this log on the Vice upon reconnection. In the event that Venus is unable to serve a particular file request made by an application or a user it reports a file miss. The ongoing work might be impeded and in such an event the extent of damage caused depends on how critical that file is for a particular task.

The reason why Coda has server replication (first-class replication) in spite of the presence of local file caching (second-class replication) is that first-class replicas are of much higher quality, are more persistent and reside on robust servers. Second-class replicas are inferior to the servers in all these aspects and are subject to loss. It is therefore required that the reliable and robust first-class replicas be maintained.

To support second-class replicas Coda uses the *optimistic* replication strategy in which the writes can be partitioned allowing reads and writes everywhere. However, for it to be viable Coda provides conflict detection and a resolution mechanism for the conflicting writes that will be caused by this strategy. The rationale behind using an optimistic strategy is that conflicting write operations constitute a very small proportion

of the total file accesses and because Coda is primarily targeted towards academic environments wherein each user has his own private space in which all files are kept.

3.2.2 Structure of Coda Client

Venus is a user-level client cache manager that services all the file requests made on the Coda namespace from its persistent cache. Venus has a tiny kernel module, the Coda FS driver that redirects all the file system requests from the kernel to the user-space and vice versa. The Coda client cache is stored generally on the local file system (e.g., Ext2 FS) in the configuration directory `/usr/coda/venus.cache`.

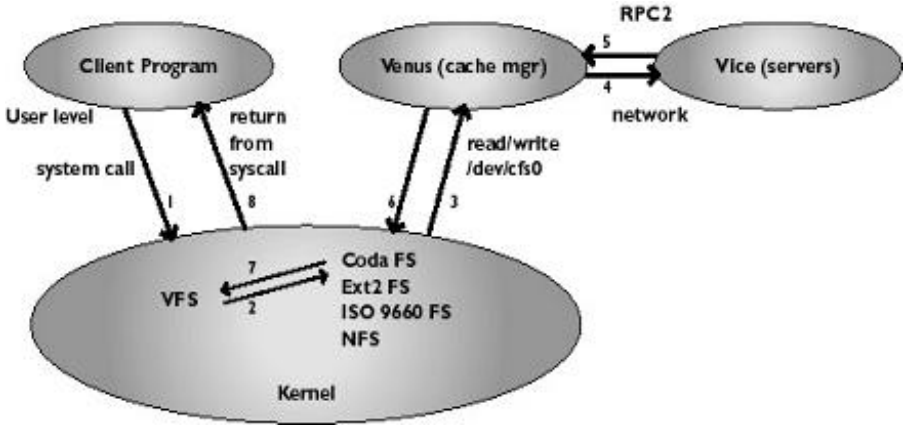

Figure 3.1: Structure of a Coda Client

Figure 3.1 shows the structure of a Coda client and how a file system call is serviced that is initiated from a user application program [BRAb]. A file system call (*read, write, open, close* and others in the Unix context) that is made by the client program is trapped by the operating system. Generally, the Virtual File System (VFS) layer of the operating system intercepts such a file system call in the kernel. The VFS redirects the system call to the Coda FS driver. The driver then communicates with Venus by passing messages. Coda uses a character device /dev/cfs0 for Venus to read from the driver and vice-versa. The kernel driver makes an upcall using a message

structure passing the current file system call to Venus. Venus now searches in its Coda cache and tries to service the system call from its cache contents. If it fails to find the file in the cache or if the cache contents are stale, Venus makes an RPC call to the AVSG to fetch the latest version of the file from one of the available Vices. If the contents are found in the cache or when the RPC returns, Venus makes a downcall to the Coda FS driver and replies to it about the system call that it is currently serving. The Coda FS driver then responds to the VFS layer that in turn returns the file request successfully to the user application that had initiated the file system call. However, in the event that the client is disconnected, Venus will report a cache miss to the user if an RPC was required to service the system call.

3.2.3 Venus States

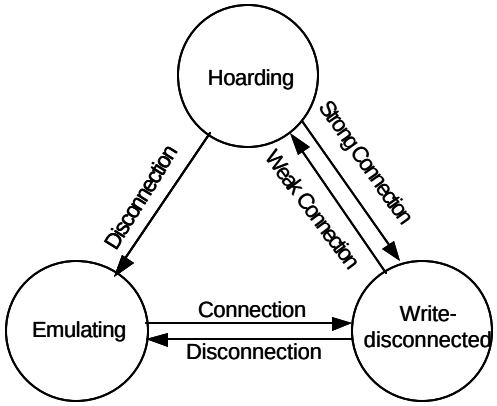

Figure 3.2: Venus States and Transitions

Figure 3.2 shows the different states in which Venus exists and how the various state transitions occur when the mobile client operates disconnected [MUM95]. Venus is normally in the hoarding state trying to preserve its cache coherence. It is responsible for caching all important and useful data for operating with minimal cache misses during disconnection. Some of the hoarding techniques have been discussed earlier in Chapter 2.

In Coda, hoarding is done by prioritized cache management and by hoard walking. Hoard priorities are assigned to file objects based on the user's activity. There is a *spy* utility that logs the recent activity of the user on the file system and assists users to specify the list of files that are to be hoarded in the per-client *hoard database*. Venus periodically does a *hoard walk* of the cache to ensure that high priority objects are in the cache and that the cache objects are consistent with the servers.

When Venus disconnects from the network its cache begins to emulate the server and services all file requests. Any file request that cannot be serviced from the cache is reported to the user as a cache-miss. All of the file operations that are made while Venus is disconnected are logged in the Client Modification Log (CML).

Upon reconnection, Venus starts its process of *reintegration*. The CML is packed together and sent to the Vice via an RPC and the file operations made while disconnected are replayed on the server side. For any `store` operations that are replayed, the Vice does a callback fetch on the corresponding Venus and fetches the actual file contents associated with that `store` operation. While disconnected, however, Venus optimizes its CML by canceling corresponding `store` and `unlink` operations. This way it can reduce the number of operations that have to be replayed and the callback fetches that will be made by the Vice. This is the basic reintegration mechanism in Coda and occurs during the write-disconnected state. In the write-disconnected state, files can be fetched from Vice to Venus but the updates from Venus cannot be propagated to Vice immediately. This state was introduced later in Coda with the aim to support weak-connectivity that is the topic of discussion in the next sub-section.

3.2.4 Handling Weak-Connectivity in Coda

Coda uses two main techniques to handle weak connectivity. They are *rapid cache validation* and *trickle reintegration*. The write-disconnected state discussed above is essentially designed to manage weak connectivity in Coda.

Coda's cache coherence is based on the concept of *callbacks,* which is a promise by the server to the client to notify it when the cache object in the client becomes stale. The message sent for invalidation is called a *callback break*. However, during disconnection there are no callbacks. So when a client reconnects after a period of disconnection it has to validate each of the objects it has in its cache. During weak connection this imposes unnecessary load on the network, wastes lot of connection time and increases latency. To counter this, Coda raises the granularity at which it performs validation. Instead of performing validation for each and every object, it validates entire volumes. This removes the need to validate every object in a volume if the entire volume has been identified to be valid. This method is called the rapid cache validation.

However, the primary mechanism by which Coda handles weak connectivity is by trickle reintegration. Trickle reintegration is a mechanism that propagates updates to servers asynchronously, while minimally impacting foreground activity [MUM95]. In the write-disconnected state, the behavior of Venus is a combination of its connected and disconnected behaviors. While file operations are logged into the CML, updates are also propagated to the Vice. During weak connectivity reintegration is a constant background activity and so it is termed *trickle reintegration*.

In trickle reintegration, instead of packing the entire set of updates in the CML and sending it to the Vice for reintegration, it now packs only those operations in the log that have *aged*. Aging is a mechanism that ensures that records have spent a sufficient

amount of time in the log and have been candidates for log optimizations. By separating the foreground activity from the slow propagation of updates, trickle reintegration improves the system for operation in weak connectivity. Figure 3.3 shows the CML while weakly connected [MUM95]. A is the aging window. The records that are in the shaded are being reintegrated while those outside have to spend time equivalent to the aging window before they can be reintegrated. The reintegration barrier separates records that are eligible for reintegration from those that are not.

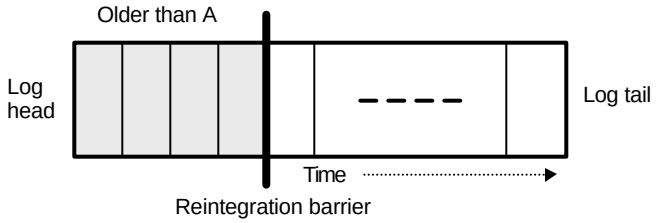

Figure 3.3: CML During Trickle Reintegration

3.2.5 Implementation of Specific Tools for Coda

We will discuss now some of the tools and mechanisms that were built to support Coda and its disconnected operation.

1. RPC2 is a Carnegie Mellon implementation of Remote Procedure Call. All communication between servers and between server and client is done using RPC calls.
2. Transaction Mechanism: Coda is a strict, stateful file system. Unlike NFS, a stateless file system, Coda operates strictly based on transactions. This guarantees atomicity to file operations.
3. Recoverable Virtual Machine (RVM) is a Coda file system mechanism to preserve the state of the file system. RVM strictly traces the state of the

file system hierarchy. All the Coda data structures are maintained by the RVM. The RVM is stored persistently so that Coda can be brought back into the same state in which it had left upon restarting.

4. SmartFTP (SFTP) is a file transfer protocol and is a side effect of the RPC2. It is a variant of FTP and it is used for all file transfers within Coda. We have used this for all the file transfer in our incremental approach.

3.3 Revision Control System

To support the incremental approach we have to maintain all the file versions that are created at the server. Saving each and every file would be inefficient and managing each version would not be easy. Instead, a good version control system is used in which files can be checked in and checked out for specific versions. Version control systems maintain a modification log relative to one initial version. Therefore versioning effectively stores only one physical version that is highly efficient and especially useful in the incremental approach where we want to maintain revisions for each file. Some of the popular version control systems on the Unix side are SCCS, RCS and CVS. After careful consideration we chose RCS because of its simplicity and ease of use.

RCS assists with the task of keeping software system consisting of many versions and configurations well organized [TIC85]. A detailed description of the commands that RCS provides can be looked up in the manual page of RCS. Some of the common commands and those that have been used in this work are *ci, co, rlog, rcs,* and *rcsdiff.*

Apart from the RCS commands we also use two other standard Unix utilities – *diff* and *patch. diff* is a utility that is used to compare the difference between two files and

generate a difference file specifying the line numbers where new lines have been added or removed. *patch* takes a patch file containing a difference listing produced by the *diff* program and applies those differences to one or more original files, producing patched versions. Normally the patched versions are put in place of the originals.

CHAPTER 4
INCREMENTAL HOARDING AND REINTEGRATION

This chapter presents the motivation and basis for using the incremental approach and describes our design and implementation that has been adopted for hoarding the files into the client and propagating updates back to the server.

4.1 Motivation for the Incremental Approach

Disconnected operation is based on the assumption that users operate without connectivity for short periods of time using their locally stored copies for lack of network connection or due to high connection costs. A businessman may work disconnected on some incomplete document while away from office, a developer may be writing programs from home on a weekend using his locally cached copies. While disconnected, users have some common characteristics –

- They frequently work on the same set of files that they had created when they were connected;
- They generally make very little changes to their files during the short periods of disconnection;
- In order to save connection cost users have the general tendency to work on their files or write emails without being connected. They then connect for a small period of time and synchronize their mobile device with the outside world.

This gives us the opportunity to exploit the common user tendency to make minimal changes while they are disconnected.

In Coda, in spite of remaining in the write-disconnected state and decoupling the foreground activity from the slow and continuous propagations of updates by doing trickle reintegration, it still suffers from a limitation: the updated files are propagated in their entirety [LEE99]. Though the response time for the user decreases the actual propagation of the changes generates heavy traffic. If a hoarded file in Venus becomes stale, then Venus simply discards it and fetches the whole up-to-date version of this file from Vice. In a high-speed LAN this does not have much effect but in a mobile environment the client is usually a laptop or a PDA connected through a slow modem connection that affects the time it takes to hoard and reintegrate complete files.

In the incremental approach we compute the changes made while disconnected. Upon regaining network connection, instead of sending the entire files on which changes have been made, we only send the changes that were computed. On the receiving side, instead of dumping the older versions, we apply to it the changes that we just received and generate the new file. Since the changes made are generally small, avoiding resending the complete files and sending only the changes saves the network bandwidth. The network savings may not necessarily be useful on a high-speed LAN network but network traffic is considerably reduced on a slow connection resulting in quicker synchronization and lower connection costs.

The incremental approach implemented by us is for transfer of file changes in either direction – from server to client and vice versa.

4.2 Incremental Hoarding

Hoarding that is done based on the incremental approach is called incremental hoarding. When Venus caches a file by hoarding, it gets a callback promise from the Vice that it will intimate the client if the cache version becomes stale. If two Venii are holding copies of the same file and if one of them updates that file, Vice makes a callback on the unchanged Venus. This Venus does a callback break by dropping the version it has and fetching the latest version from Vice. In order to reintegrate incrementally, this client will have to fetch the difference instead of the complete up-to-date version.

To support incremental hoarding, both Venus and Vice had to be modified. Our basic approach is to identify the Coda file transfer mechanism and modify it so that it transfers the difference files. We identified the RPC calls in Coda that are invoked for performing file transfers during hoarding. While hoarding, we use the same RPCs that are being used by Coda. However, we replace the full file that was being transferred with the difference file before making the RPC call as will be explained below.

For incremental hoarding we have to maintain every version of a particular file at Vice. We could store every version of the file but that would be a storage overhead. Instead a good version control system is being used. RCS that was introduced in Chapter 3 is used to maintain the versions of a file. The basic mechanism is to check-in the latest version of a particular file into the RCS server and later use the RCS server to compute file differences between file versions.

Whenever a new file is created at Vice, the first version of the corresponding RCS file is also created. This is done by the RCS check-in command *ci*. When newer versions of this file are created, they are simply checked-in into the corresponding RCS server. More on this will be discussed in incremental reintegration.

The thick arrows in figure 4.1 indicate the heavy network load between the client and server when it does not use the incremental approach. In figure 4.2 and 4.3, we indicate how the client and the server have been modified in order to support incremental hoarding and reintegration. The thin lines are an indication of the efficient network usage.

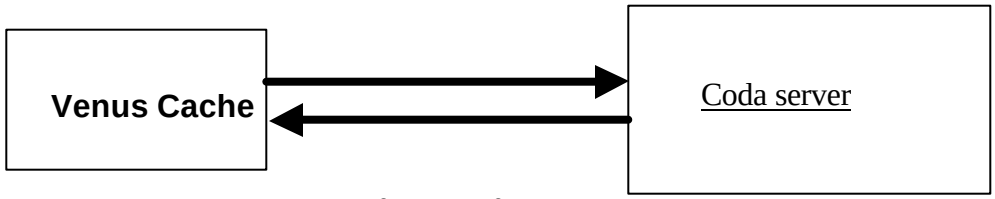

Figure 4.1: Regular Full-file Transfer between Coda Server and Client

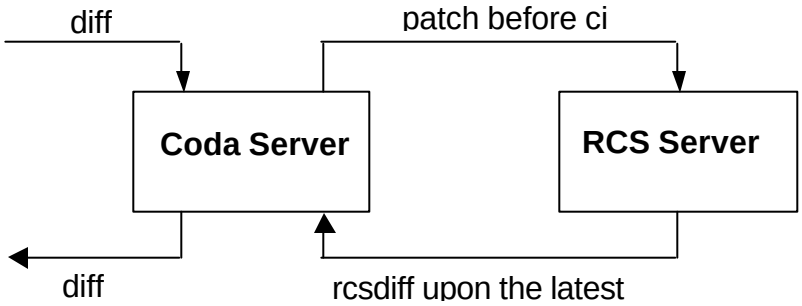

Figure 4.2: Modified Vice Supporting Incremental Approach

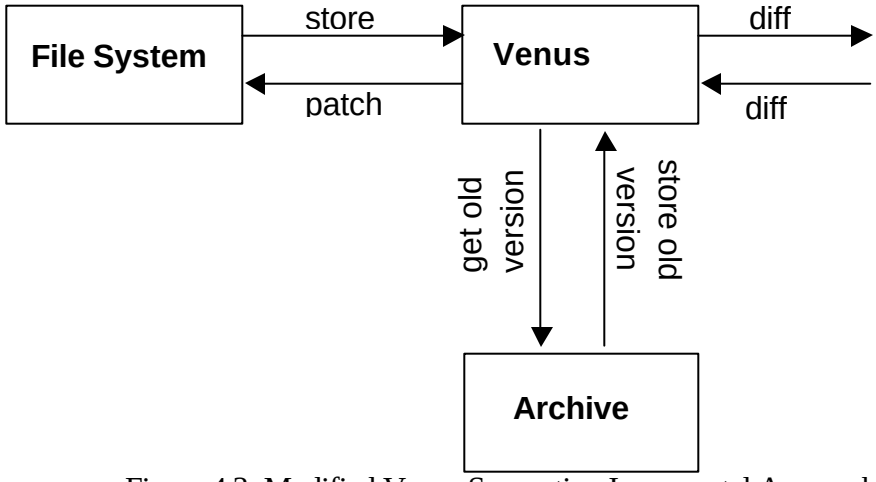

Figure 4.3: Modified Venus Supporting Incremental Approach

An archive.info file is created in `/usr/coda/archive` if it does not exist already which holds the version information of the files along with the complete path name in the Coda namespace. When a particular Venus file gets a callback break from the server, it makes a `ViceFetch` RPC call from the `Fetch` function in Venus to the Vice requesting the newer version of the file. Before it makes the RPC call Venus checks in its archive.info if it has an older version of the file that is about to be fetched. If so it prepares to fetch only the difference file instead of the complete file. To do so it reads the version number from archive.info and sets it in the `dataVersion` vector of the `ViceStatus` object. It also changes the name of the file[1] to be fetched in the `SFTP_Descriptor`. [2] The RPC call is made now to fetch the difference by passing the connection id, file id, version number and other parameters as described in the function signature below.

```
ViceFetch (IN ViceFid Fid,
           IN ViceVersionVector VV,
           IN RPC2_Unsigned InconOK,
           OUT ViceStatus Status,
           IN RPC2_Unsigned PrimaryHost,
           IN RPC2_Unsigned Offset,
           IN RPC2_CountedBS PiggyCOP2,
           IN OUT SE_Descriptor BD);
```

At the Vice, the procedure that handles this RPC call is `FS_ViceFetch`. The steps carried out by the RPC for doing the fetch are

1. Validate Parameters
2. Get Objects
3. Check Semantics (Concurrency Control, Integrity Constraints, Permissions)

---

[1] The difference files are given temporary names that is the name of the original file suffixed with *.diff*

[2] SFTP_Descriptor is a data structure that holds the parameters like name, size, file descriptor etc of the file that has to be transferred.

4. Perform Operation (Bulk Transfer, Update Objects, Set Out Parameters)
5. Put Objects

In step 4, the parameters of the bulk transfer function `FetchBulkTransfer` are modified so as to pass the version number along with it. Inside this function, before doing the actual file transfer using SFTP, it computes the difference file if the version number received is greater than 0. `rcsdiff` computes the difference file by using the version number against the latest version in the RCS server for that file object.

The `SE_Descriptor` is modified so that the file to be transferred is set to the difference file instead of the latest file available at the server. The difference file is now transferred to Venus by passing the modified `SE_Descriptor` to the SFTP RPC called `RPC2_CheckSideEffect` that does the file transfer.

At Venus, after the RPC returns, it retrieves the file it has in the archive corresponding to the version number in the archive.info. The difference file obtained from the RPC call is patched to this archive version and the latest file is generated that matches the latest version that is at the server. This file is now made available in the Venus cache and is accessible from the Coda namespace.

For a `remove` operation the particular file object that is being deleted is archived. The file is removed from the cache and the relevant RPCs are called to delete the file and its RCS server from Vice.

Similarly, if a file is renamed it is handled gracefully by changing the name of the archive file at the client and by changing the name of the RCS repository at the server.

4.3 Incremental Reintegration

Incremental reintegration is used to transfer changes from the clients to the servers efficiently based on the incremental approach. In a weakly connected mobile environment reintegrating by transferring complete files lays a heavy burden on the network. In Coda, all `STORE` operations have to be reintegrated to the server so that the server maintains cache coherence across the distributed file system. `STORE` operations occur at the client while the user is connected or when he is disconnected.

To implement incremental reintegration within Coda we used a similar approach as we did for hoarding incrementally. We identified those RPCs that are responsible for performing the `STORE` operations on the client.

There are two types of store operations in Venus. The `STORE` operations performed while the client is connected are categorized as connected stores and are handled by the `ConnectedStore` function. The `STORE` operations performed when the client is disconnected are called disconnected stores and are handled by the `DisconnectedStore` function. The RPCs called in each of these are different.

In `ConnectedStore` the `ViceStore` RPC originally transferred the newly created files in its entirety. However, to support incremental transfer of the files we create the difference file between the new version just created at the client and the old version that existed before the change was made.

To be able to create the difference file, Venus needs to maintain an archive that has the one of the older versions of this file object. If the current `STORE` operation is the first `STORE` on this file object then it has no old versions in its archive and the file has to be transferred in its entirety. This archive is generally stored in

`/usr/coda/archive/`. The entire Coda namespace from `/coda` is archived in this directory at `/usr/coda/archive/coda`. This is also the archive that is used by the `Fetch` function for incremental hoarding as explained earlier.

The `ConnectedStore` function uses the *diff* utility of Linux that is `exec`ed from within Venus to calculate the temporary difference file. It also calculates its new length and changes the file name in the `SFTP_Descriptor` that carries information about the file that will be transferred by the RPC.

The RPC called for the file transfer is `ViceStore.` This RPC has been modified so that it checks out (using *co* of RCS) the appropriate version from the RCS server of that file and applies the difference file that it obtains by calling `StoreBulkTransfer`. This function is responsible for doing the actual file transfers between the server and the client brings the difference file to the server. This difference file is patched to the checked-out version to generate the new version. The new version hence generated is checked in (using *ci* of RCS) back into the RCS server. Following is the signature of the `ViceStore` RPC.

```
ViceStore (IN ViceFid Fid,
           IN OUT ViceStatus Status,
            IN RPC2_Integer Length,
           IN RPC2_Unsigned PrimaryHost,
            IN ViceStoreId StoreId,
            IN RPC2_CountedBS OldVS,
           OUT RPC2_Integer NewVS,
            OUT CallBackStatus VCBStatus,
           IN RPC2_CountedBS PiggyCOP2,
            IN OUT SE_Descriptor BD);
```

The basic steps carried out by `ViceStore` at Vice are the same as described earlier with `ViceFetch.`

`DisconnectedStore` is responsible for logging all the file operations that are performed during disconnection. This is called the Client Modification Log (CML) and it is also stored persistently. The `STORE` operations are logged in the CML along with their length. We have modified this length that is logged in the CML to the length of the difference file that will be generated when connection is restored and reintegration occurs. We calculate the length of the difference by measuring the length of the difference between the cache file and the archive file. If another `STORE` happens on the same file while continuing to be disconnected, the new `STORE` will replace the old `STORE` also setting the appropriate length.

When connection is restored, Venus makes a transition from the Emulating state to the Write-disconnected state and starts reintegration of all the changes made while disconnected. The CML is packed and sent to the server to be replayed there. However, Vice does a `CallBackFetch` for all the `STORE` operations that it replays. The CallBackFetch RPC does a transfer of those files that correspond to a `STORE` in the CML. We modified the `CallBackFetch` RPC at Venus so that it transferred just the difference file that is computed as described in `ConnectedStore.` On the receiving side in Vice the new file is generated in the same way as it is for `ViceStore` – the difference file is patched to the corresponding RCS server version and the new file is generated and checked-in into the RCS server. The signature of the CallBackFetch RPC is as follows:

```
CallBackFetch (IN ViceFid Fid,
               IN OUT SE_Descriptor BD);
```

File removal and file rename operations are handled in the same way as described above.

4.4 Reintegration Control with Time and Money

In Coda, reintegration is an automatic process that initiates automatically and goes on until all the changes made while disconnected have been reintegrated. Reintegration after disconnection does not take a lot of time on a high-speed network and therefore doing things automatically without user intervention is desirable. However, if a mobile client is reintegrating on a weak connection, it may take a long time for the updates to be propagated to the servers. In such situations the user has to be given the choice to control reintegration. In QuickConnect and MoneyConnect we have identified two types of common necessities of users – to control the time spent during reintegration and the amount of money spent to propagate updates respectively.

4.4.1 Reintegration Controlled with Time

During weak connection the time taken for update propagation increases as the average number of packets transferred per second decreases. In a typical scenario a mobile user makes updates to the cache contents while disconnected for a period of time and then reintegrates over a weak connection from a modem. Even though doing incremental reintegration will reduce the time taken for reintegration the user may still want to specify a period after which he wants to stop reintegration. Also if the user is reintegrating from an airplane over an expensive phone connection, he would like to limit the connection time and hence the reintegration time.

We have provided a useful feature that does just this. QuickConnect is a feature that allows user to specify the time for which reintegration has to be carried out. We have used the Venus utility `cfs` to provide QuickConnect. The user can specify the time for which reintegration has to be carried out in the following way.

```
%cfs quickconnect <reint-time(s)>
```

To implement this we modified the `cfs` utility and added the `quickconnect` command. The `cfs` command makes a `pioctl` that in turn makes the file system `ioctl`. When making the `pioctl` call the time specified for reintegration is passed on to `ioctl`. Venus services this `ioctl` system call. Venus handles this by doing reintegration of one volume at a time and also notes the time spent. It reduces the remaining time for reintegration by the amount spent reintegrating the last volume. When no more time is remaining Venus stops reintegration and remains in the write-disconnected state until another reintegration is requested.

4.4.2 Reintegration Controlled with Money

With the recent advent of pricing models based on the number of packets transmitted we identified the need to give user control over reintegration so that he is able to specify the amount to be spent for reintegration. Common scenario these days is that mobile users work disconnected for short periods of time and then they connect to the Internet using their cellular phones. Some cellular services nowadays have a cost model based on the number of packets that come in and go out of the phone. In this case it does not help much by just being able to specify the time for reintegration. Users want to control the amount of money spent in this pricing model. MoneyConnect is a tool that allows the users to specify the amount of money that can be spent for reintegration.

Like QuickConnect, MoneyConnect is also implemented by modifying the `cfs` utility. To command used to do reintegration by MoneyConnect is

```
%cfs moneyconnect <amount($)>
```

MoneyConnect is also initiated as a `pioctl` call. The amount to be spent is converted into the max number of bytes that can be transmitted with that money. This

parameter is passed along with `pioctl` to the `ioctl` system call. Venus services this `ioctl` call by calling `FailReconnect`. A global variable is maintained to keep track of the number of remaining bytes that can be reintegrated. This variable is reduced each time after a Venus Callback RPC by the number of bytes that have been transferred by the last call back fetch. When the number of bytes remaining reaches zero Venus calls `FailDisconnect` and stops reintegration.

CHAPTER 5
PERFORMANCE EVALUATION OF INCCODA

The goal of the experiments for incCoda was to demonstrate quantitatively how the incremental approach is better than the full file transfer approach. We will first discuss performance of incremental hoarding followed by reintegration.

5.1 Performance of Incremental Hoarding

Our experience with hoarding has been positive. The network performance has improved owing to the smaller deltas that are being transferred in place of the complete files. We did the following experiment to evaluate incremental hoarding against full file hoarding.

A random sample of 100 files was chosen from the source code of the Linux kernel. One set of 100 files was chosen out of the version 2.2.5 and another set from version 2.2.12. We chose Linux source files since they are a good example of a developer making changes from one version to another. The developer while disconnected may have upgraded from one kernel version to another. As we know that Linux kernel changes little from one version to another and so are good candidates to demonstrate incremental hoarding.

Five groups of 20 files each are made for conducting the experiment. The scenario of updating is as follows: Initially we have two disconnected clients both working on the older version of the source code. While one of them is disconnected he upgrades his files to the higher kernel version. When he reintegrates to the network all the updates are

propagated to the server. Later when the other client reconnects, his cache contents are invalidated and the newer version of the source files are hoarded into his cache.

The payloads of the files were compared and the results obtained are show in Figure 5.1. The results are expressed as a percentage of the transferred payload of the incremental approach out of the total payload using the full file approach. The results indicate that incremental hoarding is always beneficial. In the best case the incremental payload is less than $1/20^{th}$ of the payload otherwise.

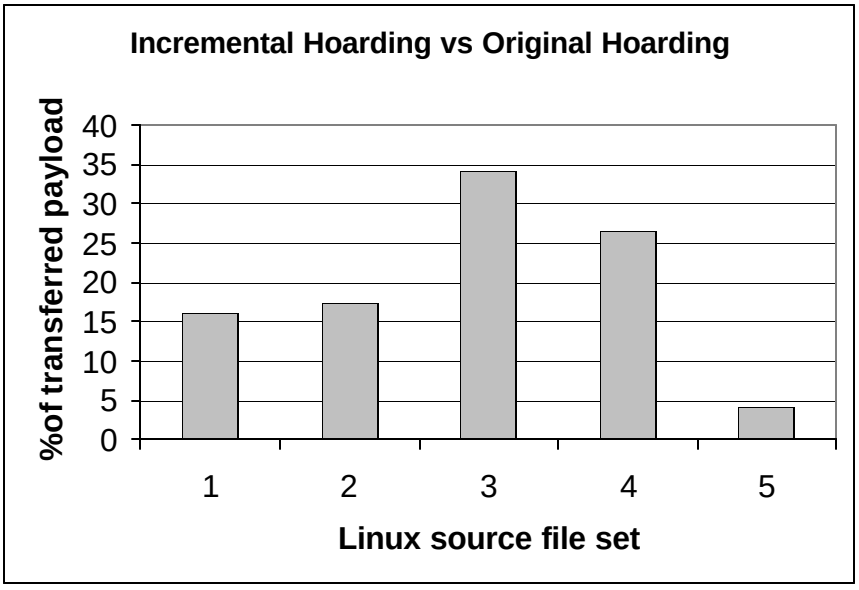

Figure 5.1: Payload Comparison Between Incremental Hoarding and Original Hoarding

From the calculations above we would like to present the benefits in a real world scenario. Lets say an IT company that has a lot of mobile users within the company roaming with their mobile devices from place to place within the premises of the organization. These users rely on Coda for their server needs and operate disconnected while mobile. In this scenario lots of mobile users are constantly connecting and disconnecting from the network and hoarding and reintegrating changes as they move

from one place to another. If the average number of updates made by a mobile user is 5MB and the actual size of the contents is about four times the size of the updates (the average value from Figure 5.1) then the average bandwidth consumption per user is reduced to one fourth. The implication of this is that since the average network usage is reduced the network can support up to four times the number of users with the same bandwidth. Also in another sense it will provide these mobile users faster hoarding thereby saving the user and the organization lot of useful time.

5.2 Performance of Incremental Reintegration

Using incremental reintegration considerably reduces the network traffic caused due to reintegration. The performance in our experiments improved by one order as opposed to using the full file reintegration.

The subject files that we chose were Linux kernel source from above and also email files. We used one single set of 100 Linux source files in this experiment. The other set of subject files that we used were day-to-day email files. We used the email files in PINE of a typical user on his disconnected mobile device. While disconnected we moved some emails from one folder to another causing moderate changes to the email files. We used about ten email folders and the total size of all the email files was about 6MB.

The idea behind using email files is that users often write emails while disconnected and actually send them upon reconnecting. Also users want to maintain the same email folders seamlessly on both – their mobile devices and the fixed network. So if the user is managing his emails while disconnected moving some of them from Inbox to another folder, the corresponding email files are updated. When he reconnects he would like to synchronize the email folders with that on the fixed network thereby requiring

reintegration of the email files. We believe that incremental reintegration will be particularly useful for email files since the updates are a small percentage of the actual file size. The results of our experiments prove the truth of this statement.

The experiment done to measure the network traffic for Linux source files is as follows. A mobile user while connected has one of the older versions of the kernel source on his mobile computer. He disconnects and while away he installs a newer version of the kernel source files in the same directory location. When he reconnects the files are reintegrated and the network traffic generated is measured in each case – incremental and full file transfer.

Table 5.1: The network traffic generated (in bytes) without and with the incremental approach for the Linux sources and the PINE email folders.

Linux kernel source files		PINE Email folders	
Without incremental (bytes)	With incremental (bytes)	Without incremental (bytes)	With incremental (bytes)
4369370	570134	12604004	615136
4356584	628010	12635300	554648
4403780	553954	12837716	577352
4322542	582818	12768556	674152
4313856	556624	12774556	539468

To measure the network traffic we use `rpc2tcpdump`. This is a utility similar to `tcpdump` and is provided by the Coda developers at CMU. `rpc2tcpdump` dumps the entire network traffic of RPC2 communication between the Coda client and the server. The port on which the network traffic is generated for reintegration can be specified and

the network traffic is dumped for all the client-server communication that takes place during reintegration. Care has been taken that Venus or Vice was not involved with some other network activity on those ports at the time of reintegration.

We repeated the experiment five times for each of the subject files using both reintegrating methods and obtained very encouraging values. The network traffic recorded is shown in table 5.2. The reduction in network traffic is by one order of magnitude. Figure 5.2 is a plot of the values in Table 5.1.

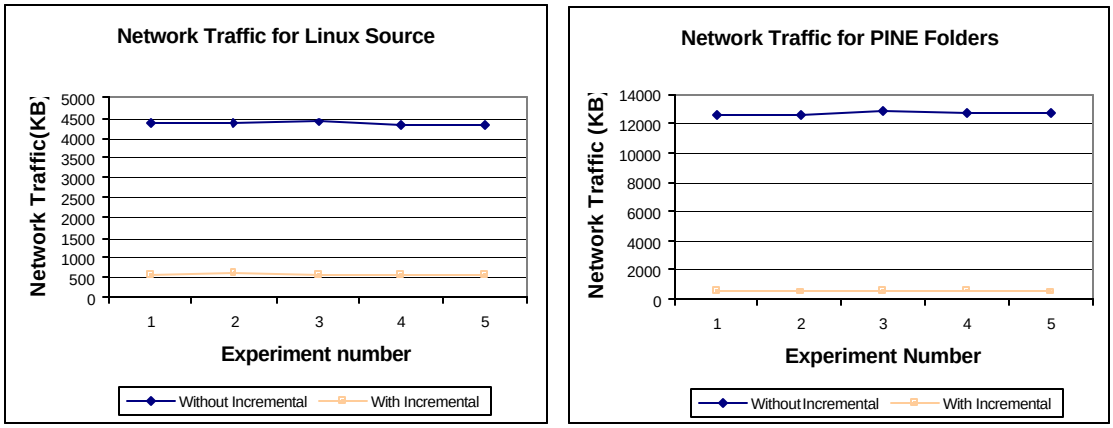

Figure 5.2: Network traffic for Linux Source and PINE Email Folders

5.3 Heuristics for Reintegrating PINE Email Folders

To impress upon the benefits of using the incremental approach for reintegration we are presenting a heuristic analysis using PINE email folders of a common email user. We chose email because it is a good example of making updates while disconnected and reintegrating over a weak connection. Email is a day-to-day application with which users spend on an average about 2.5 hours a day sending and receiving about 30 emails. For executives these figures are much higher.

Table 5.2: Heuristics for Reintegrating PINE Email Folders

Email Processing Time (min)	Count of events							Reintegration Data Size (bytes)	
	E1	E2	E3	E4	E5	E6	E7	Incremental	Traditional
5	3	1	0	0	1	0	0	9334	1481253
10	7	3	0	4	3	0	0	29704	2896004
20	14	5	0	7	5	0	0	56677	2893005
30	20	10	5	13	5	0	0	214873	2900727
45	36	12	15	20	7	1	1	321067	2905069

We identified common events that users do to manage their emails in PINE. Some of the common events are removing emails from the Inbox, moving emails to another folder, etc. We identified 7 such events as shown in Table 5.3. Also the time spent by users checking and managing emails varies. Some of the factors it depends on are the number of emails the user gets, time of the day at which emails are being checked. Less time is spent if he is rushing for a meeting and more time would be spent if the user were managing long accumulated emails. The five processing times that we chose are shown in Table 5.2.

Table 5.3: Event description for PINE folders

Event	Description
E1	Messages removed from Inbox
E2	Postponed/send later messages
E3	Messages removed from folders other than Inbox
E4	Messages added to folders other than Inbox
E5	Take address from email
E6	Create new folder
E7	Edit .pinerc

Based on the survey conducted within Harris Networking and Communication Laboratory at the University of Florida, we gathered average pattern for our events. While disconnected we performed these events on a user's email folders that are plain files in the Coda namespace. Upon reconnection the email folders are reintegrated to the server. We conducted the experiment using the traditional method and the incremental approach and noted the bytes transferred in each case for reintegrating the changes made to the email folders. The results we came up with are shown in Table 5.2. Figure 5.2 shows the huge difference between using the incremental approach and the traditional method.

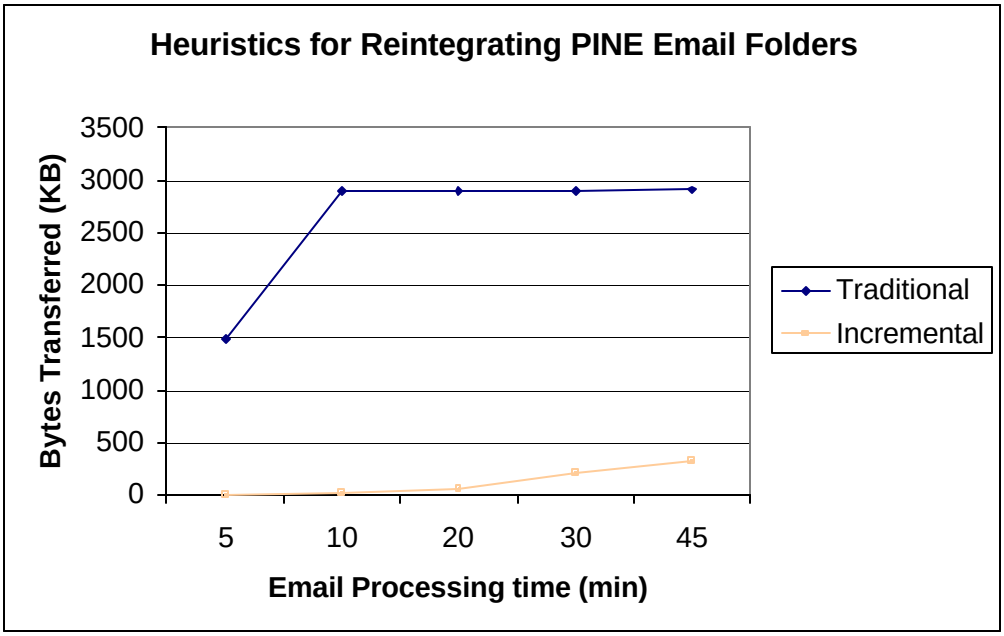

Figure 5.3: Heuristics for Reintegrating Pine Folders

5.4 Storage Overhead

The only cost that has to be paid for using the incremental approach is the storage space tradeoff because we maintain RCS server on the Vice for every file. However, RCS

stores the versions in a very efficient manner and the space overhead is overshadowed by the performance gains obtained from using the incremental approach. Figure 5.2 shows our evaluation of the extra space required on the Vice using the incremental approach as compared to the regular full file transfer using the Linux source files experiment.

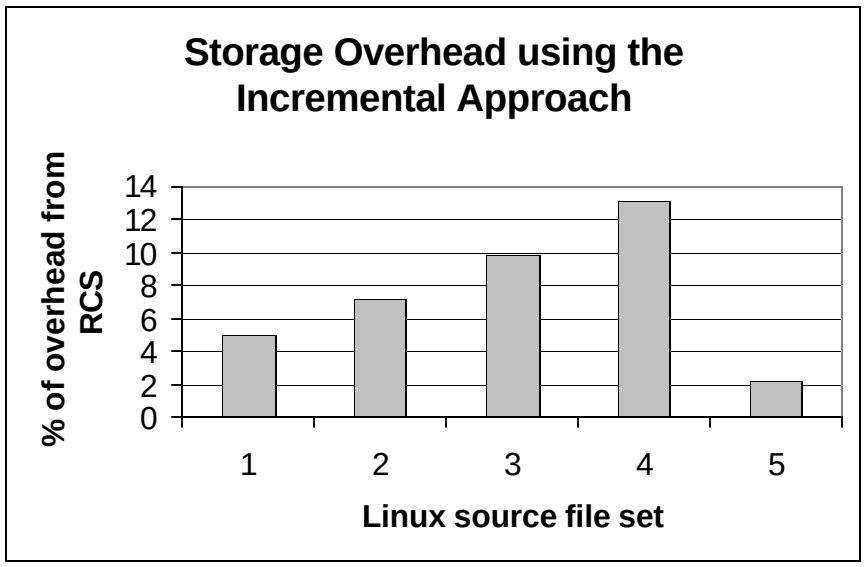

Figure 5.4: Storage Overhead due to RCS Version Files

Another storage overhead is on the client side to maintain the archives that hold the last version that the client holds before any updates are done. The extra storage required by this archive is as much as the space required by the Venus cache. However, since the Venus cache is usually small, the extra space is not really a burden since the performance gains far out weighs the space loss.

CHAPTER 6
CONCLUSION AND FUTURE WORK

We would finally like to present the conclusions that were drawn from this work, what are the benefits, limitations and drawbacks of using the incremental approach. We would also like to suggest future developments to make more out of the incremental approach.

6.1 Achievements of this Thesis

In this thesis we experimented a new way to do hoarding and reintegration. After reviewing the existing hoarding and reintegrating mechanisms against the incremental approach we conclude that this approach is very beneficial and promising. The benefits of the incremental approach are magnified in mobile computing environments when the connection is weak because the bandwidth and connection are scarce in such networks. The network traffic caused by the transfer of files is considerably reduced owing to the differential transfer of contents. This is especially beneficial since it saves users connection time and money.

The improvements on the time spent and the reduction on the network traffic vary based on the usage pattern. There is an improvement of one order magnitude when few changes are made on cache contents while disconnected. However, the gains are only reasonable when many changes are made. As mentioned earlier, our premise has been that users are disconnected only for short periods of time, and few changes are expected from the users during these short periods of disconnection. Based on this, the incremental

approach has highly positive results in realistic scenarios. Also this approach is transparent and does not require user intervention or administration unlike the operation-based approach discussed in Chapter 2.

However there are certain limitations and drawbacks to this approach. The main drawback of this approach is that it largely depends on how similar the two versions of a particular file are. In the worst case one might make global replacements in a text file to a particular string resulting in a differential file that is as large as the older version resulting in little gain from this approach.

Transfer of binary files with the incremental approach is not as attractive as for text files. However, this is attributed to the diff algorithm.

Another limitation within this implementation is that there is high dependability on the Unix *rcsdiff* and *diff* utilities that are not very sophisticated.

6.2 Future Work

This thesis is a step in the direction of using the incremental approach. There are useful improvements that can be made to this method of hoarding and reintegration.

A well-designed diff utility specifically tailored for the incremental approach can be developed that creates good quality and small differential files. The utility may also handle specific situations like the global replacement case described above.

For transferring binary files there are tools like *rsync* that can be used to transfer contents based on the difference on the blocks of data inside a file. The blocks can also be matched given the offset in the files and not just multiple of block sizes.

In this implementation we can improve version control by replacing RCS with a more sophisticated version control system that provides API's for doing the basic

operations like *ci*, *co* etc. Revision Control Engine (RCE) is a tool that is available that provides API's.

Another interesting possibility is to use the incremental approach in conjunction with operation-based update propagation. Based on the situation either differential or operation-based shipping may be used. If the differential file that is generated is small enough then we can use the incremental approach otherwise resort to the operation-based approach.

Based on our experiments and evaluations we conclude that the incremental approach is highly beneficial and useful to hoard and reintegrate after short periods of disconnection.

LIST OF REFERENCES

[BRAa] Braam, P.J., InterMezzo File System: Synchronizing Folder Collections. White Paper, Stelias Computing Inc. http://www.inter-mezzo.org/docs/intermezzo-sync-white.pdf, April 2001.

[BRAb] Braam, P.J., The Coda Distributed File System. http://www.coda.cs.cmu.edu/ljpaper/lj.html, April 2001.

[GUY90] Guy, R.G., Heidemann, J.S., Mak, W., Page, T.W. Jr., Popek, G.J., Rothmeier, D., Implementation of the Ficus Replicated File System. In Proceedings of the USENIX Conference, Anaheim, California, June 1990.

[HEL01] Helal, S., Hammer, J., Zhang, J., Khushraj, A., A Three-tier Architecture for Ubiquitous Data Access. ACS/IEEE International Conference on Computer Systems and Applications, Beirut, Lebanon, June 2001.

[KIS92] Kistler, J.J., Satyanarayanan, M., Disconnected Operation in the Coda File System. ACM Transactions on Computer Systems, Vol. 10, No.1, February 1992.

[KUE94] Kuenning, G.H., The Design of the Seer Predictive Caching System. In Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, California, December 1994.

[LEE99] Lee, Y., Leung, K., Satyanarayanan, M., Operation-based Update Propagation in a Mobile File System. In Proceedings of the USENIX Annual Technical Conference, Monterey, California, June 1999.

[MIC99] Microsoft Windows 2000 Server, Introduction to IntelliMirror Management Technologies. White Paper, Microsoft Corporation, 1999.

[MUM95] Mummert, L.B., Ebling, M.R., Satyanarayanan, M., Exploiting Weak Connectivity for Mobile File Access. In Proceedings of the Fifteenth Symposium on Operating System Principles, Copper Mountain, Colorado, December 1995.

[MUM96] Mummer, L.B., Exploiting Weak Connectivity in a Distributed File System. PhD. thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, 1996.

[SAT90] Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C., Coda: a Highly Available File System for a Distributed Workstation Environment, IEEE Transactions on Computers, Vol. 39, No. 4, April 1990.

[SAT93] Satyanarayanan, M., Kistler, J.J., Mummert, L.B., Ebling, M.R., Kumar, P., Lu, Q., Experience with Disconnected Operation in a Mobile Computing Environment. In Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing, Cambridge, Massachusetts, Aug 1993.

[SAY00] Saygin, Y., Ulusoy, O., Elmagarmid, A.K., Association Rules for Supporting Hoarding in Mobile Computing Environments. In Proceedings of the Tenth International Workshop on RIDE, San Diego, California, 2000.

[TIC85] Tichy, W.F., RCS – A System for Version Control. Software-Practice and Experience, Vol. 15, No. 7, July 1985.

BIOGRAPHICAL SKETCH

Abhinav Khushraj was born on October 17th, 1977, in Chennai, India. He received his undergraduate degree Master of Science (*Technology)* in Information Systems from Birla Institute of Technology and Science, Pilani, India, in June 1999.

He joined the University of Florida in 1999 to pursue a master's degree in computer and information science and engineering. He has worked as a research assistant with Dr. Abdelsalam Helal in the Harris Communications and Networks Laboratory. His main research interests lie in mobile computing and mobile data management.

He has worked in the industry for about a year altogether. At Software Technology Parks of India, Bangalore, he worked on two-tier client server architecture for databases. He has also worked as an intern at Citrix Technologies, Ft. Lauderdale, in the summer of 2000 and has experience in software localization and memory leak management.

After his graduation he will continue at Citrix as a full time employee working on thin-client server technologies.