

**CERIAS Tech Report 2013-11**  
**Distributed Digital Forensics on Pre-Existing Internal Networks**  
by Jeremiah J Nielsen  
Center for Education and Research  
Information Assurance and Security  
Purdue University, West Lafayette, IN 47907-2086

**PURDUE UNIVERSITY**  
**GRADUATE SCHOOL**  
**Thesis/Dissertation Acceptance**

This is to certify that the thesis/dissertation prepared

By Jeremiah Jens Nielsen

Entitled

Distributed Digital Forensics on Pre-Existing Internal Networks

For the degree of Master of Science



Is approved by the final examining committee:

Marcus K. Rogers

Chair

John A. Springer

Thomas J. Hacker

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Marcus K Rogers

Approved by: Jeffrey L. Whitten

Head of the Graduate Program

11/26/2013

Date

DISTRIBUTED DIGITAL FORENSICS ON PRE-EXISTING INTERNAL  
NETWORKS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Jeremiah J Nielsen

In Fulfillment of the

Requirements for the Degree

of

Master of Science

December 2013

Purdue University

West Lafayette, Indiana

## TABLE OF CONTENTS

	Page
LIST OF TABLES .....	iv
LIST OF FIGURES .....	v
LIST OF ABBREVIATIONS.....	vi
GLOSSARY .....	vii
ABSTRACT.....	ix
CHAPTER 1. INTRODUCTION .....	1
1.1. Scope .....	2
1.2. Significance.....	2
1.3. Research Question .....	3
1.4. Assumptions .....	3
1.5. Limitations.....	3
1.6. Delimitations .....	4
1.7. Summary.....	4
CHAPTER 2. LITERATURE REVIEW .....	5
2.1. Potential Solutions.....	5
2.2. Related Work.....	7
2.3. High Performance Computing Research .....	9
2.4. Summary.....	10
CHAPTER 3. FRAMEWORK AND METHODOLOGY .....	12
3.1. Research Approach.....	12
3.2. Instrumentation and Data Capture.....	14
3.3. Application Architecture .....	14
3.4. Application Communication.....	15
3.5. Application Class Structures .....	16
3.6. Imaging and Image Distribution.....	19
3.7. Application Buffers .....	20
3.8. Client Searches .....	20
3.9. Application GUI .....	22

	Page
3.10. Hypothesis .....	24
CHAPTER 4. OPTIMIZING APPLICATION PERFORMANCE .....	25
4.1. Imaging Buffers .....	25
4.2. Network Communication Buffer .....	26
4.3. Client Search Buffer .....	27
4.4. Image Chunk Size .....	29
4.5. Summary .....	30
CHAPTER 5. RESULTS .....	31
5.1. Test Preparation .....	31
5.2. Initial Testing on Server Hardware .....	32
5.3. FTK 4.1 Issues .....	33
5.4. Proposed Application Issues .....	34
5.5. Distributed String Searches (Sparse Hits) .....	34
5.6. Distributed String Searches (Several Hits) .....	35
5.7. Regular Expression Searches .....	37
5.8. Image Processing .....	38
5.9. Summary .....	40
CHAPTER 6. CONCLUSION .....	42
6.1. Future Work .....	43
6.1.1 Cross Platform Functionality .....	43
6.1.2 Search Improvements .....	44
6.1.3 Security Improvements .....	44
LIST OF REFERENCES .....	46
APPENDIX. CLIENT FILE SEARCH CODE IN C# .....	49

## LIST OF TABLES

Table	Page
Table 3.1 Client and Server Hardware Characteristics.....	12
Table 3.2 Communication Channels and Sizes.....	16

## LIST OF FIGURES

Figure	Page
Figure 3.1 FTK Application Process .....	13
Figure 3.2 Proposed Client/Server Application Process.....	14
Figure 3.3 Pseudo Code for Application Communication.....	15
Figure 3.4 Application Search Process .....	18
Figure 3.5 Client Search Buffers .....	21
Figure 3.6 Search Buffer Overlaps .....	21
Figure 3.7 Client Search Pseudo Code .....	22
Figure 3.8 Server Application GUI.....	23
Figure 3.9 Client Application GUI .....	23
Figure 4.1 'DD' 1 GB Imaging Time .....	26
Figure 4.2 500 MB Image Transfer .....	27
Figure 4.3 Search Runtime for a Single Image File.....	28
Figure 4.4 Search Runtime for Split Image File .....	29
Figure 5.1 10 GB Split Image Search Runtimes.....	33
Figure 5.2 100 GB String Search with Sparse Hits .....	35
Figure 5.3 100 GB String Search with Several Hits .....	36
Figure 5.4 100 GB Regular Expression Search .....	37
Figure 5.5 10 GB Image Processing Times .....	38
Figure 5.6 100 GB Image Processing Time.....	39
Figure 5.7 100 GB Overall Runtime.....	40
Figure 5.8 Application Speedup Compared to FTK.....	41

## LIST OF ABBREVIATIONS

BS - Block Size

DDF - Distributed Digital Forensics

FTK - Forensic Toolkit

GB - Gigabyte

GUI - Graphical User Interface

MPI - Message Passing Interface

RegEx - Regular Expression

TB - Terabyte

VPN - Virtual Private Network



## GLOSSARY

- Client/Server Architecture – A computer architecture in which a single server machine handles communication between several client machines.
- Data Buffer – A portion of a host’s random access memory used to temporarily store data between read/write operations.
- DD – An application used to copy raw data between devices on a computer.
- Digital Forensics – “The use of an expert to preserve, analyze, and produce data from volatile and non-volatile media storage. This is used to encompass computer and related media that may be used in conjunction with a computer” (Meyers & Rogers, 2004, p. 4).
- Distributed Digital Forensics - "Using aggressive data caching techniques and performing investigative operations in parallel" (Richard & Rousev, 2006a, p. 79).
- Forensic Toolkit - A court-accepted digital investigations platform built for speed, analytics and enterprise-class scalability (AccessData Group, 2012).
- Index Search - A fast search process in which a long preload time is used to create a rapidly searchable object from a data object.
- Live Search - A slow search process in which a data object itself is searched from beginning to end.
- MapReduce - A framework in which data is distributed across several nodes (Map function), processed, and sent back to the initializing node for summation (Reduce function).
- MD5/SHA1 - One way mathematical functions that convert blocks of data into a unique string of text providing the ability to compare files and checking for file modifications.
- Overlay Network – “A network built on top of one or more existing networks” (Stoica, 2009)

Text Search – A search process in which a single word (literal string search) or pattern (regular expression search) is located within a large block of data.

Windows Management Instrumentation - Infrastructure for management data and operations on Windows-based operating systems that supplies management data to other parts of the operating system and products. (Microsoft, 2013b)

## ABSTRACT

Nielsen, Jeremiah J. M.S., Purdue University, December 2013. Distributed Digital Forensics on Pre-existing Internal Networks. Major Professor: Marc Rogers.

Today's large datasets are a major hindrance on digital investigations and have led to a substantial backlog of media that must be examined. While this media sits idle, its relevant investigation must sit idle inducing investigative time lag. This study created a client/server application architecture that operated on an existing pool of internally networked Windows 7 machines. This distributed digital forensic approach helps to address scalability concerns with other approaches while also being financially feasible. Text search runtimes and match counts were evaluated using several scenarios including a 100 GB image with prefabricated data. When compared to FTK 4.1, a 125 times speed up was experienced in the best case while a three times speed up was experienced in the worst case. These rapid search times nearly irrationalize the need to utilize long indexing processes to analyze digital evidence allowing for faster digital investigations.

## CHAPTER 1. INTRODUCTION

Digital forensics is a rather immature discipline that has experienced a large amount of growth over its short existence. Due to this rapid growth, which can most likely be attributed to the rapid growth and increasing proliferation of technology, the discipline is plagued by many issues that have yet to be effectively resolved as described by Bebe (2009) and Garfinkel (2010). While the digital forensics research community has shifted its focus between several of these proposed issues, the well-known data deluge issue has yet to be effectively addressed. One potential solution for the data deluge issue would be to employ the use of a pool of computers to conduct distributed digital forensic investigations.

The distributed digital forensics concept has been previously investigated using a client/server architecture (Richard & Roussev, 2004), a distributed ramdisk (Richard & Roussev, 2006a), as well as the MPI reduce function (Roussev et al., 2009). Unfortunately, these approaches require a substantial investment in dedicated computer hardware and the involvement of IT professionals to configure and utilize. The ideal distributed digital forensics application would be simple to instantiate, could operate on a pool of existing machines, and would not entirely dedicate the involved clients to the investigation process.

The proposed distributed digital forensic application architecture satisfies these characteristics by running on a pool of existing Windows 7 machines. To help gauge the effectiveness of the proposed approach, dataset distribution times across a set of preexisting internal network resources was compared to that of a single workstation using AccessData's Forensic Toolkit (FTK) 4.1. The main deliverable of this research was a client/server application that can be used to create an ad-hoc overlay network on a pre-existing pool of computational resources upon which large datasets can be analyzed.

The remainder of this chapter will provide the scope and significance of the research conducted in this thesis.

### 1.1. Scope

There are many issues with the concept of distributed digital forensics in regards to security, performance, and reliability. However, the major problem that hampers its practicality in analyzing today's large datasets is scalability, as it would require a substantial investment in dedicated hardware. This thesis solves this problem by leveraging existing computational resources running the Windows 7 operating system through the utilization of a client/server application architecture. These applications were used to distribute a large suspect dataset across a pool of internally networked computational resources and conduct text searches against it.

### 1.2. Significance

It was once unheard of to have terabyte storage volumes available for average computer users to utilize on personal workstations. This is obviously no longer the case as it has become very affordable for personal workstations to employ large amounts of storage. According to Seagate, this amount averaged nearly 600 GB (Hachman, 2011) in 2011 which could increase exponentially if research in increasing mechanical hard drive densities goes mainstream (Halfacree, 2012). Unfortunately, most digital forensic methodologies cannot cope with this storage volume increase as they still rely on single machines to conduct investigations. As storage volumes continue to increase in size, tools relying on this single machine approach simply take longer to run. This in turn has led to an ever increasing backlog of digital evidence that is overwhelming criminal investigators and the tools which they employ. Having the ability to distribute digital investigations across several machines provides a viable solution to analyzing today's large datasets (Richard & Roussev, 2004).

### 1.3. Research Question

Can scalability limitations of the distributed digital forensics concept be dealt with by operating on an internal network of existing computational resources running the Windows 7 operating system?

### 1.4. Assumptions

Assumptions with regards to this research are as follows:

1. A suitable physical network architecture exists upon which the proposed distributed digital forensics applications can operate.
2. All involved hosts are able to communicate with the designated server (i.e., all hosts are routable and can communicate through firewalls).
3. Completely distributing a suspect dataset across the proposed distributed digital forensics network is the equivalent of a single workstation tool, such as FTK, conducting initial loading and indexing processes.
4. Text searches using the proposed framework are the functional equivalent of FTK's live search function.

### 1.5. Limitations

Limitations with regards to this research are as follows:

1. Several single workstation digital forensics tools exist but image distribution times using the proposed client/server application were only compared to the initial loading and indexing process used by FTK 4.1.
2. While capable of spanning the public domain of the Internet this research only focused on computational resources connected directly to the same network switch.
3. Only machines running the Windows 7 operating system with Microsoft's .Net Framework 4.0 installed were used for testing.

## 1.6. Delimitations

Delimitations with regards to this research are as follows:

1. This research did not investigate other means to analyze suspect datasets outside of text searches.
2. Comparison of variations in network hardware and architectures were not conducted as technological ambiguity makes such testing difficult and infeasible.

## 1.7. Summary

This chapter has provided a brief overview of this research in regards to its scope, significance to the digital forensics field, as well as its focus. The next chapter will look at relevant work that has been completed with respect to this research.

## CHAPTER 2. LITERATURE REVIEW

Digital forensics has overcome several issues including increased public awareness, a stronger scientific foundation, the creation of relevant journals, and the development of a strong research community (Bebe, 2009). Unfortunately there are still many other issues for the digital forensics discipline to overcome. Bebe (2009) created four "research themes" that need to be addressed by the digital forensics community which further illustrates this point. These derived themes included volume and scalability challenges, more intelligent analytical approaches, digital forensics of non-standard computing environments, and forensic tool development. The purpose of this thesis is to develop an application approach that can be used to address the volume and scalability research theme. This theme was brought about due to the fact that "data storage needs and data storage capacities are ever increasing" (Bebe, 2009, p. 24) and "the growing size of storage devices means that there is frequently insufficient time to create a forensic image of a subject device, or to process all of the data once it is found" (Garfinkel, 2010, p. S66). This chapter will look at relevant work that has been completed to help cope with this volume and scalability research theme.

### 2.1. Potential Solutions

One potential solution to analyzing today's large datasets is to employ the use of selective digital forensics in which only prudent information is taken from a suspect medium instead of making an exact bit level image (Turner, 2006). The obvious issue with such an approach is the potential for a selective digital forensic tool to potentially overlook prudent information. Such oversights could be caused by operational



inadequacies within the tool itself or by malicious users who deliberately exploit the selection process.

Another approach is to combine static analysis (sifting through a nonvolatile storage volume such as a hard drives) with live analysis (sifting through volatile storage such as random access memory) (Mrdovic et al., 2009). In this approach, a virtual machine is created using a suspect drive image in conjunction with a memory image to recreate a functional clone of the suspect's machine prior to its capture. While this solution does not solve the volume and scalability issue since a full drive image is utilized, it helps investigators to speed the investigative process along by narrowing their search scope. The glaring problem with this approach is that live analysis tools necessitate the use of an active suspect system which is inherently un-trusted.

While other unmentioned approaches, such as data mining, exist to combat this data volume issue, none appear to provide an effective mitigation strategy as the data deluge issue still persists. In an ideal world refined imaging processes would create smaller drive images by capturing only relevant information. These smaller images would then be analyzed using refined analytical approaches that are fast, highly accurate, and repeatable. User friendly tools would then present analysis results in a format that even an untrained investigator would be able to understand. Garfinkel (2010) as well as Richard and Roussev (2004b) describes some of these solutions in more detail.

While many investigators would enjoy working in such an ideal world, getting the digital forensics discipline to such a point is excruciatingly difficult given the rapid pace of technological development and its inherent ambiguity. Unfortunately, while researchers continue to try and create the aforementioned ideal world, datasets are only getting larger. A solution is necessitated to analyze these large datasets until the research community is able to catch up. One possible solution is to utilize the concept of distributed digital forensics. Distributed digital forensics is not much different than a typical criminal investigation of the past in which no form of technology was utilized (which is obviously highly unlikely today). One or more detectives could be assigned to a case and information would be collected to ascertain a suspect's involvement. If a case

were to be time sensitive, such as a kidnapping case, then the investigating organization could potentially speed the process along by doing one or more of the following:

- Use more efficient specialized internal resources (a kidnapping division)
- Assign more internal resources (add more detectives to the case)
- Utilize external resources (request assistance from other investigative organizations)

Many digital forensics tools are designed to work on a single workstation which is essentially the equivalent of assigning a single investigator to an investigation. With the concept of distributed digital forensics, several computer resources are pooled together to work on one suspect dataset, which is essentially the equivalent of assigning several detectives to one case.

## 2.2. Related Work

Distributed digital forensics is not new concept as a working prototype has been created, implemented, and tested (Richard & Rousev, 2004). The network used in their approach consisted of a network file server and eight physical nodes with each housing 1 GB of random access memory. A 6 GB suspect dataset was loaded onto the network storage server and one of the eight nodes involved, called the coordinator, was responsible for splitting the files in the dataset across the remaining seven nodes on a file by file basis. Each file from the dataset that was copied to a node was loaded entirely into the node's random access memory to prevent the induction of disk I/O latency. The coordinating node then sent commands to and received commands from the worker nodes via a customized command structure which operated over HTTP. In a performance analysis of their implementation, Richard and Rousev showed some substantial improvements in load times and string search times when compared to a single machine utilizing FTK. Their results showed 34% faster load times, an eighteen times speed up for string search times, and an eighty-nine times speedup for regular expression searches.

Of course the glaring issue with the aforementioned approach is that a rather small image by today's standards, only 6 GB in size, was used for testing their

implementation. This is understandable seeing as testing was conducted in 2004, but currently hard drives average several hundred gigabytes (Hachman, 2011) and in many cases even surpass the one terabyte mark. If an organization were to have machines with 12 gigabytes of random access memory, then over 160 machines would be necessary to load a 2 TB dataset. Even if an adequate architecture existed and the dataset was loaded, the machines involved would be entirely dedicated to conducting the investigation due to resource saturation. Many organizations simply cannot afford or even rationalize the dedication of such large clusters of machines for the sole purpose of digital forensic investigations.

In another approach, Richard and Roussev attempted to speed up digital forensics applications by creating a distributed ramdisk (Richard, Roussev, & Tingstrom, 2006). This block level ramdisk was created on a pool of machines that shared a portion of their ram creating a single logical ram pool that was addressable from a single machine. Their benchmark results showed a 3.5 times speed up for sequential read/write operations while random operations showed a 22 times speedup. Such results are expected as random access memory excels at random operations which mechanical hard drives find difficult to deal with. While this speed up can be seen as rather substantial, this approach also suffers from the same problems as their previous approach in that it would require a large investment in hardware and would be difficult for average law enforcement personnel to configure and use.

While they may not be distributed digital forensics approaches per se, two other approaches created by Craiger et al. (2008) and Davis et al. (2005), try to solve the volume and scalability issue by utilizing virtualization technology to gain high speed access to suspect images stored on high speed network storage devices. The idea is to have a device shipped to a nearby lab where it is imaged. (Kechadi and Scanlon (2010) suggested transferring device images over the internet from the crime scene). The resulting image is stored on high speed network storage devices residing on an internal optical network. A virtual machine server, also connected to the optical network, hosts virtual machines that are leased out to remote users to analyze a desired image. Permissions to the network storage devices and the data they contain are then assigned to

individual virtual machines. When investigations are conducted through these virtual machines they would gain a substantial performance boost as all devices communicate through an internal low latency optical network without the induction of internet latency.

There is one major drawback with the approaches created by Craiger et al. (2008) and Davis et al. (2005) in that it would require a substantial investment in a dedicated infrastructure. Since a VPN is used to grant outside users access to the system, usernames and passwords must be managed. Permissions for the data store and individual files must also be managed to prevent users from damaging suspect images be it intentionally or unintentionally. Policies would need to be created and enforced for data retention, destruction, and integrity. Addressing all of the aforementioned issues would most assuredly necessitate the involvement of several fulltime IT personnel. These additional requirements may simply be infeasible for smaller local investigative agencies.

Since many digital investigators are aware of the data deluge issue and distributed digital forensics has the potential to be a feasible solution, then why hasn't distributed digital forensics gained enough momentum for a company to capitalize on the idea? One potential explanation that has been repeated several times throughout this thesis is that distributed digital forensics is viewed as infeasible from a resource or management point of view. Another potential explanation is that researchers are logically concentrating their efforts into solving the causes of the deluge by researching analytical/presentation approaches and frameworks instead of trying to deal with the symptoms. Unfortunately, research in this area still appears to be inconclusive as digital investigators still rely on a 'capture all, analyze all' mentality using single workstation tools that utilize time consuming indexing processes.

### 2.3. High Performance Computing Research

Thanks to large computational projects, such as CERN's Large Hadron Collider (CERN, 2008) and NASA's cyclone analysis (Vishwanath et al., 2008) project, several approaches have come to fruition to capture and analyze petabytes of information through

the use of multidimensional data, massively parallel processing, and large ultrafast networks. This is quite a feat seeing as the digital forensics community is finding it excruciatingly difficult to manage several hundred gigabytes of data (Bebe, 2009). Unfortunately, these approaches seem to be effective for data that is largely homogeneous and predictable; characteristics that fit nicely with typical large scale instrumental data but not with largely heterogeneous file system data (Douceur & Bolosky, 1999) made up of several different types of data. Such approaches also operate best using very fast hardware interconnected with large low latency networks. These networks are typically located in close proximity to one another and consist of an optical communication medium.

Massive parallel processing can potentially be directly applied to DDF through the creation of an MPI cluster. These clusters are simply a pool of machines that use a standardized protocol to communicate. Typically machines in such clusters run a rudimentary operating system to limit resource utilization and are dedicated to running MPI functions. In one DDF approach, the MPI MapReduce function was used to split large text files and conduct string searches against them (Roussev et al., 2009). Their implementation showed a linear 15X speed up when compared to Hadoop which also utilizes the MapReduce function. Some advantages of MPI clusters include scalability, the ability to use heterogeneous computer environments with respect to both hardware and software, and low level code optimizations. Some disadvantages of MPI clusters are that they are difficult to configure, do not typically tolerate node failures, and MPI software development can be very difficult. These disadvantages make it difficult for investigators without a background in networking and distributed computing to effectively utilize MPI based clusters.

#### 2.4. Summary

While dealing with the causes of the data deluge is only logical via research in imaging and analysis, a stopgap solution is needed to deal with the current backlog of digital evidence caused by massive storage volumes. The solution developed in this

thesis employed the distributed digital forensics concept. While several similar prototypes of this concept have been developed, they do not appear to scale well with today's large datasets as it they would require a substantial investment in dedicated hardware and or IT personnel. In an attempt to address these limitations and make DDF a practical reality, this thesis leveraged existing computational resources for distributed digital investigations. This was accomplished using a client/server application architecture operating within the confines of an internal network. Analysis of a large suspect volume was split across several clients providing an increase in performance when compared to a single workstation using FTK 4.1.

## CHAPTER 3. FRAMEWORK AND METHODOLOGY

This chapter will discuss the overall approach used for this research including the research approach, instrumentation, application architecture, data capture, and data analysis.

### 3.1. Research Approach

The computer forensics lab at Purdue University was used to run the proposed implementation across varying counts of physical hosts running Windows 7 Ultimate Service Pack 1. All hosts were physically connected to the same Cisco 3750 gigabit network switch. Due to the inability to access sata connections within the available client machines, a separate machine configuration was used for the designated server. This designated server was also used to test single machine processing using FTK 4.1. Hardware specifications for both the clients and designated server are shown in Table 3.1.

Table 3.1 *Client and Server Hardware Characteristics*

	Client	Server
Model	iMac 21.5 Inch Mid 2011	N/A
Processor	Intel Core i5 @ 2.5 GHz	AMD Phenom II X6 1055T @ 2.8 GHz
Memory	8 GB DDR3 1333	8 GB DDR3 1333
Storage	Seagate ST3500418AS (500 GB)	Seagate ST1500DL (1.5 TB) Seagate ST1500DL (1.5 TB) Western Digital (320 GB)
Network	Broadcom NetXtreme	Realtek PCIe GBE
OS	Windows 7 Enterprise SP1	Windows 7 Enterprise SP1

A mock suspect volume was created and filled with textual data using a custom in-house text generator as described in Section 5.1. This text generator continually writes a specified block of text until a specified size or iteration count is reached and upon task completion presents the user with the total number of write operations. This approach was chosen as it allowed known match counts to be compared to those returned from searches conducted using both the proposed implementation and live search functionality in FTK. Text search runtimes were recorded using various image sizes distributed across varying counts of physical hosts.

The total wall clock runtime of the distribution process using the proposed implementation was recorded for each host and image configuration. Text search runtime as well as match counts were recorded for the proposed implementation. Values were captured for the initial loading/indexing and search functionality of FTK 4.1 running on a single workstation. The application process utilized for FTK testing is illustrated in Figure 3.1 while the application testing process used for the proposed client/server application architecture is illustrated in Figure 3.2.

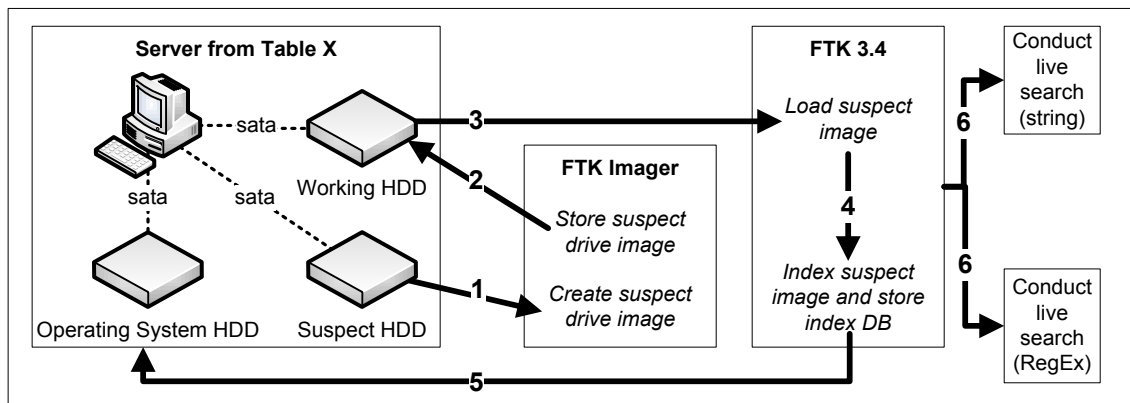


Figure 3.1 FTK Application Process



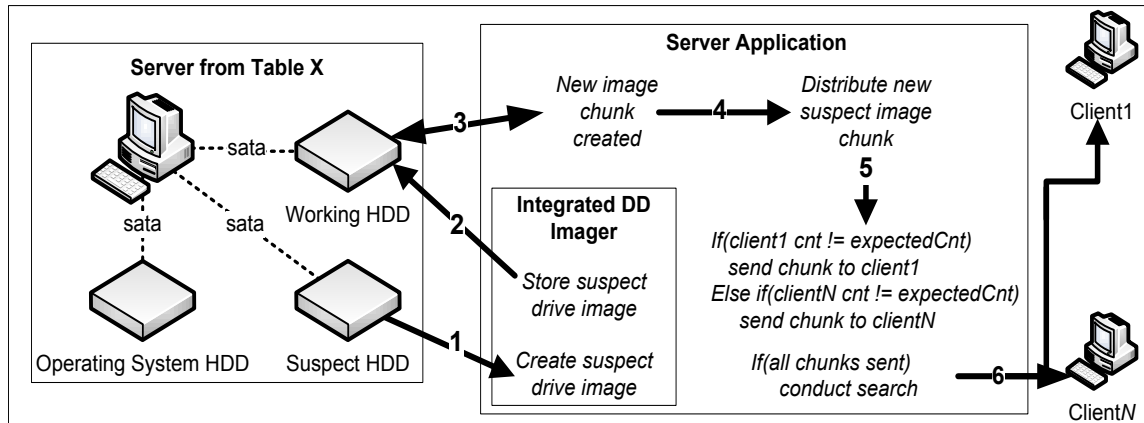


Figure 3.2 Proposed Client/Server Application Process

### 3.2. Instrumentation and Data Capture

As previously stated, wall clock runtimes and string match counts were used to gauge the effectiveness of the proposed implementation and FTK 4.1. Runtime tracking was built into both the client and server applications using the difference between a start and end time while counting was handled by a simple incremented counter. The timers were used for both text search and imaging operations while the counter was only used for counting search matches. FTK 4.1 automatically provided these values upon task completion.

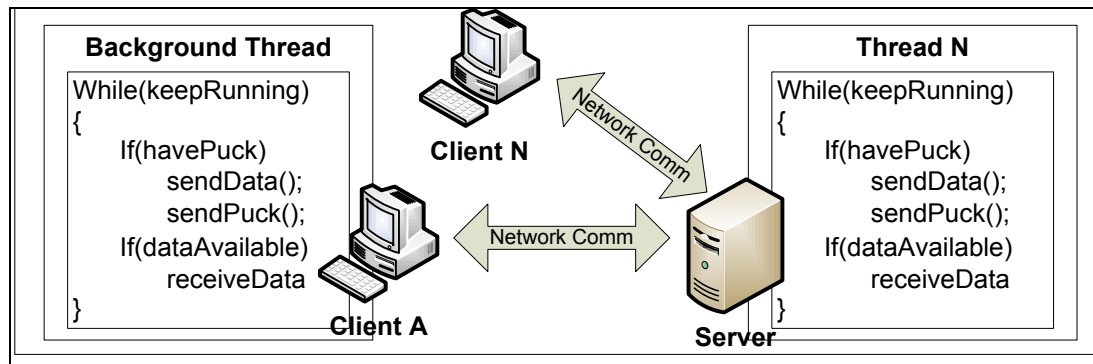
### 3.3. Application Architecture

A rudimentary overlay network was created with preexisting computational resources using a two tier application architecture written in Microsoft's C# programming language. This architecture consisted of a single server application instance and multiple client application instances. The single instantiated server application was responsible for monitoring the status of all involved clients, imaging a suspect volume, and distributing the image across all connected clients. All involved clients were required to receive files from the server as well as conduct text searches against them. Results for each of these operations were compounded and sent to the designated server using several class objects (see section 3.5). This process was done for

all clients whereupon task completion the user is presented with the match results on the designated server. In the high performance computing realm this would be the equivalent of a MapReduce function.

### 3.4. Application Communication

Once initialized, the user designated server simply listens for clients to connect. When a new client connects to the server a new instance of a custom class is instantiated using a newly spawned processor thread. All clients communicate with the server simultaneously and independent of all other clients. Communication between the server and involved clients is accomplished using separate channel identifiers, a known packet size, and a synchronization bit referred to as a “puck”. This puck assigns network write permissions to either the client or server so as to prevent collisions during data transfers. The pseudo code for this communication process is shown in Figure 3.3 below.



*Figure 3.3* Pseudo Code for Application Communication

In order to differentiate between messages and message types, the channel identifier and expected packet size are written to the network stream before outgoing data. When a channel identifier is received the recipient knows what type of data is incoming. The packet size is necessary so the recipient knows when to stop reading data from the network stream so as not to overlap message data thus causing data corruption. Types of data and their respective size and channel are shown in Table 3.2.

Table 3.2. *Communication Channels and Sizes*

Data Identifier	Packet Size	Channel
Synchronization Bit (puck)	1 byte	0
Channel Identifier	1 byte	N/A
Packet Size	2 bytes	N/A
clsPerformanceValues	Varies	200
clsSearchResults	Varies	205
clsReconnect	Varies	210
File Packet	Varies	253
Message Packet	Varies	254

Two things to note about Table 3.2 are the items with no channel as well as those with varying packet sizes. Channel identifiers are not necessary for the channel identifier and packet size because they are configured with a static size and will always precede data traversing the network stream. However, some items vary in size based on the amount of data being sent. Very large data streams, such as a large file transfer, are split into several chunks based on a specified block size (more details in section 4.2) and written to the network stream individually. Smaller messages, and in many cases the tail end of a file, manipulate the block size to only accommodate the amount of data being sent. This approach helps to prevent the induction of unnecessary computational overhead from writing redundant data to the network stream.

### 3.5. Application Class Structures

Quite possibly the most important component of the proposed application process is the use of structured class objects to send several client values to the server via a single data transfer. Three such objects were used including a reconnect class (aptly named clsReconnect), a performance values class (aptly named clsPerformanceValues), and a client search results class (clsSearchResults)

The clsPerformanceValues class is used to ensure that all involved clients provide enough combined computational resources to feasibly analyze a suspect volume. Once a

client is started, this class is instantiated and used to store resource values gathered using the Windows Management Instrumentation. While several resource values can be captured, the most important ones with regard to this thesis are the available hard drives for a client and their respective capacities. The hard drive with the greatest amount of free space and read/write capability is used as the destination for incoming file chunks received from the server.

Subsequent resource updates are only made for dynamic resource values after the performance values class has been instantiated on a client. For instance, total drive capacity of a selected drive will not change but the amount of free space will change as files are received from the server. Resource updates uploaded to the server after client instantiation only include updates for these dynamic values. Once these values are updated, the `clsPerformanceValues` object is serialized and sent to the server where it is processed and stored in an array.

The second class, `clsReconnect`, is only used by clients during the initial connection handshake with the application server. Some form of mitigation strategy was necessitated to deal with client or server connection failures even though this thesis did not address reliability issues. If nodes were to fail during testing the entire imaging and distribution process would have to start over. A large amount of testing time would have been lost solely on application re-instantiation given the amount of time these processes can take.

The reconnect class prevents this time loss by providing the server with a list of files that already exist on a client. Once the class is instantiated on a client it is populated with names, sizes, and hash values of all files located in the client's configured working directory. The class is serialized and transmitted where its contents are compared to that of the server working directory. If a discrepancy is found the client connection attempt is denied until the issue is resolved.

The third and final class, `clsSearchResults`, is slightly more involved as it necessitates the use of two other class objects named `clsSearchMatch` and `clsSearchResultsSub`. The `clsSearchMatch` class is used to store information about a specific search match found in a file. This includes the file ID of the image file it was

found in, the index location of the match, its byte offset, and the match itself including fifteen characters before and after. While the index and offset values were not used in this thesis, their inclusion provides the ability to correlate search matches to locations on the physical suspect volume.

When a search match has been found a new `clsSearchMatch` object is instantiated, all afore mentioned values for the match are captured, and the resulting object is saved in an array within `clsSearchResults`. Unfortunately, all instantiated class objects are stored in the host machine's memory. This can be problematic in the event several million matches are found. Each host is configured to only store the first one million matches as `clsSearchMatch` objects to prevent excessive memory consumption. Matches discovered after the one million match limit are not stored as `clsSearchMatch` objects but are included in the total match count.

Resource saturation can also be a problem on the server as it must receive and display search matches from all involved hosts. This is where the `clsSearchResultsSub` class comes into play as it will only store a subset of the search results until it occupies a single network transfer block (a 65KB block size was used as described in Chapter 4). This technique simplifies the transfer of the `clsSearchResultsSub` object as the client does not have to split the object and the server does not have to reconstruct it. The `clsSearchResultsSub` class is sent to the server when a client has finished searching all files. The user can then review each search result list from all involved clients from the single sever instance. The overall search process using the three class objects is shown in Figure 3.4.

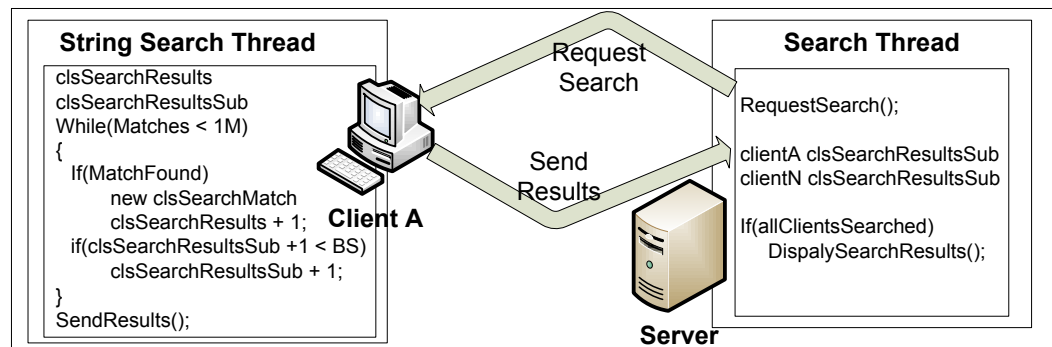


Figure 3.4 Application Search Process

### 3.6. Imaging and Image Distribution

Two Seagate 1.5 terabyte 5900 rpm ST1500DL001 hard drives were connected directly to the designated server via an integrated sata controller. Some form of write blocker would be used in an actual digital investigation but was not used in this research. Since this research is only concerned with search performance a write blocker would only increase imaging times but have no effect on search times.

Imaging of the suspect volume was handled via a separate thread on the server using two variations of 'DD' for windows. One variation of 'DD' (version 0.5) (<http://www.chrysocome.net/dd>) was used simply for its volume listing functionality which is faster than querying the Windows Management Instrumentation for hard drive identifiers. The second variation of 'DD' (Garner, 2011) was chosen because it provided image splitting functionality but lacked volume listing functionality. Both of these 'DD' variants are compiled into the server application and extracted to a temporary directory when the server is started. They are then run transparent to the user as hidden background processes.

Before the imaging phase is started, image chunking can be manually configured by the user with regard to chunk size and count. These image chunks are stored in a working directory on a separate internal SATA hard drive connected to the designated server. The server application monitors this specified working directory for new files. When a new file is discovered the file is hashed, its name and resulting hash are stored in a Microsoft Access Database, and the file is sent to one of the connected clients.

The file distribution process is conducted by first predicting the number of expected file chunks using the suspect volume size in conjunction with the configured chunk size. The total number of chunks is determined and divided by the host count to determine how many chunks a client should receive from the server. If a certain client does not have enough free space for this expected client chunk count then these files are simply sent to the next client until its client chunk count is reached (Figure 3.2). This approach was chosen as it helps prevent image fragmentation of a suspect volume while also evenly distributing the processing load. Total client free space is calculated to

ensure that enough free space is available to store the entirety of a suspect volume before starting the image distribution process.

One final aspect of the imaging and distribution process is the utilization of hashing to ensure that files are not changed during the file transfer process. When a client receives a file chunk it has the ability to request an MD5 and/or SHA1 hash from the server. The server sends these hashes to the client whereupon the client also calculates the hash value. The client and server values are compared to ensure that the data was successfully transferred from the server to the client.

### 3.7. Application Buffers

Both the client and server applications utilize multiple data buffers for various tasks. These include a data buffer for data being transferred over the network stream, reading from the suspect volume, writing to the storage volume, and searching the image chunks on a client. Minor changes to these buffer sizes have drastic effects on application performance and resource utilization. Ideal buffer sizes were determined by manipulating them across several scenarios and monitoring the effect on application performance. Results from this testing including the ideal values determined are described in Chapter 4 in more detail.

### 3.8. Client Searches

When the server requests a search, the query is forwarded to all clients. Upon receipt the client searches all received file chunks using the specified query and the Microsoft Regex library. This is accomplished by reading portions of a file into a temporary file buffer, converting the buffer to a string, and passing the string to the Microsoft Regex library configured with the server designated query. The total number of matches is accumulated and the process repeated until the end of the file is reached.

One major issue with such an approach is that matches overlapping file buffer occurrences will not be found. This is true at both the individual chunk level as well as

between distinct file chunks on a client. This issue was dealt with at the client level by developing a search overlap function. Two secondary buffers were used to store the beginning and end of the primary search buffer when appropriate. Once the primary buffer has been searched, the beginning portion of the next file is copied to a start buffer while the tail end of the current file is copied to an end buffer. The beginning and end buffers are combined and searched before the next portion of the file is loaded into the primary buffer. To ensure that matches at the end and beginning of the primary buffer are not counted multiple times, the beginning and end buffers are configured with a length equal to one character less than the query length. An example of this buffered search process is illustrated in Figure 3.5 below.

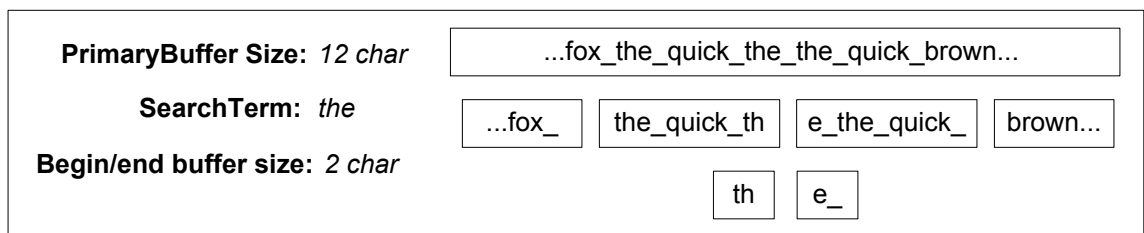


Figure 3.5 Client Search Buffers

As shown in Figure 3.5, the second ‘the’ occurrence would be missed if overlapping buffers were not used. It should also be noted that if the beginning and end buffers were too long then the first ‘the’ occurrence would be counted twice. This overlapping buffer approach was used between search buffers for a single file and between files on a client but was not used between clients. Figure 3.6 illustrates these overlapping search buffers and depicts discovered matches with a circle and missed matches with a diamond.

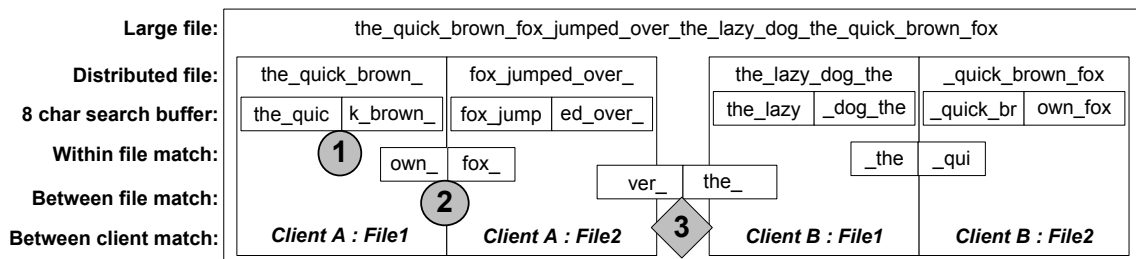


Figure 3.6 Search Buffer Overlaps



The total match count and a subset of the client's matches are sent back to the server via a `clsSearchResultsSub` class object. Pseudo code for the client search process is illustrated in Figure 3.7 below while the actual search code written in C# can be found in the Appendix.

**Input:** Drive image file and desired search term  
**Output:** `clsSearchResultsSub` and `clsSearchResults` object

- 1: Retrieve list of image chunks from client working directory sorted numerically
- 2: Copy portion of drive image to searchBuffer
- 3: Convert buffer to string, search for term, compound matches into output objects
- 4: bufferOverlap: End of previous searchBuffer with beginning of next searchBuffer
- 5: Convert bufferOverlap to string, search, and compound into output objects
- 6: Finish searching all buffer iterations for single image chunk
- 7: betweenFileOverlap: End of previous file with beginning of next file
- 8: Convert betweenFileOverlap to string, search, and compound into output objects
- 9: Select next file in directory and repeat steps 2 – 9

*Figure 3.7 Client Search Pseudo Code*

### 3.9. Application GUI

One of the major advantages of the proposed implementation is simple instantiation. While usability testing was not within the scope of this thesis the GUI for both the client and server application were designed for simplicity. The GUI for the sever side allows users to create volume images, manage clients, and conduct searches. The user can start the imaging process on the server application by selecting a volume from the imaging tab. The server will start to listen for clients after the user specifies an IP address on the server tab. Connected clients are also listed and can be managed from the server tab as illustrated in Figure 3.8 below.

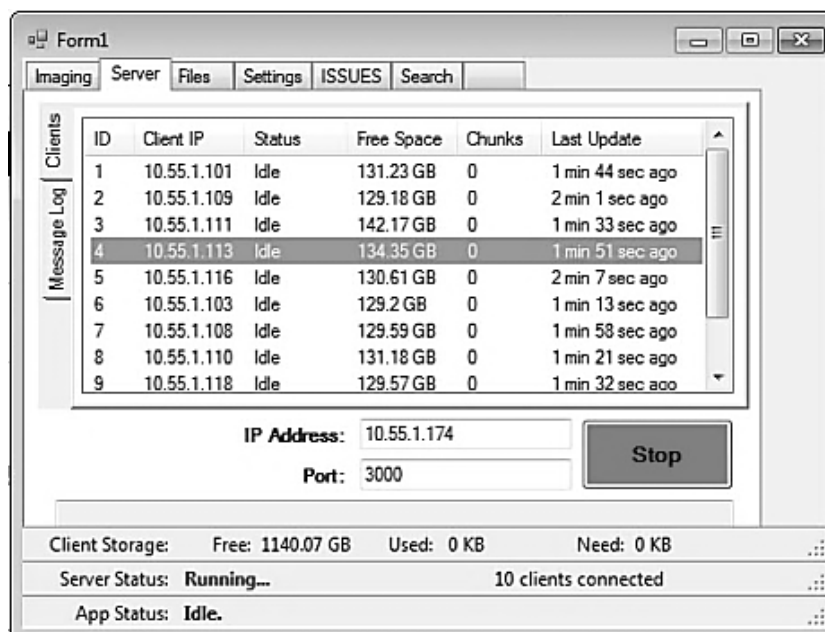


Figure 3.8 Server Application GUI

The client GUI is even simpler than the server application since a client simply receives server commands and returns results back to the server for processing. Several options were included in the client GUI including a message log for debugging, a display listing all client files, and several configurable options as shown in Figure 3.9 below.

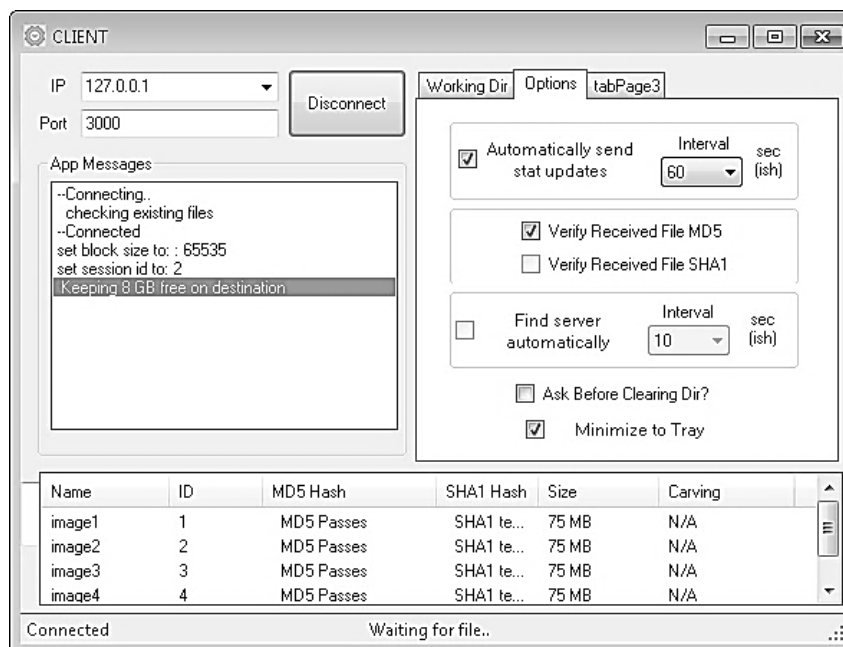


Figure 3.9 Client Application GUI

### 3.10. Hypotheses

Hypotheses with regard to this research were as follows:

1. The proposed implementation should show a decrease in text search times as the number of physical hosts involved increases.
2. The proposed approach should show a lower number of text search matches compared to the actual match count due to matches overlapping files between clients.

The first hypothesis was formulated simply due to the nature of distributed systems. Distributed systems are meant to spread processing load across several nodes thus decreasing the work load of any one single node. Adding more nodes decreases the amount of work for each single node resulting in an increase in performance. In this thesis, performance was measured by recording search runtimes thus increasing the number of nodes should shorten overall search runtime.

The second hypothesis was formulated due to a limitation with regard to the client search function. This search function is unable to find matches overlapping client instances due to the inability of clients to communicate directly. Increasing the number of clients involved increases the occurrence of such gaps thus increasing the probability of matches being missed.

The first hypothesis looks at scalability directly by measuring the effect of increasing the number of clients. The second hypothesis looks at the effect of a scalability limitation that was made apparent during application development. Both of these hypotheses are directly applicable to the research question posed in this thesis in that they help to illustrate the scalability of the proposed client/server application.

## CHAPTER 4. OPTIMIZING APPLICATION PERFORMANCE

Several application parameters are available which can be manipulated resulting in drastic changes with regard application performance. While the degree of these effects may vary across hardware architectures, values chosen for use in the application testing section were determined using the hardware specified in Table 3.1. The following section describes why values were chosen based on tracking application performance across several parameter configurations.

### 4.1. Imaging Buffer

As stated in section 3.6, two variations of 'DD' are used during the image process with one being used for volume listing while the other is used for creating images. The variation used in creating images (Garner, 2011) allows input and output buffers to be configured by the user. Eight different buffer sizes were used in creating a 1 GB solid image to determine the ideal imaging configuration. Both the input and output buffers were set to identical values. This image was created twenty times for each buffer size and the average runtimes, standard deviations, and memory usage are illustrated in the Figure 4.1 below.

Memory usage by 'DD' during the image process is essentially twice the buffer size as both an input and output buffer are allocated. Run times decreased as the buffer size increased up until the 15 MB buffer size. After this point the runtime decreases leveled off while memory usage continued to increase. Therefore the 15 MB buffer was chosen for application testing as it provided the fastest imaging times with the least amount of memory usage.

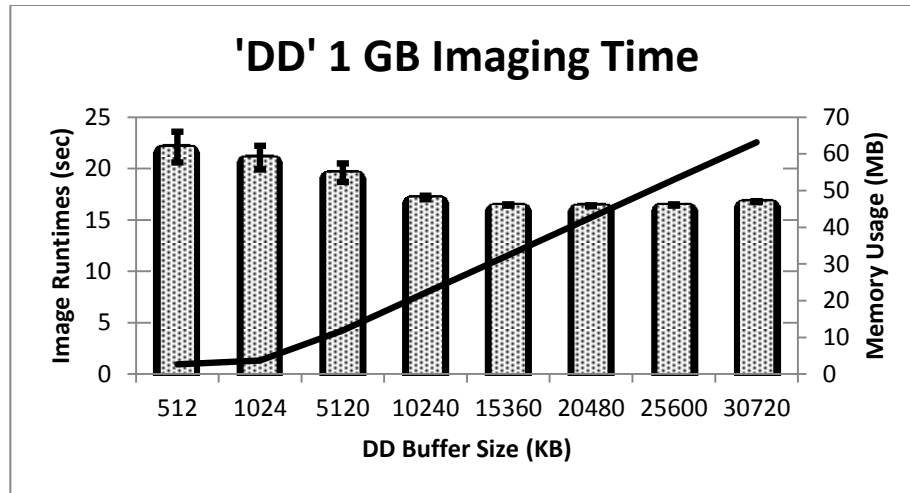
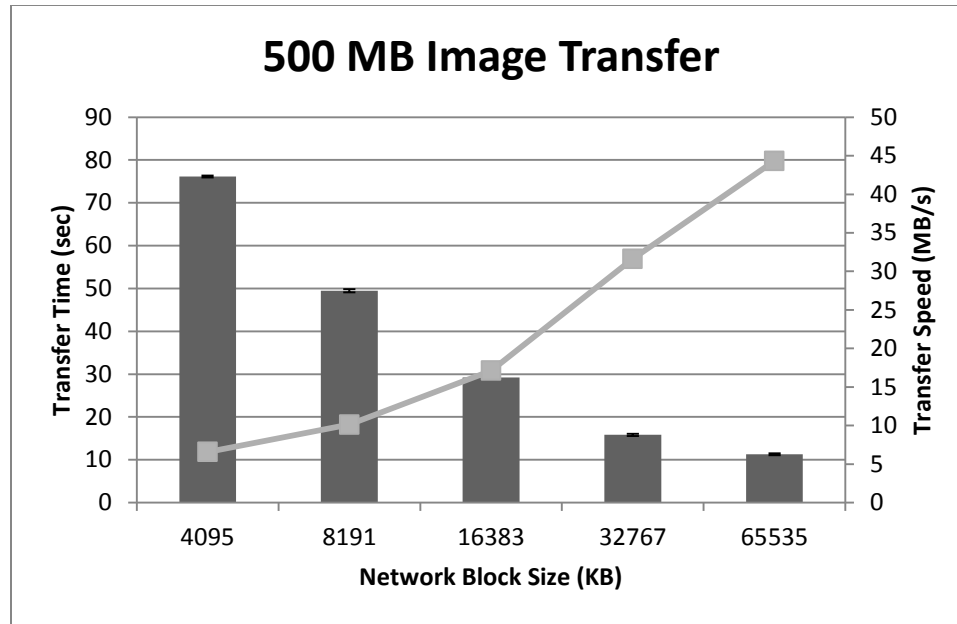


Figure 4.1 'DD' 1 GB Imaging Time

#### 4.2. Network Communication Buffer

Network communication buffers can be configured in two places which include the .Net TcpClient (Microsoft, 2013a) class and the byte array used to store information when writing data to and receiving data from the network stream. These values do not have to be identical as the .Net framework will handle chunking data accordingly. However, testing showed that having the byte array size and TcpClient buffer size differ resulted in much higher CPU utilization due to these background chunking operations. Therefore, both buffer sizes were set identically during file transfer testing. Fifteen file transfer operations were completed using a 120 MB and a 525 MB file for each of five different buffer sizes. No file transfer failures were experienced with any of the illustrated transfer iterations. The average runtime time for the fifteen iterations and their standard deviations are shown in Figure 4.2.

As expected, increasing the buffer sizes lead to faster file transfer times since more data is allowed to be sent across the network channel at a given time. Another thing to note about the graph is the increase in standard deviation as the buffer size decreased. This was simply caused by a greater number of fluctuations in the data transfer rate across a longer duration resulting in a more profound effect on transfer time.



*Figure 4.2 500 MB Image Transfer*

The 65 KB network buffer was chosen as it resulted in the fastest image transfer times. Any transfers using a buffer size greater than 65 KB failed. While the cause of this failure was not determined it is believed to be caused by the limitations of TCP which has a max window size of 65 KB. This hypothesis could be tested by configuring TCP window scaling; however, delving into the finer points of network tuning is outside the scope of this thesis. Details with regard to TCP window size and window scaling can be found in RFC 1323 created by the Internet Engineer Task Force (Jacobson et al., 1992).

#### 4.3. Client Search Buffer

The client search process is conducted by reading small portions of a drive image into a memory buffer with a configurable size. This buffer is then converted into a string which is then searched using Microsoft's Regex library. To determine the ideal search buffer size, several searches were conducted using seven different buffer sizes across five different file sizes. Ten iterations for each file and buffer size combination were averaged and graphed. The same search term was used across all iterations and no

matches were found. Since computation time is used to process discovered matches it was important that no matches be found as match processing would directly affect overall runtime. The runtime results for several continuous image files of various sizes (images that were not split into chunks) are illustrated in Figure 4.3 below.

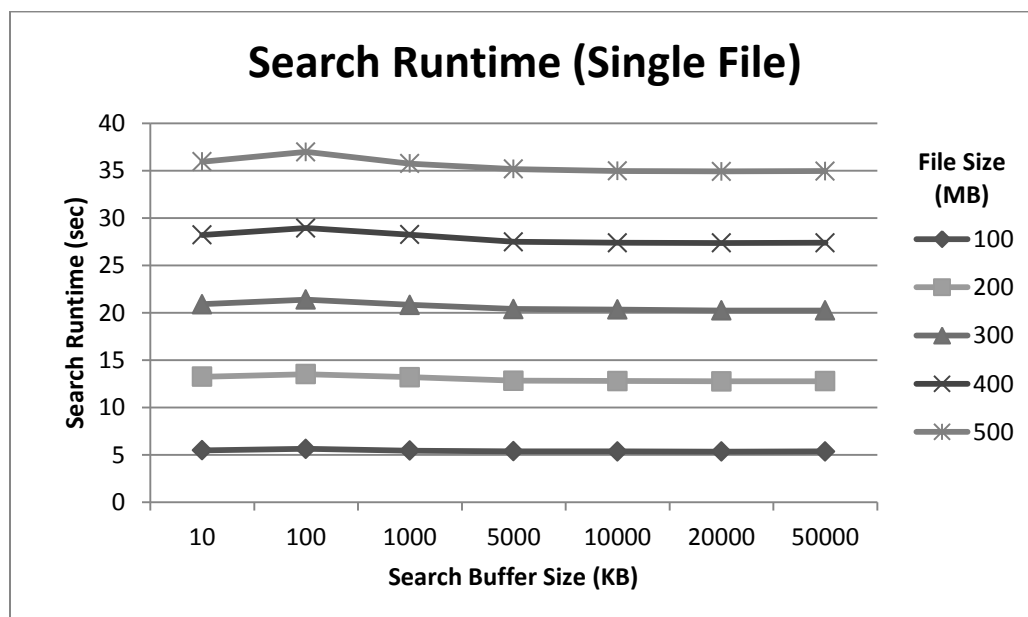


Figure 4.3 Search Runtime for a Single Image File

As expected, larger files take longer to search and increasing the search buffer size resulted in slightly faster runtimes. Search runtimes across buffer variations appeared relatively consistent but was more pronounced with larger file sizes. Unfortunately, single continuous files cannot be used as images must be distributed across several clients. Consequently search runtimes were also gauged using images that were split into several chunks.

Figure 4.4 below depicts search runtimes using a 400 MB image that was split into various chunk counts. Chunk counts were varied from a minimum of one file to a maximum of forty files. The legend in Figure 4.4 shows the number of files involved and the resulting chunk size in parenthesis.

As was the case with the single continuous files, increasing the buffer size decreased search runtimes in most cases. However, search runtimes lengthened with large and small buffer sizes as the number of chunks increased. This was particularly

apparent when looking at the performance of the forty file case. This is most likely caused by the induction of I/O operations from opening and closing a greater number of files.

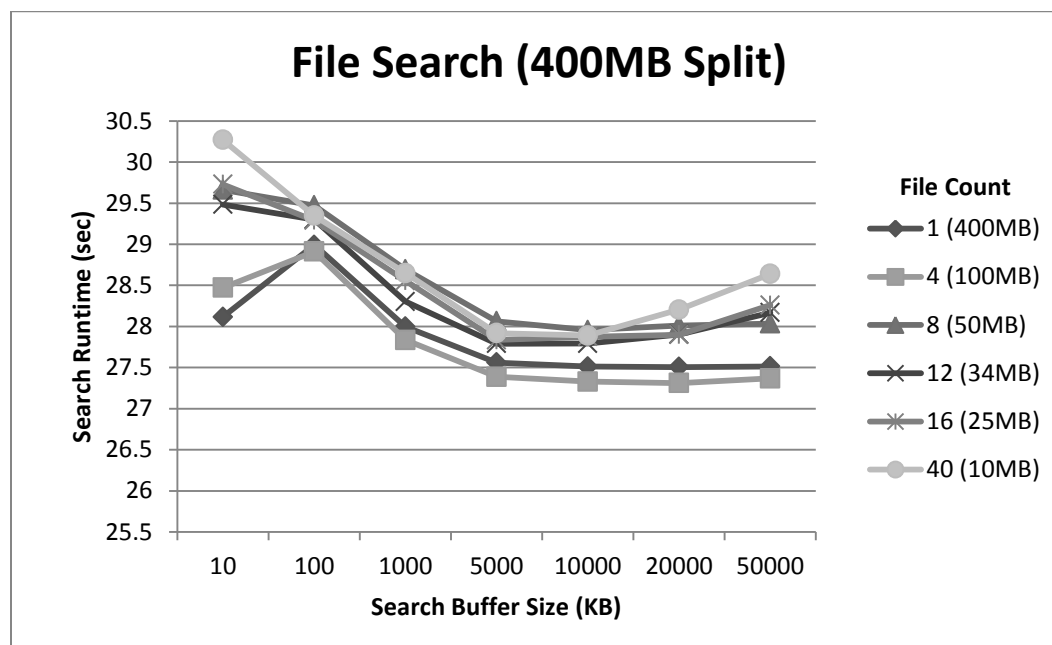


Figure 4.4 Search Runtime for Split Image File

The 10 MB search buffer was chosen as it resulted in the fastest search runtimes. This became much more apparent when testing across variations in chunk size where increasing the chunk count resulted in an increase in search times.

#### 4.4. Image Chunk Size

Image chunking operations are an important component of the proposed implementation as their inclusion makes the distribution of large suspect volumes feasible. Setting an appropriate chunk size is important but not necessarily vital for the proposed implementation to function well. Large chunk sizes would result in fewer image files but lead to longer file transfers and hashing operations. Small chunk sizes would lead to faster transfers and hashing operations but result in greater fragmentation thus inducing file management overhead as discussed in the previous section.



The ideal chunk size would result in the least number of files while allowing hashing and transfers to complete in a reasonable amount of time. While an ideal size was not determined through testing, a 500 MB chunk size was arbitrarily chosen as it appeared to satisfy the afore mentioned criteria.

#### 4.5. Summary

This chapter measured the effect of configurable performance values for several components of the proposed application architecture. These included imaging, searching, and data transfer functions. Ideal values were chosen after running each process against variations in configurable parameters for each function. These ideal values were then set as the default configuration to ensure optimum application performance.

## CHAPTER 5. RESULTS

Performance was gauged by comparing the runtime of live search operations in FTK with the runtime of the text search function using the proposed implementation. Accuracy was gauged by comparing the number of match results across both applications. This was made possible since a known block of text was written a known number of times to fill volumes before they were imaged.

Text searches were run using two scenarios which included searching for a word with a low frequency and a word with high frequency. This approach helped to extrapolate the effect match processing had on overall search runtime. Regular expression searches were also used as their computationally intensive nature results in longer runtimes when compared to their literal counterparts.

### 5.1. Test Preparation

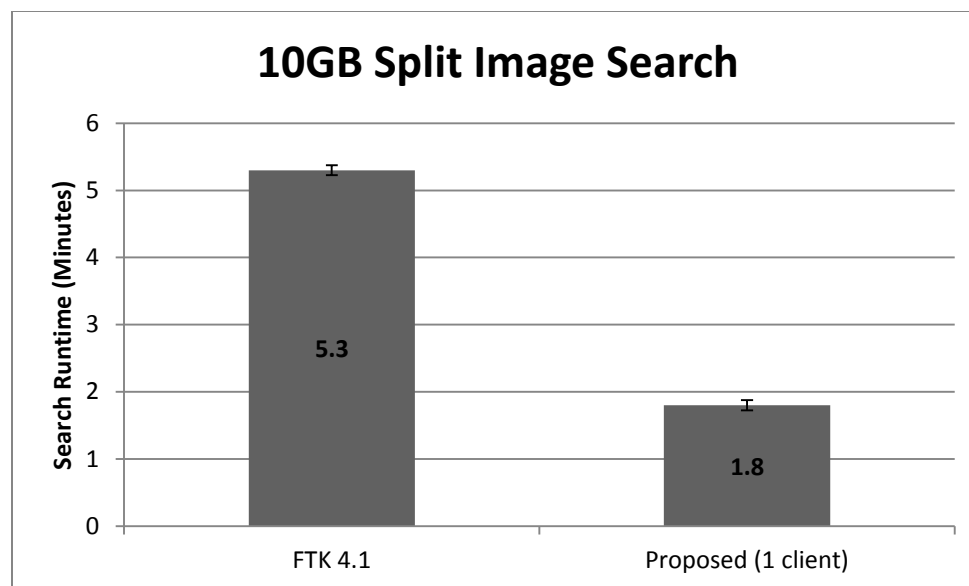
Desired images had to be created for each testing scenario without contamination from other tests. This was accomplished on the server by first creating a partition of desirable size on one of the Seagate ST1500DL 1.5 TB hard drives. This partition was then wiped with a one pass zero algorithm using DiskWipe 1.7 (<http://www.diskwipe.org>). A blank text file was created on the zeroed partition and an in house text generator was used to write text to the file until a specified size was met. Les Miserables (<http://www.gutenberg.org/files/135/135.txt>) was used as the text block as it provided a large enough block of text which helped to eliminate repetitiveness. After the text file was created the partition was then imaged and stored on the second Seagate 1.5 TB hard drive in 500 MB chunks. These chunks were then searched using FTK and the proposed implementation. This process was repeated for each testing scenario.

Another important step for test preparation was the configuration of the live search function in FTK. Live searches in FTK are run directly against suspect images which results in long search times. FTK limits the number of live search results to 200 matches per file by default to expedite the search process. This is unacceptable as searches were conducted against a single large text file which would result in the early termination of the live search process. During testing this live search limit was set to zero which allowed the entire file, and consequently the entire image, to be searched in full.

It should be noted that search runtimes essentially equate to the average of the slowest running host. Some host may have slower search runtimes due to irrelevant background processes and or hard drive problems such as a large number of reallocated sectors.

## 5.2. Initial Testing on Server Hardware

The first test conducted was used to record a base line performance measurement. This was accomplished by first following the preparation steps described in section 5.1 to create a 10 GB image split into 500 MB chunks. The text generator wrote the Les Miserables text block 3,131 times to fill the 10 GB partition before it was imaged. Both FTK and the client portion of the proposed application were run on the designated server meaning there was no variation in hardware. The images chunks were searched for the term 'crowell' due to its low frequency per text block iteration. The term 'crowell' occurred only once per text block iteration meaning a match count of 3,131 was expected. Both FTK and the proposed client application returned this expected match count. Search runtime results for both applications averaged across five iterations are illustrated in Figure 5.1 below.



*Figure 5.1* 10 GB Split Image Search Runtimes

The search process used in the proposed client application was nearly three times faster for literal string searches than the search process used in FTK 4.1. Variations in hardware were not the cause since both applications were run on the server described in Table 3.1.

### 5.3. FTK 4.1 Issues

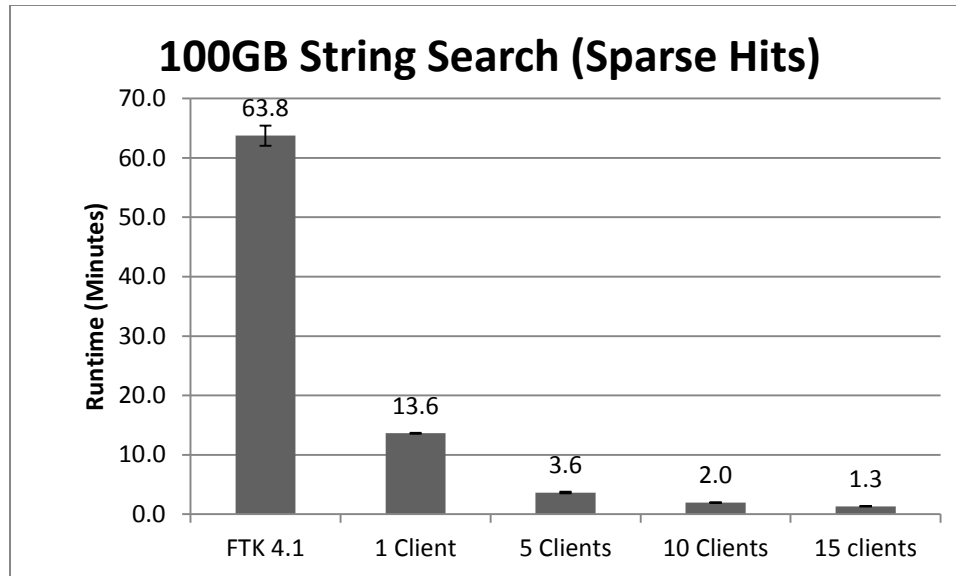
Several problems were experienced when using FTK 4.1 against the 100 GB test images filled with text. The FTK indexing process would hang when adding the 100 GB test image to FTK 4.1 with default settings. The process was allowed to run for a 48 hour period but a “failed work event error” was continually written to the job error log. This error was further described with “Could not search objects: bad allocation”. The 100 GB image was recreated several times but the error still occurred. The error was finally remedied when the FTK indexing process was disabled during the add evidence process. While the direct cause of the error was not determined, it was surmised to be caused by the excessive amount of repetitive text in the test dataset which the FTK data processor simply could not handle.

#### 5.4. Proposed Application Issues

An issue was encountered with the proposed application that affected both the client and server variants. Several hundred matches were being missed between files on a single client even though debugging eliminated the search function as the cause. On the server side, the folder monitor function would hang even though the 'DD' imaging process would complete successfully. It was eventually surmised that both the search function on the client and the folder monitor function on the server relied on a file listing function within Microsoft's .NET Framework. This file listing function returns a list of files in a directory sorted alphabetically instead of numerically. A third-party file listing function was used to retrieve a list of files in a directory sorted numerically. Both the client and server issues were resolved once the original file listing function was replaced. Vcepa (2005) describes the problem in more detail including the third-party solution that was used.

#### 5.5. Distributed String Searches (Sparse Hits)

The distributed string search test scenario described in this section used a word with a low frequency against a 100 GB image file split into 500 MB chunks. Exactly 31,529 text block iterations were written to fill the 100 GB volume before it was imaged. The search term 'crowell' was used as it has the lowest frequency with only one occurrence per text block iteration. Five runs were averaged for each of the five testing scenarios which are illustrated in Figure 5.2 below.



*Figure 5.2* 100 GB String Search with Sparse Hits

No failures were experienced during any of the test runs and all implementations returned the expected 31,529 result count. FTK 4.1 had the longest search run time taking nearly four times longer than a single client running the proposed application. As expected, adding more clients increased search performance nearly in proportion to the number of clients added. The fifteen client scenario showed nearly a fifty times speed up when compared to FTK and a thirteen times speed up when compared to the single client scenario.

#### 5.6. Distributed String Searches (Several Hits)

The distributed string search test scenario described in this section used a word with a high frequency against a 100 GB image file split into 500 MB chunks. The purpose of this test was to illustrate the effect match processing has on overall search times. The same process from the previous section was used with the only difference being the search term. The search term ‘his’ was used for this test because it had a reasonably high frequency but not absurdly so like many noise words such as ‘the’. One Les Miserables text block iteration contained 10,753 iterations of the term ‘his’ resulting

in an expected total match count of 339,031,337. Five runs were averaged for each of the five testing scenarios and are illustrated in Figure 5.3 below.

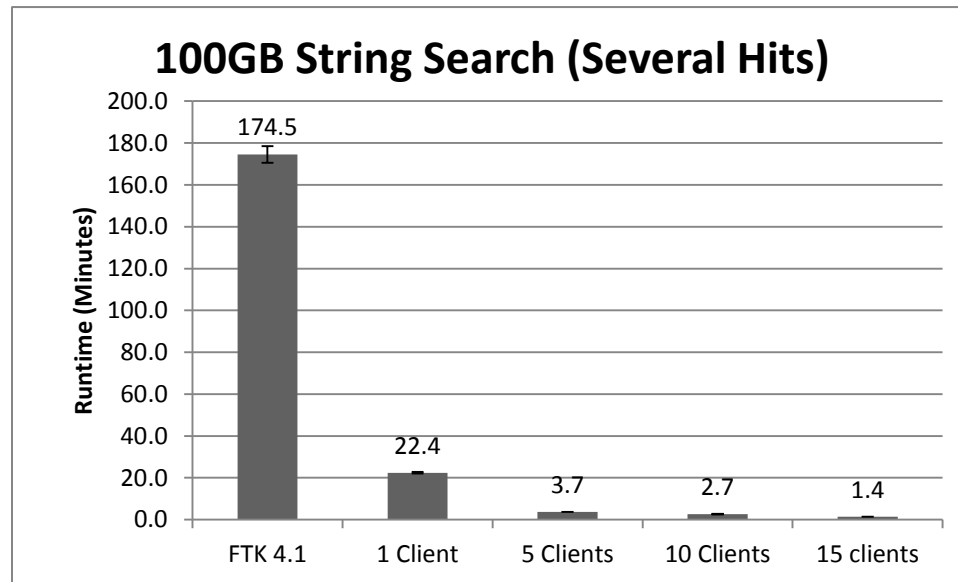


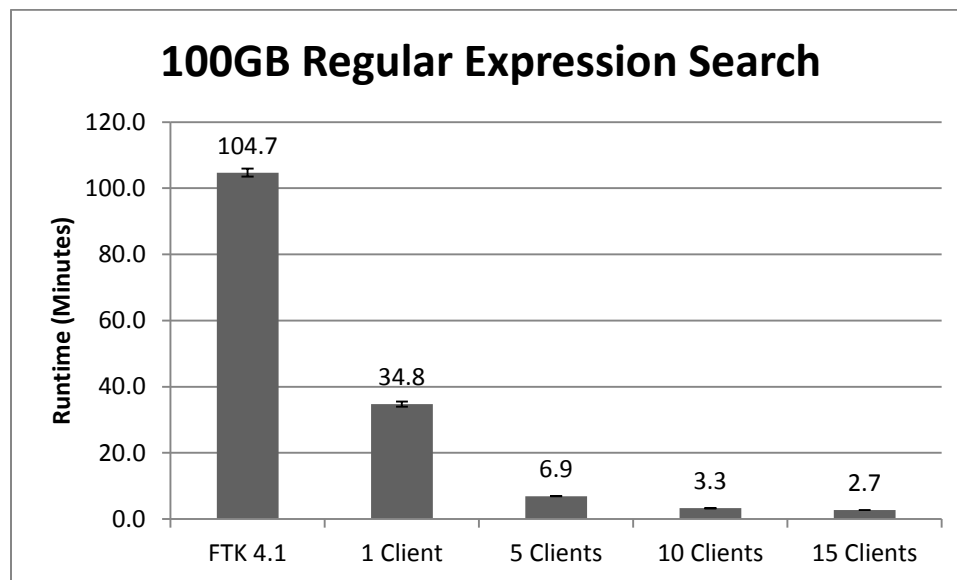
Figure 5.3 100 GB String Search with Several Hits

The proposed application experienced a problem in that it did not return the expected count of results. The ten and fifteen client scenarios were three hits shy of the expected 339,031,337 hits while the five client scenario was only two hits shy. These results confirm the hypothesis that matches overlapping clients would be overlooked and therefore not counted. This was further reinforced by the fact that the single client scenario returned the expected result count exonerating the client search function as the cause.

The speed up experienced in the several hits scenario was much greater than the speed up in the sparse hits scenario. The fifteen client case showed a 126 times speed up when compared to FTK 4.1 and a fifteen times speed up when compared to the single client case. All testing scenarios experienced an increase in search times when compared to the sparse hits scenario in the previous section. This can most likely be attributed to processing and storing an increased number of matches.

### 5.7. Regular Expression Searches

A regular expression search test was conducted against the 100 GB test image to gauge the performance of processor intensive searches. The query “[qer] was arbitrarily chosen due to the relatively high frequency of quoted text in the Les Miserables text block. Figure 5.4 show the results of running the selected regular expression query across several different testing scenarios with each scenario averaged across five iterations.



*Figure 5.4* 100 GB Regular Expression Search

Regular expression searches increased search runtimes across all testing scenarios when compared to literal string searches. When a potential match is found the regular expression is referenced for all expression combinations. This is a CPU intensive task which causes the longer search runtimes. FTK had the longest search runtime time while the proposed application showed a decrease in search time as clients were added. The fifteen client scenario provided the fastest regular expression search time which was approximately thirty-eight times faster than FTK 4.1.

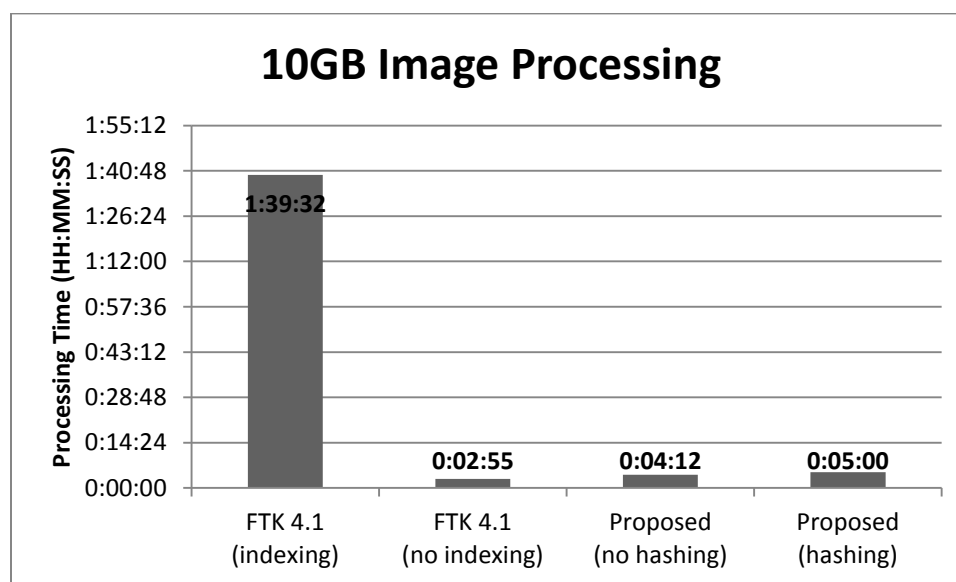
All client scenarios showed an identical match count of 5,174,931 which was 4,175 matches greater than expected. This was believed to be caused by file system meta data located at the beginning of the drive image since it was consistent across all



iterations. FTK 4.1 returned 476 fewer results than expected with a total value of 5,171,232 which was also consistent across all five iterations. The cause of these differing match counts between applications was not fully determined.

### 5.8. Image Processing

Both the proposed application and FTK 4.1 must process images before they can be searched. FTK 4.1 conducts several processing operations when an image is added as evidence including indexing, file carving, and thumbnail generation just to name a few. Image processing for the proposed application consists of hashing individual image files and distributing them across all clients. Image processing times for the 10 GB test image are illustrated in Figure 5.5 below.

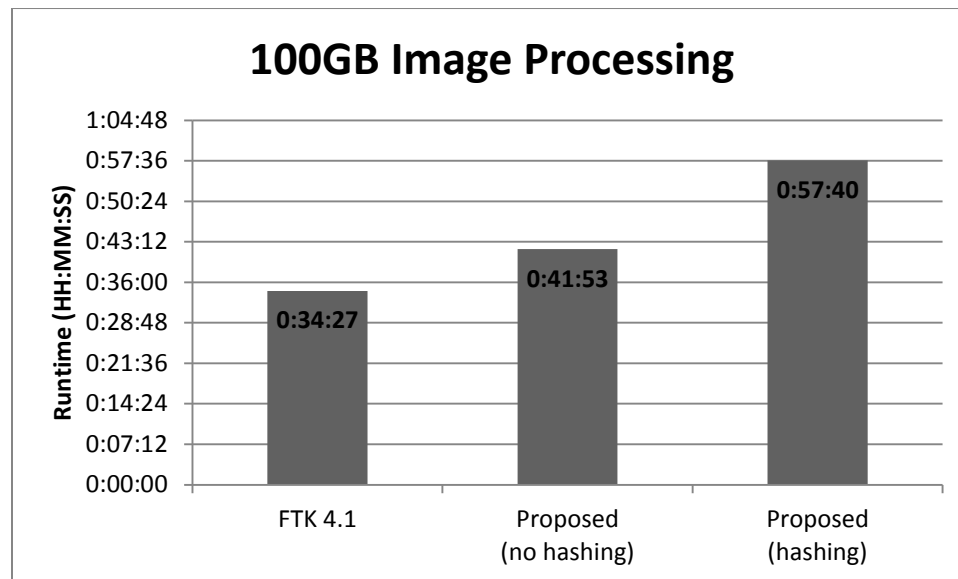


*Figure 5.5* 10 GB Image Processing Times

FTK 4.1 had the longest processing time for the 10 GB test image but was the fastest processor when its indexing functionality was disabled. Hashing accounted for less than 20% of the proposed application processing time while Indexing accounted for more than 95% of the processing time in FTK. Indexing processes lead to very long evidence load times in FTK but allowed for nearly instantaneous index searches. A 10

GB image is not a realistic representation of modern datasets so imaging processing times were also recorded for a 100 GB test image.

FTK 4.1 had difficulty handling the 100GB test images as described in section 5.3 and therefore an indexing time for the 100 GB image could not be determined. All FTK 4.1 settings were kept at default with the exception of disabling the indexing process. Figure 5.6 below shows the 100 GB image process times for FTK 4.1 with indexing disabled along with the proposed application with and without hashing functionality.



*Figure 5.6* 100 GB Image Processing Time

FTK 4.1 had the fastest processing time with the 100 GB image as its indexing functionality was disabled due to program failure. Hashing functions were used in the proposed application to ensure images were not corrupted during the transfer process. Both the server and the client must calculate hash values and the server compares the values upon transfer completion. The inclusion of this functionality accounted for nearly 27% of the processing time which is slightly higher when compared to the 10 GB image case.

### 5.9. Summary

The DDF implementation developed by Richard and Roussev (2004) excelled at regular expression searches while also being slightly faster with regard to load time. The application proposed in this thesis showed a massive runtime advantage with regard to literal string searches. Direct comparison of the two approaches is difficult given the multitude of differences in the implementations and testing approaches. For example, the difference in regular expression speedups could simply be the result of variations in FTK versions.

In the best case scenario the client/server application showed a 125 times speed up in text search times using fifteen clients when compared to FTK 4.1. Indexing evidence in FTK takes a rather lengthy amount of time but allows for nearly instantaneous indexed searches. The proposed application allows live searches to complete very quickly almost eliminating the need for indexing operations. Figure 5.7 illustrates overall runtime of operations across several testing scenarios. FTK indexing was disabled as explained in Section 5.3.

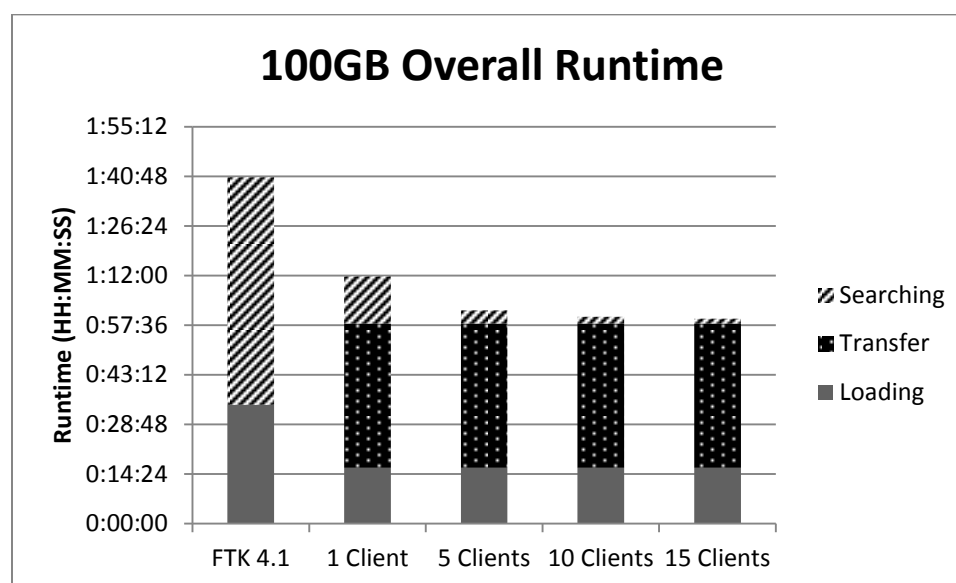
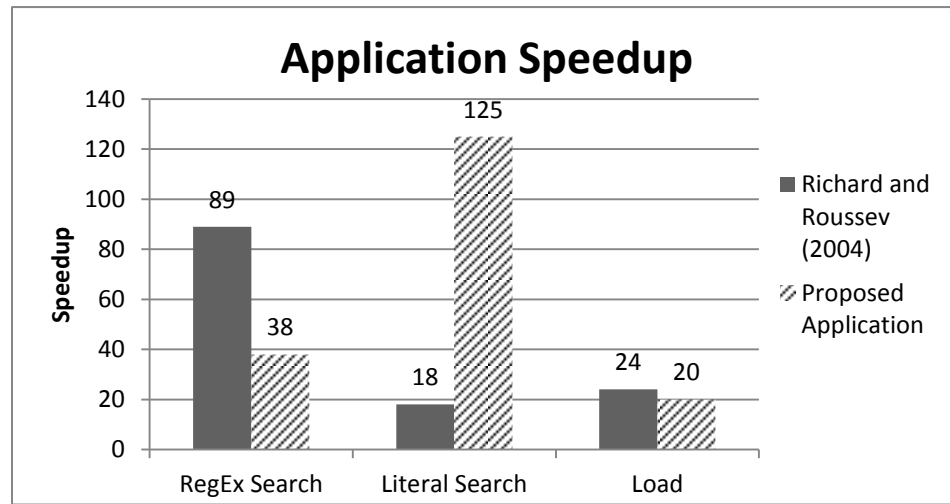


Figure 5.7 100 GB Overall Runtime

Richard and Roussev (2004) conducted similar testing of their distributed digital forensics implementation using both literal string and regular expressions searches. Their

best case speedups as well as the best case speedups for the implementation proposed in this thesis are illustrated in Figure 5.8 below.



*Figure 5.8 Application Speedup Compared to FTK*

During testing, all hypotheses in Section 3.10 were proven correct. Increasing the number of clients running the developed architecture resulted in increasingly faster search runtimes for all testing scenarios. Increasing the client count also led to matches being missed due to an increased probability of overlapping client instances. This was only apparent in the ‘several hits’ testing scenario due to the large quantity of search matches.

## CHAPTER 6. CONCLUSION

The result of this thesis was a client/server application architecture employing the distributed digital forensics concept. Similar approaches have been developed but this thesis resolves some of their inherent infeasibilities. These include substantial financial investments and difficulty in application instantiation. This thesis provides a solution to these infeasibilities by leveraging existing desktop workstations on an internal network.

The application architecture developed in this thesis was written entirely in C# and allows for one server instance to manage several client instances. Its effectiveness was gauged by testing it against FTK which is one of the most widely used tools in the digital forensics discipline. Testing consisted of using fabricated volume images to compare search match counts and runtimes across both applications. Literal string searches and regular expression searches were run across various counts of physical hosts and on a single machine running FTK 4.1.

The client/server architecture in this thesis showed a decrease in search runtimes in all testing scenarios when compared to FTK 4.1. With a 100 GB volume image, a 125 times speed up was experienced in the best case while a three time speed up was experienced in the worst case. The hypotheses stated in this research were validated in that search runtimes decreased as clients were added and matches were missed due to the inability to search between clients.

While the search runtime results were faster than FTK 4.1 in all testing scenarios, the client/server architecture could greatly benefit from additional functionality. This includes cross platform operation, search browsing beyond the client maximum, and the securing of evidence on remote clients as described in Section 6.1 below. Adding such functionalities would make the proposed application a practical tool for use in real world digital investigations.

Distributed digital forensics can be a viable option for analyzing today's very large suspect datasets. The quick live search times of DDF implementations almost eliminate the need for time consuming indexing processes. This time savings consequently results in faster investigation times which are vital in time sensitive investigations. While other DDF approaches have been developed, the approach in this thesis is novel in that it is simple to instantiate while also eliminating the need to purchase and or dedicate additional hardware. As such it is a practical tool for an investigative agency of any size to utilize. The end result is the ability to conduct more digital investigations in less time helping to alleviate the strains of an ever increasing backlog of digital evidence.

## 6.1. Future Work

While the performance of the proposed distributed digital forensics application is noteworthy, it suffers from several shortcomings. These include limited operating system compatibility, search matches not being displayed, search matches not being found, and remnant evidence on the clients. These shortcomings were made apparent during application development and solutions were partially implemented due to their irrelevancy to the scope of the thesis. The proposed application can become a much more viable solution for the digital forensics community if these shortcomings were addressed. This chapter describes these shortcomings and potential solutions that may be fully implemented in the future.

### 6.1.1. Cross Platform Functionality

The proposed application was developed entirely in Microsoft's C# programming language. This is not ideal as it severely hampers cross platform operation on networks consisting of machines running various operating systems. The obvious solution would be to port the application to a more cross platform friendly programming language such as C++.

### 6.1.2. Search Improvements

As stated in section 3.5, each client sends a list of search results back to the server for display to the user. The number of matches sent to the server is limited based on both the size of the results and the size of the communication buffer. This limit helps prevent resource saturation since the server is responsible for storing results from every client. Matches not stored in the initial batch of matches are not stored on the server and thus cannot be displayed to the user.

This shortcoming was apparent during development and as such an index pointer was configured on the client side. This pointer is updated to the index of the last search match located at the end of the last batch of matches. The final step required to complete this solution would be to implement search querying on the server. The client would simply send the next batch of matches greater than the stored index value when the server requests additional matches.

Quite possibly the greatest shortcoming of the proposed search process is the inability to search files overlapping separate clients. A simple solution would be to store the beginning of the first file and the end of the last file from each client on the server. The resulting loss of resources on the server should essentially equate to two times the search term length multiplied by the number of clients. This resource loss would be justified as it would eliminate the need to implement any type of direct communication between clients.

### 6.1.3. Security Improvements

Quite possibly the biggest issue with the proposed implementation is the security concerns of the distribution process. This application was created with the intention of operating on existing computational resources. Doing so facilitates easier instantiation while also eliminating the need to purchase and or dedicate hardware. Unfortunately, such machines may be publicly accessible and or compromised which could lead to possible evidence tampering and or malicious file recovery.

One potential solution would be to employ some sort of encryption on the remote clients. Employing encryption could lead to significant performance loss depending on what algorithm is used and how it is implemented. This performance loss may be considered acceptable depending on the sensitivity of the suspect data being analyzed. However, encryption may not be necessary if all machines are considered secure and unoccupied during analysis. Secure file wiping may be a more viable if encryption is deemed impractical or unnecessary. Involved clients could securely wipe all suspect image files once an investigation is completed. Deciding between either of these options would depend largely on the situation as well as the impact on overall application performance.



## LIST OF REFERENCES

## LIST OF REFERENCES

- AccessData Group. (2013). FTK overview. Retrieved November 24, 2013, from <http://accessdata.com/products/computer-forensics/ftk>
- Allcock, B., Bester, J., Bresnahan, J., Chervenak, A., Kesselman, C., Meder, S., et al. (2001). *Secure, efficient data transport and replica management for high-performance data-intensive computing*. In *Mass Storage Systems and Technologies, 2001. MSS'01. Eighteenth IEEE Symposium on* (pp. 13-13). IEEE.
- Ames, J., Bresnahan, J., Chervenak, A., Feller, M., Foster, I., Keator, D., et al. (2010, December). A data management framework for distributed biomedical research environments. In *e-Science Workshops, 2010 Sixth IEEE International Conference on* (pp. 72-79). IEEE.
- Beebe, N. (2009). Digital forensic research: The good, the bad and the unaddressed. In *Advances in digital forensics V* (pp. 17-36). Springer Berlin Heidelberg.
- Burke, P., Craiger, P., Marberry, C., & Pollitt, M. (2008). A virtual digital forensics laboratory. *Advances in digital forensics IV* (pp. 357-365). New York: Springer.
- Davis, M., Manes, G., & Sheno, S. (2005). A network-based architecture for storing digital evidence. In *Advances in Digital Forensics* (pp. 33-42). Springer US.
- Douceur, J., & Bolosky, W. (1999). A large-scale study of file-system contents. *ACM SIGMETRICS Performance Evaluation Review*, 27(1), 59-70.
- Garfinkel, S. L. (2010). Digital forensics research: The next 10 years. *Digital Investigation*, 7, S64-S73.
- Garner, G. (2013, August 30). Forensic Acquisition Utilities. *Forensic Acquisition Utilities*. Retrieved November 24, 2013, from <http://gmgsystemsinc.com/fau/>
- Hachman, M. (2011, July 20). Average drive capacity tops 500 GB as Seagate reports profit. *PCMag*. Retrieved November 26, 2013, from <http://www.pcmag.com/article2/0,2817,2388807,00.asp>

- Halfacree, G. (2012, March 20). Seagate HAMRs hard drives to 1Tb per square inch. *Bit-tech*. Retrieved November 26, 2013, from <http://www.bit-tech.net/news/hardware/2012/03/20/seagate-hamr/1>
- Jacobson, V., & Borman, D. (1992, May). TCP extensions for high performance. *The Internet Engineering Task Force*. Retrieved November 24, 2013, from <http://www.ietf.org/rfc/rfc1323.txt>
- Kechadi, M., & Scanlon, M. (2010). Online Acquisition of Digital Forensic Evidence. In *Digital Forensics and Cyber Crime* (pp. 122-131). Springer Berlin Heidelberg.
- Microsoft. (2013a). TcpClient Class. Retrieved November 24, 2013, from [http://msdn.microsoft.com/en-us/library/system.net.sockets.tcpclient\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/system.net.sockets.tcpclient(v=vs.110).aspx)
- Microsoft. (2013b). About WMI. Retrieved November 24, 2013, from [http://msdn.microsoft.com/en-us/library/aa384642\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/aa384642(v=vs.85).aspx)
- Mrdovic, S., Huseinovic, A., & Zajko, E. (2009, October). Combining static and live digital forensic analysis in virtual environment. In *Information, Communication and Automation Technologies, 2009. ICAT 2009. XXII International Symposium on* (pp. 1-6). IEEE.
- Richard III, G. & Roussev, V. (2004, August). Breaking the performance wall: The case for distributed digital forensics. In *Proceedings of the 2004 Digital Forensics Research Workshop* (Vol. 94).
- Richard III, G., & Roussev, V. (2006a). Next-generation digital forensics. *Communications of the ACM*, 49(2), 76-80.
- Richard III, G. & Roussev, V. (2006b). Digital forensics tools: the next generation. *Digital Crime and Forensic Science in Cyberspace*. Idea Group Publishing, 75-90.
- Richard III, G., Roussev, V., & Tingstrom, D. (2006). dRamDisk: Efficient RAM sharing on a commodity cluster. In *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International*, 193-198. IEEE.
- Roussev, V., Wang, L., Richard III, G., & Marziale, L. (2009). MMR: A platform for large-scale forensic computing. *Research Advances in Digital Forensics*, 5, 201-214.

- Stoica, I. (2009). Overlay networks. *University of Virginia Department of Computer Science*. Retrieved November 24, 2013, from <http://www.cs.virginia.edu/~cs757/slidespdf/757-09-overlay.pdf>
- Turner, P. (2006). Selective and intelligent imaging using digital evidence bags. *Digital Investigation*, 3, 59-64.
- Vishwanath, V., Burns, R., Leigh, J., & Seablom, M. (2009). Accelerating tropical cyclone analysis using LambdaRAM, a distributed data cache over wide-area ultra-fast networks. *Future Generation Computer Systems*, 25(2), 184-191.
- Vcepa. (2005, August 8). Numeric string sort in C#. *Code Project*. Retrieved November 23, 2013, from <http://www.codeproject.com/Articles/11016/Numeric-String-Sort-in-C>

## APPENDIX

## APPENDIX. CLIENT FILE SERACH CODE IN C#

client search code.txt

12/1/2013

```

public clsSearchResults searchResults = new clsSearchResults();
public clsSearchResultsSub searchResultsSub = new clsSearchResultsSub();
public bool searchingFiles = false;
public bool searchMatchCase = false;
public bool searchPreview = true;
public bool searchMatchWholeWord = false;
public int searchMatchesMax = 1000000;
public int searchMatches = 0;
private string searchFor = "";
private RegexOptions regexOptions;
public DateTime searchStart;
public DateTime searchEnd;
public ArrayList SearchResults = new ArrayList();
private void searchBackgroundWorker_DoWork(object sender, DoWorkEventArgs e)
{
    int searchBufferSize = 10000;
    MatchCollection matches;
    //set the options for whole word and case sensitive matching
    if (!searchMatchCase)
        regexOptions = RegexOptions.IgnoreCase;
    if (searchMatchWholeWord)
        searchFor = " " + searchFor + " ";

    bool addMatches = true; //simply for debugging
    int searchLength = searchFor.Length;
    //set overlap buffers so matches arent counted multiple times (cant be
    length of search term)
    byte[] fileChunkOverlap = new byte[(searchLength * 2) - 2];
    byte[] betweenFileOverlap = new byte[(searchLength * 2) - 2];

    searchingFiles = true;
    systemMessage("searching");
    searchStart = DateTime.Now;
    try
    {
        addMatches = true;
        searchMatches = 0;
        searchResults.resetResults();
        searchResultsSub.resetResults();
        byte[] beginningOfFile = new byte[searchLength - 1];
        byte[] endOfFile = null;
        foreach (clsFile singleFile in receivedFiles)
        {
            //loop through all files
            if (singleFile.beingReceived)
                continue;
            string filePath = singleFile.filePath;
            FileInfo newInfo = new FileInfo(filePath);
            long fileSize = newInfo.Length;
            //set the temp buffer to the desired size
            byte[] searchBuffer = new byte[searchBufferSize];
            if (fileChunkOverlap.Length != ((searchLength * 2) - 2))
                Array.Resize(ref fileChunkOverlap, (searchLength * 2) -
                2);

            FileStream fileReader = new FileStream(filePath,
            FileMode.Open, FileAccess.Read, FileShare.None, 320000);
            long bytesRead = 0;
            bool beginning = true;
            string searchString = "";
            while (bytesRead < fileSize)
            {
                //read entire single file in buffer chunks and compound

```

client search code.txt

12/1/2013

```

the results
int counter = fileReader.Read(searchBuffer, 0,
searchBufferSize);
bytesRead += counter;
//resize buffer for tail end of file
if (counter < searchBufferSize)
    Array.Resize(ref searchBuffer, counter);
//convert byte buffer to string
searchString = clsMethods.byteArrayToString(searchBuffer);
matches = Regex.Matches(searchString, searchFor,
regexOptions);
//search all microsoft regex match objects and compound
them
foreach (Match singleMatch in matches)
    searchMatchAdd(bytesRead, singleFile.fileID,
        singleMatch, searchString);
if (addMatches)
    searchMatches += matches.Count;

if (searchBuffer.Length > searchLength)
{
    if (beginning)
    {
        //copy file beginning of file for between file
        search
        Array.Copy(searchBuffer, (counter - searchLength)
+ 1, fileChunkOverlap, 0, searchLength - 1);
        Array.Copy(searchBuffer, 0, beginningOfFile, 0,
searchLength - 1);
        beginning = false;
    }
    else if (!beginning)
    {
        //if it is not the beginning then search between
        buffers
        Array.Copy(searchBuffer, 0, fileChunkOverlap,
searchLength - 1, searchLength - 1);
        searchString =
        clsMethods.byteArrayToString(fileChunkOverlap);
        matches = Regex.Matches(searchString, searchFor,
regexOptions);
        foreach (Match singleMatch in matches)
            searchMatchAdd(bytesRead, singleFile.fileID,
                singleMatch, searchString);
        if (addMatches)
            searchMatches += matches.Count;
        Array.Copy(searchBuffer, (counter - searchLength)
+ 1, fileChunkOverlap, 0, searchLength - 1);
    }
}
else
{
    Array.Copy(searchBuffer, 0, fileChunkOverlap,
searchLength - 1, searchBuffer.Length);
    Array.Resize(ref fileChunkOverlap, (searchLength - 1)
+ (searchBuffer.Length));
    searchString =
    clsMethods.byteArrayToString(fileChunkOverlap);
    matches = Regex.Matches(searchString, searchFor,
regexOptions);
    foreach (Match singleMatch in matches)
        searchMatchAdd(bytesRead, singleFile.fileID,
            singleMatch, searchString);
}
}

```



client search code.txt

12/1/2013

```

        if (addMatches)
            searchMatches += matches.Count;
    }
}
fileReader.Close();
//copy the end of file for between file overlap
fileReader = new FileStream(filePath, FileMode.Open,
    FileAccess.Read, FileShare.None, 3200000);
if (endOfFile == null)
{
    endOfFile = new byte[(searchLength - 1)];
    fileReader.Seek(fileReader.Length - (searchLength - 1),
        SeekOrigin.Begin);
    fileReader.Read(endOfFile, 0, searchLength - 1);
}
else
{
    //if its the end of the file then compound buffers to
    search between files
    Array.Copy(endOfFile, 0, betweenFileOverlap, 0,
        searchLength - 1);
    Array.Copy(beginningOfFile, 0, betweenFileOverlap,
        searchLength - 1, searchLength - 1);
    searchString =
        clsMethods.byteArrayToString(betweenFileOverlap);
    matches = Regex.Matches(searchString, searchFor,
        regexOptions);
    foreach (Match singleMatch in matches)
        searchMatchAdd(bytesRead, singleFile.fileID,
            singleMatch, searchString);
    if (addMatches)
        searchMatches += matches.Count;
    //copy end of file for next file overlap
    fileReader.Seek(fileReader.Length - (searchLength - 1),
        SeekOrigin.Begin);
    fileReader.Read(endOfFile, 0, searchLength - 1);
}
fileReader.Close();
}
searchResultsSub.totalMatches = searchMatches;
searchEnd = DateTime.Now;
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
//if searchResultsSub is larger than buffer size error will occur
if (!sendBytes(clsMethods.objectToByteArray(searchResultsSub), 205))
    clsMethods.showBadDialogBox("send bytes error", "serialize search
        results fail");
}
private void searchBackgroundWorker_WorkComplete(object sender,
    RunWorkerCompletedEventArgs e)
{
    searchingFiles = false;
    systemMessage("searchComplete");
}
private byte[] tempByteArray = null;
private byte[] tempByteArray2 = null;
private bool searchMatchAdd(long inputOffset, int inputFileID, Match
    inputMatch, string inputSearchBuffer)
{
    //store newly discovered matches but limit them to prevent unnecessary

```

client search code.txt

12/1/2013

```

memory usage
if (searchMatches < searchMatchesMax)
{
    string matchStart = "";
    string matchEnd = "";
    if (searchPreview)
    {
        //store a preview of matches of content before and after a
        match
        if (inputMatch.Index - 15 > 0)
            matchStart = inputSearchBuffer.Substring(inputMatch.Index
            - 15, 15);
        if (inputMatch.Index + 15 + inputMatch.Value.Length <
        inputSearchBuffer.Length)
            matchEnd = inputSearchBuffer.Substring(inputMatch.Index +
            inputMatch.Value.Length, 15);
    }
    clsSearchMatch newMatch = new clsSearchMatch(inputOffset,
    inputMatch.Index, inputFileID, matchStart, inputMatch.Value,
    matchEnd);
    //add the match to the master search list on the client
    searchResults.addSearchMatch(newMatch);
    //no need to waste processing time if subset matches are full
    if (searchResultsSub.addMatches)
    {
        tempByteArray = clsMethods.objectToByteArray(newMatch);
        tempByteArray2 =
        clsMethods.objectToByteArray(searchResultsSub);
        //add the match to the matches subset that is sent to the
        server
        if ((tempByteArray2.Length + tempByteArray.Length) <
        blockSize)
            searchResultsSub.searchResults.Add(newMatch);
        else
            searchResultsSub.addMatches = false; //sub matches is full
            so dont add anything
        tempByteArray = null;
        tempByteArray2 = null;
    }
    return true;
}
else
    return false;
}

```