Breaking the Performance Wall:
The Case for Distributed Digital Forensics

Vassil Roussev
Golden G. Richard III

Department of Computer Science
University of New Orleans
New Orleans, LA 70148
{vassil, golden}@cs.uno.edu

1. Abstract

Current trends in computing and communications technologies are putting vast amounts of disk storage and abundant bandwidth in the hands of ordinary computer users. These trends will very soon completely overwhelm digital forensics investigators attempting investigations using a single workstation as a platform. The symptoms: Performing simple preprocessing operations such as indexing of keywords and image thumbnail generation against a captured image will consume vast amounts of time before an investigation can even begin. Non-indexed, "live" searches, such as those involving regular expressions, are already unbearably slow and will become completely intolerable. Even worse, it will be impossible to raise the level of sophistication of digital forensics analysis because single forensics workstations will simply not be up to the task. It is therefore inevitable that forensic investigation tools will have to employ the distributed resources of a pool of computer systems in order to make investigations manageable.

In this paper, we make the case for distributed digital forensic (DDF) tools and provide several real-world examples where traditional investigative tools executing on a single workstation have clearly reached their limits, severely hampering timely processing of digital evidence. Based on our observations about the typical tasks carried out in the investigative process, we outline a set of system requirements for DDF software. Next, we propose a lightweight distributed framework designed to meet these requirements and describe an early prototype implementation of it. Finally, we present some performance comparisons of single- versus multiple-machine implementations of several typical tasks and describe some more sophisticated forensics analysis techniques, which will be enabled by a transition to DDF tools.

2. Introduction

Digital forensic investigations, like any other type of investigation, are more of a craft than an exact science. In essence, the investigator must wade through vast amounts of digital evidence (files and fragments thereof) to find the pieces of the puzzle and put them together in a way that will hold up under scrutiny in court. Conceptually, analyzing computer files is no different than going through files of physical documents and analyzing their content. At first glance it may seem like computer systems would make investi-

gations much faster by virtue of being able to perform hundreds of millions of operations per second. The reality is, however, that the sheer (and growing) amount of processing needed to perform digital investigations largely negates the inherent advantages of automated processing. In other words, finding the evidence on a single hard drive may take the same amount of time as finding it in an archive room full of paper documents. A large part of this is "think" time, which is the time needed for the human investigator to analyze the data. While it may be possible for a computer system to accumulate experience and reduce this time, our concern here is the other component, which is the machine processing time needed to execute queries given by the investigator.

In the archive room scenario, it is relatively straightforward to distribute the work to a bigger team in order to speed up the process. Interestingly enough, we have seen relatively little in the way of distributed processing capabilities in current digital forensics tools (other than for password cracking) despite the fact that distributed computing has been around for decades. In our view the time is ripe to take the next logical step and add such capabilities. There are a number of factors that support the need for such effort.

Growth of high-capacity storage devices. It is well-known that storage capacity has grown exponentially for a number of years, while the unit costs have dropped dramatically and consistently. There are few reasons to believe that these trends cannot be extrapolated into the near future. The end result is that the average consumer can easily afford massive amounts of storage that a few years back would have been accessible only to corporate clients. To illustrate, consider that a 200GB hard disk drive can now be purchased for under $200, which means that a terabyte-class storage system can be built at well under $1000. The fast growth of network bandwidth available to end-users means that they will actually *want* to have this kind of capacity to store the large amount of data they can easily access.

Thus investigators should be prepared to handle targets with massive amounts of data. Our experience is that very large data sets are difficult to handle with existing tools and that there is little room for enhancing the sophistication of automated analysis of data. For example, using one of our favorite forensics tools, FTK [1], it took us about 2 hours to open a case for an old 6GB hard disk using the default options. If we extrapolate to a 200GB one, under the conservative assumption that processing time grows linearly as a function of size, it would take some 60 hours. In reality, things are even harder—we had to wait for over 4 days to open a case on a 80GB target. We should explicitly point out that we do not believe that this kind of experience is due to a bad implementation on part of the vendor. Quite the opposite—to us it is a warning sign that even a very mature tool like *FTK*, which tries to optimize performance, cannot do much better on a single workstation. Users are simply raising the bar too high. We are convinced that the system is fundamentally resource constrained and that any optimizations will only have a marginal effect on execution time.

Growth of the I/O speed vs. capacity gap. Another well-known fact is that, courtesy of the "megahertz wars", CPU speeds have also grown exponentially. Unfortunately, I/O transfer speeds have grown only linearly and, therefore, have not kept up the pace with increases in storage capacity. Put another way, if that were not the case, it would take about the same amount of time to image a 20GB drive as it would take to image a 200GB

drive. Anybody who has tried a similar experiment knows that this is absolutely not the case.

Thus digital forensic tools should limit to the bare minimum (preferably once) the number of times they have to read in massive amounts of data to process. The obvious choice is to go over the data once and preprocess it to save on subsequent processing. While this is a valid approach, it has two drawbacks: one is the long initial wait mentioned above and the second is that no tool can possibly anticipate all possible queries and prepare for them. As a simple example, consider the search for a case-specific pattern—for example, an account number discovered after a drive has been processed. The only choice is to perform a "live" search of the entire file system, which could take quite a bit of time.

Increased sophistication of digital forensics analysis. Many of the functions implemented by current tools, such as keyword indexing, cryptographic hashing of files, extraction of human-readable ASCII sequences, and thumbnail generation, necessarily require relatively little CPU processing power. With single boxes pressed to their limits, it is difficult to imagine adding even more time-consuming analysis as an integral part of the investigative process. But there is a distinct need for more sophisticated automated analysis to help investigators target "interesting" evidence, and these will require substantially more CPU power. For example, currently, the only help a typical investigator gets for image analysis is galleries of thumbnail images that must be examined individually. Research in image analysis has produced usable systems that can automatically identify and classify the contents of images. But classifying tens or hundreds of thousands of images on a single digital forensics workstation will be terrifyingly slow. Other examples are the identification and extraction of data hidden using steganography, speech recognition technology to analyze voice messages, and generation of summaries for digital video files using keyframe extraction.

In other words, while any kind of forensics analysis is inherently I/O-constrained because of the need to process vast amounts of data, it can also become CPU-constrained if more sophisticated analytical techniques are used. A distributed solution can help alleviate both the I/O and the CPU constraints. For example, a 64-node Beowulf cluster with 2GB of RAM per node can comfortably cache over 100GB of data in main memory. Using such a system, the cost of the I/O transfer of a large forensic image can be paid once and any subsequent I/O can be performed at a fraction of the cost. Taking the idea a step further, the data cached by each node can be made persistent so that if the system needs to shutdown and restart, each node need only autonomously read in its part of the data from a local disk. At the same time, having multiple CPUs performing the CPU-intensive operations obviously has the potential to dramatically improve execution time.

In summary, the case for going distributed in forensic analysis is not fundamentally different than in any other application domain where high performance is needed. The obvious question is: should we try to use generic distributed frameworks (GDF) and adopt them for our purposes, or are we better off developing a more specialized solution?

Our own survey of available GDFs (in the Related Work section) leads us to believe that a specialized solution is a better choice for a number of reasons:

- A targeted solution can be better optimized for its specific purpose and, hence, achieve better performance with less overhead. Generic frameworks tend to be

heavyweight since they try to be everything to everyone. Usually they provide simple programming abstractions, such as distributed shared memory, that are convenient but may become serious performance bottlenecks.

- GDFs typically require a pre-installed infrastructure on all the machines. We want to minimize that administrative overhead so that regular users can run the system with ease.
- GDFs come with their own specialized programming interfaces that typically require effort and experience to use properly. One of our main goals is to eventually present a component-based solution, where the programming effort of writing and incorporating a new distributed component is practically the same as writing it in the sequential case.

Drawing on our discussion so far, we can formulate the following list of requirements for a DDF toolkit:

- *Scalability*. The system should be able to employ in the order of tens to hundreds of machines over a fast LAN and the addition of more resources should lead to a close to linear improvement in execution time for forensic analysis.
- *Platform-independence*. Ideally, an investigator should be able to employ any unused machine on her local network, regardless of the machine architecture and operating system. While we think that labs will eventually invest in dedicated compute clusters (once they have the software tools to justify them), it should be possible to pool together, for example, the machine resources of a group of investigators working on the same case to speed up the processing of critical evidence.
- *Lightweight*. By lightweight in this context we mean two things:
    - *Efficiency*—the extra work performed to distribute data and queries, as well as to collect results, should constitute a small fraction of the total execution time.
    - *Easy administration*—it should be easy, in fact trivial, for an end user to install and run the distributed system. In particular, the toolkit should make minimal assumptions about the underlying infrastructure, e.g., only common operating system services.
- *Interactivity*. A forensic investigation is an interactive process in which the investigator issues queries against the target datasets and based on the results asks more questions until he builds a sufficiently clear picture of the case. Arguably, on a single machine where the disk and the CPU are 99% busy analyzing the target, there is little that can reasonably be done to improve the interactive experience. However in the distributed case, there is no such excuse and it should be possible to let the time-consuming processing take place in the background (on remote machines) while allowing the investigator to (almost) immediately issue queries and to view partial results as they become available. For example, thumbnail generation might be backgrounded when a new case is started, allowing an investigator to begin viewing the filesystem structure and searching for keywords.

- *Extensibility*. This is a standard software engineering requirement and it mandates that it should be easy to add new or replace existing functions. We explicitly mention it here because it is well known that distributed programs are more difficult to develop. Therefore, we require that writing a new processing function for the DDF toolkit should require the same effort and skills as writing it in the sequential case.
- *Robustness*. In a distributed system there are many components that can fail at arbitrary times. It should fall on the DDF toolkit to detect and recover from such exceptional conditions and ensure the same level of confidence in the end result as in the sequential case.

In the following sections, we present our approach to satisfying the above requirements and present some early experimental results with our prototype implementation.

3. System Architecture

3.1. Overview

Virtually all digital forensics analysis targets files or fragments of files. From a distribution point of view, this file-based processing is very good news since it provides us with a natural unit of distribution. Even better—operations are typically performed on single files and not on groups of files. In other words, there are few dependencies that place constraints on the distribution of data or require complex synchronization. This is in sharp contrast to distributed problems like multiplication of large matrices, where dependencies are numerous and there is no single correct answer to the problem of optimal data distribution.

In summary, our problem can be reduced to the following: we are given a list of files (most likely, part of a captured image) and, based on the file type, a number of operations (indexing, searching, image analysis, password cracking, etc) that have to be performed on the files. To do the job, we have a set of distributed processes capable of caching files in RAM and executing the required operations on these cached copies of the files. The files can remain cached during the entire investigation, allowing complex operations to be quickly performed on the file set.

We have chosen an architecture that is based on one central process, called a *coordinator*, and a number of *worker* processes. The coordinator accepts commands from the user, distributes the tasks among the worker processes, aggregates the results, and displays these results to the user. To implement this system we need a communication protocol that is simple, generic, and extensible. Simplicity is dictated by the need to process messages with minimal overhead. Generality means that we want to assume a minimal set of communication services and, therefore, be able to accommodate the largest set of different computer systems. We also need the ability to easily extend the protocol to accommodate new functions.

The overall conceptual design and a sketch of the associated communication protocol are shown on Figure 1.

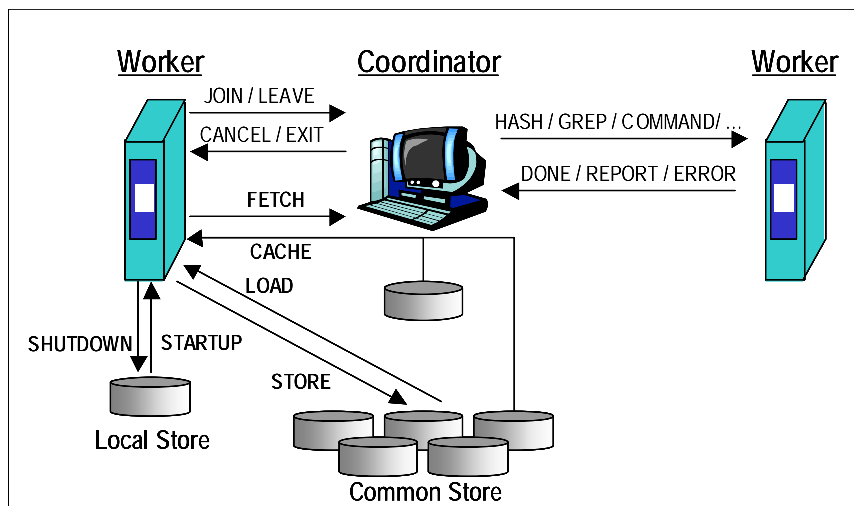

Figure 1 System Architecture and Communication Protocol

3.2. Communication Protocol

To accommodate our basic protocol requirements we have opted for a simple text-based message protocol. We have deliberately stayed away from generic standardized solutions, such as *GENA* [3] and *SOAP* [2]. Our rationale is that these generic frameworks are designed for a wide area network and are overlaid on top of HTTP. In our case, we do not foresee the communication leaving the private LAN of a forensic lab and, hence, there is little point in paying the additional protocol-processing overhead, given that performance is our primary consideration.

All protocol commands consist of a single line of text—the message *header*—followed (optionally) by a block of binary data:

```
<id> <keyword> [<argument_1> … <argument_N>]CR-LF[binary_data]
```

The `<id>` is an integer that uniquely identifies a request. By convention, a response from a worker must carry the same identifier to allow the coordinator to match requests and responses. The keyword identifies the nature of the request/reply followed by the necessary arguments and the optional binary data (e.g., the contents of a file). Whenever binary data is transferred, two of the arguments specify the length of the data and a cryptographic hash, so that correct transmission can be verified.

It is useful to split the message-encoded commands into two types—system commands and processing commands. The system commands constitute the core of the protocol, which we expect to change slowly and be applicable to domains outside of digital forensics. We expect the number of processing commands to grow rapidly as we expand the functionality of the system. In the following description, note that many of the request commands include a reply port number—this is to enable asynchronous communications for requests that may take more time and to allow a more convenient multi-threaded coordinator implementation where specialized threads listen for specific responses.

3.2.1. System Commands

Initialization and Termination

```
<id> JOIN <name> <cache_size> <IP> <port>
<id> LEAVE <name>
```

`JOIN/LEAVE` are issued by worker processes to manage their participation in the system. `JOIN` is the first message sent by a worker processes to the coordinator to announce its availability. The worker provides a human-readable name and available space for an in-memory cache, along with the IP address and the port number to which subsequent commands should be sent. The coordinator is expected to reply with a `DONE/ERROR` message as appropriate (see below). `LEAVE` notifies the coordinator that the worker is about to terminate—no reply is expected.

```
<id> EXIT
<id> SHUTDOWN <name>
<id> STARTUP <name>
```

`EXIT` and `SHUTDOWN` are issued by the coordinator during termination. `EXIT` indicates that the worker should immediately drop all tasks, free all resources, and terminate. `SHUTDOWN` means that currently running tasks should be terminated but the contents of the in-memory cache should be swapped to disk and saved under the given name. The `STARTUP` command instructs a worker to load the specified saved cache. After `EXIT/SHUTDOWN`, the workers are expected to send a `LEAVE` message before terminating.

Cache Management

```
<id> CACHE <file_name> <file_size> <hash> <reply_port>CR-LF<file_data>
<id> FETCH <file_name> <reply_port>
```

`CACHE` instructs the worker process to cache (in RAM) the binary data following the header under a given filename. The size and the hash code support verification of the transmitted data. The worker process is expected to reply with a `DONE/ERROR` message sent to the given port. `FETCH` requests the specified file from the worker—reply is a `REPORT/ERROR` message.

```
<id> LOAD <files> <reply_port>
<id> STORE <files> <reply_port>
```

`LOAD` is a coordinator-issued message that commands a worker to load into its in-memory cache the named file (presumably from a shared file store). `STORE` instructs the worker to commit the file back to a stable store. Note that the file name may contain wild cards to specify a group of files. A `DONE/ERROR` reply is expected.

```
<id> FREE <files> <reply_port>
```

By default, all cached content is locked into main memory and allocated space is not freed unless there is an explicit `FREE` command. A `DONE/ERROR` reply is expected.

Reply Messages

```
<id> DONE
```

DONE confirms that a particular request has completed. It is typically issued by workers but it is also used by the coordinator in replying to JOIN messages.

```
<id> ERROR <code> <message>
```

ERROR reports any problems encountered during execution using error codes and explanation messages.

```
<id> REPORT <report>
```

REPORT provides partial results for a request, usually on a per file basis. For example, the partial results of applying a HASH processing command on a group of files are the corresponding hash codes for each file and they are sent in REPORT messages. No reply is expected.

```
<id> PROGRESS <processed> <all>
```

The PROGRESS message is issued by workers and provides feedback to the coordinator on the amount of processing already completed compared to the overall size. This feedback is optional and is provided at the specified frequency in the processing command. No reply is expected.

```
<id> CANCEL <req_id>
```

Issued by the coordinator, the CANCEL command instructs a worker to stop the processing of the specified request and free up associated resources. A DONE/ERROR reply is expected.

3.2.2. Processing Commands

Processing commands are issued by the coordinator and instruct a worker process to execute a specific operation on a file or a set of files. The common <progress> parameter has the format PROGRESS:n indicating progress report preferences. The number *n* after the colon indicates that a progress report should be sent after every n^{th} result is produced. Thus, PROGRESS:0 means *no* progress reports, PROGRESS:1 means report after each partial result, PROGRESS:10—report after every 10th result, and so on. The <reply_port> has the same meaning as above—the port to which all results should be sent. In all cases, a REPORT /PROGRESS /DONE/ ERROR reply is expected.

```
<id> CLASSIFY <files> <progress> <reply_port>
```

The CLASSIFY command prepares the worker for subsequent commands that identify targets by file type rather than name. The classification of file types is based on MIME types. Workers are expected to verify the actual format of the files and use that information as a basis for classification rather than relying on hints based on the filenames.

```
<id> HASH <method> <files> <progress> <reply_port>
```

Perform cryptographic hashing of the specified files using method <method>. Currently MD5, SHA1, and CRC32 are supported.

```
<id> GREP <expr> <files> <progress> <reply_port>
```

Search <files> for regular expression <expr>—the same semantics as the UNIX grep command apply.

```
<id> THUMB {<files>|<type>} <thumb_dir> <progress> <reply_port>
```

Generate thumbnail images of the specified files and place them in the specified directory (in memory cache).

```
<id> STEGO {<file>|<type>} <progress> <reply_port>
```

Search files for evidence of steganography.

```
<id> CRACK <file> <key_range> <progress> <reply_port>
```

Attempt a password cracking operation against the specified file using the given key range.

```
<id> EXEC <command> <arguments> <reply_port>
```

This allows executing a generic shell command that takes its input from the standard input and sends its output to the standard output. The worker must act like a shell by creating a separate process, providing the appropriate input and redirecting the output back to the coordinator. We should explicitly note that is intended as an exceptional case, in which a command line tool for which no source is available must be used. No progress reports can be provided because the worker has no idea what the command is doing.

4. Prototype Implementation

The current version of our system is only a skeleton implementation of the presented design. The main purpose of this early prototype is to produce some initial results that justify the full system development and its optimization.

The entire implementation is written entirely in *C* (from scratch) and assumes nothing more than a TCP/IP network service. Both the coordinator and worker processes are multi-threaded for performance reasons although more can be done to further improve concurrency. We have not used any specialized libraries for distributed computing, such as PVM [4] or MPI [5], so our code can currently work on an arbitrary set of Linux machines (running any modern kernel), connected to a LAN. As the code becomes more mature, a Windows port will be undertaken. The user interface is currently a shell-like environment where the user types in commands and those are distributed to worker processes.

We have made two basic assumptions about the LAN environment, which we believe are quite realistic:

- All the processing takes place on a private and trusted local area network. Having a dedicated (physically separate) network should be the standard procedure for any forensics lab in order to guarantee that the results are not contaminated in any way. From our point of view, having a trusted environment allows us to focus entirely on performance and eliminates the overhead associated with security and authentication mechanisms.
- The LAN works at speeds of at least 1 Gbps. Gigabit Ethernet technology is nowadays well established and practically all new machines support it by default. Given that the network is the performance bottleneck, anything slower would not justify distributed processing. This is confirmed by the performance measurements presented in the next section.

5. Experience

To investigate the performance of our prototype, we conducted a series of simple experiments. They were designed to validate our basic hypothesis that having a DDF toolkit would improve performance in three different ways:

- Reduce hard disk I/O by reading all the data once and caching it in main memory;
- Reduce the initial loading time through improved utilization of high-performance storage systems;
- Reduce processing time for CPU-intensive tasks by utilizing multiple machines.

Overall, we expected our system to show significant improvement in response time from the point of view of the investigator.

5.1. Experimental Setup

As a baseline, we performed single machine experiments with *FTK* v1.43a on a dedicated high-end *Windows XP* machine. This machine contains a 3 GHz *Intel Pentium 4* processor with 2GB of RAM and two 15,000rpm SCSI hard drives.

Our prototype was running on 8 of the 72 nodes in the *Gumbo-72* Beowulf cluster in the Department of Computer Science at the University of New Orleans. Each node has a 2.4 GHz *Intel Pentium 4* processor, 1GB of RAM and a gigabit network interface connected to a switch with a 24 Gb backplane. The cluster is connected to a file server, which is a dual-CPU 1.4GHz *Intel Xeon* with 2GB of RAM and 504 GB of RAID-5 storage (Figure 2). We used the cluster solely for the convenience of its private network segment, which isolates our experiments from all other traffic on the LAN.

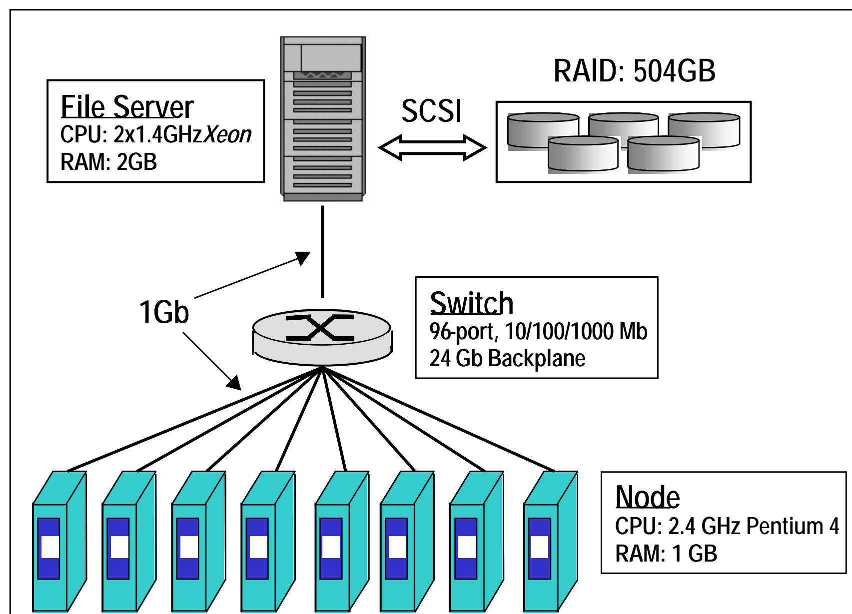

Figure 2 System Setup for Distributed Experiments

5.2. Initial Results

All tests were conducted against a 6GB forensic image, captured using *dd* (via a Linux boot disk). The target machine, a Dell Optiplex GX1, was pulled from our computer literacy laboratory. The machine contained a 6448.6MB Western Digital IDE hard drive with a single partition, formatted as NTFS and contained about 110,000 files in about 7,800 directories. The machine was running Windows 2000 before we pulled the plug. To exclude a number of system files that would not normally be searched, we limited the size of target files to less than 20MB in size and made sure both *FTK* and our system worked on the same set of files.

The original impetus for our work came from the observation that current tools offer two extreme choices in performing an investigation. The first one, offered by FTK, is a long initial data preprocessing, followed by quick answers (for most operations). The alternative, offered by Encase [6] (among many), is little preprocessing but lengthy searches. Our working hypothesis is that this situation is entirely the result of resource constraints and, therefore, a distributed solution provides the best of both worlds, or something close to it.

Initial Processing

On the 3GHz Pentium 4, *FTK* took 1 hour and 38 minutes to perform initial processing of the image with image thumbnail generation OFF and 2 hours with thumbnail generation ON. None of the measurements for our prototype are directly comparable to these numbers, because FTK is performing a lot of initial preprocessing of the (forensic) image and we only have a general idea of the implementation. We simply note that no investigative operations can be performed during this time.

In our current implementation, the initialization step is loading the image from disk into the in-memory cache of all the nodes. This can be performed in two different ways: via a `CACHE`, or via a `LOAD` operation.

The comprehensive `CACHE` operation involves reading the entire image and distributing the files (over the network) in round-robin fashion to the workers. This scenario approximates a setup in which the nodes have no shared file space. Note also that all network I/O is performed sequentially so the presence of a high-capacity switch is inconsequential. In other words, this is our minimal setup and, hence, worst case performance.

For the 6GB image, each of the 8 workers was caching approximately 1/8 of the 6GB image in RAM. As shown on Table 1, it took 576 seconds (9:36min) to complete this task. Since we could not spare 6GB on the local disk, we placed the image on the RAID and accessed it through the file server (Figure 2). Thus, all data had to traverse the network twice to reach the worker processes with the exception of one node that was also running the coordinator. Note also that local I/O with the hard disk is faster than the network. Therefore, the above CACHE number should be considered inflated.

The LOAD operation causes workers to independently load specified files from the file server, rather than receiving file contents from the coordinator over the network. On our test image, the eight nodes completed the operation in 238 seconds (3:58min), which as an improvement of about 142% over the CACHE case.

Recall that one of our contentions was that a distributed system may be able to better utilize a high-performance I/O system, such as our RAID. To quantify that effect, we modified the system to run with a single node, which simply throws away all file data it receives. In this case, the LOAD of the entire image took 319 seconds (5:19min). Thus, the eight nodes were able to collectively load the image 34% faster than a single node could, which clearly indicates better utilization. At this time, we do not know at what point the addition of processing nodes reaches the point of diminishing returns or, perhaps, even backfires by creating too much contention, but the jump from one to eight was clearly beneficial.

Initial Operation	Time (hh:mm:ss)
FTK Add Evidence	1:38:00
CACHE	0:09:36
8-node LOAD	0:03:58
1-node LOAD	0:05:19

Table 1 Initialization Overhead

Finally, we should note that on a slower network, the time to distribute the data make become unacceptable. For example, on a 100 Mb Ethernet, we would expect to take about 10 times as long to CACHE/LOAD the image.

Online Queries

The only intensive operation that was fully implemented in time for inclusion here was online ("live") regular expression searching using the GREP command against files in the image the image.

We performed two test searches—the first one was the to find all occurrences of the simple phrase "Vassil Roussev" across all undeleted files (the machine was used by students to access course home pages so we got a few hits). The second one was using a more complicated regular expression:

```
v[a-z]*i[a-z]*a[a-z]*g[a-z]*r[a-z]*a
```

which matches all strings containing the letters "v i a g r a" (in order), with an arbitrary number of letters between each (we took inspiration from our junk mail folders for this one). We limited the number of hits per file to 10 (standard on FTK) and verified that the end results are the same for both tools. The results are summarized in Table 2.

	Search time: String Expression (mm:ss)	Search time: Regular Expression (mm:ss)
FTK	08:27	41:50
8-node System	00:27	00:28

Table 2 Search Times

FTK took 8:27 minutes to complete the first "live" search and 41:50 minutes to complete the second one. Our prototype took a total of 0:27 minutes for the simple search, that is, it took this much for the slowest of nodes to complete. Individual times varied between 24

and 27 seconds. For the second case, it only took a second longer—0:28 minutes—with similar marginal variations. In relative terms, our times are 18 and 89 times faster, correspondingly and these numbers should speak by themselves of the value of distributed processing to the investigative process. They also show the cumulative effect of caching and parallel processing (speedups are well in excess of the distribution factor of 8).

We were a little puzzled by the fact the *FTK*'s numbers for the searches were so different, whereas ours were almost the same. However, since the search results were the same for both tools, the only reasonable explanation we can offer is implementation differences for regular expression searching. Ours is based on a standard implementation of POSIX regular expression matching with some optimizations to skip over parts of files that are certain not to match.

5.3. Discussion

While we are the first to admit that our experiments do not qualify as comprehensive and that the prototype needs additional work before it can be put to practical use in the investigative process, we believe that the numbers we obtained are quite compelling and will prompt more efforts to take advantage of distributed processing in digital forensics. It already raises several exciting possibilities.

Is it worth indexing the target? If it takes a few minutes to load the data and only seconds to complete regular expression searches, it may well not be worth the effort to build huge word indices of which only a small fraction will ever be used in subsequent queries. Furthermore, although it took half a minute to get the complete answer for a regular expression search, the user is pressed with partial results as they are discovered. Chances are, the human investigator will not be finished reviewing the partials before the final result are in. Furthermore, multiple outstanding searches could be submitted (with different priorities) giving the investigator ample opportunities to do useful work instead of waiting. Note that when *FTK*, for example, loads a file during its preprocessing phase it is forced to perform all preprocessing (indexing, thumbnailing, etc.) before moving on to the next because it is prohibitively expensive to walk the disk more than once. Clearly, it is preferable to run a number of these in the background and allow the investigator to do useful work in the mean time.

What would you do with the extra CPU cycles? There are a number of CPU-hungry processing tasks that now are simply out of the question but may well be within reach on a distributed platform. A few examples:

- Video processing:
    - Keyframe extraction (e.g., generating a gallery of images extracted every 5 second),
    - Image fingerprinting, e.g., discovery of contraband images
- Audio processing:
    - Voice identification of a specific person or more general characteristics, e.g., child voices in the search for child pornography.

- Generation of searchable text for a movie dialog through speech recognition.

All of the above ideas have the potential to tremendously speed up the processing of digital evidence by performing analysis much closer to the one performed by the human investigator only on a much larger scale.

6. Related Work

6.1. Generic Distributed Frameworks

Grid computing has been a buzzword for a number of years and there has been significant work in that area. Grid computing attempts to build a standardized middleware platform that makes distributed computing scalable, secure, and widely available. The problem from our point of view is that grid computing today is anything but easy to use or deploy. Notably, the work on standards is far from complete and, therefore, the universal availability of grid computing will not be forthcoming for at least a few more years. Another problem is that grid computing platforms, such as the most popular *Globus* [7], provide only low-level primitives (e.g., low-level message passing, thread management). Higher-level programming abstractions, such as remote procedure calls (a.k.a. *GridRPC* [8]) are still in the works and not yet standardized or widely accepted. In summary, grid computing is perhaps the ultimate example of trying to cover all possible uses of distributed computing, from multi-year projects like SETI to time-sensitive weather forecasting. Such generality is invariably paid for with overhead and that is incompatible with our goal to make forensic investigations more efficient.

MPI [5] (Message Passing Interface) and *PVM* [4](Parallel Virtual Machine) have been a staple of cluster computing for many years. Each provides a basic set of relatively lightweight mechanisms for inter-process communication and synchronization. They also provide higher-level primitives for dissemination/aggregation of shared data. The issues with using MPI or PVM are that they are not universally available (e.g., it's typically installed on clusters) and requires experience in distributed programming to be used effectively. Proper administration of a cluster running MPI or PVM programs can also be challenging.

6.2. Digital Forensics Toolkits

There are a number of integrated toolkits for processing digital evidence. Most run under Microsoft Windows, though a few target Unix platforms, typically Linux. For example, Encase [6] is a powerful package for forensics which runs under Microsoft Windows. Imaging can be performed using a boot floppy on the target or by removing drives from the target and attaching them to a forensics workstation. Once forensic images are available, investigative operations such as keyword searches and file/partition recovery can be performed. Encase Enterprise Edition runs on a number of machines, but primarily to support remote investigation operations, rather than to distribute the burden of analysis. iLook [9] has a similar set of capabilities and is free to law enforcement agents. The Forensics Toolkit (FTK) [1] by AccessData takes a different approach and is built on a database model. An extensive analysis of drive images is performed immediately after the imaging process, which can be very time-consuming, but subsequent operations, such as

keyword searches, determining which deleted files can be recovered, how many graphics files are present, etc. are much faster. SMART [10] is a Linux-based counterpart to Encase, FTK, and iLook. Open source tools such as The Coroner's Toolkit (TCT) and Autopsy [11] have similar capabilities. With all of these toolkits, an investigator uses one workstation as an analysis platform, with the exception of the password cracking tool Distributed Network Attack (DNA), which is provided with FTK. Thus all of the current tools, while generally very full-featured and powerful, will become increasingly difficult to use as forensics data sets become much larger.

7. Conclusions

In this paper, we argued that the current implementations of digital forensics tools rely on single-machine processing and, therefore, are incapable of performing the analysis of even modest (by today's standards) forensic targets at interactive rates. The restrictions come from fundamental resource limitations and, given current technological trends, the problem will only get worse.

To address the issue, we proposed a new design based on distributed processing and an open protocol. We presented an early prototype implementation and the results from some initial experiments in which we compared our approach to one of the most popular tools currently on the market—*FTK*. Our results show that initial pre-processing can be reduced from hours to a few minutes and that live "spur of the moment" regular expression searches can be performed 18-89 times faster using only 8 machines. Furthermore, our system can stay interactive while presenting partial results and can allow multiple simultaneous prioritized searches. Also, it enables the long processing jobs, such as thumbnailing of images, to be performed in the background and be controlled by the user.

8. Future Work

We are currently completing the basic protocol operations and optimizing the implementation of the coordinator and worker applications. Implementation of additional processing commands, such as registry analysis, file carving, and image thumbnailing are underway. A GUI is also under development. Once the DDF toolkit has functionality similar to that of commercial forensics suites, we will investigate the implementation of "fancier" operations such as audio and video processing and detection of steganography.

9. References

[1] Forensics Toolkit (FTK), http://www.accessdata.com.

[2] Simple Object Access Protocol (V1.1), http://www.w3.org/TR/SOAP/.

[3] GENA: Generic Event Notification Architecture, http://www.upnp.org/download/draft-cohen-gena-client-01.txt

[4] Parallel Virtual Machine (PVM), http://www.csm.ornl.gov/pvm/pvm_home.html.

[5] Message Passing Interface (MPI), http://www-unix.mcs.anl.gov/mpi/.

[6] Encase forensics software, http://www.encase.com.

[7] The Globus Alliance, http://www.globus.org/.

[8] Grid Remote Procedure Call Working Group, https://forge.gridforum.org/projects/gridrpc-wg/.

[9] iLook Investigator forensics software, http://www.ilook-forensics.org/.

[10] SMART forensics software, http://www.asrdata.com/SMART/.

[11] Sleuthkit and Autopsy, http://www.sleuthkit.org.