Demystifying Distributed Systems: Uniform and Noninvasive Instrumentation for Cloud Testbeds

Nicholas D. Pattengale, Sean M. Crosby, Craig D. Ulmer
Sandia National Laboratories
Albuquerque, New Mexico, USA
Email: {ndpatte,smcrosb,cdulmer}@sandia.gov

***Abstract*—Despite ever increasing adoption of distributed systems, there continues to be a dearth of general purpose tools for capturing, analyzing, displaying, and communicating the operational manifestation of complex distributed systems for other than performance purposes. Such tools could be highly useful in (for example) reducing the learning curve for new users of a distributed deployment, enhancing/aiding knowledge transfer between users or administrators of such a deployment, comparing differences between versions of a distributed software package, and comparing disparate competing packages. We present "LAASER-ttag," a prototype for noninvasively capturing the operation of distributed systems in a testbed setting. By leveraging and extending a modern Linux tracing toolkit (LTTng), we effortlessly collect and incorporate into our analyses data from different subsystems such as disk and network. Our prototype imposes no source code modification (or recompilation), and is completely agnostic to the application under study. After presenting the design and rationale behind LAASER-ttag, we show select samples of its output across a number of use cases.**

I. Introduction

The widespread adoption of cloud computing technologies in industry means that many software users now rely on complex, distributed systems to solve their day-to-day problems, whether they know it or not. There are many instances where cloud computing technologies have made it easy for general users to take advantage of distributed systems without having to face the steep learning curve associated with traditional parallel processing architectures. Large-data frameworks such as Hadoop[1] make it easy for users to store and analyze massive amounts of data in a cluster without having to worry about the specifics of how data and computations flow through the system. Open source cluster file systems such as Ceph or GlusterFS make it easy to present a cluster's distributed storage as a single mount point that legacy web servers can utilize for scale-out storage. Infrastructure-as-a-Service (IaaS) cloud software such as OpenStack[2] provide a convenient means of provisioning a cluster's resources out to end users in the form of virtual machines. All of these technologies utilize software frameworks to manage distributed resources and simplify the amount of work end users must do to take advantage of a cluster.

While it is important to make distributed systems more usable, quite often developers want or need to know what exactly their framework is doing under the hood. For example, performance-oriented users need to understand how resources and tasks are scheduled in a framework when refactoring applications to maximize performance. In situations where sensitive data is involved, security researchers need to be able to inspect a framework's behavior to verify that sufficient safeguards are in place and that the frameworks do not provide new opportunities for attackers. Users with high reliability requirements often need to verify that data and computations are in fact distributed by a framework in a way that the system could survive a known number of failures. Finally, application developers often want to inspect a framework's behavior to help discover race conditions and bugs in their own applications.

While there are many tools available today for analyzing different aspects of complex distributed software systems, we have yet to find one that covers all of our needs in a generic manner. The vast majority of distributed analysis tools focus on providing performance information. Ganglia, Nagios, Supermon, OVIS, and Bright Cluster Manager provide an effective means for collecting runtime performance information about applications in a cluster. Unfortunately, these statistics generally do not reveal enough information to infer a detailed understanding of a distributed application's low-level behavior. There are a variety of application-specific instrumentation and monitoring efforts for specific frameworks, including Hadoop's Chukwa and Cassandra's JMX interface, as well as approaches that simply parse a specific framework's log files. These approaches are extremely insightful for understanding applications that utilize the intended framework. However, each has its own learning overhead, and the application-specific nature of these approaches prohibits generality.

Our research is in developing tools and techniques to help rapidly understand how different distributed software frameworks behave. We argue that this work is best accomplished by finding a middle ground between capturing high-level system statistics and application-specific instrumentation: instead, use kernel-level instrumentation to generically capture important, system-level events in the life of the framework that can be analyzed offline to extract meaningful behavior over time.

We have prototyped a solution that largely achieves our goals. By leveraging and extending a modern trace framework – The Linux Tracing Toolkit, next generation (LTTng)[3], we have been able to rapidly assemble a relatively non-invasive high-fidelity platform spanning subsystems such as disk and network. We have used this platform for collecting, analyzing, and displaying the operations and interactions carried out by a variety of distributed software packages.

The basic reasoning behind leveraging a system-level trace framework is that system-level calls (e.g. syscalls) are the well-defined crossings between computer programs and various subsystems of interest, such as disk and network. Thus tracing these points is a natural and parsimonious approach for observing how applications in general treat data.

The version of the prototype covered here focuses almost exclusively on 'tag tracking,' which refers specifically to placing short prespecified strings (the so-called 'tracked tags') into input data and subsequently observing them they traverse a cluster during distributed computations. This tag tracking prototype is part of a larger program (beyond the scope of this publication) toward 'Live All-encompassing Automated Scoring and Event Reconstruction' (LAASER) in computer network testbeds, and thus we refer to the prototype detailed here as LAASER-ttag.

The remainder of this paper is structured as follows: Section II explains general approach as well as our prototype platform in great depth. Section III shows our system in action by showcasing and discussing a variety of tag-tracking analyses. Section IV comments on the implications of our tool as well as future directions.

II. Approach and Methods

A. High Level Design Rationale

At the highest level, our goal for this work is loosely stated. We desire a solution for recording the detailed operation of testbed cloud systems in a form which lends itself straightforwardly to high level human understanding. The solution space for this loosely stated problem is immense. The most natural starting points, perhaps, lie in using already resident system functionality such as (on Linux, at least) `netstat`, `ps`, and the `/proc` filesystem to cobble together snapshots of testbed nodes as the cluster operates. Other natural (but typically not system resident) data sources are network packet capture (e.g. `libpcap`) and filesystem watches (e.g. `inotify`). Pushing these various and disparate datasources through log aggregators such as Splunk has in fact shown promise, but in practice causes heavy system loads due to their polling nature[4].

Avoiding such performance hits is one of the reasons we chose the path leading to LAASER-ttag, which is based upon system level *tracing*. LTTng (the tracing framework we extended) works by leveraging kernel *tracepoints* – prespecified locations in kernel code which call out to functions provided by custom (LTTng provided) loadable kernel modules for dumping structured, packed binary, trace entries to disk. Much more information on system level tracing, including the list of tracepoints used by LTTng, is available via LTTng's documentation[5] or a variety of other sources[6], [7]. Our prototype only uses a subset of these tracepoints (mainly file system and network operations).

Tracing, by design, is inherently event driven; upon events of interest control is transferred to code LTTng provides for inspecting and recording system state at that instant. Other event-driven approaches include instrumented library code for subsystems of interest and custom instrumented application code[8]. As mentioned in the introduction, we desire a 'more uniform and less invasive' solution. Now we are prepared to define these terms more precisely – by uniform we mean that it should be possible to instrument and analyze a wide variety of tools according to a common methodology and technology substrate, and by less invasive we mean that we want to avoid having to modify or recompile the source code of the tools under observation. We have achieved these goals with our prototype. As our system collects data at the operating system level, it is agnostic to the application under study. For the exact same reason, it imposes no source code modification (or recompilations) requirements on the application under study.

The space of possible analyses enabled by trace data is large. For example, it is straightforward to produce a listing of all Hadoop (see Section III-A for a more detailed discussion of Hadoop) components annotated with process ID and role, (e.g. DataNode) along with a record of all of the files that they accessed during a distributed computation. Unfortunately, even for simple computations, this listing can be large and difficult to visualize. There are many strategies worth exploring for managing the complexity (and sheer size) in such general purpose analyses. However, we chose a different path, and focused our attention on a rather simple analysis – putting tracked tag observations into the context of operations that were handling them. This analysis in our experience has a wonderful filtering effect, and renders our datasets manageable in size. That we can present both meaningful and readable timeline graphics (e.g. Figure 2 in Section III-A1) on normal sized paper is evidence of this filtering effect.

As we subsequently learned throughout development and testing of LAASER-ttag, even tag tracking presents many challenges. Foremost is comprehensiveness, for example, in order to read from disk, applications have a choice in the routine they employ. They can use the well-known `read` call, they can use the scatter-gather `readv`, or they can use `mmap` to map the file and read it as if it were in main memory, among others. In order for LAASER-ttag to comprehensively catch *every* traversal of tracked tags through subsystems of interest requires covering all of the independent paths that data can take through a system. Given the limited scope and funding for LAASER-ttag development, we chose essentially to defer to LTTng in selecting a sufficient set of tracepoints, and deal with blind spots as they arise. For example, Section III-B details a known blind spot of LTTng, memory mapped file I/O.

B. Trace Gathering and Pipeline Specifics

To conduct the analyses described in this study, we use a (locally) modified version of the Linux Tracing Toolkit next generation (LTTng)[3]. We made modifications to the 0.19.11 LTTng loadable kernel modules to enable 'tag tracking.' More specifically, we have enhanced the LTTng modules to search for each member of a predetermined fixed-size array of strings (specific strings shown for reference in Table II) upon calls to, e.g., `fs.write`, `fs.read`, `net.socket_sendmsg`, `net.socket_recvmsg`. By strategically seeding input data with instances of strings from the predetermined list, our LTTng modules enable reconstructing high fidelity

synchronized[9] timelines of data traversal through network cards and file systems of an instrumented cluster during distributed computations.

In Table I we outline the versions of the various LTTng components used in our current prototyping cluster. For simplicity, our traced clusters (so far) have mainly been homogeneous populations of Ubuntu 11.04 virtual machines. The version of LTTng that was available when we were building our prototype (LTTng 0.19.x) required kernel patches that are no longer required by newer generation LTTng (2.x) releases. As such we patched Ubuntu's 2.6.38-9 build with LTTng's 0.249 kernel patch set (written against mainline kernel 2.6.38.6).

The machine suite used for the analyses in this paper consists of a collection of virtual machines. All but one of the machines (the cluster) perform the distributed computations, and other than normal system software contain installs of the various distributed software packages as well as LTTng. The additional node is the *instrumentation control and analysis* machine, and contains scripts for batch-controlling tracing on the cluster as well as analyzing the collected traces.

Figure 1 depicts our collection scheme and analysis pipeline, and works as follows:

1) After receiving a command from the *instrumentation control* node, individual cluster nodes begin tracing and storing results to local disk. In our current prototype, the commands are issued via ssh with a command such as `ssh nodeX lttctl -C sampletrace -w /home/ltt/sampletrace`). We prefer that the control box reside on a separate control network, such that commands arrive at (and trace data leaves) cluster nodes via an independent network interface than cluster inter-node network traffic proper. This affords stronger experimental pedigree as the two types of network traffic are not comingled.
2) Once an experiment has concluded (or periodically, for long running experiments), the control box commands each cluster node to stop tracing and proceeds to download the trace results, in their packed binary form, for subsequent synchronization and analysis. Our current prototype simply uses scp for downloading individual traces. In the future we intend to explore LTTng's streaming capability.
3) Individual node traces are globally synchronized by LTTng's lttv (trace viewer) tool. This ability to globally synchronize traces is another notable attribute of, and one of of the major reasons that we chose, LTTng. Their global synchronization method is detailed in [9], and essentially amounts to a clever implementation of [10] using each cluster node's TSC (timestamp counter) register as a local timestamp along with inter-node events having a known ordering (TCP packet transmit/receive) to find a globally consistent linear mapping of each node's TSC values to a global time. For simplicity, we store the globally synchronized event transcript in lttv's textDump format. This text-based format is space inefficient (especially relative to LTTng's binary format), but has been manageable so far. An example line from the globally synchronized trace is as follows:

```
fs.read: 356245.554562472
(/home/.../hdfs1/fs_0), 18788,
18766, /.../bin/java, , 18766, 0x0,
SYSCALL { count = 545, fd = 5, therep
= 4194312, ret = 545 }
```

This event indicates that a filesystem `read` was carried by java (pid 18788, tgid 18766, parentpid 18766) on node 'hdfs1' resulting in a buffer full of 545 characters which were pulled from tgid 18766's fifth file descriptor. The 'therep' field is detailed below.

4) As can be inferred from the example trace line above, accumulating state is necessary to put any individual event into context. For example, to appropriately ascribe the example `fs.read` to a meaningful filename (or socket, or pipe) requires keeping track of file descriptor creation events. The corresponding event in this case is as follows:

```
fs.open: 356244.909153060
(/home/.../hdfs1/fs_0), 18788,
18766, /.../bin/java, , 18766, 0x0,
SYSCALL { fd = 5,
filename = "/home/ltt/gettysburg.txt"}
```

To accumulate this state information across the various cluster machines, we have written a highly modular tool called `LTTngcrunch`. For tag tracking, LTTngcrunch's operation is fairly simplistic. It consumes the globally synchronized textual output as shown above, parses it into an object representation, which is then passed through a pipeline of user-specified modules. For tag tracking, our events pass through modules which perform file descriptor bookkeeping (for regular files and TCP/IP sockets) and process bookkeeping. Bookkeeping refers rather simply to accumulating (python) dictionaries of system state (e.g. file descriptor tables), in order to decorate an event's object representation with more comprehensive information (such as a filename instead of merely a file descriptor number). Thus, in later stages of the pipeline, the data need not be traversed serially in order to retrieve corresponding state. The output of LTTngcrunch, for ease in portability, is in javascript object notation (JSON). With some fields omitted for brevity and readability, the following is an example of an LTTngcrunch output object:

```
{count:545, event_type:fs.read,
tgid:18788, pid:18766
local_timestamp:356245.554562472,
fdext:"/home/ltt/gettysburg.txt",
ret:545, seqid:431501, fd:5,
therep:4194312}
```

For tag tracking, the most important field in these objects is the 'therep' field, which reveals whether tracked tags were seen in this event. The title 'therep' is meant to be interpreted as a predicate (as in predicate logic), i.e. 'is it there?' In this case 'it' refers to tracked tags, and therep is interpreted as a bitmask. In other words, therep being nonzero implies that a tracked tag was seen in the corresponding operation. For example, 'therep=4194308' in an `fs.read` event means that the

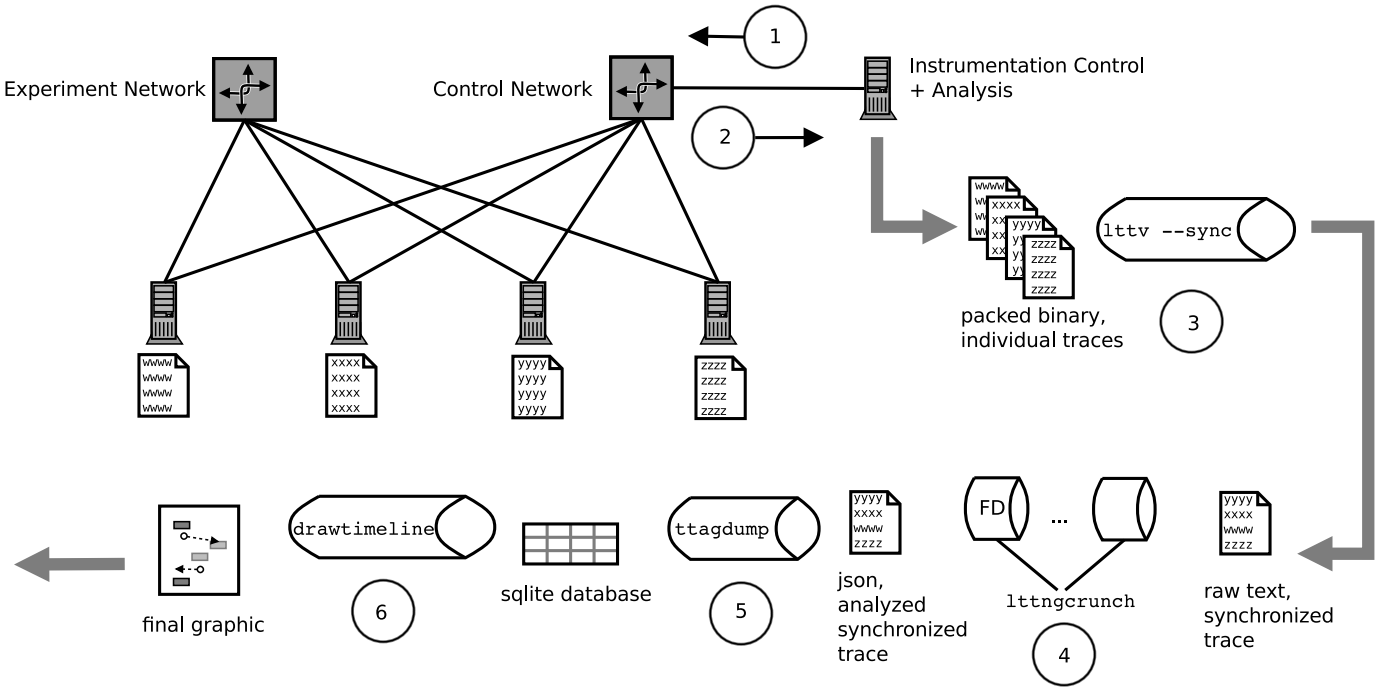

Fig. 1. A sketch of our overall collection scheme and data processing pipeline

tracked tags 'ulr821' and 'fix283' were seen in the buffer being returned by `fs.read` since $4194308_{10} = 00000000100000000000000000000100_2 = 2^{22}_{10} + 2^2_{10}$ and 'fix283' is the 22^{nd} and 'ulr821' the 2^{nd} zero-indexed entries, respectively, in our prespecified tag array (Table II).

5) The final data refinement step in our pipeline is to store all of the (now JSON formatted) events where therep is nonzero in a SQLite database (with a schema detailed in [11]). This is accomplished by a fairly simple python script which consumes JSON objects, and writes data to a SQLite database, appropriately formatted per our schema. At this point we consider our analysis complete, and the SQLite product is amenable to interpretation in any way desired (perhaps, most easily, as a spreadsheet). It is of note that the number of events where therep is nonzero is typically orders of magnitude smaller than the original number of traced events, and as such the resulting SQLite files are typically small.
6) The standard fashion in which we inspect the SQLite files produced by our pipeline is via an in-house developed timeline generator. We will see a number of examples of these timelines in subsequent sections (e.g. Figure 2 in Section III-A1), which depict (global) time on their vertical axis and contain a column for each process (thread group id (TGID), more specifically) that handled a tracked tag. This medium has proved natural for understanding node-to-node interactions, as well as intra-node operations, where tracked tags are involved.

C. Limitations

Our system also suffers from a number of minor limitations:

component name	version	description
lttv	0.12.37-17022011	Visualizer
ltt-control	0.88-09242010	Trace daemon, etc.
ltt-modules	0.19.11 (+ in house mods)	Kernel modules
patches	0.249 (targeting 2.6.38.6)	Kernel patch set

TABLE I
LTTNG TOOL VERSIONS USED IN OUR PROTOTYPE

- The 0.19.X LTTng kernel patches place tracepoints at the locations where traced functions are about to return control to their callers. In general this is not a problem, but in certain cases makes for difficulty in deciphering results. For example, if a socket is sending data asynchronously (i.e. with the socket option O_NONBLOCK set), the net.socket_send event typically occurs before the corresponding net.socket_receive at the other end of the socket. This is contrary to the order a user of LTTng comes to expect, because normally the net.socket_receive will *return* before the net.socket_send (which blocks until receipt is confirmed).
- While the global time synchronization feature of LTTng is certainly distinguishing, it is not without its own limitations. For example, every node must exchange traffic with every other node at least once during each tracing session (albeit only a few packets for each node pair). This all-to-all communication requirement scales quadratically with number of nodes, and may be prohibitive in large clusters.
- Tracing has the potential, especially on highly utilized systems, to produce huge amounts of data. We save approximately one order of magnitude in storage requirements (versus LTTng in its standard configuration) by selectively deactivating tracepoints which are non-

essential to our analyses. This savings has been sufficient to enable all of the experiments we have conducted to date. If larger savings are needed in the future, it will not be prohibitively difficult to modify LTTng to selectively save events produced by, e.g., white listed applications. This is only one of many potential space saving strategies.

D. Scanning Algorithm

A key challenge in developing an effective tagging systems is implementing an efficient system for inspecting data that moves through the instrumentation points. Our needs require that a small number (30) of fixed-length (6) strings be used as a search dictionary, and that tags can start at any position in the stream. Since we have control over the tags used in a system, we can simplify the search task by using non-overlapping tags that remove the need for tracking multiple potential hits at the same time.

We considered multiple strategies for string matching in the streams. While efficient algorithms such as Boyer-Moore and Knuth-Morris-Pratt would be ideal, we constructed the simple but effective approach listed in Algorithm 1. This scanning algorithm was straightforward to implement and met our performance objectives. It is invoked in the following LTTng tracepoints to check for the existence of tracked tags in input/output buffers:

Algorithm 1 Simplistic scan for fixed length prespecified strings in a buffer. While this routine has $O(nmw)$, we reasonably assume m and w as constant. Further, n is also typically small, and thus this strategy is not time-prohibitive.

Input: a character buffer of length n
Input: a tag array of length m, containing tags with fixed width w, e.g. Table II
Output: a bitmask b where bit i being set indicates that tag i exists in the buffer

```
1:  function SCAN-FOR-TTAGS(buffer)
2:      for all positions i from 0 to n − w + 1 do      ▷ O(n)
3:          for all tags t with pos j in tag array do   ▷ O(m)
4:              if t =buffer[i : i + w] then            ▷ O(w)
5:                  b ← b ∨ 2^j          ▷ ∨ denotes bitwise OR
6:              end if
7:          end for
8:      end for
9:  return b
10: end function
```

III. Case Studies

We now present a variety of LAASER-ttag analyses in order to illustrate its utility in understanding cluster operations. The scenarios covered in this paper are intentionally short, simplistic, and involve only a few processes across a small number of cluster nodes. That they are short is in an effort to save space, but not at the expense of showcasing a meaningful set of operations.

0 yqz958	1 wbu365	2 ulr821	3 jrs036	4 rkf168	5 jxm820
6 ori894	7 yko871	8 ftu070	9 srf502	10 grl148	11 lyr428
12 dpp223	13 roc357	14 ddj250	15 vio154	16 pzz933	17 bjk412
18 wqv139	19 yvl354	20 wfb150	21 bwj563	22 fix283	23 ogd030
24 oie495	25 ggh069	26 wyc894	27 hpn120	28 riu782	29 bbt515

TABLE II
Current prototype's tracked tags array. For example, if *fix283* is found in a buffer, Algorithm 1 will return a bitmask with the $22nd$ 0-indexed bit set. In the notation of Algorithm 1, $m = 30$ and $w = 6$.

A. Hadoop

The case studies in this section were conducted with Apache Hadoop (http://hadoop.apache.org/). Hadoop is a framework for distributed processing of large data sets. Hadoop has two primary components: MapReduce, which is patterned after Google's MapReduce, and the Hadoop Distributed File System (HDFS)[1] which is patterned after Google's GFS. HDFS is a replicated block store and functions as the storage layer of the Hadoop framework. Hadoop MapReduce runs a TaskTracker process on each node for processing data. It's JobTracker process manages the processing tasks. HDFS's NameNode server process runs on a single node and stores the metadata (file names, permissions, replication factors, etc.) for all the files in the file system. The SecondaryNameNode process assists the NameNode in compacting the metadata stored on disk. File content is divided up into blocks and stored by the DataNode processes running on all the nodes in the cluster.

The case studies in this section were conducted in a cluster where DataNode and TaskTracker processes were running on every node and the NameNode, JobTracker, and SecondaryNameNode processes were running on one of those nodes. The HDFS replication factor was set to two, which means that each HDFS block will be replicated to at least two distinct nodes.

1) Simple Tag in Data File: In our first case study, we traced movement of file content in HDFS. HDFS writes are performed by a client process that reads files from the local disk and then sends the file metadata to the NameNode. The client's communication with the NameNode is over Java Remote Procedural Calls (RPCs). The NameNode returns to the client a list of DataNodes to write each file block. The client then sends each block and its DataNode list to the first DataNode in the list using a custom binary protocol. The DataNodes replicate the blocks they receive to the next DataNode in the provided list.

Figure 2 shows the output of LAASER-ttag after executing

```
bin/hadoop dfs -copyFromLocal
  ~/Documents/
  /user/ltt/gutenberg
```

from one of our cluster nodes. This command copies all of the files used in this Hadoop tutorial[12], as well as two additional files seeded with tracked tags (the Gettysburg Address and the U.S. Constitution) into an HDFS directory

with path `/user/ltt/gutenberg`. The picture shows that the cluster is configured to replicate blocks to two nodes, and that the replication is done as a relay (as opposed to a broadcast). Second, it gives hints as to HDFS' block structure and naming scheme, as the block file names and underlying directory structure imply some sort of hashing scheme.

2) Tag as Username: By placing tracked tags in locations other than input data, we are able to learn about other aspects of the system under study. In this case, we set the username of our HDFS transaction to be a tracked tag. By invoking the same command as in the previous subsection (Section III-A1), we observe that in the default HDFS configuration, the username rarely manifests in the network or on disk. Specifically in this case, the username only traversed the network as the HDFS client initiated conversation with the NameNode, and only manifested on disk as the NameNode updated its filesystem journal (**`dfs-root`**`/name/current/edits`). The SecondaryNameNode did not compact the metadata on disk during the trace.

In contrast, the username manifests much more copiously in a fairly simple MapReduce job. When a MapReduce job is submitted by the Hadoop client, the JobTracker instructs the TaskTracker which part of the job to process. The job's code and configuration settings are distributed to the TaskTrackers via HDFS. By cursory inspection, the username manifests in a dozen TaskTracker configuration and job specification files, another dozen logs, and in blocks across all nodes of the distributed file system. We omit detailed trace graphics for this case due to space constraints.

3) Tag in Code: As a final case for HDFS, we seeded executable code (in the form of a Java jar file) with a tracked tag in order to examine HDFS from yet another angle. Figure 3 shows the analysis produced after running MapReduce job from the previously mentioned Hadoop tutorial [12]. The figure shows the distribution of the jar file from the client to the DataNodes and then from DataNodes to the TaskTrackers. This analysis, of all we have conducted to date, gives us hope that it will be possible to use LAASER-ttag data to infer highly abstracted characterizations of distributed systems at the distributed-protocol level.

B. Tag in MapReduce job input (with the details of Memory Mapped File I/O shortcomings)

As discussed earlier (Section II-A), while LAASER-ttag makes a best effort at comprehensively tracking data as it traverses through our clusters, the current prototype undoubtedly has blind spots. As an instructive example, we were suspicious of one of our early analyses (of a MapReduce job, analysis figure omitted due to space constraints) in that there were network send/recv events of tracked tags without preceding filesystem reads. To wit, how can Hadoop send HDFS blocks out to the network without reading them first from disk? Digging lightly into the HDFS source revealed that the BlockSender class (`BlockSender.java` in `hdfs/server/datanode/`) reads HDFS blocks from disk via Java's "new" I/O (`java.nio.*`) FileChannel class, which is Java's abstraction for memory-mapped file I/O. Since our current prototype ignores memory-mapped file I/O (mainly because LTTng 0.x does not provide a natural tracepoint for `mmap` and friends), we essentially missed the disk reads. Rerunning the same command on an older version of Hadoop (namely 0.17.2, the last version that did not use Java FileChannel) yielded an analysis in which our system correctly observes the disk reads before the network sends.

C. GlusterFS

GlusterFS[13] is a popular open source distributed file system (DFS) that enables users to aggregate a cluster's distributed storage resources into one or more mountable volumes. The software is comprised of a daemon for servers that manages local storage resources, and a FUSE software interface for clients that enables end applications to transparently connect to the data maintained by the system. Unlike other DFSs, GlusterFS does not utilize a metadata server to handle data placement within the cluster. Instead, it computes a hash of a file's filename to determine which servers in the cluster are responsible for maintaining the desired data. GlusterFS can be configured to stripe (discouraged) and/or replicate (encouraged) data across multiple storage nodes in the system. If striping is not utilized, a file is stored in its entirety on a single storage node, as well as every other replicate storage node. Many users favor GlusterFS because data files are generally stored as-is in the underlying storage devices, and could easily be recovered if the GlusterFS system were to suffer a catastrophic failure.

Figure 4 shows a simple scenario where one cluster node (node2) performs a `sha1sum` of a file that resides in the Gluster file system and turns out not to reside on the local disk, necessitating a network transfer. As was the case with HDFS above, this picture can be quite informative to those not deeply familiar with Gluster's inner workings. First, it is straightforward to infer from the reads and writes to/from `/dev/fuse` that this Gluster deployment is via a FUSE filesystem. Second, Gluster appears to use a client-server model for retrieving data from remote nodes, as 'glusterfs' on node2 reaches out to a daemon 'glusterfsd' on node1 which in turns invokes its local 'glusterfs' on node1.

D. Ceph

Ceph[14] is a relatively new DFS that has received a fair amount of recent attention due to the inclusion of its client interface code in the Linux kernel. Ceph was designed to be a distributed object store upon which additional storage services such as a DFS could be layered. This object store decomposes objects into smaller (8MB) blocks that are internally replicated on multiple storage nodes within the cluster.

The analysis is shown in Fig. 5 presents a scenario nearly identical to that shown in GlusterFS (Sect. III-C and Fig. 4) whereby an invocation of `sha1sum` of a file necessitates network transfer of the desired content via the underlying distributed file system. In contrast with GlusterFS, the Ceph client is part of the Linux kernel, as is evident by the kworker thread contacting the ceph-osd for the file in question. Additionally, the analysis makes evident that Ceph is journaled, and writes

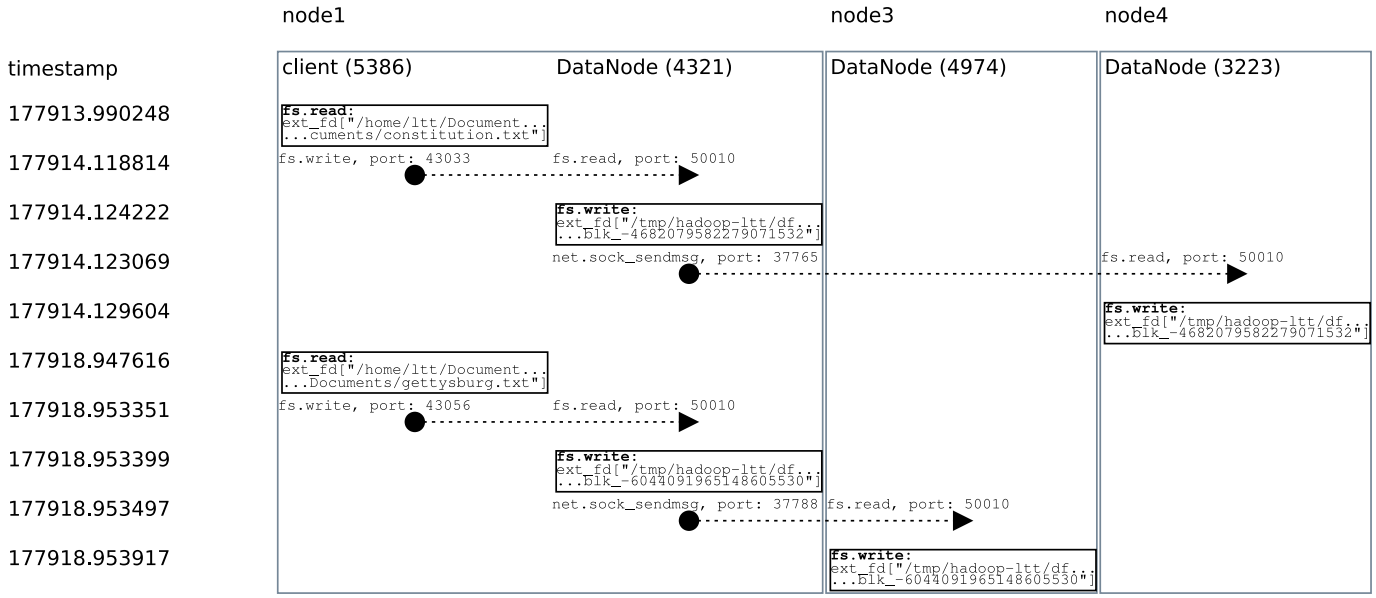

Fig. 2. A Hadoop client puts two files into an HDFS by submitting them as blocks to its local DataNode. In turn, the local DataNode replicates each block to a second DataNode somewhere else in the cluster.

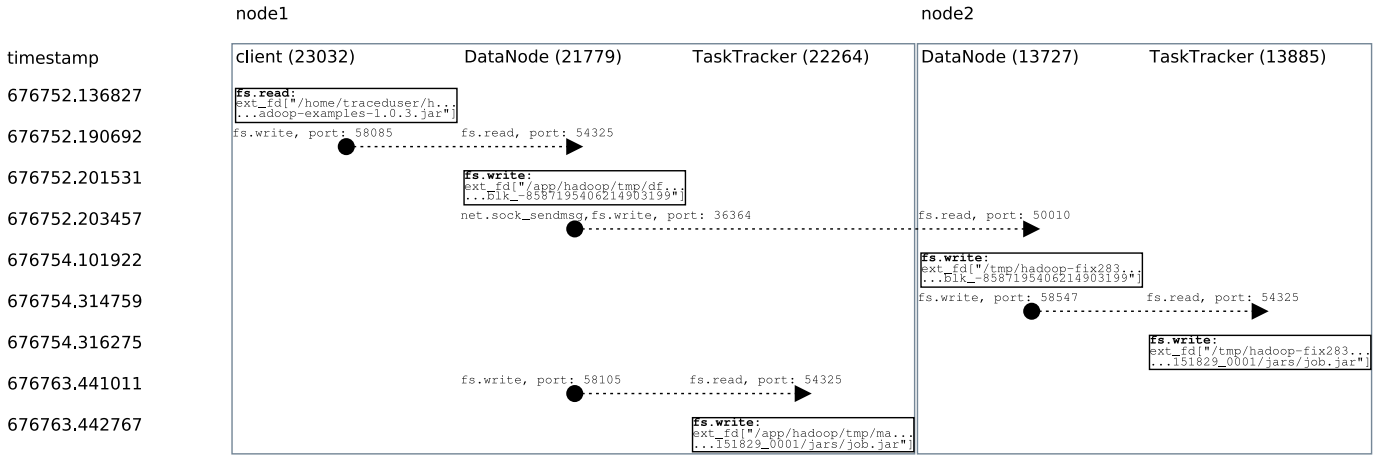

Fig. 3. Upon invocation of a MapReduce job (by the client on node1), Hadoop distributes copies of the tagged Jar file to TaskTrackers across the cluster.

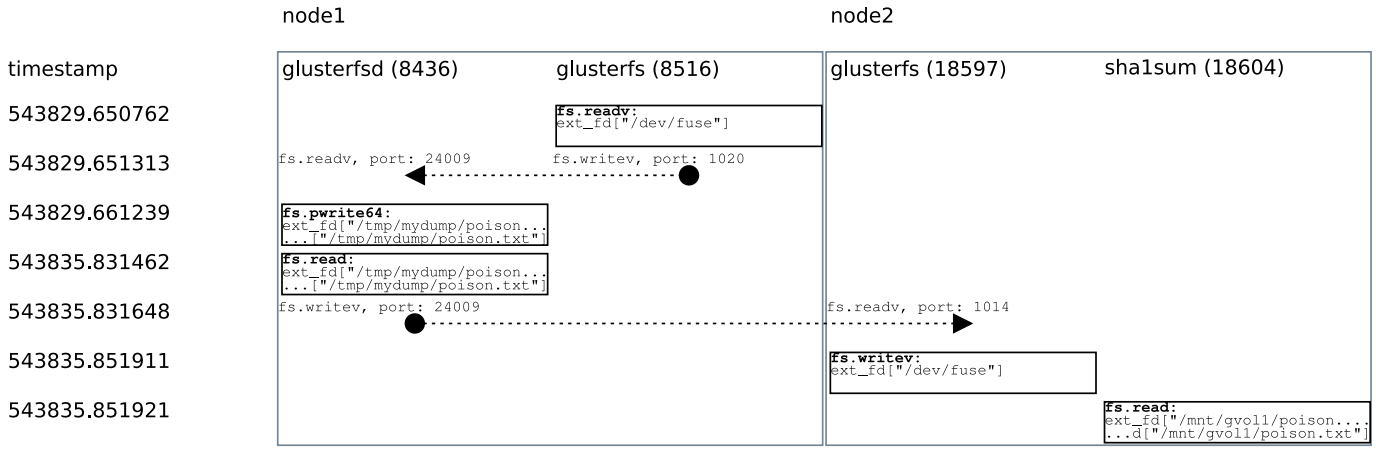

Fig. 4. Taking the hash (sha1sum) of a file resident in a Gluster filesystem in this case triggers the underlying DFS to retrieve the file via the network.

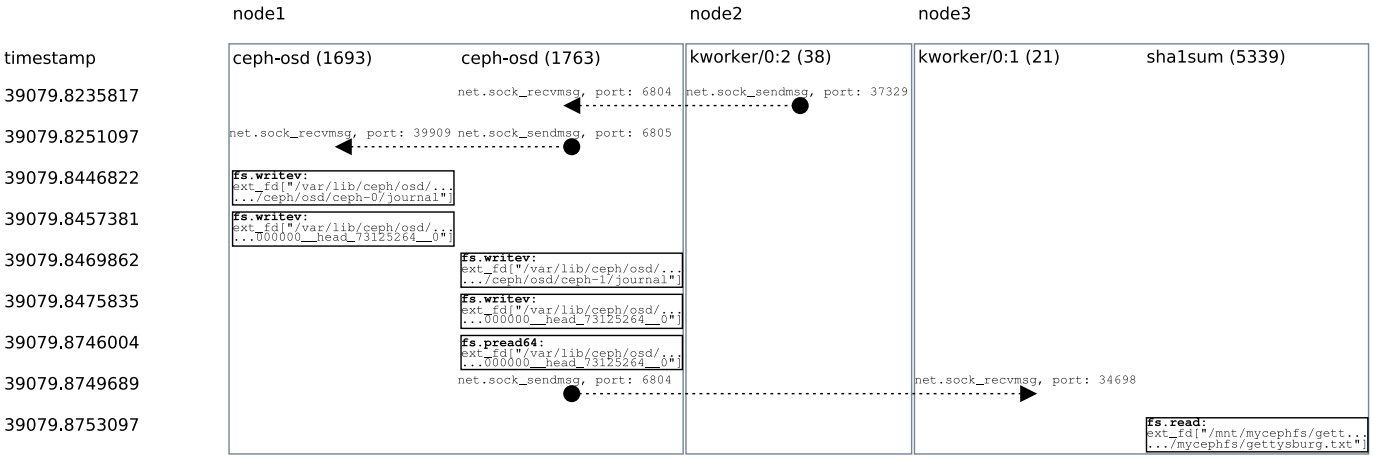

Fig. 5. Taking the hash (sha1sum) of a file resident in a Ceph filesystem in this case triggers the underlying DFS to retrieve the file via the network.

to the journal are done lazily (upon request, rather than upon insertion into the filesystem).

IV. Discussion and Conclusions

Our research is in developing techniques to help rapidly understand how different distributed software frameworks behave. We have argued that this work is best accomplished by finding a middle ground between capturing high-level system statistics and application-specific instrumentation: instead, use kernel-level instrumentation to generically capture system-level events in the life of the framework that can be analyzed offline to extract meaningful behavior over time.

We have presented a prototype system, LAASER-ttag, for noninvasively and uniformly making sense of data flows throughout distributed system testbeds. As our system collects data at the operating system level, it is agnostic to the application under study, and imposes no source code modification (or recompilation) requirements on the application under study. We have also presented a number of simple case studies carried out by LAASER-ttag, thereby illustrating its utility.

In addition to diving deeper into any particular distributed software package (natural follow on steps), there are many other potential uses for LAASER-ttag.We plan to explore the space of different analyses. That is, while tag tracking is no doubt useful, we are interested in characterizing other aspects of distributed systems operational manifestation. For example, it would be worthwhile to explore how the overwhelming number of operations being carried out by a distributed system every second could be portrayed to human in a summarized but insightful fashion.

Another direction is in exploring LAASER-ttag as an independent data source for finding indicators of anomalous system activity. We envision pushing LAASER-ttag data through log aggregators along with more standard system security logs in order to statistically associate known violations (i.e. the ground truth provided by LAASER-ttag) with non-obvious but discriminatory side-effects (i.e. non-obvious indicators that will reliably manifest in fielded systems).

Acknowledgements

This paper has report number SAND2012-9604C, and was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10, 2010, pp. 1–10.

[2] "OpenStack," http://openstack.org.

[3] M. Desnoyers and M. R. Dagenais, "LTTng, Filling the Gap Between Kernel Instrumentation and a Widely Usable Kernel Tracer," in *Linux Foundation Collaboration Summit 2009 (LFCS 2009)*, Apr. 2009.

[4] V. E. Urias and M. Merza, "Splunking the cloud," Presented at Splunk User's Conference, 2011.

[5] "LTTng," http://lttng.org.

[6] "TracingWiki," http://lttng.org/tracingwiki/index.php/Main_Page.

[7] B. Jacob, P. Larson, B. H. Leitao, and S. A. M. M. da Silva, "Systemtap: Instrumenting the linux kernel for analyzing performance and functional problems," IBM, Tech. Rep. REDP-4469-00, 2009.

[8] A. Konwinski and M. Zaharia, "Finding the elephant in the data center: Tracing hadoop," University of California, Berkeley, Tech. Rep. CS294, 2008.

[9] B. Poirier, R. Roy, and M. Dagenais, "Accurate offline synchronization of distributed traces using kernel-level events," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 3, pp. 75–87, Aug. 2010.

[10] A. Duda, G. Harrus, Y. Haddad, and G. Bernard, "Estimating global time in distributed systems." in *ICDCS*, pp. 299–306.

[11] S. Crosby, N. Pattengale, C. Ulmer, and V. Urias, "Nephelae LDRD Project Summary," Sandia National Laboratories, Tech. Rep. SAND2012-8807, 2012.

[12] M. G. Noll, "Running Hadoop on Ubuntu Linux (Multi-node Cluster)," http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/.

[13] "GlusterFS," http://www.gluster.org.

[14] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06, 2006, pp. 22–22.