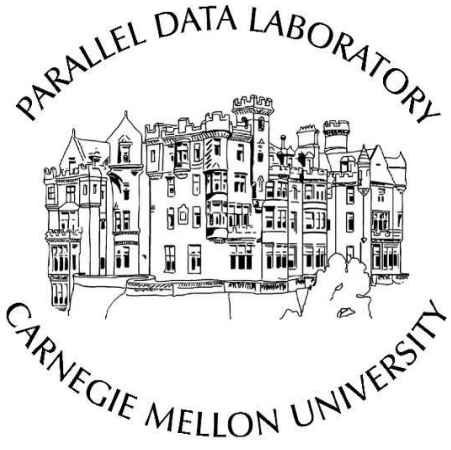

SALSA: Analyzing Logs as StAte Machines

Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi and Priya Narasimhan

CMU-PDL-08-111
September 2008

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

SALSA examines system logs to derive state-machine views of the sytem's execution, along with control-flow, data-flow models and related statistics. Exploiting SALSA's derived views and statistics, we can effectively construct higher-level useful analyses. We demonstrate SALSA's approach by analyzing system logs generated in a Hadoop cluster, and then illustrate SALSA's value by developing visualization and failure-diagnosis techniques, for three different Hadoop workloads, based on our derived state-machine views and statistics.

Acknowledgements: This work is partially supported by the NSF CAREER Award CCR-0238381, NSF Award CCF-0621508, and the Army Research Office grant number DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to the Center for Computer and Communications Security at Carnegie Mellon University.

Keywords: Log Analysis, Hadoop, Failure Diagnosis

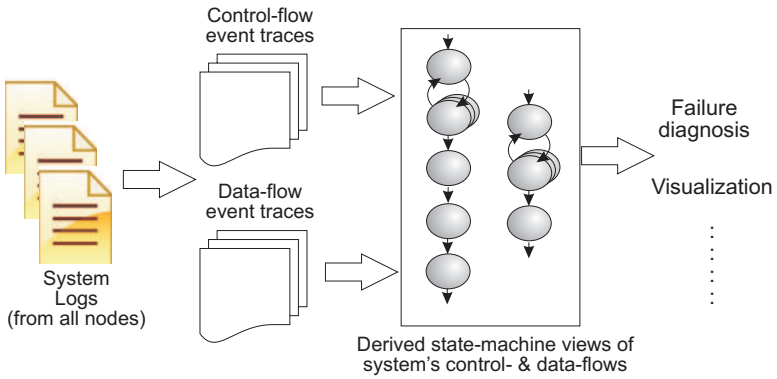

Figure 1: SALSA's approach.

1 Introduction

Most software systems collect logs of programmer-generated messages for various uses, such as troubleshooting, tracking user requests (e.g. HTTP access logs), etc. These logs typically contain relatively unstructured, free-form text messages, making them relatively more difficult to analyze than numerical system-data (e.g., CPU usage). However, logs can often contain more semantically rich information than numerical system/resource utilization statistics, particularly since the log messages often capture the intent of the developer or programmer of the system to record events of interest.

SALSA, our approach to automated system-log analysis, involves examining the logs to *trace control-flow and data-flow execution in a distributed system*, and to *derive state-machine-like views of the system's execution on each node*. Figure 1 represents the core of SALSA's approach. Because log data is only as accurate as the developer/programmer who implemented the logging points in the system under inspection, we can only infer the state-machines that execute within the target system. We cannot (from the logs), and do not, attempt to verify whether our derived state-machines faithfully capture the actual ones executing within the system. Instead, we leverage these derived state-machines to support different kinds of useful analyses: to understand/visualize the system's execution better, to discover data-flows in the system, to pinpoint performance bottlenecks, to discover bugs, and to localize performance problems and failures.

To the best of our knowledge, SALSA is the first log-analysis technique that aims to derive state-machine views from unstructured text-based logs, to support visualization, failure-diagnosis and other uses. In this paper, we apply SALSA's approach to the logs generated by Hadoop [7], the open-source implementation of Map/Reduce [4]. Concretely, our contributions are: (i) a log-analysis approach that extracts state-machine views of a distributed system's execution, with both control-flow and data-flow, (ii) a usage scenario where SALSA is beneficial in preliminary failure diagnosis for Hadoop, and (iii) a second usage scenario where SALSA enables the visualization of Hadoop's distributed behavior.

2 SALSA's Approach

SALSA aims to analyze the target system's logs to see whether we can derive the control-flow on each node, the data-flow across nodes, and the state-machine execution of the system on each node. In this process of parsing the logs, SALSA also extracts key statistics (state durations, inter-arrival times of events, etc.) of interest. To demonstrate SALSA's value, we exploit the SALSA-derived state-machine views and their related statistics for visualization and failure diagnosis. SALSA does not require any modification of the hosted applications, middleware or operating system.

To describe SALSA's high-level operation, consider a distributed system with many producers, $P1, P2, ...$, and many consumers, $C1, C2, ...$ Many producers and consumers can be running on any host at any point in time. Consider one execution trace of two tasks, $P1$ and $C1$ on a host X (and task $P2$ on host Y) as captured

by a sequence of time-stamped log entries at host X:

```
[t1] Begin Task P1
[t2] Begin Task C1
[t3] Task P1 does some work
[t4] Task C1 waits for data from P1 and P2
[t5] Task P1 produces data
[t6] Task C1 consumes data from P1 on host X
[t7] Task P1 ends
[t8] Task C1 consumes data from P2 on host Y
[t9] Task C1 ends
   :
```

From the log, it is clear that the executions (control-flows) of $P1$ and $C1$ interleave on host X. It is also clear that the log captures a data-flow for $C1$ with $P1$ and $P2$.

SALSA interprets this log of events/activities as a sequence of *states*. For example, SALSA considers the period $[t1, t6]$ to represent the duration of state $P1$ (where a state has well-defined entry and exit points corresponding to the start and the end, respectively, of task $P1$). Other states that can be derived from this log include the state $C1$, the data-consume state for $C1$ (basically, the period during which $C1$ is consuming data from its producers, $P1$ and $P2$), etc. Based on these derived state-machines (in this case, one for $P1$ and another for $C1$), SALSA can derive interesting statistics, such as the durations of states.

SALSA can then compare these statistics and the sequences of states across hosts in the system. In addition, SALSA can extract data-flow models, e.g., the fact that $P1$ depends on data from its local host, X, as well as a remote host, Y. The data-flow model can be useful to visualize and examine any data-flow bottlenecks or dependencies that can cause failures to escalate across hosts.

Discussion. On the stronger side, SALSA provides a picture of the system's progress (control-flow and data-flow) on various hosts, and models the sequence of logged events as state machines. SALSA also extracts statistics that can be compared across different hosts, and can also be compared historically on the same host, for useful capabilities such as failure diagnosis.

On the weaker side, SALSA's views might not be accurate, since the control-flow, data-flow and state-machines are derived/inferred purely from the log data. SALSA has no way of determining what the system is actually doing. Thus, SALSA's inferences are undoubtedly affected by the quality of the log data. Another constraint of using the log data, as-is, is the fact that SALSA might need to be upgraded for every new version of the target system, if system's log messages or its logging points are modified by the developers.

Regardless of all these drawbacks, the fundamental research question that we sought to ask was: *how can we leverage log data, as-is, to construct state-machine views, control-flow and data-flow models of the target system's execution, in order to support analyses for problem diagnosis, visualization, etc.?*

Non-Goals. We do not seek to validate or improve the accuracy or the completeness of the logs, nor to validate our derived state-machines against the actual ones of the target system. Rather, our focus has been on the analyses that we can perform on the logs in their existing form.

It is not our goal, either, to demonstrate complete use cases for SALSA. For example, while we demonstrate one application of SALSA for failure diagnosis, we do not claim that this failure-diagnosis technique is complete or perfect. It is merely illustrative of the sorts of useful analyses that SALSA can support.

Finally, while we can support an online version of SALSA that would analyze log entries generated as the system executes, the goal of this paper is not to describe such an online log-analysis technique or its runtime overheads. In this paper, we use SALSA in an offline manner, to analyze logs incrementally and to generate the derived state-machines accordingly.

Assumptions. We assume that the logs faithfully capture events and their causality in the system's execution. For instance, if the log declares that event X happened before event Y, we assume that is indeed the case, as the system executes. We assume that the logs record each event's timestamp with integrity, and as close in time (as possible) to when the event actually occurred in the sequence of the system's execution. Again, we

recognize that, in practice, the preemption of the system's execution might cause a delay in the occurrence of an event X and the corresponding log message (and timestamp generation) for entry into the log. We do not expect the occurrence of an event and the recording of its timestamp/log-entry to be atomic. We do not expect the clocks to be synchronized across the different hosts.

3 Related Work

Event-based analysis. Many studies of system logs treat them as sources of failure events. Log analysis of system errors typically involves classifying log messages based on the preset severity level of the reported error, and on tokens and their positions in the text of the message [14] [13]. More sophisticated analysis has included the study of the statistical properties of reported failure events to localize and predict faults [15] [13] [11] and mining patterns from multiple log events [10].

Our treatment of system logs differs from such techniques that treat logs as purely a source of events: we impose additional semantics on the log events of interest, to identify durations in which the system is performing a specific activity. This provides context of the temporal state of the system that a purely event-based treatment of logs would miss, and this context alludes to the operational context suggested in [14], albeit at the level of the control-flow context of the application rather than at a managerial level. Also, since our approach takes the log semantics into consideration, we are able to produce views of the data that can be intuitively understood by users. However, we note that our log analysis is amenable only to logs that capture both normal system activity and errors.

Request tracing. Our view of system logs as providing a control-flow perspective of system execution, when coupled with log messages with unique identifier of the relevant request or processing task, allows us to extract request-flow views of the system. Much work has been done to extract request-flow views of systems, and these request flow views have then been used to diagnose and debug performance problems in distributed systems [2] [1]. However, [2] used instrumentation in the application and middleware to track requests and explicitly monitor the states that the system goes through, while [1] extracted causal flows from messages in a distributed system using J2EE instrumentation developed by [3]. Our work differs from these request-flow tracing techniques in that we can causally extract request flows of the system without added instrumentation given system logs, as described in § 7.

Log-analysis tools. Splunk [12] treats logs as searchable text indexes, and generates visualizations of the log; Splunk treats logs similarly to other log-analysis techniques, considering each log entry as an event. There exist commercial open-source [6] tools for visualizing the data in logs based on standardized logging mechanisms, such as `log4j` [8]. To the best of our knowledge, none of these tools derive the control-flow, data-flow and state-machine views that SALSA does.

4 Hadoop's Architecture

Hadoop [7] is an open-source implementation of Google's Map/Reduce [4] framework that enables distributed, data-intensive, parallel applications by decomposing a massive job into smaller tasks and a massive data-set into smaller partitions, such that each task processes a different partition in parallel. The main abstractions are (i) `Map` tasks that process the partitions of the data-set using key/value pairs to generate a set of intermediate results, and (ii) `Reduce` tasks that merge all intermediate values associated with the same intermediate key. Hadoop uses the Hadoop Distributed File System (HDFS), an implementation of the Google Filesystem [16], to share data amongst the distributed tasks in the system. HDFS splits and stores files as fixed-size blocks (except for the last block).

Hadoop uses a master-slave architecture to implement the Map/Reduce programming paradigm. As shown in Figure 2, there exists a unique master host and multiple slave hosts, typically configured as follows. The master host runs two daemons: (1) the JobTracker, which schedules and manages all of the tasks belonging to a running job; and (2) the NameNode, which manages the HDFS namespace by providing a filename to block mapping, and regulates access to files by clients (which are typically the executing tasks).

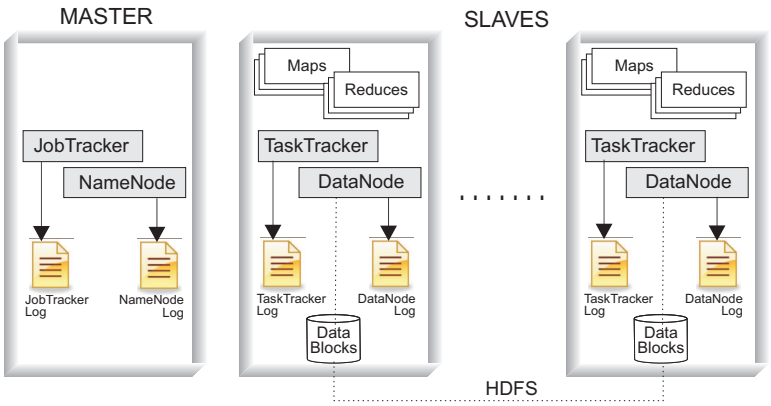

Figure 2: Architecture of Hadoop, showing the locations of the system logs of interest to us.

Hadoop source-code

```
LOG.info("LaunchTaskAction: " + t.getTaskId());
LOG.info(reduceId + " Copying " + loc.getMapTaskId()
    + " output from " + loc.getHost() + ".");
```

⇓ TaskTracker log

```
2008-08-23 17:12:32,466 INFO
    org.apache.hadoop.mapred.TaskTracker:
    LaunchTaskAction: task_0001_m_000003_0
2008-08-23 17:13:22,450 INFO
    org.apache.hadoop.mapred.TaskRunner:
    task_0001_r_000002_0 Copying
    task_0001_m_000001_0 output from fp30.pdl.cmu.local
```

Figure 3: `log4j`-generated TaskTracker log entries. Dependencies on task execution on local and remote hosts are captured by the TaskTracker log.

Each slave host runs two daemons: (1) the TaskTracker, which launches tasks on its host, based on instructions from the JobTracker; the TaskTracker also keeps track of the progress of each task on its host; and (2) the DataNode, which serves data blocks (that are stored on its local disk) to HDFS clients.

4.1 Logging Framework

Hadoop uses the Java-based `log4j` logging utility to capture logs of Hadoop's execution on every host. `log4j` is a commonly used mechanism that allows developers to generate log entries by inserting statements into the code at various points of execution. By default, Hadoop's `log4j` configuration generates a separate log for each of the daemons– the JobTracker, NameNode, TaskTracker and DataNode–each log being stored on the local file-system of the executing daemon (typically, 2 logs on each slave host and 2 logs on the master host).

Typically, logs (such as syslogs) record events in the system, as well as error messages and exceptions. Hadoop's logging framework is somewhat different since it also checkpoints execution because it captures the execution status (e.g., what percentage of a `Map` or a `Reduce` has been completed so far) of all Hadoop jobs and tasks on every host.

Hadoop's default `log4j` configuration generates time-stamped log entries with a specific format. Figure 3 shows a snippet of a TaskTracker log, and Figure 4 a snippet of a DataNode log.

Hadoop source-code

```
LOG.debug("Number of active connections is: "+xceiverCount);
LOG.info("Received block " + b + " from " +
    s.getInetAddress() + " and mirrored to "
    + mirrorTarget);
LOG.info("Served block " + b + " to " + s.getInetAddress());
```

⇓ DataNode log

```
2008-08-25 16:24:12,603 INFO
    org.apache.hadoop.dfs.DataNode:
    Number of active connections is: 1
2008-08-25 16:24:12,611 INFO
    org.apache.hadoop.dfs.DataNode:
    Received block blk_8410448073201003521 from
    /172.19.145.131 and mirrored to
    /172.19.145.139:50010
2008-08-25 16:24:13,855 INFO
    org.apache.hadoop.dfs.DataNode:
    Served block blk_2709732651136341108 to
    /172.19.145.131
```

Figure 4: `log4j`-generated DataNode log. Local and remote data dependencies are captured.

5 Log Analysis

To demonstrate Salsa's approach, we focus on the logs generated by Hadoop's TaskTracker and DataNode daemons. The number of these daemons (and, thus, the number of corresponding logs) increases with the size of a Hadoop cluster, inevitably making it more difficult to analyze the associated set of logs manually. Thus, the TaskTracker and DataNode logs are attractive first targets for Salsa's automated log-analysis.

At a high level, each TaskTracker log records events/activities related to the TaskTracker's execution of `Map` and `Reduce` tasks on its local host, as well as any dependencies between locally executing `Reduces` and `Map` ouputs from other hosts. On the other hand, each DataNode log records events/activities related to the reading or writing (by both local and remote `Map` and `Reduce` tasks) of HDFS data-blocks that are located on the local disk. This is evident in Figure 3 and Figure 4.

5.1 Derived Control-Flow

TaskTracker log. The TaskTracker spawns a new JVM for each `Map` or `Reduce` task on its host. Each `Map` thread is associated with a `Reduce` thread, with the `Map`'s output being consumed by its associated `Reduce`. The `Map` and `Reduce` tasks are synchronized to the `MapReduceCopy` and `ReduceCopy` activities in each of the two types of tasks, when the `Map` task's output is copied from its host to the host executing the associated `Reduce`.

The `Maps` on one node can be synchronized to a `Reduce` on a different node–SALSA derives this distributed control-flow across all Hadoop hosts in the cluster by collectively parsing all of the hosts' TaskTracker logs. Based on the TaskTracker log, SALSA derives a state-machine for each unique `Map` or `Reduce` in the system. Each log-delineated activity within a task corresponds to a state.

DataNode log. The DataNode daemon runs three main types of data-related threads: (i) `ReadBlock`, which serves blocks to HDFS clients, (ii) `WriteBlock`, which receives blocks written by HDFS clients, and (iii) `WriteBlock_Replicated`, which receives blocks written by HDFS clients that are subsequently transferred to another DataNode for replication. The DataNode daemon runs in its own independent JVM, and the daemon spawns a new JVM thread for each thread of execution. Based on the DataNode log, SALSA derives a state-machine for each of the unique data-related threads on each host. Each log-delineated activity within a data-related thread corresponds to a state.

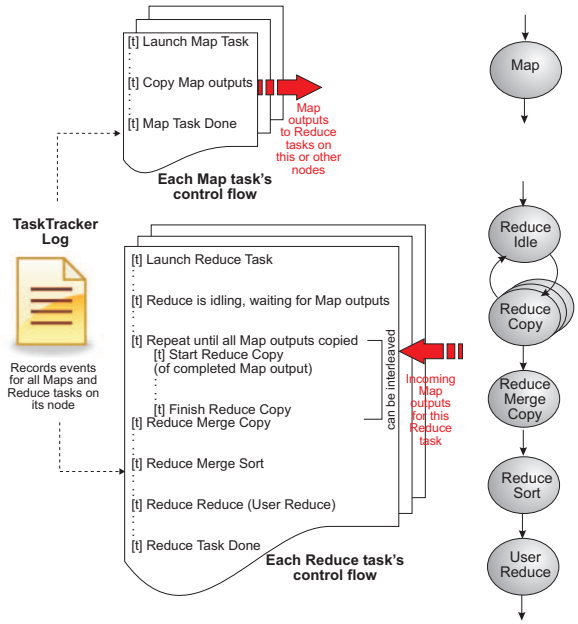

Figure 5: Derived Control-Flow for Hadoop's execution.

Processing Activity	Start Token	End Token
Map	LaunchTaskAction: [MapID]	Task [MapID] is done.
MapCopy	N/A	N/A
ReduceIdle	LaunchTaskAction: [ReduceID]	Task [ReduceTaskID] is done.
ReduceCopy	[ReduceID] Copying [MapID] output from [Hostname]	[ReduceID] done copying [MapTaskID] output from [Hostname].
ReduceMergeCopy	[ReduceID] Thread started: Thread for merging in memory files	[ReduceID] Merge of the 3 files in InMemoryFileSystem complete. Local file is [Filename]
ReduceSort	N/A	N/A
ReduceUser	N/A	N/A

Table 1: Tokens in TaskTracker-log messages for identifying starts and ends of states.

5.2 Tokens of Interest

SALSA can uniquely delineate the starts and ends of key activities (or states) in the TaskTracker logs. Table 1 lists the tokens that we use to identify states in the TaskTracker log. `[MapID]` and `[ReduceID]` denote the identifiers used by Hadoop in the TaskTracker logs to uniquely identify `Maps` and `Reduces`.

The starts and ends of the `ReduceSort` and `ReduceUser` states in the `Reduce` task were not identifiable from the TaskTracker logs; the log entries only identified that these states were in progress, but not when they had started or ended. Additionally, the `MapCopy` processing activity is part of the `Map` task as reported by Hadoop's logs, and is currently indisguishable.

SALSA was able to identify the starts and ends of the data-related threads in the DataNode logs with a few provisions: (i) Hadoop had to be reconfigured to use `DEBUG` instead of its default `INFO` logging level, in order for the starts of states to be generated, and (ii) all states completed in a First-In First-Out (FIFO) ordering. Each data-related thread in the DataNode log is identified by the unique identifier of the HDFS data block. The FIFO ordering assumption was necessary to attribute log messages corresponding to the starts of processing activities to those corresponding to the ends of processing activities. The DataNode log recorded the "Number of active connections" in the DataNode daemon whenever a thread of execution

Processing Activity	End Token
ReadBlock	Served block [BlockID] to [Hostname]
WriteBlock	Received block [BlockID] from [Hostname]
WriteBlock_Replicated	Received [BlockID] from [Hostname] and mirrored to [Hostname]

Table 2: Tokens in DataNode-log messages for identifying ends of data-related threads

was created or completed, and the starts of processing activities were detected from increasing pairs of the number of active connections in the daemon. Hence, the starts of processing activities as detected from the logs identified neither the activity itself nor its instance that had started. Only log messages for the ends of processing activities identified the execution-step and its instance. Hence, the assumption of FIFO completion of execution-steps was used to match the earliest unmatched log message for the start of an processing activity with each subsequent message for the end of an processing activity seen to identify the starts of processing activities. The log messages identifying the ends of states in the DataNode- logs are listed in Table 2.

Race condition in DataNode logging

In examining the Hadoop DataNode logging mechanism, we discovered a race condition in the generation of DataNode log messages for the starts of processing activities. On the spawning and completing of each thread of execution, the DataNode respectively incremented and decremented an internal counter; however, the incrementing and decrementing of the counter, together with the generation of the log message which contained the connection count, was not atomic, so that the connection count could change before the log message was generated. This prevented us from accurately extracting the control-flow model of the DataNode from the candidate version of Hadoop.

Augmented DataNode logging

We addressed this race condition the the DataNode's logging mechanism by introducing a unique sequence number for each DataNode thread of execution that was incremented and reported by that thread of execution on its start and end in an atomic manner. This addressed both the race condition in the logging, and also identified the event for the start of that execution-thread with the event for its end, allowing us to do away with the assumption of the FIFO completion of DataNode threads of execution.

5.3 Data-Flow in Hadoop

A data-flow dependency exist between two hosts when an activity on one host requires transferring data to/from another node. The DataNode daemon acts as a server, receiving blocks from clients that write to its disk, and sending blocks to clients that read from its disk. Thus, data-flow dependencies exist between each DataNode and each of its clients, for each of the `ReadBlock` and `WriteBlock` states. SALSA is able to identify the data-flow dependencies on a per-DataNode basis by parsing the hostnames jointly with the log-messages in the DataNode log.

Data exchanges occur to transfer outputs of completed `Maps` to their associated `Reduces` in the `MapCopy` and `ReduceCopy` phases. This dependency is captured, along with the hostnames of the source and destination hosts involved in the `Map`-output transfer. Tasks also act as clients of the DataNode in reading `Map` inputs and writing `Reduce` outputs to HDFS. However, these activities are not recorded in the TaskTracker logs, so these data-flow dependencies are not captured.

5.4 Extracted Metrics & Data

We extract multiple statistics from the log data, based on SALSA's derived state-machine approach. We extract statistics for the following states: `Map`, `Reduce`, `ReduceCopy` and `ReduceMergeCopy`.

• Histograms and average of duration of unidentified, concurrent states, with events coalesced by time, allowing for events to superimpose each other in a time-series.

Symptom	[Source] Reported Failure	[Failure Name] Failure Injected
Processing	[Hadoop users' mailing list, Sep 13 2007] CPU bottleneck resulted from running master and slave daemons on same machine	[CPUHog] Emulate a CPU-intensive task that consumes 70% CPU utilization
Disk	[Hadoop users' mailing list, Sep 26 2007] Excessive messages logged to file during startup	[DiskHog] Sequential disk workload wrote 20GB of data to filesystem

Table 3: Failures injected, the resource symptom category they correspond to, and the reported problem they simulate

• Histograms and exact task-specific duration of states, with events identified by task identifer in a time-series;
• Duration of completed-so-far execution of ongoing task-specific states.

We cannot get average times for `ReduceReduce` and `ReduceSort` because these have no well-defined start and termination events in the log.

For each DataNode and TaskTracker log, we can determine the number of each of the states being executed on the particular node at each point in time. We can also compute the durations of each of the occurrences of each of the following states: (i) `Map`, `ReduceCopy`, `ReduceMergeCopy` for the TaskTracker log, and (ii) `ReadBlock`, `WriteBlock` and `WriteBlock_Replicated` for the DataNode log.

On the data-flow side, for each of the `ReadBlock` and `WriteBlock` states, we can identify the end-point host involved in the state, and, for each of the `ReduceCopy` states, the host whose `Map` state was involved. However, we are unable to compute durations for `UserReduce` and `ReduceSort` because these have no well-defined start and termination events in the logs.

6 Data Collection & Experimentation

We analyzed traces of system logs from a 6-node (5-slave, 1-master) Hadoop 0.12.3 cluster. Each node consisted of an AMD Opeteron 1220 dual-core CPU with 4GB of memory, Gigabit Ethernet, and a dedicated 320GB disk for Hadoop, and ran the amd64 version Debian/GNU Linux 4.0. We used three candidate workloads, of which the first two are commonly used to benchmark Hadoop:
• *RandWriter* : write 32 GB of random data to disk;
• *Sort* : sort 3 GB of records;
• *Nutch* : open-source distributed web crawler for Hadoop [9] representative of a real-world workload

Each experiment iteration consisted of a Hadoop job lasting approximately 20 minutes. We set the logging level of Hadoop to `DEBUG`, cleared Hadoop's system logs before each experiment iteration, and collected the logs after the completion of each experiment iteration. In addition, we collected system metrics from `/proc` to provide ground truth for our experiments.

Target failures. To illustrate the value of SALSA for failure diagnosis in Hadoop, we injected three failures into Hadoop, as described in Table 3. A persistent failure was injected into 1 of the 5 slave nodes midway through each experiment iteration.

We surveyed real-world Hadoop problems reported by users and developers in 40 postings from the Hadoop users' mailing list from Sep–Nov 2007. We selected two candidate failures from that list to demonstrate the use of SALSA for failure-diagnosis.

7 Use Case 1: Visualization

We present automatically generated visualizations of Hadoop's aggregate control-flow and data-flow dependencies, as well as a conceptualized temporal control-flow chart. These views were generated offline from logs collected for the *Sort* workload in our experiments. Such visualization of logs can help operators quickly explain and analyze distributed-system behavior.

Aggregate control-flow dependencies (Figure 6) The key point where there are inter-host dependencies in Hadoop's derived control-flow model for the TaskTracker log is the `ReduceCopy` state, when the

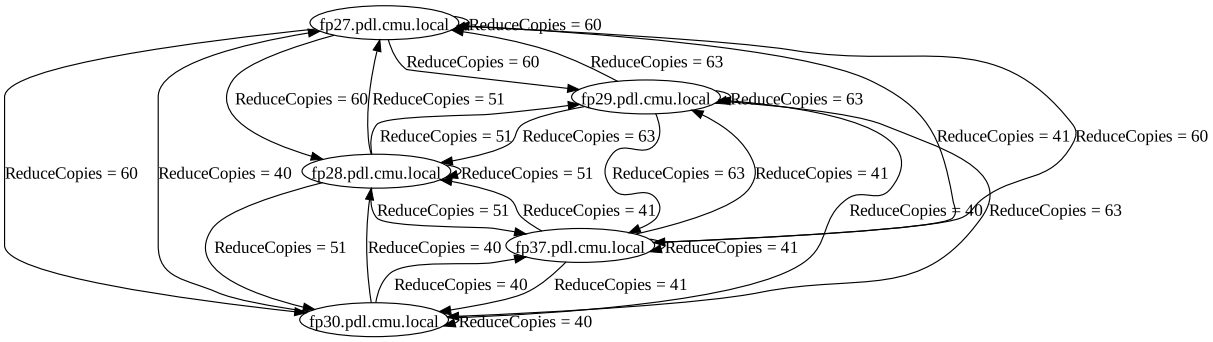

Figure 6: Visualization of aggregate control-flow for Hadoop's execution. Each vertex represents a TaskTracker. Edges are labeled with the number of `ReduceCopy`s from the source to the destination vertex.

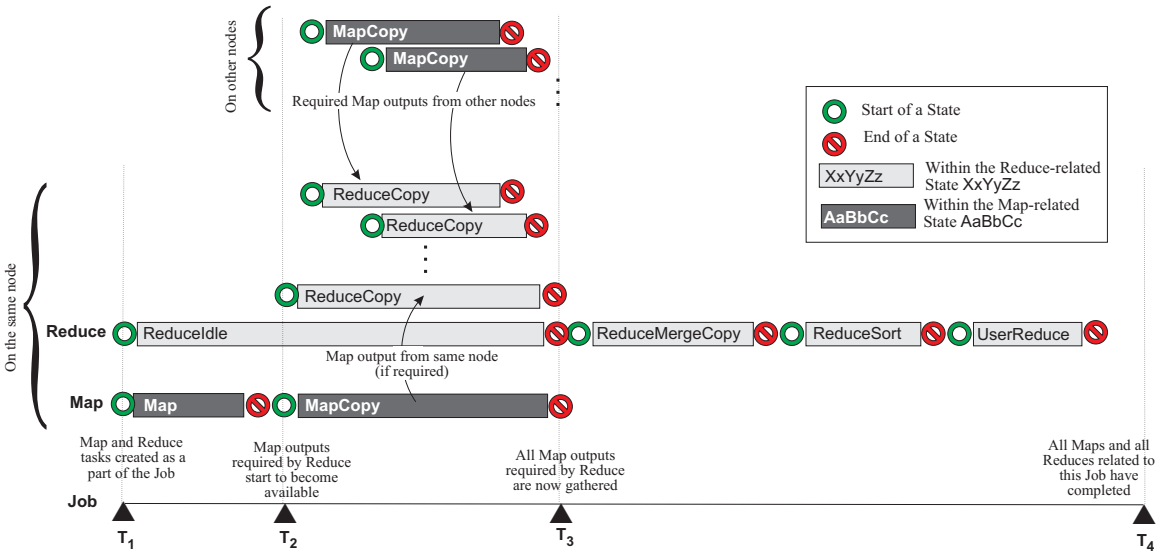

Figure 7: Visualizing Hadoop's control- and data-flow: a sample depiction.

`ReduceCopy` on the destination host for a `Map`'s output is started only when the source `Map` has completed, and the `ReduceCopy` depends on the source `Map` copying its map output. This visualization captures dependencies among TaskTrackers in a Hadoop cluster, with the number of such `ReduceCopy` dependencies between each pair of nodes aggregated across the entire Hadoop run. As an example, this aggregate view can reveal hotspots of communication, highlighting particular key nodes (if any) on which the overall control-flow of Hadoop's execution hinges. This also visually captures the equity (or lack thereof) of distribution of tasks in Hadoop.

Aggregate data-flow dependencies (Figure 8) The data-flows in Hadoop can be characterized by the number of blocks read from and written to each DataNode. This visualization is based on an entire run of the *Sort* workload on our cluster, and summarizes the bulk transfers of data between each pair of nodes. This view would reveal any imbalances of data accesses to any DataNode in the cluster, and also provides hints as to the equity (or lack thereof) of distribution of workload amongst the `Map`s and `Reduce`s.

Temporal control-flow dependencies (Figure 7) The control-flow view of Hadoop extracted from its logs can be visualized in a manner that correlates state occurrences causally. This visualization provides a time-based view of Hadoop's execution on each node, and also shows the control-flow dependencies amongst nodes. Such views allow for detailed, fine-grained tracing of Hadoop execution through time, and allow for

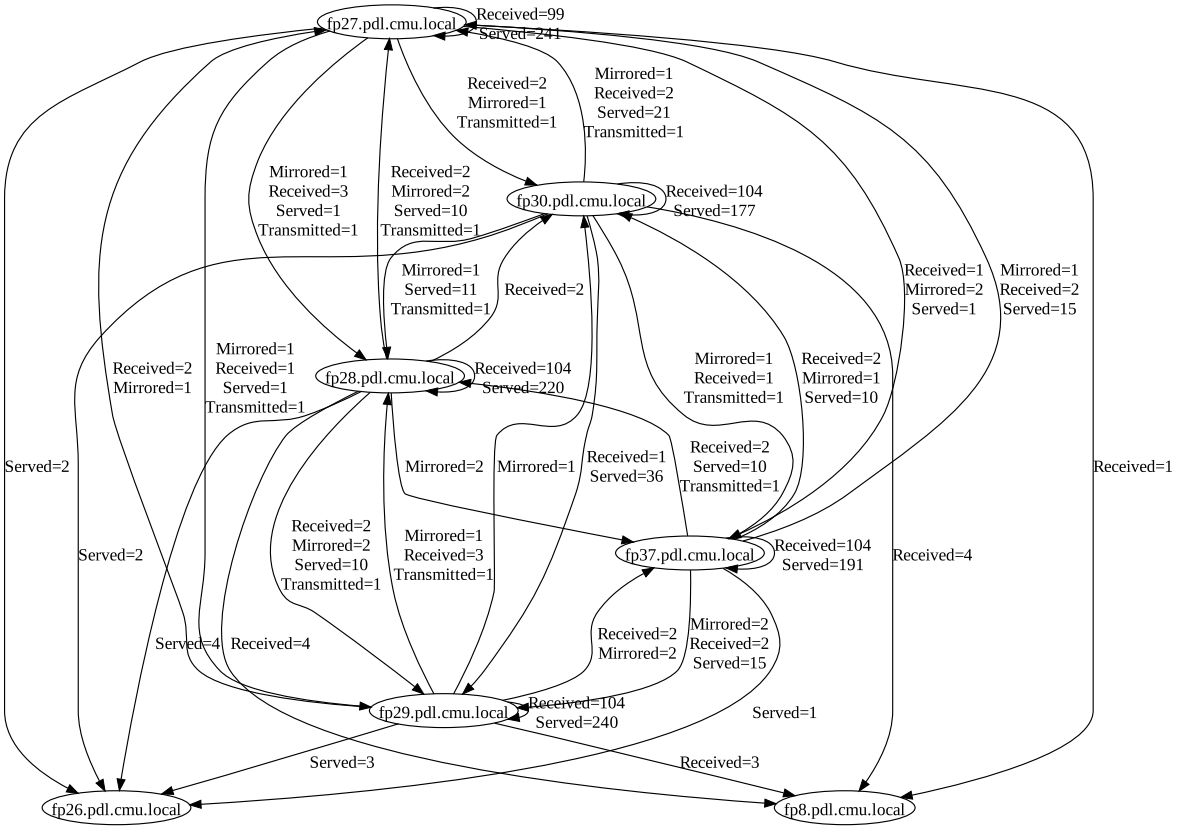

Figure 8: Visualization of aggregate data-flow for Hadoop's execution. Each vertex represents a DataNode and edges are labeled with the number of each type of block operation (i.e. read, write, or write_replicated, which traversed that path.

inter-temporal causality tracing.

8 Use Case 2: Failure Diagnosis

8.1 Algorithm

Algorithm 1 Algorithm to fingerpoint faulty node using durations of states. Note: $JSD(distrib_i, distrib_j)$ is the Jensen-Shannon divergence between the distributions of the states' durations at nodes i and j.

procedure SALSA-FINGERPOINT($prior$, $threshold_p$, $threshold_h$)
  for all i, initialize $distrib_i \leftarrow prior$
  for all i, initialize $lastUpdate_i \leftarrow 0$
  initialize $t \leftarrow 0$
  while job in progress **do**
    for all node i **do**
      while have new sample s with duration d **do**
        if $1 - CDF(distrib_i, d) > 1 - threshold_h$ **then**
          indict s
        end if
        $distrib_i \leftarrow distrib_i \times e^{-\lambda \frac{lastUpdate_i - t}{\alpha(lastUpdate_i - t)+1}}$
        add d to $distrib_i$ with weight 1
        $lastUpdate \leftarrow t$
      end while
    end for
    for all node pair i, j **do**
      $distMatrix(i, j) \leftarrow \sqrt{JSD(distrib_i, distrib_j)}$
    end for
    for all node i **do**
      if $count_j(distMatrix(i, j) > threshold_p) > \frac{1}{2} \times \#nodes$ **then**
        raise alarm at node i
        if 20 consecutive alarms raised **then**
          indict node i
        end if
      end if
    end for
    $t \leftarrow t + 1$
  end while
  for all node i **do**
    if node i associated with at least $\frac{1}{2}$ of indicted states **then**
      indict node i
    end if
  end for
end procedure

Intuition. For each task and data-related thread, we can compute the histogram of the durations of its different states in the derived state-machine view. We have observed that the histograms of a specific state's durations tend to be similar across failure-free hosts, while those on failure-injected hosts tend to differ from those of failure-free nodes, as shown in Figure 10. Thus, we hypothesize that failures can be diagnosed by comparing the probability distributions of the durations (as estimated from their histograms) for a given state across hosts, assuming that a failure affects fewer than $\frac{n}{2}$ hosts in a cluster of n slave hosts.

	TP	*FP*	*TP*	*FP*	*TP*	*FP*
Workload	*RandWriter*		*Sort*		*Nutch*	
Fault	`Map`					
CPUHog	1.0	0.08	0.8	0.25	0.9	0
DiskHog	1.0	0	0.9	0.13	1.0	0.1
	`ReduceMergeCopy`					
CPUHog	0.3	0.15	0.8	0.1	0.7	0
DiskHog	1.0	0.05	1.0	0.03	1.0	0.05
	`ReadBlock`					
CPUHog	0	0	0.4	0.05	0.8	0.2
DiskHog	0	0	0.5	0.25	0.9	0.3
	`WriteBlock`					
CPUHog	0.9	0.03	1.0	0.25	0.8	0.2
DiskHog	1.0	0	0.7	0.2	1.0	0.6

Figure 9: Failure diagnosis results of the Distribution-Comparison algorithm for workload-injected failure combinations; $TP =$ true-positive rate, $FP =$ false-positive rate

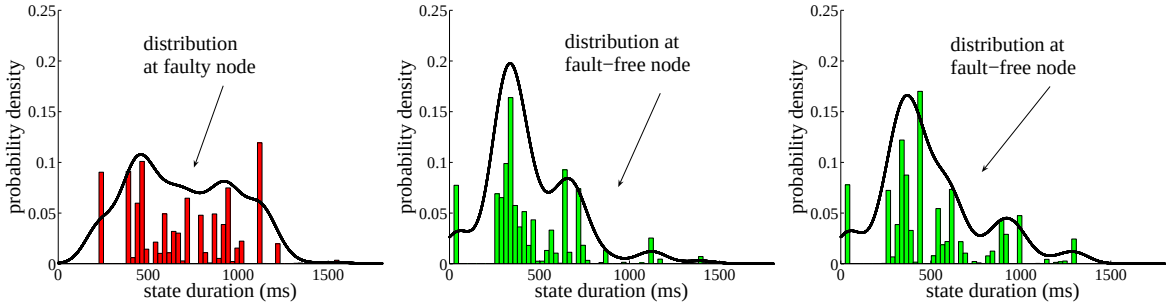

Figure 10: Visual illustration of the intuition behind comparing probability distributions of durations of the `WriteBlock` state across DataNodes on the slave nodes.

Algorithm. First, for a given state on each node, probability density functions (PDFs) of the distributions, $distrib_i$'s, of durations at each node i are estimated from their histograms using a kernel density estimation with a Gaussian kernel [17] to smooth the discrete boundaries in histograms.

In order to keep the distributions relevant to the most recent states observed, we imposed an exponential decay to the empirical distributions $distrib_i$'s. Each new sample s with duration d would then be added to the distribution with a weight of 1. We noticed that there were lull periods during which particular types of states would not be observed. A naive exponential decay of $e^{-\lambda\Delta t}$ would result in excessive decay of the distributions during the lull periods. States that are observed immediately after the lull periods would thus have large weights relative to the total weight of the distributions, and thus effectively result in the distributions collapsing about the newly observed states. To prevent this unwanted scenario, we instead used an exponential decay of $e^{-\lambda\frac{lastUpdate_i - t}{\alpha(lastUpdate_i - t)+1}}$, where $lastUpdate$ is the time of the last observed state, and t is the time of the most recent observation. Thus, the rate of decay slows down during lull periods, and in the limit where $lastUpdate - t \to 0$, the rate of decay approaches the naive expoential decay rate.

The difference between these distributions from each pair of nodes is then computed as the pair-wise distance between their estimated PDFs. The distance used was the square root of the Jensen-Shannon divergence, a symmetric version of the Kullback-Leibler divergence [5], a commonly-used distance metric in information theory to compare PDFs.

Then, we constructed the matrix $distMatrix$, where $distMatrix(i, j)$ is the distance between the estimated distributions on nodes i and j. The entries in $distMatrix$ are compared to a $threshold_p$. Each $distMatrix(i, j) > threshold_p$ indicates a potential problem at nodes i, j, and a node is indicted if at least

half of its entries $distMatrix(i, j)$ exceed $threshold_p$. The pseudocode is presented above.

Algorithm tuning. $threshold_p$ is used for the peer-comparison of PDFs across hosts; for higher values of $threshold_p$, greater differences must be observed between PDFs before they are flagged as anomalous. By increasing $threshold_p$, we can reduce false-positive rates, but may suffer a reduction in true positive rates as well. $threshold_p$ is kept constant for each (workload, metric) combination, and is tuned independently of the failure injected. $threshold_h$**:** Threshold value for comparison of states' durations against the historical distribution on each node. Specifically, states whose durations are larger than the $(threshold_h \times 100)$-percentile of the distribution are flagged as outliers. Therefore, increasing $threshold_h$ reduces false positive rates but also reduces true positive rates. In our experiments, we have kept $threshold_p$ and $threshold_h$ constant for each workload and metric pair.

*bandwidth***:** The bandwidth for Gaussian kernel density estimation is also known as the "smoothing parameter". If the bandwidth is too small, undersmoothing occurs, and noise in data become prominent, leading to a jagged PDF. Oversmoothing occurs when the bandwidth is too large, and details may be lost. There are rules of thumb that can be followed in choosing the bandwidth. In general, we have selected bandwidths for each workload and metric that produced PDFs that are neither oversmoothed nor undersmoothed.

8.2 Results & Evaluation

We evaluated our initial failure-diagnosis techniques based on our derived models of Hadoop's behavior, by examining the rates of true- and false-positives of the diagnosis outcomes on hosts in our fault-injected experiments, as described in § 6. True-positive rates are computed as:

$$\frac{count_i(\text{fault injected on node } i\text{, node } i \text{ indicted})}{count_i(\text{fault injected on node } i)}$$

, i.e., the proportion of failure-injected hosts that were correctly indicted as faulty.

False-positive rates are computed as:

$$\frac{count_i(\text{fault not injected on node } i\text{, node } i \text{ indicted})}{count_i(\text{fault not injected on node } i)}$$

, i.e., the proportion of failure-free hosts that were incorrectly indicted as faulty. A perfect failure-diagnosis algorithm would predict failures with a true-positive rate of 1.0 at a false-positive rate of 0.0. Figure 9 summarizes the performance of our algorithm.

By using different metrics, we achieved varied results in diagnosing different failures for different workloads. Much of the difference is due to the fact that the manifestation of the failures on particular metrics is workload-dependent. In general, for each (workload, failure) combination, there are metrics that diagnose the failure with a high true-positive and low false-positive rate. We describe some of the (metric, workload) combinations that fared poorly.

We did not indict any nodes using `ReadBlock`'s durations on *RandWriter* . By design, the *RandWriter* workload has no `ReadBlock` states since its only function is to write data blocks. Hence, it is not possible to perform any diagnosis using `ReadBlock` states on the *RandWriter* workload. Also, `ReduceMergeCopy` on *RandWriter* is a disk-intensive operation that has minimal processing requirements. Thus, *CPUHog* does not significantly affect the `ReduceMergeCopy` operation, as there is little contention for the CPU between the failure and the `ReduceMergeCopy` operations. However, the `ReduceMergeCopy` operation is disk-intensive, and is affected by the *DiskHog* .

We found that *DiskHog* and *CPUHog* could manifest in a correlated manner on some metrics. For the *Sort* workload, if a failure-free host attempted to read a data block from the failure-injected node, the failure would manifest on the `ReadBlock` metric at the failure-free node. By augmenting this analysis with the data-flow model, we improved results for *DiskHog* and *CPUHog* on *Sort* , as discussed in § 8.3.

The list of metrics that we have explored for this paper is by no means complete. There are other types of states like `Reduces` and heartbeats that we have not covered. These metrics may yet provide more information for diagnosis, and will be left as future work.

8.3 Correlated Failures: Data-flow Augmentation

Peer-comparison techniques are poor at diagnosing correlated failures across hosts, e.g., `ReadBlock` durations failed to diagnose *DiskHog* on the *Sort* workload. In such cases, our original algorithm often indicted failure-free nodes, but not the failure-injected nodes.

We augmented our algorithm using previously-observed states with anomalously long durations, and superimposing the data-flow model. For a Hadoop job, we identify a state as an outlier by comparing the state's duration with the PDF of previous durations of the state, as estimated from past histograms. Specifically, we check whether the state's duration is greater than the $threshold_h$-percentile of this estimated PDF.

Since each DataNode state is associated with a host performing a read and another (not necessarily different) host performing the corresponding write, we can count the number of anomalous states that each host was associated with. A host is then indicted by this technique if it was associated with at least half of all the anomalous states seen across all slave hosts.

Hence, by augmenting the diagnosis with data-flow information, we were able to improve our diagnosis results for correlated failures. We achieved true- and false-positive rates, respectively, of $(0.7, 0.1)$ for the *CPUHog* and $(0.8, 0.05)$ for the *DiskHog* failures on the `ReadBlock` metric.

9 Conclusion

SALSA analyzes system logs to derive state-machine views, distributed control-flow and data-flow models and statistics of a system's execution. These different views of log data can be useful for a variety of purposes, such as visualization and failure diagnosis. We present SALSA and apply it concretely to Hadoop to visualize its behavior and to diagnose documented failures of interest.We also initiated some early work to diagnose correlated failures by superimposing the derived data-flow models on the control-flow models.

For our future directions, we intend to correlate numerical OS/network-level metrics with log data, in order to analyze them jointly for failure diagnosis and workload characterization. We also intend to automate the visualization of the causality graphs for the distributed control-flow and data-flow models. Finally, we will aim to generalize the format/structure/content of logs that are amenable to SALSA's approach, so that we can develop a log-parser/processing framework that accepts a high-level definition of a system's logs, using which it then generates the desired set of views.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed system of black boxes. In *ACM Symposium on Operating Systems Principles*, pages 74–89, Bolton Landing, NY, Oct 2003.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec 2004.

[3] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *IEEE Conference on Dependable Systems and Networks*, Bethesda, MD, Jun 2002.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 137–150, San Francisco, CA, Dec 2004.

[5] D. M. Endres and J. E. Schindelin. A new metric for probability distributions. *Information Theory, IEEE Transactions on*, 49(7):1858–1860, 2003.

[6] Apache Software Foundation. Chainsaw, 2007. http://logging.apache.org/chainsaw.

[7] The Apache Software Foundation. Hadoop, 2007. http://hadoop.apache.org/core.

[8] The Apache Software Foundation. Log4j, 2007. http://logging.apache.org/log4j.

[9] The Apache Software Foundation. Nutch, 2007. http://lucene.apache.org/nutch.

[10] Joseph L. Hellerstein, Sheng Ma, and Chang-Shing Perng. Discovering actionable patterns in event data. *IBM Systems Journal*, 41(3):475–493, 2002.

[11] Chengdu Huang, Ira Cohen, Julie Symons, and Tarek Abdelzaher. Achieving scalable automated diagnosis of distributed systems performance problems, 2007.

[12] Splunk Inc. Splunk: The it search company, 2005. http://www.splunk.com.

[13] Yinglung Liang, Yanyong Zhang, Anand Sivasubramaniam, Morris Jette, and Ramendra K. Sahoo. BlueGene/L failure analysis and prediction models. In *IEEE Conference on Dependable Systems and Networks*, pages 425–434, Philadelphia, PA, 2006.

[14] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *IEEE Conference on Dependable Systems and Networks*, pages 575–584, Edinburgh, UK, June 2007.

[15] A. Oliner and J. Stearley. Bad words: Finding faults in Spirit's syslogs. In *8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008)*, pages 765–770, Lyon, France, May 2008.

[16] H. Gobioff S. Ghemawat and S. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, pages 29 – 43, Lake George, NY, Oct 2003.

[17] L. Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer, 1st edition, Sep 2004.