Stage-Aware Anomaly Detection Through Execution Flow Tracking

Cristiana Amza

Department of Electrical and Computer Engineering
University of Toronto
Logging in Large Scale Systems

• Distributed storage systems of large-scale websites:
  ▸ 1000s of servers

• Designed to be fault tolerant (automatic failure masking)

• Generate large volume of log data
Why Logging?

• We log detailed information for anomaly diagnosis

• Verbose logging: Calls, network, control flow, errors

• Goal: Understand call graph, execution flow, root cause analysis
Call Graph for FS

37 errors out of 188 calls

Slide courtesy of Haryadi S. Gunawi
HDFS: 3-way Replication (3 nodes)

- Storage for Hadoop MapReduce Jobs

- Yahoo!: 4000 nodes
- Facebook: 2000 nodes
- Ebay: 700 nodes

Single failures

Multiple failures
In An Ideal World ...
In the Real World of Logs, Logs, ...

FileTransfer

```c
LOG("Sending file "+file.name); //L1
int i=0;
while(i < file.size){
    int ack=send_recv(dest,file.block(i));
    if( ack == 0){
        i++;
        LOG("Sent block "+ i); //L2
    }else{
        LOG("Failed to send file "+ file.name); //L3
        return -1;
    }
}
LOG("Sent file "+ file.name); //L4
return 0;
```

Logs

1:10.032 Sending file A
1:10.042 Sent block 1
1:10.099 Sent block 2
...
1:11.030 Sent block 1000
1:11.032 Sent file A

Logs contain information about the execution flow and duration
Logs Contain Detailed Information

FileTransfer

```c
LOG("Sending file "+file.name);  //L1
int i=0;
while(i < file.size){
    int ack=send_recv(dest,file.block(i));
    if( ack == 0){
        i++;
        LOG("Sent block "+ i);  //L2
    }
    }else{
        LOG("Failed to send file "+ file.name); //L3
        return -1;
    }
}LOG("Sent file "+ file.name); //L4
return 0;
```

Logs

1:10.023 Sending file F
1:10.042 Sent block 1
1:10.099 Sent block 2
....
1:11.130 Sent block 300
1:11.131 Failed to send file F
Logs Contain Detailed Information

FileTransfer

```cpp
LOG("Sending file "+file.name); //L1
int i=0;
while(i < file.size){
    int ack=send_recv(dest,file.block(i));
    if( ack == 0){
        i++;
        LOG("Sent block "+ i); //L2
    }
    else{
        LOG("Failed to send file "+ file.name); //L3
        return -1;
    }
}
LOG("Sent file "+ file.name); //L4
return 0;
```

Logs

1:10.032 Sending file B
1:10.042 Sent block 1
1:10.099 Sent block 2
....
1:14.030 Sent block 1000
1:14.032 Sent file B
Inferring Execution Flow from Logs

1:10.023 Sending file F
1:10.042 Sent block 1
1:10.099 Sent block 2
....
1:11.130 Sent block 300
1:11.131 Failed to send file F
Challenges

• Identify tasks from logs

• Distinguish normal vs. anomalous tasks

• Overheads
  ‣ Storage and processing
Challenge 1: Identify tasks from logs

Normal Execution Flow
1:10.023 Start sending file A
1:10.042 Sent block 1
1:10.099 Sent block 2
...
1:11.130 Sent block 1000
1:11.132 Sent file A

Anomalous Execution Flow
1:10.033 Start sending file F
1:10.032 Sent block 1
1:11.049 Sent block 2
...
1:11.050 Sent block 300
1:11.131 Failed to send file F

Server runs forever, how to delimit tasks?
Challenge 1: Identify tasks from logs

<table>
<thead>
<tr>
<th>Normal Execution Flow</th>
<th>Anomalous Execution Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:10.023 Start sending file A</td>
<td>1:10.033 Start sending file F</td>
</tr>
<tr>
<td>1:10.042 Sent block 1</td>
<td>1:10.032 Sent block 1</td>
</tr>
<tr>
<td>1:10.099 Sent block 2</td>
<td>1:11.049 Sent block 2</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
</tr>
<tr>
<td>1:11.130 Sent block 1000</td>
<td>1:11.050 Sent block 300</td>
</tr>
<tr>
<td>1:11.132 Sent file A</td>
<td>1:11.131 Failed to send file F</td>
</tr>
</tbody>
</table>

Thread reuse, data flow tracking may obscure task execution flow
Challenge 2: Normal vs Anomaly

• No baseline

• Anomalies do not always generate an explicit error/warning log message

How to identify normal vs. rare execution flows?
Challenge 3: Overhead

- Log messages are text to be read by human
- Verbose logging generates huge log volume!
  - 2600 times more than INFO-level (default) logging
- Text processing is expensive and imprecise

2.4TB per day for a 100 node Cassandra cluster
Not many people can look into a 2TB log file
Key Observation

Many server codes have a modular or stage-based architecture

• Stage-aware anomaly detection
  ‣ on-the-fly
  ‣ with low overhead
  ‣ almost completely automated
Staged-Architecture

Server

Foo
Bar
Baz

stage

Task 4
Task 5
Task 6
We capture execution flow from log points on-the-fly without generating logs.
We exploit similarity between tasks for statistical anomaly detection.
Stage-Aware Anomaly Detection

• Leverage the staged code structure
  ‣ To track start and end of each task

• Log statements as trace points
  ‣ To track execution flow of the tasks

• Exploit the statistical similarity of tasks to detect
  ‣ Flow anomalies: rare execution paths
  ‣ Performance anomalies: unusually high duration
Prototype

Hosts

- Server
- Task Execution Tracker
- Logger

Statistics Analyzer

- Tracks log points encountered by each task
- Generates *synopses* of task execution flows
- Streams to *Statistical Analyzer* for *real-time* anomaly detection
Instrumentation

FileTransfer

Tracker.setContext("FileTransfer", task_id);
LOG("Sending file "+file.name);  //L1
int i=0;
while(i < file.size){
    int ack=send_recv(dest,file.block(i));
    if( ack == 0){
        i++;
        LOG("Sent block "+ i);  //L2
    }else{
        LOG("Failed to send file "+ file.name);  //L3
        return -1;
    }
}
LOG("Sent file "+ file.name);  //L4
return 0;
Tracking Log Points

FileTransfer

Tracker.setContext("FileTransfer", task_id)

LOG(1,"Sending file "+file.name); //L1
int i=0;
while(i < file.size){
    int ack=send_recv(dest,file.block(i));
    if( ack == 0){
        i++;
        LOG(2,"Sent block "+ i); //L2
    }else{
        LOG(3,"Failed to send file "+ file.name); //L3
        return -1;
    }
}
LOG(4,"Sent file "+ file.name); //L4
return 0;

Synopsis: <134, "FileTransfer", 3.5 ms, [L1, L2, L4]>
Automatic Instrumentation

**producer-consumer model**

*Foo* → *Consumer Threads* → *Bar*

**dispatcher-worker model**

*Dispatcher Threads* → *Worker Threads* → *Tracker*

**consumer thread**

```
while (1){
    req=dequeue()
    Tracker.setContext("FileTransfer", task_id)
    do(req)
}
```

**worker thread**

```
void Thread.run(){
    Tracker.setContext("FileTransfer", task_id)
    do()
}
```

<table>
<thead>
<tr>
<th>server</th>
<th>#stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS</td>
<td>10</td>
</tr>
<tr>
<td>HBase</td>
<td>38</td>
</tr>
<tr>
<td>Cassandra</td>
<td>78</td>
</tr>
</tbody>
</table>
Model Building (per Stage)

Anomaly: statistically significant increase of outliers

FileTransfer

99.9% [L₁, L₂, L₄ ]

0.1% [L₁, L₂]

99.1% 1 ms

0.9% 4 ms

flow outlier

performance outlier
Evaluation

• Three distributed storage systems:
  ‣ Hadoop Distributed Filesystem (HDFS)
  ‣ HBase
  ‣ Cassandra

• Write intensive workload of Yahoo Cloud Serving Benchmark (YCSB)
  ‣ Similar to real-world scenarios
  ‣ Stress the system
Experiment Setup

Workload Emulator

Cassandra 1
- Task Execution Tracker
- Logger

Cassandra 2
- Task Execution Tracker
- Logger

Cassandra 3
- Task Execution Tracker
- Logger

Cassandra 4
- Task Execution Tracker
- Logger

Statistical Analyzer
Cassandra write I/O path

Write-Ahead-Log (WAL)

Buffer pool (in-memory tables)

Error on write I/O to WAL

Drop 1%
Error on write I/O

Fault: failing 1% of write to WAL

Execution Flow

<table>
<thead>
<tr>
<th>Log Id</th>
<th>Log Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>L101</td>
<td>“memtable is already frozen; another thread must be flushing it”</td>
</tr>
<tr>
<td>L102</td>
<td>“Applying”</td>
</tr>
<tr>
<td>L103</td>
<td>“Adding hint for”</td>
</tr>
<tr>
<td>L104</td>
<td>“... applied. Sending response to”</td>
</tr>
<tr>
<td>L105</td>
<td>“applying mutation of row {}”</td>
</tr>
</tbody>
</table>

Normal

Anomalous

L101  “memtable is already frozen; another thread must be flushing it”

Error logs

Input/output error

Fatal exception – crash!
**Error on write I/O**

Error log message
Anomaly in execution flow
Fault: failing 1% of write to WAL

```
Cassandra 1
```

```
memtables frozen
```

```
Cassandra 2
Cassandra 3
Cassandra 4
```

```
IncomingTcpConnection
CassandraDaemon
StorageProxy
LogRecordAdder
Table
```

```
Throughput (op/sec)
```

```
Time (Minute)
```

```
5.4 million tasks
```

```
```
### Uncovering Bugs & Misconfigurations

<table>
<thead>
<tr>
<th>#</th>
<th>Type</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bug</td>
<td>HDFS Data Node</td>
<td>Empty Packet</td>
</tr>
<tr>
<td>2</td>
<td>Bug</td>
<td>Hbase Regionserver</td>
<td>Distributed Log splitting gets stuck</td>
</tr>
<tr>
<td>3</td>
<td>Misconfig.</td>
<td>HBase Regionserver</td>
<td>No live nodes contain current block</td>
</tr>
<tr>
<td>4</td>
<td>Misconfig.</td>
<td>Hbase Regionserver</td>
<td>Zookeeper missed heartbeat due to lengthy GC</td>
</tr>
</tbody>
</table>
Analysis Cost

• State of the Art, based on regular expression matching:
  ‣ Offline processing of 1.6GB log data
    - 12 millions log messages
  ‣ 12 Minutes on 8 cores full utilization

• Our solution
  ‣ Real time on one core
  ‣ Average of 3% CPU utilization
Storage Overhead

Up to 1000x storage saving vs. verbose logging
No overhead when using SAAD vs. default (INFO) logging
False Positive: Flow anomalies

Each experiment: 90min (10 times)

1 false alarm per hour
Stage Aware Anomaly Detection

Provides context to the anomaly

- Code module (stage)
- Execution flow (log points)

In Real Time, with Low overhead

- Minimal runtime overhead
- Low storage cost (synopses instead of logs)
- Low processing cost

Portable, evaluated on 3 distributed storage systems
Questions


and in ACM/IFIP/Usenix Middleware 2014

In collaboration with

Saeed Ghanbari and Ali Hashemi
Ongoing Work: Model Transformation

Cache L1 L2 L3
Core i7 256 kB 1024 kB 8192 kB

Intel Core i7

Read throughput (MB/s)

Size (Bytes)

Stride (x8 Bytes)

- 0-2,000
- 2,000-4,000
- 4,000-6,000
- 6,000-8,000
- 8,000-10,000
- 10,000-12,000
Guidance: e.g., 3D to 2D reduction

<table>
<thead>
<tr>
<th>Size</th>
<th>Xeon L1</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>s16</td>
<td>256 kB</td>
<td>2048 kB</td>
<td>20480 kB</td>
</tr>
<tr>
<td>Core i7</td>
<td>256 kB</td>
<td>1024 kB</td>
<td>8192 kB</td>
</tr>
</tbody>
</table>
With minimum new samples...

Cache | L1 | L2 | L3
-----|----|----|----
Xeon | 256 kB | 2048 kB | 20480 kB

Read throughput (MB/s)

Stride (x8 Bytes)

Size (Bytes)

Colors:
- Green: 10,000-12,000
- Blue: 8,000-10,000
- Orange: 6,000-8,000
- Yellow: 4,000-6,000
- Brown: 2,000-4,000
- Grey: 0-2,000