D3: Declarative Distributed Debugging

Byung-Gon Chun
Kuang Chen
Gunho Lee
Randy H. Katz
Scott Shenker

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-27
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-27.html

March 31, 2008

Acknowledgement

We would like to thank Rodrigo Fonseca and George Porter for their help on X-Trace and Andy Konwinski and Matei Zaharia for providing us X-Trace enabled Hadoop. We also thank Joe Hellerstein and Petros Maniatis for their valuable feedback.

D3: Declarative Distributed Debugging

Byung-Gon Chun[†], Kuang Chen[⋆], Gunho Lee[⋆], Randy Katz[⋆], Scott Shenker[⋆†]
[†]International Computer Science Institute, [⋆]University of California at Berkeley

Abstract

Large-scale distributed systems, like MapReduce, are increasingly being used, but debugging such systems is still very difficult. In this paper, we propose D3, a new debugging system that answers diverse debugging queries by processing logs formally and efficiently. D3 specifies queries concisely in a declarative language and executes the queries in a distributed fashion, thereby lowering debugging overhead and bandwidth consumed. We demonstrate the effectiveness of our D3 design on a local cluster with Hadoop, an open-source MapReduce framework.

1 Introduction

Distributed programs are becoming larger and more sophisticated. For example, typical MapReduce [10] applications run on thousands of nodes, and have complex threads of execution that span multiple distributed components. However, our ability to build such programs is outstripping our ability to debug them.

Debugging[1] typically relies on logs that capture such execution, which are produced locally by nodes in the systems. In most cases, these logs are specific to applications and generated by *printf* statements annotated by developers. These logs are scattered everywhere, numerous, and large.

Common approaches to debugging perform *centralized* log processing. In a debugging system, a centralized node collects logs from other nodes, stores them, and processes them to answer debugging questions. Such a system often uses *ad hoc* scripts for processing logs.

These approaches have the following weaknesses. Fetching logs to a centralized point is inefficient and not scalable. Large and numerous logs require significant bandwidth. Furthermore, debugging is often like looking for a needle in a haystack; to find answers most logs are not necessary and do not need to be fetched. Using ad hoc scripts causes troubleshooters to focus on low-level details like how to parse and interpret logs rather than on high-level query goals.

Moreover, centralized processing becomes more challenging as the scale of systems increases, systems run in wide-area networks, and troubleshooters want to do online debugging, which notifies automatically, in real time, when faults or performance problems occur. In addition, if systems operate across multiple administrative domains (due to privacy concerns or policy issues) domains may not want to ship their logs to the centralized point; however, they may be willing to expose summary results computed locally on their logs. To solve these problems and challenges, we need smart log processing that is a more efficient and formal way to process logs for debugging.

In this paper, we present D3, our new approach to debugging distributed systems. In contrast to traditional approaches, D3 is *distributed* and *declarative*. D3 does not move logs but sends debugging queries to nodes, processes logs locally in situ, and receives concise query results from nodes. In particular, it supports queries that trace particular execution paths of interest across multiple distributed components. D3 specifies queries declaratively to separate *what to query* from *how to query*. Once queries and logs are specified compactly in high-level rules, D3 translates these specifications into low-level code and executes it on its distributed query engine.

Our D3 design is a hybrid approach, combining the insights and implementations of three existing systems (Section 2). D3 combines a high-level declarative language for debugging specifications and a distributed database based on P2 [17] with execution tracing by X-Trace [11] and resource utilization monitoring by Ganglia [1]. To illustrate D3's expressiveness, we show how D3 can implement distributed matching in one rule, distributed queries following the paths of specific executions in six rules, and distributed join of tracing and resource utilization data in three rules (Section 3). Through these examples and discussion we show that D3 can express diverse debugging queries.

We have evaluated the feasibility of D3 using an X-Trace enabled Hadoop [2], an open-source MapReduce [10] framework, in our local cluster with respect to query conciseness and network efficiency (Section 4). We find that D3 rules are compact and easy to customize. Furthermore, D3 can save bandwidth by not fetching unnecessary logs to a centralized point. In the queries we ran, we achieved two orders of magnitude bandwidth reduction even though the tracing and resource utilization data we collected were coarse grained. We discuss re-

[1]We use the term "debugging" broadly to represent both code debugging in development and test stages and troubleshooting of running applications in deployment stages.

lated work in Section 5, and conclude in Section 6.

2 D3 Architecture

The D3 system architecture implements the principles of declarative, distributed debugging. In this section, we first motivate our architecture by discussing user debugging requirements and discuss how users may use D3 for debugging. We then describe the high-level D3 system model focusing on how different components fit together to realize D3 and explain main components.

2.1 Debugging Distributed Systems

Distributed systems can incur faults or performance problems when components of the systems fail or there are unintended interactions between components. Analogous to local debuggers that locate bugs in a program, a distributed debugging tool is used to localize hard or performance faults in a distributed system. Typically, a user takes a top-down approach, looking first at higher level behavior, then drilling down lower, following the tracks of unintended actions. A key difference from local debugging is that events are causally dependent across nodes. Thus, we need to trace across nodes in the network. This can be done by explicitly tracking or heuristically inferencing causality across nodes.

Debugging performance problems is especially difficult in large-scale systems. For performance debugging, it is useful to monitor system resource utilization information alongside application states. With this information, the user can figure out, for example, what operations are bottlenecks during particular execution time spans.

The user needs to gain various views on subsets of the logs and interactions in the system. In a large-scale system, creating a single global, master view may incur high communication overhead and latency. Instead, the user should specify subsets of nodes that involve particular threads of execution. In addition, the debugging system must be scalable as the debugged system scales.

2.2 D3 System Model

Thus motivated, we envision the following D3 debugging model. Debugging is done in three steps: a user writes query programs in our high-level declarative language (called NDLog, as explained later)[2], the user creates data models of logs (along with relations, etc.), and the user applies programs to incoming data.

This system model leads to the design of D3 architecture shown in Figure 1. D3 nodes form a distributed query engine. In each node, a trace-enabled application produces logs from its execution tracer and a resource utilization monitor produces logs from its logger. The local logs are imported into a local D3 query process through user-provided *adapters* or standard D3 adapters. The declarative programs written in our language process these local logs and produce high-level information useful for debugging. These programs are compiled and executed by the *distributed query engine*. The user typically selects a node as the *front node*, which serves as a portal from which a query may be issued, and query results may be retrieved.

[2]It might be through a sophisticated GUI-based debugging tool that compiles user actions down into the appropriate lower-layer actions described in our language.

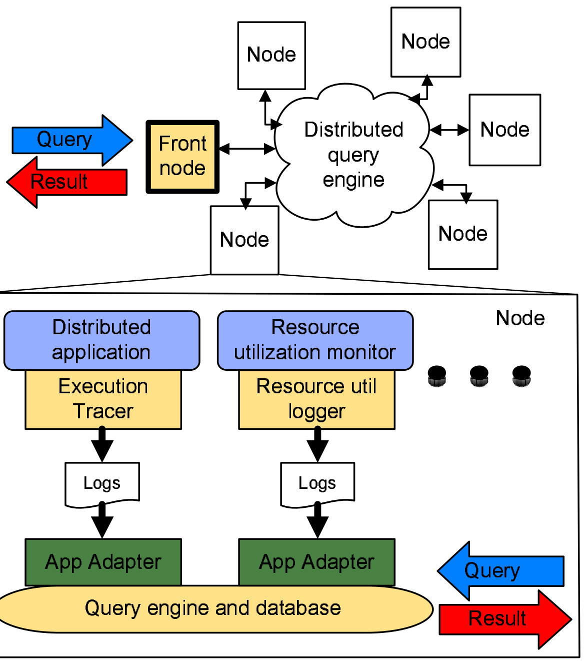

Figure 1: The D3 system ties together distributed applications that generate execution logs with declarative debugging queries. Local logs are streamed into a local D3 process, and queries are processed in a distributed manner.

2.3 Components

We have built D3 by extending existing systems: P2 [17] as our query engine, X-Trace [11] for execution tracing, and Ganglia [1] for resource utilization monitoring.

P2: We chose P2 as our distributed query engine because of its recursion and high-level language support. Recursion across nodes is key to tracing particular execution paths across nodes of networked systems. P2 executes distributed algorithms specified in a high-level logic programming language NDLog, which is a network-aware extension of Datalog. NDLog programs are constructed from rules which specify how tuples of relations are generated from each other. A relation is described by a list of fields and a tuple in a relation is an assignment of values to the fields. In P2, there are materialized and streaming relations. Materialized relations are like database tables and streaming relations are like events.

The rules of NDLog are translated into a data flow graph and directly executed. The data flow graph is instantiated on each node and implements the algorithm specified. Messages traverse the graph for distributed processing (e.g., distributed join), and can be manipulated by relational operations *(join, selection, projection, aggregation)*. As well, messages can trigger other messages or can be generated periodically. We extend P2 to import different types of static or dynamic logs through adapters.

X-Trace: X-Trace is a tracing framework which allows a user to insert tracing metadata on network operations, on different layers, resulting from the task with the same task identifier. The result is trace entries that can be used to construct the task graph, which provides the causal relationships in a call path. X-Trace requires that implementations be modified to carry X-Trace metadata. An X-Trace enabled application logs each tagged operation locally.

Currently, X-Trace does not log the data for tracing forward across nodes since distributed processing of logs (e.g., examining particular execution paths) was not considered. We added a *NextHost* field in X-Trace metadata. X-Trace enabled applications create a *NextHost* field in the trace entries when they cross node boundaries so that our distributed query engine can follow execution paths.

Ganglia: Ganglia is a distributed monitoring system for cluster computing. The system produces resource utilization information for nodes in the cluster. Ganglia is a full fledged resource utilization monitoring system with its own statistics and user interface. We use Ganglia as a local resource monitoring tool and extend it to write the information to local logs.

3 Logs and Queries

3.1 Adapters and Logs

D3 is designed to be extensible to various log types. The user creates an NDLog schema for the application log and builds a log adapter which processes logs produced by the application and injects tuples to our runtime. To provide user data to our runtime, the user simply extends the data producing application to write logs according to the input format of an existing standard application log adapter. Alternatively, the user may create a custom application log adapter to interpret existing application log formats. D3 handles both static and dynamic logs. Static logs include the configurations and policies of systems and offline logs produced by applications. Dynamic logs include online logs produced by running applications and those produced by resource utilization monitors.

Data provided to D3 can be kept in a materialized tuple store or injected as stream events. D3 has the ability to limit the number of tuples, and to keep logs in different time scales in its soft-state table whose record has a timeout ranging from 0 to ∞. In debugging scenarios where the last N records or the last T seconds worth of records is of interest, the soft-state table makes retiring tuples easy.

Relation	Fields
TraceTask	TaskID, OpID, Host, NextHost, Agent, Label, Timestamp
TraceEdge	TaskID, OpID, Type, ParentOpID

Field	Description
TaskID	The task identifier; same for all operations of the task.
OpID	The operation identifier.
Host	The host on which the operation occurs.
NextHost	The host at which the next operation occurs.
Agent	The application specific agent responsible for starting the operation.
Label	The application specific operation name.
Timestamp	The timestamp in milliseconds.
Type	The type of the edge.
ParentOpID	The identifier of the parent operation which led to this operation.

Table 1: Execution tracing data schema — TraceTask and TraceEdge.

For our D3 prototype, we designed NDLog schemas of execution tracing and resource utilization data, and corresponding log adapters. The system can easily handle other types of logs — we list a few of them below.

Execution tracing data: As described above, we are using X-Trace to instrument the run-time behavior. We normalize X-Trace log records and create two D3 table schemas since an X-Trace log record can contain multiple edges from previous parent events. The schemas that represent X-Trace tasks and edges are presented in Table 1.

Resource utilization data: As described above, we are using Ganglia. This is useful for debugging since it allows for important observations to be made. For instance, we can use resource utilization data to do bottleneck analyses — whether an application's bottleneck is CPU or I/O. We can also do correlation analysis between hotspots and resource utilization. Table 2 shows the fields of our resource utilization data obtained from Ganglia.

Other logs: We briefly describe other types of logs, which are potentially useful for debugging.

System logs OSes create system logs that store activities of various devices. This is widely used for system

Category	Fields
CPU	CPU speed, Busy CPU, Sys CPU, Free CPU, CPU WaitIO
Load	1 Min load, 5 Min load
Memory	Mem size, Mem act, Free mem
Disk	Swap in, Swap out, Disk in, Disk out, Disk size, Free disk, Swap used
Network	Tx rate, Rx rate

Table 2: Main resource utilization fields

management and security auditing. We can use these logs for debugging hard faults of nodes, for example.

System call logs Local system call tracking such as DTrace is useful for debugging when problems occur during system call invocations.

Policy data Distributed systems typically use several different components, which have their own policies (e.g., load-balancing, fail-over, and access-control policies). D3 can use this information to detect whether the system follows the load-balancing policy, or it follows SLA specifications and monitors for tasks or operations that most contribute to SLA violation.

Configuration data Network, hardware, software, and operating system information is useful for debugging faults occurred under specific configurations or under misconfigurations. For example, D3 can pinpoint a particular problem occurs in a particular version of software.

3.2 Queries

Diverse queries can be easily implemented with the D3 system to find hard faults or performance problems — from queries that simply find events of interest, to queries that trace an application task's execution path, to queries that aggregate statistics on a particular subset of nodes in a distributed system.

D3 is designed to handle both online and offline queries. In an offline debugging setting, data is loaded into the tables of the distributed query engine after application execution has been completed, and queries are posed over the data. In an online setting, log data streams into soft-state tables in the runtime. Online queries fit naturally to automated debugging. Users can set up triggers and receive alerts when events of interest occur.

Below, we discuss how to construct a query in our system, walk though several representative queries, and then discuss more sophisticated queries we hope to explore.

Constructing queries: Since we use NDLog, a deductive query and rule language, we gain its recursive expressiveness and clean semantics for expressing debugging queries. The ability to create concise and formal logic-based representations of debugging tasks simplifies analyzing distributed systems of increasing complexity.

A query in NDLog is a program that runs in the P2 runtime. These programs are made up of rules that specify how tuples are generated from each other. The simple query in Listing 1 is a single rule named `rft`. NDLog rules have three parts: the rule name, head and body. The head and body are separated by the delimiter `:-`, and are mostly easily interpreted from right to left. When the body is true, the head is true. The distributed nature, or network-awareness, of this programming model is represented by the special symbol "@", which gives a tuple an intended network destination. For instance, in Listing 1, the "@" on the right hand side refers the the address:port of the current location. On the left hand side, "@" represents the destination that the tuple should be delivered.

```
rft taskFound(@F, TID, OpID) :-
  taskReq(@Me, F, TID),
  TraceTask(@Me, TID, OpID).
```

Listing 1: An NDLog rule that finds operations with given TaskID

Constructing queries is straightforward. The user can trace execution paths, aggregate statistics, join data together by identifier, time, or data content. An initial decision point requires that the user takes into account the available log and trace information. Depending on the type of query, and whether it is online or offline, the user must choose to represent the data in the P2 runtime as a stream, a soft-state tuple store or a regular table. That done, the user writes NDLog rules that make up a query program. In the current prototype, queries are based on execution tracing and resource utilization data. Next, we describe exemplary NDLog query programs in detail.

Basic queries: We start with a basic query presented in Listing 1. The query poses the following question: what is the x-trace report of this TaskID?

When we receive a `taskReq`, we lookup the `TID`(TaskID) in the `TraceTask` table. Effectively, we are joining the `taskReq` tuple with matching tuples in the `TraceTask` table based on the key `TID`. When a `TID` is found, we generate a tuple called `taskFound`, address it to the *F*(Front) node from which the request came, and send the details of the `TraceTask` tuple.

This example in Listing 1 can be easily extended to implement queries such as distributed *grep*. Instead of matching on a particular task ID, we can query on a substring or regular expression match of the entire trace statement. For example, with distributed grep, we can answer the following question: find all nodes whose logs contain a particular keyword (e.g., HTTP 500 error).

Execution tracing queries: Listing 2 shows the NDLog rules for tracing the execution path of a particular task. This query is the core of causality tracing. By customizing this query, users can create queries that ask specific measures of execution paths easily.

```
/* initiate tracing execution paths */
rs trFound(@Me, F, TID, POpID, COpID) :-
  trReq(@Me, F, TID), POpID=="00000000",
  TraceEdge(@Me, TID, COpID, _, POpID).

/* trace causal paths */
rl trTry(@Me, F, TID, POpID, COpID) :-
  trFound(@Me, F, TID, POpID, COpID).
rr trTry(@Y, F, TID, POpID, COpID) :-
  trFound(@Me, F, TID, POpID, COpID),
  nextHosts(@Me, TID, Y).
rb trFound(@Me, F, TID, COpID, NOpID) :-
  trTry(@Me, F, TID, POpID, COpID),
  TraceEdge(@Me, TID, NOpID, _, COpID).

/* find next-hop hosts */
rn nextHosts(@Me, TID, NextHost) :-
  trFound(@Me, Front, TID, POpID, COpID),
  TraceTask(@Me, TID, COpID, _, _, _,
    _, _, NextHost), Me!=NextHost.

/* report found tuples to the Front */
rp trReport(@F, Me, TID, POpID, COpID) :-
  trFound(@Me, F, TID, POpID, COpID).
```

Listing 2: NDLog rules that trace the execution paths of a task.

In short, this query finds the start of a task, and follows the NextHost information across nodes. At each step, the trace entry with our `TID` of interest is sent to the Front node as the record of the execution path. This query starts on rule `rs`. A node starting the task in the system learns of a `trReq` received from the Front node, and checks its `TraceEdge` table to see if it has the first operation of a task. In this case, the rule checks that `POpID` is undefined (or equal to "00000000"). In rules `rl` and `rr`, `trFound` tuples generate `trTry` messages. `rl` is a rule for local searching, and `rr` is a rule for forwarding the tracing to the next-hop node (`NextHost`). In rule `rb`, on a `trTry` insert, if the request matches an `TraceEdge`, then we issue `trFound` tuple. Rule `rp` is for reporting; the `trFound` tuple triggers a message `trReport` to the Front node.

Some other path queries, which we can construct by customizing the above query, include: find the longest execution path of this TaskID, find which operation occurred the most number of times, and find the longest or critical path of this TaskID.

Performance queries: Listing 3 shows the NDLog rules for averaging CPU utilization for the jobs of a task during the execution period of the task. This query demonstrates how D3 expresses performance debugging concisely.

The query consists of only three rules. The first rule `rf1` receives an `avgCPUUserReq` from the Front node, and finds an appropriate `TraceTask` with the label that indicates the end of a run. The rule `rf2` finds

```
/* find the end of a run */
rf1 runEnd(@Me, F, TID, OpID, Ag, TS) :-
  avgCPUUserReq(@Me, F),
  TraceTask(@Me, TID, OpID, CID, Host,
    Agent, Label, TS, NextHost),
  Label=="run end".

/* find the start and end of the run */
rf2 runStartEnd(@Me, F, TID, Ag, SOpID,
    STS, EOpID, ETS) :-
  runEnd(@Me, F, TID, EOpID, Ag, ETS),
  TraceEdge(@Me, TID, EOpID, _, SOpID, _),
  TraceTask(@Me, TID, SOpID, _, _,
    _, SLabel, STS, _),
  SLabel=="run start".

/* mean of CPU user during the run */
rc avgCPUUser(@F, TID, Ag,
    AVG<CpuUser>) :-
  runStartEnd(@Me, F, TID, Ag, _,
    STS, _, ETS),
  ganglia(@Me, TS, _,_,_,CpuUser,_),
  TS in [STS, ETS].
```

Listing 3: NDLog rules that find average CPU utilization.

the start of the same run and summarizes the information in the tuple `runStartEnd`. Note that both `rf1` and `rf2` can fire multiple tuples. The rule `rc` finds all CPU data in the relevant time period and computes the average with the `AVG` function. The result is sent back to the Front node.

Some other performance queries, which we can construct by customizing the above query, include: find paths of tasks whose latency is over x seconds, find all paths which took more than y seconds, and find resource utilization and jobs run in a time span.

More sophisticated queries: System developers often want to know how an input change (e.g., workload or system code change) affects the system behavior. *Delta queries* address this problem. Given the delta of an input change, we keep track of the delta of an output change such as execution time and resource utilization changes. If the output is deviant beyond a threshold, we can pinpoint the change of input that affects the system behavior.

An invariant query can be seen as an assertion statement in a distributed system. Users can specify invariants in a D3 declarative language; these invariants can then be checked in a distributed fashion. If an invariant violation occurs, D3 can automatically generate an alert for the users. For example, the load balancing policy may specify that all packets in a session must go to the same host. We hope to investigate these queries in the future.

4 Preliminary Evaluation

We developed a prototype D3 implementation by integrating P2 [17], X-Trace [11], and Ganglia [1] in C++ and Java. We implemented adapters and log generators for X-Trace and Ganglia data, and extended P2 to support features required for our log processing.

We ran our experiments in a local cluster consisting of 16 machines. We used an X-Trace instrumented version of Hadoop 1.2.4 [2], a widely-used large-scale data processing application. It produces X-Trace logs at important execution points to capture the causality of events of Hadoop MapReduce and Distributed File System. As workload, we used a Hadoop wordcount application, which counts the occurrences of each word from input files, with a 16GB input file.

Query conciseness: We implemented NDLog rules presented in Listings 1, 2, and 3 run for our experiments. The rules of the queries are simple. Our common data loading module for all our queries consists of six statements. As we exclude common data loading rules, we can write the matching query in one rule, the execution tracing query in six rules, and the performance query in three rules. Listing 3 computes average CPU utilization per node. By adding extra rules that define a set of nodes (e.g., nodes in the same rack in a data center), we computed the statistics aggregated over the set easily.

Network usage efficiency: For the performance query shown in Listing 3, we report bandwidth usage averaged over four runs. The size of raw log files was 4.28GB, which represents the network usage of a centralized approach, and the network usage of D3 was 62KB, which accounts for the bytes transferred for executing the query and reporting the result of the query. Network bandwidth required for the D3 query is approximately two orders of magnitude less than the size of total log files. This improvement is achieved even though tracing and resource utilization data that we collected were coarse grained. The X-Trace instrumented Hadoop contained only 76 method calls manually annotated.

Hadoop troubleshooting: We used D3 to analyze the characteristics of Hadoop execution runs. We report an instance of our experiments that displayed performance problems. The entire Hadoop job took approximately 16 minutes. The job consisted of 346 map tasks and 1 reduce task. We obtained the statistics of nodes: the number of tasks executed and the minimum, maximum, and mean of task execution time per node, each of which was represented by one rule.

From the query results shown in Table 3, we observe that nodes 2, 4, and 13 have larger maximum execution time than the other nodes have. Node 13 runs a single big reduce task for the entire job, so it takes more time to finish this task. This is a normal behavior given our Hadoop setup. However, behaviors of nodes 2 and 4 are hard to explain. The minimum task execution time of the nodes exceeds the maximum task execution time of the other mapper nodes. This is not likely to be within a normal variance.

Node	MapReduce task time (s)			CPU usage (%)	
	Min.	Max.	Mean	User	WaitIO
1	2.8	102.7	47.7	90.1	0.0
2	129.7	281.8	203.9	55.3	29.6
3	2.7	100.2	83.9	90.6	0.0
4	203.6	274.0	240.3	42.0	39.3
5	3.5	99.7	83.1	88.2	0.0
6	3.2	100.8	83.3	88.4	0.0
7	3.4	127.2	77.0	88.4	0.0
8	4.8	101.2	89.0	89.0	0.2
9	3.1	98.1	71.8	89.2	0.0
10	0.3	108.8	35.6	89.8	0.0
11	87.7	99.4	94.3	91.8	0.0
12	2.8	105.3	54.3	90.0	0.0
13	0.2	688.3	45.8	83.9	0.4
14	82.0	102.5	93.2	87.8	0.0
15	81.1	103.5	92.5	87.5	0.0
16	90.4	101.1	96.3	88.6	0.0

Table 3: Local statistics

To examine what caused this performance problem in nodes 2 and 4, we retrieved resource utilization statistics. We collected the mean CPU usage in user mode during processing particular tasks by running rules in Listing 3. We also collected the CPU usage of waiting I/O by replacing the third rule of Listing 3. The results show that CPUs of nodes 2 and 4 spent significant time in waiting I/O; this is the major cause of the performance problem. We could debug problems of Hadoop execution by writing simple, concise D3 queries.

5 Related Work

D3 is different from other work due to its approach to distributed and declarative debugging. D3's key debugging primitive is analyzing particular threads of execution by tracing the execution paths recursively.

Project 5 [5] and WAP5 [20] aim to debug distributed systems by taking a black-box debugging approach. Magpie [7] and Pinpoint [8] take a gray-box debugging approach that combines prior knowledge, observations, and inference. Pip [19] detects deviation of distributed systems by comparing actual behavior with expected behavior. Sherlock [6], Shrink [15], and SCORE [16] are the systems that localize hard faults or performance problems. They infer dependencies among components and build models for fault localization. All these approaches collect traces at a centralized point for processing.

Friday [12] supports distributed watchpoints and breakpoints (as in local `gdb` for local program debugging). Friday is not scalable to large systems and not efficient for debugging high-level goals. Singh et al. [21] developed an execution tracing of P2 NDLog rules. This system enables debugging P2 rules by following P2 rule and element execution and storing execution information into P2 tables. In contrast, D3 targets debugging *native* applications using P2.

Both Sawzall [18] and Pig [3] are designed for analyzing large data set atop MapReduce. D3 is tailored towards debugging and supports both online and offline queries. Splunk [4] is a commercial IT search engine that indexes logs. It collects all indexes in a centralized point, and uses keyword queries instead of formal declarative queries.

Distributed triggers [14] were proposed for distributed network monitoring (e.g., distributed rate limiting, quota management, and intrusion detection). Other query driven approaches in network monitoring and management include Sophia [22], PIER [13], and Knowledge plane [9].

6 Conclusion

In this paper, we have presented a new debugging system, D3, that realizes formal debugging of distributed systems efficiently. The key insight is that one should leverage the fact that queries are simple but logs are large and numerous, and focus on high-level debugging goals instead of low-level details. Thus, rather than using a centralized and ad hoc approach, D3 takes a distributed and declarative approach. The preliminary evaluation of our D3 prototype is promising. We hope to report our long term experience of D3 with distributed applications and to run large-scale experiments in Amazon's EC2/S3 cluster in the future.

Acknowledgments

We would like to thank Rodrigo Fonseca and George Porter for their help on X-Trace and Andy Konwinski and Matei Zaharia for providing us X-Trace enabled Hadoop. We also thank Joe Hellerstein and Petros Maniatis for their valuable feedback.

References

[1] Ganglia. http://ganglia.sourceforge.net/.

[2] Hadoop. http://lucene.apache.org/hadoop/.

[3] Pig. http://incubator.apache.org/pig/.

[4] Splunk. http://www.splunk.com/.

[5] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.

[6] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM*, 2007.

[7] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, 2004.

[8] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI*, 2004.

[9] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski. A knowledge plane for the internet. In *ACM SIGCOMM*, 2003.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[11] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *NSDI*, 2007.

[12] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *NSDI*, 2007.

[13] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *VLDB*, 2003.

[14] A. Jain, J. M. Hellerstein, S. Ratnasamy, and D. Wetherall. A wakeup call for internet monitoring systems: The case for distributed triggers. In *HotNets*, 2004.

[15] S. Kandula, D. Katabi, and J.-P. Vasseur. Shrink: A tool for failure diagnosis in ip networks. In *MineNet Workshop*, 2005.

[16] R. R. Kompella, J. Yates, A. Greenberg, and A. Snoeren. Ip fault localization via risk modeling. In *NSDI*, 2005.

[17] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, 2005.

[18] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Prog. Journal*, 13(4):277–298, 2005.

[19] P. Reynold, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, 2006.

[20] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: Black-box performance debugging for wide-area systems. In *WWW*, 2006.

[21] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Using queries for distributed monitoring and forensics. In *EuroSys*, 2006.

[22] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *HotNets*, 2003.