Rake: Semantics Assisted Network-Based Tracing Framework

Yao Zhao, Yinzhi Cao, Yan Chen, Ming Zhang, and Anup Goyal

***Abstract*—The ability to trace request execution paths is critical for diagnosing performance faults in large-scale distributed systems. Previous black-box and white-box approaches are either inaccurate or invasive. We present a novel semantics-assisted gray-box tracing approach, called Rake, which can accurately trace individual request by observing network traffic. Rake infers the causality between messages by identifying polymorphic IDs in messages according to application semantics. To make Rake universally applicable, we design a Rake language so that users can easily describe necessary semantics of their applications while reusing the core Rake component. We evaluate Rake using a few popular distributed applications, including web search, distributed computing cluster, content provider network, and online chatting. Our results demonstrate Rake is much more accurate than the black-box approaches while requiring no modification to OS/applications. In the CoralCDN (a content distributed network) experiments, Rake links messages with much higher accuracy than WAP5, a state-of-the-art black-box approach. In the Hadoop (a distributed computing cluster platform) experiments, Rake helps reveal several previously unknown issues that may lead to performance degradation, including a RPC (Remote Procedure Call) abusing problem.**

***Index Terms*—Rake, tracing framework.**

I. Introduction

LARGE-SCALE distributed system and cloud computing have undergone unprecedented growth in recent years. Parallel computing platforms, such as Hadoop [10], enable Yahoo! to search through the entire Library of Congress in less than 30 seconds [7]. Many of these systems employ load balancing, caching, and replication to enhance capacity and availability. On the positive side, if some nodes misbehave, the whole system may still function properly. On the negative side, debugging such systems becomes extremely challenging because many performance problems are not only transient but also unpredictable.

Traditional troubleshooting systems monitor individual services and machines independently. For example, many commercial network management products [4]–[6, 25] keep track of resource usage, such as CPU and disk, and generate syslog messages and various alerts. However, it is well known that the performance of individual machines or network elements may not directly correlate with user-perceived performance. As a result, these commercial products often raise too many alerts. In fact, most of the alerts are simply ignored because they do not affect users.

Recently there has been a plethora of research on debugging performance problems that affect individual user requests. Such work normally leverages the *task tree*[1] to diagnose faults either deterministically or statistically. A task tree encapsulates the set of recursive messages that result from a particular task or user request. For example, accessing a web page usually involves DNS, HTTP, and database queries and responses. By analyzing delays between messages, we can pinpoint the faulty nodes or sometimes even the root causes. However, extracting a task tree from a large number of messages has proven to be extremely challenging, and hence has been intensively studied [8, 9, 18, 19, 22].

We should consider the following factors when designing a system to extract task tree:

- **Accuracy.** Accurately identifying the causality between different messages is the key to diagnosing performance problems.
- **Non-invasiveness.** Some approaches require modifications to the OS, middleware, and/or applications. Generally, an invasive approach may provide accurate tracing results, but its invasiveness often hinders its wide adoption.
- **Applicability.** It is desirable to be applicable to any applications. Due to various practical issues such as model limitation, accuracy requirement and instrumentation overhead, no single approach can be applicable to all systems. Our goal is actually to develop a tracing approach for a wide range of distributed systems.

Most previous approaches for tracing task trees can be classified into either the *black-box* or *white-box* ones. Existing white-box approaches insert some unique IDs into messages by instrumenting the application, middleware, or OS [18, 19]. In contrast, a black-box approach does not require any instrumentation or understanding of application's architecture and/or semantics [8, 9, 22]. Instead, it only relies on temporal correlation between messages. While a black-box approach is non-invasive, it tends to have limited accuracy. This motivates us to develop a novel, "gray-box" approach for task tree extraction that is both non-invasive and accurate.

In this paper, we introduce Rake, a semantics assisted gray-box approach to extract execution path of distributed systems and further use this information to diagnose performance problems and failures. The basic idea stands on the observation that there exist polymorphic "IDs" in the messages of the same task which can be utilized to infer the task tree. In designing

Manuscript received January 18, 2012; revised July 7, 2012. The associate editors coordinating the review of this paper and approving it for publication were B. Lin, J. Xu, and P. Sinha.

Y. Zhao is with Bell Labs, Murray Hill, NJ, USA (e-mail: jingoshine@gmail.com).

Y. Cao and Y. Chen are with Northwestern University, Evanston, IL, USA.

M. Zhang is with Microsoft Research, Redmond, WA, USA.

A. Goyal is with Yahoo! Inc., Sunnyvale, CA, USA.

Digital Object Identifier 10.1109/TNSM.2012.12.120224

[1]Similar terms such as execution path or causality path are also used.

TABLE I
CLASSIFICATION OF MANAGEMENT AND DIAGNOSIS SYSTEMS.

Invasiveness / App Knowledge	Non-Invasive			Invasive
	Network sniffing	Interposition	Logs	Source code modification
Black-box	Project 5, Sherlock	WAP5	Footprint	
Grey-box	Rake		Magpie,SALSA	
White-box				X-Trace, Pinpoint

Rake as a generic tracing and diagnosis tool, we make the following contributions:

- We propose a novel, non-invasive, grey-box tracing and diagnosis approach and only requires limited application semantics provided by application developers.
- We propose general guidelines to identify necessary semantics of applications that can be used to link messages. Two simple rules are demonstrated to be general and powerful enough to allow Rake to be applied in plenty of popular applications.
- We design an XML-based Rake language to allow users to provide application semantics, which makes Rake a general tool that can be quickly adopted to different applications with different semantics. It is also easy to extend Rake to a new or an updated application by just writing an XML file with a few user libraries if necessary.
- We demonstrate the feasibility and accuracy of Rake using some testbed experiments including a content distribution network – CoralCDN [20] and Hadoop. We release our source code online at [26]. In addition, we execute the accuracy analysis based on real measurement data of one major web search infrastructure. Evaluation results demonstrate that the semantics based approach is much more accurate than the black-box approaches while requiring no modification to OS/applications or any logs.

The rest of this paper is organized as follows. We give related work in Section II, problem definition in Section III and introduce Rake in Sections IV. Diagnosis approaches are discussed in Section V. We present evaluation results in Section VI and conclude in Section VII.

II. RELATED WORK

Significant recent research has been done on debugging or troubleshooting service problems in the view of the entire distributed systems. Many of these systems model the dependencies between components with a task tree [8, 13, 19, 22]. A task tree embodies control flows, resources, and performance characteristics associated with servicing a request.

A. Task Tree Extraction Approaches

Table I shows a classification of previous diagnosis and workload extraction systems. We will present them as follows.

a) Black-box approaches: Project5 [8] attempts to identify execution paths of messages with no knowledge of applications. Two algorithms, the nesting algorithm and the convolution algorithm, for inferring the dominant causal paths are proposed in Project5. Reynolds *et al.* further proposes WAP5 [22] to improve Project5. WAP5 also uses time correlation between incoming and outgoing messages on a node to link messages with probabilities. Anandkumar *et al.* studied the linking of transaction footprints and reduced the maximum likelihood rule to the minimum weight bipartite matching problem [9].

Another research work, Sherlock [12], considers an aggregated dependency graph instead of individual task trees. A dependency graph models dependent relationships among components in the network. The follow-up work of Sherlock, Orion [15], uses the delay spike based analysis to further increase the accuracy of discovered dependencies.

These black-box based approaches can be easily applied to different applications; however, the accuracy heavily depends on cross traffic and application properties because time correlation is the only information to link messages.

b) White-box approaches: X-Trace [18] tags all network operations resulting from a particular task with the same task identifier. To do so, the TCP/IP stack is enhanced and applications must be instrumented to invoke X-Trace. However, for a large distributed system using many softwares from different vendors, some even on different platforms, X-Trace may be limited to a certain part of the system where software source codes are available and modifiable. Similarly, Pinpoint [19] also instrument middleware to track the requests as the flow through the system.

c) Gray-box approaches: A gray-box approach is something between the white-box approach and the black-box approach. It does use certain general application knowledge, but does not require the detailed implementation of applications such as data structures. Magpie [13] works with events generated by the operating system, middleware, and application instrumentation. Instead of unique identifiers, Magpie relies on experts with deep knowledge about the system to construct a schema of how to correlate events in different components. SALSA [24] is another log-based approach which relies on the application logs to derive state-machine views of the system's execution. In comparison, Rake is on message level, while Magpie and SALSA rely on the event logs generated by the operating system and applications. Log based approaches may suffer from the insufficient log content and coarse diagnosis level.

d) Intrusiveness Classification: Table I shows a classification of previous diagnosis and workload extraction systems. Sherlock [12] and Project5 [8] only use network sniffed traces, which have no modification to the OS and applications. Reynolds *et al.* develop their own library to collect OS level traces such as system calls [22], which can obtain richer information than pure network sniffing, but is more invasive. X-Trace [18], however, requires users to modify both the OS and the application to inject unique IDs in all messages. Apparently, this approach is extremely invasive. Interestingly, the previous works usually are of two poles, either very invasive white-box or non-invasive black-box. This motivates

our research of semantic-based diagnosis system, Rake, which is non-invasive and very accurate in terms of message linking.

B. Other Related Works

It is worth mentioning that the gray-box concept and semantics are general and used in other research areas as well. For example, in [11], Arpaci-Dusseau *et al.* studied how to treat an OS as a gray box, and then disseminate OS research ideas without requiring any changes to the underlying OS. Also protocol semantics are widely used in the security arena, such as in network intrusion system [17] for packet classification.

III. Problem Definition

Unlike X-Trace [18] that injects IDs into messages, we argue that such IDs already exist in messages and can be dug out. We designed Rake based on the following key observation:

Generally, in distributed system implementations, there are no explicit unique IDs between all the messages in a given task tree; there are, however, polymorphic IDs along the paths of the task tree. Further, the polymorphic IDs can be extracted with proper semantic knowledge of the system implementation.

A. How Does Rake Work?

In this paper, we assume the tree model, *i.e.*, a child message is triggered by only one parent message. This simple model works for many applications and adopted by most diagnosis approaches [8, 12, 22].

Basically, Rake takes four steps to identify task trees: 1) Identify message types based on signatures; 2) Extract ID set of messages based on their type; 3) Given a message, deduce the ID set of its triggered messages; 4) Construct task trees by matching IDs.

For example, consider the recursive DNS query process. DNS packets can be identified by port number. One can take the DNS query target as the ID of the query and its triggered response messages. Hence all the DNS messages for the same query task can be easily connected.

B. Why Does Rake Work?

Generally, if one message triggers another, they must have causality relationship and the causality is also reflected in the contents. In reality, a portion of the messages (so-called IDs) are often enough to uniquely identify the causality. We can find a function to map contents of a message to the IDs of triggered messages. On the other hand, there are so many different distributed systems running different software and protocols. Hence the challenge of this research is to find as universal solutions as possible, and to demonstrate the wide application of the Rake framework.

When will Rake fail? In theory, if Rake can reproduce all states of the monitored system, Rake can always derive exactly the output messages triggered by one input message. In practice building such a system is too expensive if not impossible, and we choose not to infer any internal states of systems. Therefore, if the discovery of the causality relationship between two messages is impossible without some internal states of the distributed system, Rake will fail. For example, imagine in a distributed file system (DFS), the input query is a file name and the output can be a number as the file ID, which is generated based on an internal counter in the server's memory. However, often other part of the input and output messages can disclose the causality as well, as we see in the Hadoop DFS system.

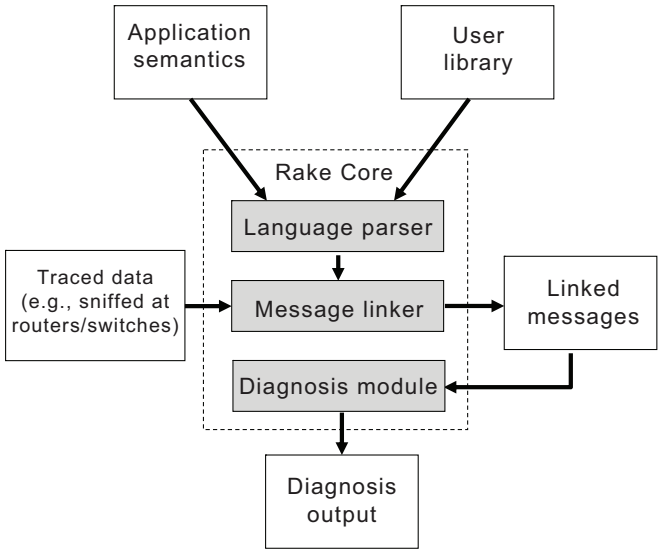


Fig. 1. Architecture of Rake.

C. How to Make Rake Work

While the high-level idea of Rake is very simple, we need to answer the following key questions to build practical tracing systems:

- There is no single universal mapping function. Therefore, we leverage on the semantics of applications. How do we utilize the abstract semantics concept in the real systems?
- Different applications have different semantics. How can we design Rake to be general and easily adopted by different applications with various semantics?
- What accuracy and efficiency can Rake achieve in real applications?

IV. Design of Rake

In this section, we describe our semantic assisted task tree extraction scheme, Rake. We first describe the high-level philosophy of Rake, and then describe, in detail, the design of Rake, including selection and utilization of semantics.

A. System Architecture

Figure 1 shows the architecture of our Rake system. The core components of Rake include three modules: a language parser, a message linker and a diagnosis module. To decouple Rake core from the various application semantics, Rake takes unified semantics as the input, and the language parser reads the application semantics in an XML based language (See

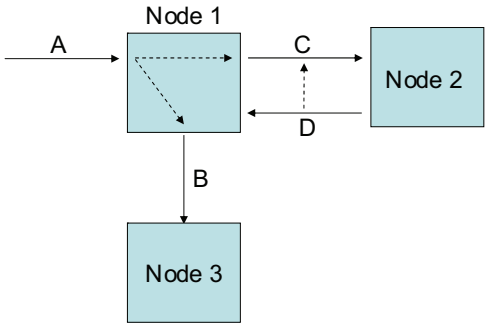


Fig. 2. Example of message linking.

Section IV-C). The message linker then extracts message IDs and links related messages according to the IDs. Finally, the diagnosis module takes the task trees as the input and output the diagnosis results.

B. Semantics Used in Rake

Given a new application, a natural question to ask is what kind of knowledge in the application is needed? First of all, we need the high-level flow information of the messages through the system. For example, for the DNS system, we need to know the recursive/iterative DNS query procedure. Furthermore, to extract the IDs from messages, certain knowledge of the message format is necessary. On the other hand, we find that Rake does not require detailed implementation knowledge. For example, it does not require the internal data structures, multi-threading usage, queue maintenance or others. Protocol specifications with complete state machines and packet format are enough to find the causal relationship of messages. Taking DNS as the example again, the knowledge in the DNS RFC is sufficient.

Consider the triggered event of a message. A message may trigger the node to communicate with other nodes, or trigger a response back (See Figure 2). We elaborate on the two cases as follows:

- *Message ID transformation*: This is for linking an outgoing message to its triggering incoming message, when the incoming message triggers further communication to other nodes (*e.g.* linking messages B and C to A in Figure 2). Often times, the incoming and outgoing messages are also related in their content, as well as in logic. Especially in many applications of query style, the query target usually is embedded in the query messages, though probably in different formats. For example, consider a chat message going from the sender to the IRC server. The IRC server simply forwards the chat message to another IRC server. In this case, taking the chat content as the message ID, this ID is kept in the incoming and outgoing messages.
- *Communication protocol*: This is for linking the query and the response messages between two nodes (*e.g.* linking message D to C in Figure 2). The query and response style is prevalent and the communication protocol itself guarantees that the sender can link its multiple queries to the responses, even with reordering. For example, the protocol can specify a query ID in the query and match the response ID with the query ID. With the knowledge of the communication protocol, Rake can link the query and response as the sender does. For example, Hadoop RPC [10] uses a unique ID to match every pair of calls and returns in one communication channel (or socket).

```
<Rake>
  <Message name="IRC_PRIVMSG">
    <Signature>
      <Protocol> TCP </Protocol>
      <Port> 6667 </Port>
      <Regex> PRIVMSG </Regex>
    </Signature>
    <Link_ID>
      <Type> Regular expression </Type>
      <Pattern> PRIVMSG\s+(.*) </Pattern>
    </Link_ID>
    <Child_ID>
      <Type> Link_ID </Type>
    </Child_ID>
    <Query_ID>
      <Type> None </Type>
    </Query_ID>
  </Message>
</Rake>
```

Fig. 3. Example of IRC XML description.

C. Rake Language to Utilize Semantics

Different application and distributed systems have different semantics. Implementing separate codes using different semantics for each application will waste much programming time on similar or identical components. Therefore, we attempt to design a unified Rake infrastructure, to which users can supply the semantics of their applications easily. We provide a simple *language* to allow users to present their semantics, and the Rake infrastructure works as an interpreter, understanding user provided semantics and using it to link messages.

1) Basic Rake Language: We choose XML to present the Rake language, which is widely used for creating custom markup languages. The Rake language is message driven, and it mainly defines properties of the messages. Take IRC as an example; assume we are interested in tracking the chat messages. We define a message named "IRC PRIVMSG" with the XML tag *<Message name="IRC PRIVMSG">* (See Figure 3). There are five basic properties for a message to specify:

- *Signature*: The signature property is used to identify the message type. Usually different messages have different format, different IDs carried and different following messages triggered. Therefore, it is necessary to provide accurate signatures to classify messages correctly. We provide a simple content-based matching mechanism.
- *Link_ID*: Link_ID is the ID that this message carries and is used to match with the parent message triggering this message. For example, in IRC chat messages, the chat content (including channel, sender and the chat words) can be used as the Link_ID, and the regular expression extracts the content out (See Figure 3).
- *Child_ID*: The Child_ID specifies the IDs that will be in the future messages triggered by this message. Note, one message may trigger several messages and the Child_ID may be a set of IDs. The Child_ID is used to match the aforementioned Link_ID. For example, when an IRC server first receives a chat message from the client, the Child_ID is

```
<Message name="DNS_Response">
  ......
  <Link_ID>
    <Type> User_Function </Type>
    <Libray> dns.so </Libray>
    <Function> Get_DNS_Dest </Function>
  </Link_ID>
  ......
</Message>
```

Fig. 4. Example of DNS XML description.

the chat content. When the IRC server delivers the message to another server, the second message's Link_ID is also the chat content. Hence the two messages can be linked together because the first one's (one of) Child_ID matches the second one's Link_ID. If the Child_ID is the same as the Link_ID of the same message, the type of Child_ID can be set to some particular value indicating the equality (*e.g.* in IRC case in Figure 3).

- *Query_ID* and *Response_ID*: The Query_ID and Response_ID pair is similar to the Link_ID and Child_ID pair. But these IDs are for the query/response or RPC style communication. Usually, based on the programming habit, the query and response can be matched by five tuple (source IP, source port, destination IP, destination port, protocol), and some user-defined query/response ID. In the IRC example, Query_ID and Response_ID are not applicable, and hence these IDs can be set to the "None" type in the XML file.

2) Signatures: We provide a content-based signature matching to classify messages. Currently, Rake supports four types of signatures: packet header field matching, expression testing, regular expression matching and user defined function. The first two types are borrowed from TCPDUMP filters.

For the packet header field matching, the user can specify some fields in IP, UDP and TCP headers. For example, the IP protocol field specifies whether the payload is UDP or TCP. The port in UDP and TCP header is also useful.

The expression matching allows users to specify some complex signature matching. For example, as shown in Figure 4, to differentiate the DNS query and response messages, we check if expression $udp[10]\&128$ is 0 or not. The eleventh byte since the UDP header [2] is the flag byte for DNS packets. The expression format is similar to that in TCPDUMP.

The regular expression matching is useful for messages with text format, *e.g.*, IRC and HTTP messages. Users can write regular expression to classify messages. In the IRC example, we simply use the regular expression "PRIVMSG" which checks if the message contains the string or not.

While we believe most signature can be expressed in the previous three pre-defined ways, there may be some special, complex signature patterns. Hence, as the last resolve, we allow users to provide their functions. The details of the user specified function will be introduced in the Section IV-C3.

Note, all the matching rules defined in the same signature tag are combined using the "And" operation, which means the message classified as this type should satisfy all the rules. If the users need to specify some alternative matching rules, they just need to write multiple Signature definitions.

3) Matching IDs: We define four types of IDs in Rake: Link_ID, Child_ID, Query_ID and Response_ID. We first describe the common properties they share, and then describe the unique properties of some of them.

a) Common properties: The common properties specify how to get the IDs from the message. The first property is TYPE, specifying the method to extract the ID. Currently, we define the following types:

- *Regular expression:* For some applications with payloads in text format, the IDs of messages can be extracted out by regular expressions. For example, a simple expression can extract the URL in the HTTP packets as the ID.
- *Block:* User can specify some blocks in the message as the ID. This may be useful for some messages with binary format. For example, for the DHT query and response messages in CoralCDN [20], the first four bytes (actually an integer) are the query and response ID.
- *User defined functions:* In some application, the IDs in a message may not be extracted from the packet payload using simple methods, *e.g.* hash functions. In Rake implementation, we utilize the dynamic (or shared) library techniques to allow user to define their own functions. Rake specifies the interfaces, and defines their input and output. The user implements the interface accordingly. In the current Rake implementation on Linux platform, we utilize the libtool [3] to call the functions in the shared library written by users. For the DNS example (See Figure 4), for the Link_ID of DNS Query messages, the type is *User_Function*, the user provided library is *dns.so* and the function is *Get_DNS_Dest*.
- *Special types:* One special type is NONE, which means some ID (usually Query_ID or Response_ID) may not exist. Another type is to reuse another ID, *e.g.*, when the Child_ID is the same as the Link_ID.

b) Special matching of Query_ID and Response_ID: As we described, the Query_ID and Response_ID matching is usually for the query/response or RPC style protocols. So implicitly, the query and the response are in the same network connection (or socket).

c) ID inheritance: In some cases, a message may need to inherit some IDs from its parent message to link its own triggered messages. This may happen in the query/response style communication. For example, in Hadoop distributed file system (Hadoop DFS), to download a file, the client will submit two sequential RPC "getFileInfo" (with the query Q_A and the response R_A) and "getBlockLocations" (with the query Q_B and the response R_B). In the two queries, the target filename can be extracted as the ID, and we can link the two queries (Q_A and Q_B). Meanwhile, the query and its response (*e.g.* Q_A and R_A) can be linked using the RPC ID in the messages. However, the correct linking should link the response of the first "getFileInfo" call (R_A) to the second query (Q_B). Unfortunately, R_A contains some file properties, but not filename. Therefore, it is desirable to let the response R_A to inherit the ID (*i.e.* filename) from its parent message, the query Q_A. In the semantics description of the response R_A, we can use the following tags to specify the inheritance:

[2]Actually this is the 3rd byte of the UDP payload.

```
<Inherit_ID name="Filename">
  Parent.Link_ID
</Inherit_ID>
<Follow_ID>
  <Type> Inherit </Type>
  <Value> Inherit_ID.Filename </Value>
</Follow_ID>
```

In this example, the message inherits its parent message's Link_ID and renamed it to be "Filename". Then the Child_ID of the message is specified to be the "Filename" inherited.

D. Practical Issues and Diagnosis

1) Software Evolution: When the application evolves, some semantics in the application may change. For example, we noticed the quick update of Hadoop, which comes up with a new version nearly every month.

To Rake, the evolution of some applications is easy to deal with. Often time the overall protocol and the basic message format does not change much, although the software implementation may update significantly. For example, Hadoop took about one year to evolve from v0.14.0 to v0.18.0, but there are no major changes in its network protocol. So the user function for parsing Hadoop messages only need to change a couple of lines. On the other side, X-Trace is not built in Hadoop's development so far. It is painful to patch X-Trace manually for the new Hadoop version, even if the new changes are not related to the network component at all. When we ask the authors for the latest source of X-Trace on Hadoop, we were told "[Hadoop] changes too quickly for us [X-Trace] to be able to migrate the current patch forward.".

2) Trace Collection: Rake takes the sniffed network traffic as the input trace. Although sniffing is a mature and widely used technique, sniffing and collecting data from a large network can still be a challenging problem.

Sniffers can be put on hosts and/or on the routers/switches. Sniffing every host seems to be lots of work, but it is actually very easy in some systems. For example, in our evaluation of CoralCDN over PlanetLab, simple scripts can start Tcpdump on every server and download all sniffed data.

a) Partial Sniffer Deployment: For various reasons, fully sniffing the whole network may be infeasible or too costly. Partial sniffer deployment degrades the power of Rake by causing the diagnosis granularity to be coarser. Fortunately, Rake deployer or administrator is able to evaluate and select a good trade-off between deployment cost and diagnosis power. A general approach to evaluate different trade-offs is to analyze the semantics graph (*e.g.* Figure 10). By deleting nodes (servers) and edges (network communications), one can evaluate the cost reduction on sniffing (*e.g.* number of sniffers saved), as well as the change on task trees and diagnosis granularity (*e.g.* completeness of partial task tree compared to full task tree with complete sniffer deployment). For example, in our Hadoop experiment (See Section VI-E), we usually adopt partial sniffer deployment. We noticed a few master servers control relatively large number of the slave nodes, and generally the slave nodes rarely talk between each other. Therefore, we choose sniffing on the few master nodes only, which covers most part of the task tree. Rake may only miss the latest layer of the tree involving the communication between slave nodes in rare cases.

b) Time synchronization: With multiple sniffing points, time synchronization is a practical problem to consider. If different nodes are not synchronized, calculated latency also includes clock offset. What's more, packet reordering may happen after merging traces from different sniffers, causing message linking algorithm to fail. Synchronization itself is a challenging research problem, and there exist lots of solutions for it. Particularly, we leverage on Paxson's technique [21] in Rake to adjust time in captured traces before merging them together. Paxson's technique [21] is simple and fast, however it assumes symmetric network delay. This assumption may work well in enterprise networks which usually are built on LANs and have very small network delay. Internet-scale deployment may violate the assumption, but packet reordering can be avoided by Paxson's technique generally. Also Rake can freely adopt more sophisticated synchronization algorithm if necessary.

c) Preprocessing Collection Data: Sniffing and sending all data packet back to a central machine may not be practical. Rake mainly use the IDs extracted from a few packets. Therefore, simple preprocessing after sniffing, and sending back only a summary of the packets is desirable. This way, the network overhead of collecting traces is negligible. For example, in CoralCDN, we only need to extract 20 bytes from one large packet.

d) Encryption and Compression of Packets: Encryption and compression may prevent Rake from understanding the semantics of the communication. This is the common problem of many security applications such as deep packet inspection. While this is true, we would not worry about it much due to the following reasons:

- Many popular distributed systems such as DNS systems, MSN and the search system do not encrypt or compress their communication. The reasons for not using encryption are diverse. For example, the data communications need not be secure (*e.g.* DNS and IRC), or the system is isolated from the external Internet, and encryption adds additional overhead costs (*e.g.* Search system, MSN core network).
- There may still be approaches to overcome the encryption problem. For example, if the communication is encrypted using IPSec, it is possible to interposition between the application and the dynamic library of IPSec to extract the raw data.

V. Diagnosis with Task trees

Diagnosing large distributed systems is non-trivial even if accurate task trees are at hand. Since diagnosis is not the focus of this paper, we only describe some simple diagnosis algorithms that are used in our evaluations.

A. Diagnosis Using Processing Time

In many times sensitive applications such as web search, IRC and CDN, the processing time may be a good indication of the performance of the nodes. For example, if an index server in a search system has an elevated processing time for search queries on average, it is quite likely that this server has performance problems and needs detailed diagnosis, such as CPU/disk load investigation.

When the messages in a task tree are linked together, it is easy to calculate the time interval between linked messages using the timestamp in the messages. These time intervals can be viewed as the processing time. Therefore, task trees are very helpful for such time sensitive applications. In our evaluation on CoralCDN, we use the processing time for diagnosis.

B. Diagnosis with User Knowledge

In parallel computing systems such as Hadoop, it is not appropriate to simply use the processing time to diagnose the system. The normal interval between two linked messages can be quite volatile, *e.g.*, varying from seconds to minutes. For such applications, it is challenging to find a single diagnosis algorithm for all applications even if sophisticated machine learning approaches are used. Instead of struggling with the challenging diagnosis algorithm design, we apply user knowledge in the Rake system to make the diagnosis job much easier.

While users provide the semantics of the system, the user can also provide the expected processing time or maximum normal processing time as well. In Rake language, for some time sensitive messages, the user may use the "Diagnose" tag to specify the expected maximum processing time (an example of Hadoop is shown below). While generating task trees, Rake also checks if the processing time of the messages is over the maximum processing time or not. If it is true, Rake generates warnings for the unexpectedly long processing time. In the evaluation of Hadoop (See Section VI-E), this simple diagnosis approach actually helps us identify the *Slow master node* problem.

```
<Message name="Hadoop_HeartbeatResponse">
  ......
  <Diagnose>
    <MaxProcessTime> 1 </MaxProcessTime>
  </Diagnose>
</Message>
```

VI. Evaluation

In this section, we first talk about our implementation experience of Rake on different applications. Then we describe the extensive experiments on some distributed systems.

A. Implementation

We implemented Rake in C++ on the Linux platform. The Rake framework requires about 3000 lines of code. The XML configuration files for applications usually have hundreds of lines. For Hadoop, the message parsing and ID extracting rely on the dynamic library, which are implemented in around 2000 C++ lines and can be found at [26]. For CoralCDN, DNS and IRC, usually less than 300 lines are enough for user provided library.

1) Interface between Rake and User Defined Functions: In our Rake implementation, we utilize the libtool [3] to call the functions in the dynamic library written by users. Users can write functions in any language and compile it into standard Linux shared library. Rake defines two interfaces, one for determining the type of the message and the other for extracting ID sets (P_m or F_m). For example, the interface for message type takes the packet payload as the input and then outputs a boolean to tell if the message is of a particular type or not. The XML configuration files specify the name of the user library and the function names, and hence Rake can dynamically load the library and call them.

B. Experience of Applying Rake to Applications

Ideally the Rake users are the application designer but this may not always be the case. Next, we describe our experiences on applying Rake to IRC, DNS, CoralCDN and Hadoop, as only non-designers.

1) Task Trees Discovery: For network protocols such as DNS and IRC, we find that it is very convenient to simply study the RFCs of them. The RFCs usually clearly describe the task trees of the protocols and defines the message format. The level of details of RFCs is just what Rake needs. No software programming details are required.

For CoralCDN and Hadoop that are not well documented, the semantics study is a little bit more troublesome. For CoralCDN, we mainly rely on source code reading to understand its potential task trees. But we only focus on the network module of CoralCDN, ignoring other modules such as cache management. On the other hand, for Hadoop, because most packets are in plain text, we find it is very helpful to learn the message flows from the network traffic dump.

2) Task Trees Construction: In our real experience, we find it is quite straightforward to find out the IDs used to link messages. This may be because applications we studied are mostly query or task driven applications. The query target (*e.g.* query host name in DNS and URL in CoralCDN) or its transformation (*e.g.* hashed value) is embedded in most messages of the task tree. For the task based applications (such as Hadoop), there is a built-in task ID that is contained in most of the messages in the same task to differentiate concurrent jobs. Therefore, finding the IDs to link messages becomes a simple job of learning the packet format of the messages.

C. Evaluation Methodology

We evaluated two large distributed systems to show the feasibility and accuracy of our Rake: (i) CoralCDN – Coral content distribution network, and (ii) Hadoop – an open source distributed cluster computing platform. Meanwhile, we also analyzed the accuracy of task tree extraction of Rake on the web search system of a top search provider. Similar accuracy analysis of the IRC system is omitted due to space limit.

With the ground truth provided by a log-based approach, we compared our Rake algorithm to previous studies using the black-box approach WAP5 [22]. Since WAP5 does not work for computation intensive applications such Hadoop, as the gap between messages are general very large and the time correlation fades quickly, we mainly compare WAP5 with Rake in the evaluation of CoralCDN. For Hadoop, we show how we can use Rake to find out some design problems and performance problems, which cannot be identified by Hadoop's own tools or logs.

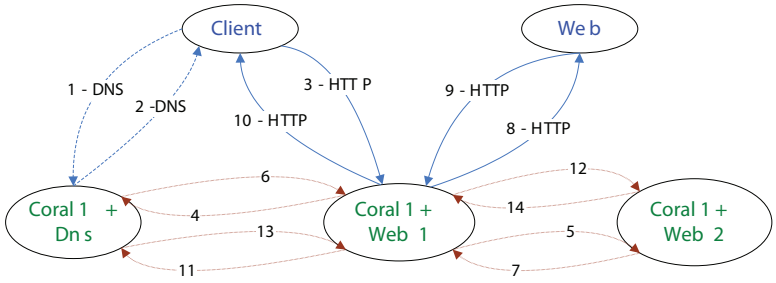


Fig. 5. Coral execution path from Rake.

Fig. 6. Semantic information flow in CoralCDN system .

D. Evaluation on CoralCDN

1) CoralCDN Background: CoralCDN is a decentralized peer-to-peer web-content distribution network. Figure 5 shows a detailed example of execution path of messages in CoralCDN. The numbers in the path represent the sequence number of messages linked by Rake algorithm.

2) Semantics used in Diagnosis of CoralCDN: Figure 6 shows the semantic information flowing through the coral system. The URL requested by client in HTTP request serves as the intrinsic ID to link all the related messages in a task tree. Coral hashes the URL requested and converts it into a 20 byte sha1 hash ID called KeyID. This KeyID serves as the ID (both Link_ID and Child_ID) for all the later DHT communication, and it is used to link the HTTP and DHT query messages. Each pair of DHT query and response messages share a unique MsgID, serving as a linking point.

3) Experiment Setup: We deployed CoralCDN on PlanetLab, using the public CoralCDN source code [20]. In our current deployment, 25 PlanetLab nodes are installed with Coral daemons and web server daemons. However, because PlanetLab nodes are not always available and sometimes heavily overloaded, usually we have about 18 Coral nodes in our experiments. One of our university server acts as the DNS server, handling all the customized DNS requests. The source code of Rake in CoralCDN can be found at [26].

We replayed two different datasets of about half an hour's duration on CoralCDN. These two different data-sets are:

- URLSet1 – The sniffed network traffic of Tsinghua University in China. We replayed a total of 21 GB HTTP traces collected from the university on coral CDN.
- URLSet2 – The sanitized access log from [23]. The logs are sanitized and each line contains information of a HTTP connection. We replayed a total of about 20,000 HTTP connections.

4) Message Linking Accuracy: Due to the lack of ground truth for CoralCDN task trees, we rely on a log-based approach which uses the CoralCDN logs to estimate the accuracy of task tree extraction of both Rake and WAP5. CoralCDN writes logs when some important events occur, *e.g.*, receiving a HTTP request, making a DHT query and starting download from the real web servers. CoralCDN does not log any DHT message at all, making it impossible to diagnose CoralCDN solely using the logs.

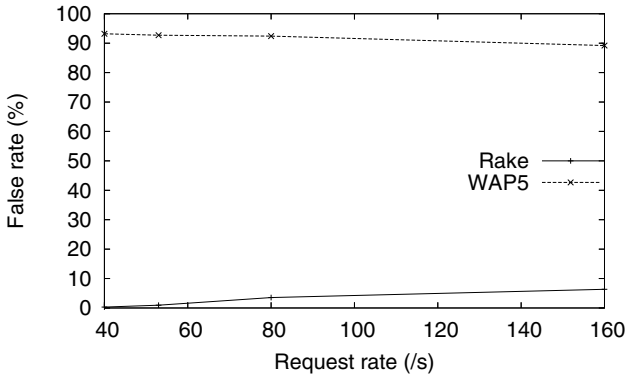


Fig. 7. False rate of WAP5 vs. Rake.

By modifying the CoralCDN source code (for CoralCDN, this approach is invasive), we enhance the CoralCDN logs so that we can link the events for the same HTTP requests in the log into event trees. An event tree is simpler than the corresponding message-level task tree. Typically an event tree has four nodes, receiving HTTP request, starting DHT query, start downloading from real web server and sending the webpage to the client. To evaluate the accuracy of Rake and WAP5, we compare the tree structures from Rake and WAP5 with the event trees generated from logs. Basically, using the timestamp and URLs in the HTTP request, we first identify the event trees and their corresponding task trees (from Rake or WAP5). Next, given an event tree and its corresponding task tree, for each node in the event tree, we check if we can find a corresponding node in the task tree. For example, for the "starting DHT query" message in an event tree, we check if there are DHT query messages and response messages with the same DHT ID in the task tree. If any node in the event tree is missing a corresponding node in the task tree, the match of the event tree and the task tree is false. Finally, we count all the false cases and use the false rate to evaluate the accuracy of task tree extraction for both Rake and WAP5.

Figure 7 shows the false rate of Rake and WAP5. Generally, Rake is very accurate even when the HTTP request load is very high, *e.g.*, 160 requests/second. The higher request load

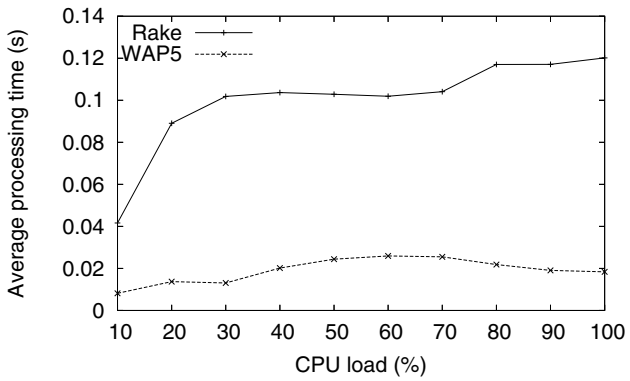


Fig. 8. Processing time of WAP5 *vs* Rake.

causes the higher ambiguity, which hence affect the accuracy of Rake. On the other hand, WAP5 has very low accuracy, and the false rate is around 90%. Actually given certain high HTTP request load (*e.g.* 40 requests/second), the messages of different task trees interleave and time correlation is really not a good way to link messages in task trees. This suggests that WAP5 is better used in low load scenarios such as finding performance bugs due to design or coding errors [22].

5) Diagnosis Ability: We calculate the processing time of each coral node using both algorithms, WAP5 and Rake. We take the difference of receiving and sending time for each pair of linked messages as the processing time. Since both sending and receiving timestamps are local to the node, we do not have synchronization problem. For both Rake and WAP5, we calculate the mean processing time for the HTTP and DHT request seen under the HTTP request tree. We compare only the processing time of the linked messages so that we can have fair comparison between WAP5 and Rake.

We run CoralCDN on multiple PlanetLab nodes and log the CPU load of these nodes. We conjecture that the CPU load may correlate with the real processing time, because naturally one may think a busy machine should be slow. However, we find that processing time calculated from neither Rake nor WAP5 correlate with the CPU load. One reason can be the heterogeneity of PlanetLab nodes and load. Therefore, we further conducted a controlled experiments on a single Coral node installed in one of our own server which we have full control on.

The controlled node runs Coral server daemon solely at first. Then we use Lookbusy [14] to keep the CPU(s) at the chosen utilization level. Figure 8 shows the processing time calculated by Rake and WAP5 under different CPU loads. As for Rake, we can see the processing time increases significantly when the CPU load increases from 10% to 30%, and then the line becomes quite flat. This phenomenon is probably because Coral itself is not a computational intensive program. The increase of processing time may be mainly caused by the process scheduling of the operating system. When the CPU load increases while it is still low, the Coral program needs more and more time to get back CPU. To make CPU busier and busier, Lookbusy does not increase the number of processes, but reduces the sleeping time of its processes. Therefore, when the CPU utilization is high (*e.g.* over 40%), Coral process should have high priority and switch back to running status quickly and this is not quite affected by the CPU load.

On the other hand, the processing time calculated by WAP5 increases slowly and then drops a little bit as the CPU load increases. And obviously, the processing time from WAP5 is much smaller than that of Rake. WAP5 underestimate the processing time because it always attempts to link the closest messages which might not be related. Actually lots of unrelated control messages are also linked in the HTTP request tree by WAP5. Since these messages are close in time with other messages, the overall processing time in WAP5 is lower than the actual time.

E. Evaluation on Hadoop

In this section, we present an example to use Rake to diagnose Hadoop [10], an open-source parallel computing framework, which is widely used by many companies such as Yahoo! and Amazon.

1) Hadoop Background: Hadoop [10] is an open-source implementation of Google's MapReduce [16].

Hadoop enables distributed and parallel computation by decomposing a massive job into smaller tasks and a massive data-set into smaller partitions. Each task processes a different partition of data in parallel on different machines. Hadoop abstracts two types of tasks, Map task and Reduce tasks. Hadoop uses the Hadoop Distributed File System (HDFS), an implementation of Google Filesystem, 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 has a master-slave architecture for both HDFS and the job computing. Usually there are a couple of master hosts and multiple slave hosts. For HDFS, a NameNode (with a backup) manages the HDFS file indexing and processes the file access from clients, and the slave nodes act as DataNode to store the file contents. For computing, the JobTracker schedules and manages all of the tasks belonging to a running job and the tasks are executed finally on the slave nodes, tracked by TaskTrackers on each slave node. Note a Hadoop job involves data file uploading and downloading as well as long time computation on data. Usually there are a large number of data packets for file transferring but infrequent job status report messages. We find WAP5 generally cannot find meaningful task trees and hence do not report WAP5's diagnosis results in the following evaluation.

Hadoop use log4j to log useful information. There are different levels of log, such as ERROR, WARNING, INFO and DEBUG. To save the log data size, DEBUG level logs are not enabled by default, and Hadoop only logs some significant events related to job progress. Even in the DEBUG level, the log is far from the granularity of packet level. Therefore, Hadoop logs are generally data-mined in the event or state level (*e.g.* [24]).

2) Semantics used in Diagnosis of Hadoop: In this section, we use two examples to show the semantics of Hadoop utilized to link messages into task trees. Hadoop uses a general term IPC (Inter-Process Communication) call to denote their remote procedure call. In this paper, we will use IPC call and RPC interchangeably.

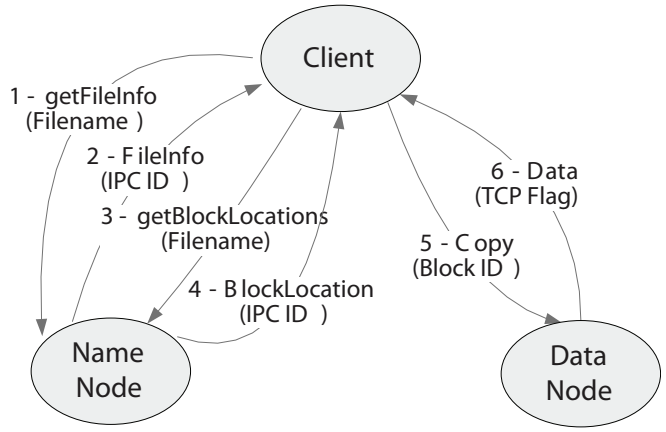


Fig. 9. Semantics of Hadoop DFS - Get operation.

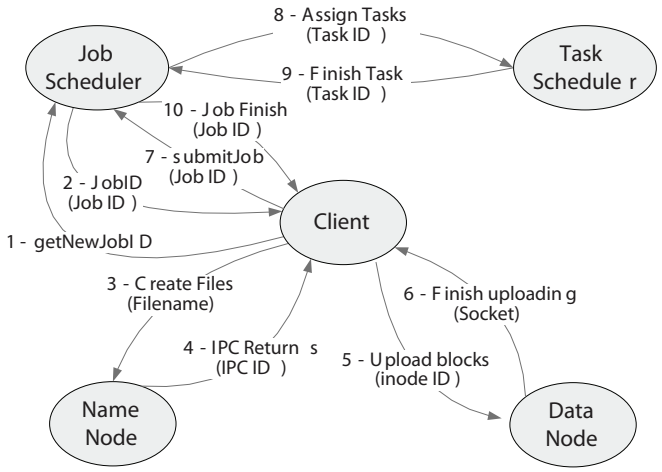


Fig. 10. Semantics of Hadoop - Grep operation.

a) Hadoop DFS - Get Operation: Figure 9 shows the message flow as well as the semantics that can be used to link the messages together. First, the client will send a IPC call "getFileInfo" to the NameNode to get the status of the files, *e.g.*, existence, its owner and group information. Note the file name of the target file can be used as the ID to link following messages. File name is not a unique ID in some cases, as different clients may get the same file from DFS at the same time. To further reduce the ambiguity, we also add socket information (client IP and port) to make the ID to be unique. The NameNode returns the status of the file status via the IPC mechanism. As we described above, remote procedure calls and returns are matched via the unique IPC IDs, and the IPC ID is used to link messages in this case. Next, the client uses a second IPC call "getBlockLocations" to get the location of the blocks of the target file, including the inode IDs and the hostname of the datanodes storing the blocks. In the IPC call of "getBlockLocations", it also contains the file name which is used to link with the previous "getFileInfo" IPC call. Then the NameNode replies with the block information. This time, the Link_IDs generated are the inode IDs, which should be unique in the DFS system. Last, the client sends the "Copy" command to the DataNode to download the file blocks, presenting the inode IDs. When the TCP session of downloading ends (normally or exceptionally), the last message (with TCP FIN or RST flag) is linked to the beginning of the downloading. In a word, to link the messages in the *Get* operation, the polymorphic IDs are first the file name, then the IPC IDs and inode IDs, and finally the socket tuples.

b) Hadoop Job Running Operation: Running a job in Hadoop is much more complex than simple DFS operations. Actually, during the running of a job, many files are created and read. Due to space limit, we only briefly introduce the semantics Rake can utilize to link the whole job running process, omitting many details.

Figure 10 shows the general steps of a job running. Note each step in the graph may contain multiple sub-steps and Rake does link the messages in these detailed sub-steps. First, the client requests a new job via the "getNewJobID" IPC call (Step 1). In the reply from the JobTracker, a JOB ID is returned, which is one of the basic ID that Rake uses to link the whole task tree (Step 2). Then the client uploads a couple of files using DFS operations, *e.g.*, the user code and configuration files (Steps 3∼6). Note the file names of these files all have the fixed format and contain the JOB ID as part of the file names. This is how Rake link the DFS uploading operations into this task. After uploading the job files, the client submits the job to the JobTracker, including the JOB ID in the message. Then the JobTracker assigns different Map and Reduce tasks to several different slave nodes. In each assignment, there is a TASK ID, which contains the JOB ID and some additional information, such as the ID to differentiate this task to others of the same job and the type of the task (Map or Reduce). The JOB ID is used to link the assignments to the job (*e.g.* linking Step 8 to step 7), and the longer TASK ID is used to link the actions in the task (*e.g.* linking Step 9 to step 8).

3) Experiment Setup: We deployed the Hadoop v0.18.1 on a small cluster of four machines in our department as well as 10 PlanetLab hosts. One of our machine acts as the master (both NameNode and JobTracker) and the other nodes act as slaves (DataNode and TaskTracker). We generate two candidate workloads, which are commonly used to benchmark Hadoop:

- *Reader*: read different size of files from Hadoop DFS
- *Grep*: grep target strings from files in Hadoop DFS

In the controlled experiments, we manually inject some failures to some nodes to cause the node to be very slow, as we did in CoralCDN experiments.

4) Evaluation Results:

a) Accuracy: We manually checked message linking results of Rake and found that Rake can link the messages without any error. The socket information helps to solve the ambiguity that may potentially caused by downloading the same file simultaneously.

b) Problem Diagnosed with Linked Message: In this section, we will discuss how to diagnose the problems using our linked messages. We find an existing problem, an abused RPC problem in Hadoop DFS reader. Meanwhile we inject some problems in Hadoop MapReduce system and use Rake to find the problems we injected.

Abused RPCs in DFS Reader: In this experiment, we use Rake to inspect the *Get* operation of Hadoop DFS. In each

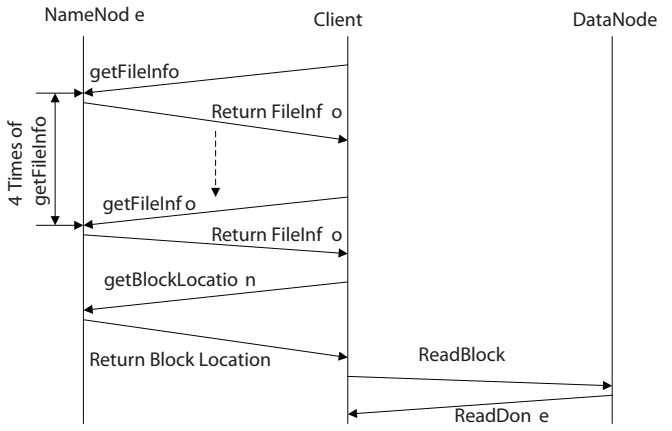


Fig. 11. Abused RPCs in Hadoop.

single run of the experiment, two Hadoop clients download the same file from the DFS system simultaneously and we conducted the experiments five times.

Figure 11 shows the messages linked by Rake. Surprisingly, it shows that the RPC "getFileInfo" is called four times with the same parameter (*i.e.* the file name). By inspecting the source code of Hadoop, we find that the problem indeed exists and Hadoop redundantly call the same function four times. The reason may lie in the convenience of the RPCs, and the programmer may not realize that he makes some RPCs. In this case, the RPC "getFileInfo" is called in function "getFileStatus", which is further called in other functions such as "isDirectory" or "isFile". For example, in Hadoop implementation, both "isDirectory" and "isFile" are called to determine the file type and hence cause two RPCs. To the best of our knowledge, we are the first one to find this problem.

Injected Problem in Hadoop MapReduce Grep Job: In these experiments, we run the general *Grep* application on some middle size files of about 200MB. The data file is partitioned into three blocks and hence the job has three Map tasks and one Reduce tasks. We specifically make one of the nodes (either the master node or slave node) to be slow and check if Rake can help on diagnosing the slow nodes.

Figure 12 shows an example that Rake outputs the general running time of each steps as well as some substeps zoomed in. Note Hadoop itself has a web based visualization which shows the running time of each Map and Reduce task, which is in very coarse level.

Case 1: Slow slave node. In this case, some Map or Reduce task runs slowly. Both Rake and Hadoop web visualization can clearly show the running time of the tasks, but the running time cannot directly reflect the status of the slave nodes, slow or fast. For example, a Map task can be fast simply because it processes a small block (*e.g.*the last block of a data file). Further data-mining approaches such as the distMatrix [24] can be used to diagnose more accurately, but this is not our focus in this paper.

Case 2: Slow master node. The problem is more interesting when the master node is made slow. Unlike the slow slave node case, Hadoop's native web visualization cannot really give implication on the problem, while Rake can potentially show some symptoms of the slowness of the master node.

When the master node is slow, the RPCs may become very

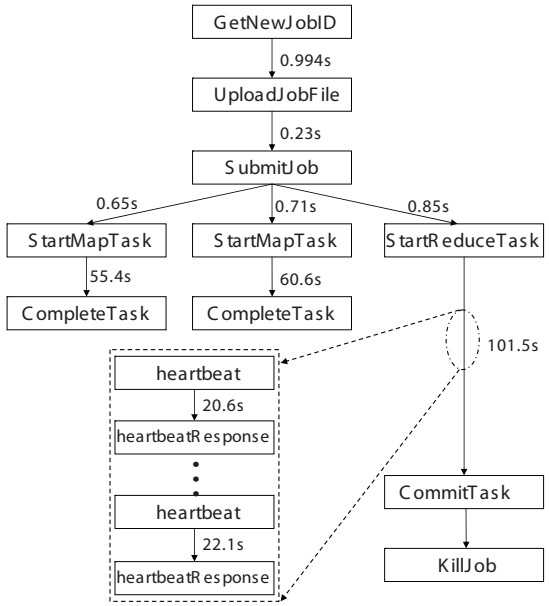


Fig. 12. Running time of Hadoop steps.

slow and hence cause the whole job to be slow. For example, in the experiment without injected failures, a Map task takes about 20 seconds. In one controlled experiment, we found all three Map tasks took about 50 seconds. However, we only injected fault into the master node in the experiment, while the results from Hadoop web tool might imply that the slave nodes are slow. By looking into the time consumption in the message layer via Rake, we can clearly see that when the slave nodes reported the running status of the Map tasks to the master node, master node took about 20 seconds to reply back. The slave node reports the different stage of the Map task to the master node,*e.g. BEGINNING* stage, multiple *RUNNING* stage and *SUCCEEDED* stage. The RPCs are blocked due to the master's slow response and finally it seemed the Map task was finished slowly. Rake can clearly identify the time between the RPCs and responses and hence is able to disclose the problem in the master node. It is worth mentioning that it is hard to identify the problem with Hadoop's own logs because Hadoop does not log every heatbeat or their response messages.

F. Internet Relay Chat (IRC) System

Internet Relay Chat (IRC) is a form of real-time Internet chat or synchronous conferencing. It is mainly designed for group communication in discussion forums called channels, but also allows one-to-one communication via private message. Generally, in IRC, a group of servers cooperatively serve a large number of clients, and an IRC chat message may cross multiple servers and paths to the other users in the same channel.

1) Semantic used in Diagnosis of IRC Servers: The messages in IRC servers are relayed in plain-text in client-to-server or server-to-server communication. While there are extended IRC to support encrypted communication, we find in most famous IRC networks such as EFnet [2] and DALnet [1] still relay messages of plain-tex between the server. Therefore, the chat content itself is a non-transforming ID which can

mark the message flow through the IRC system. Actually we previously use IRC as the example to describe Rake and Figure 3 shows the Rake configuration file for IRC to submit its semantics.

2) Message Linking Accuracy: Similarly, we did the theoretical analysis on the ambiguity of Rake on IRC system because of lack of real traces. We collected and analyzed one day of data from a channel in a IRC server of EFnet [2]. We connected server irc.servercentral.net and collected 40,222 lines of messages passed between the clients and servers on channel "fw". We filtered the control messages (e.g. someone joins or leaves the channel) from the messages. Two identical chat messages are ambiguous if they come within the time less than the processing time of message. We find that only 20 (.047%) requests out of 40,222 are duplicate. Thus IRC system has very low ambiguity.

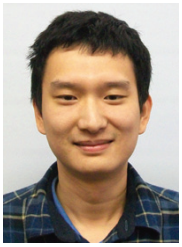
VII. Conclusions

In this paper, we propose Rake, a semantics assisted gray-box tracing framework for distributed system diagnosis. The key idea is that in most cases, related messages can be linked together by extracting some (transformed) IDs based on application semantics. We achieve aforementioned three goals. *1. non-invasiveness:* we do not need changing any client/server applications/OSes; *2. applicability:* we use several popular distributed systems to demonstrate why Rake is widely applicable ; and *3. accuracy:* we designed and implemented Rake, evaluate it over CoralCDN and Hadoop. The results showed that the accuracy is much better that of the black-box approaches and comparable to those of the white-box schemes.

References

[1] DALnet - The DALnet IRC Network, http://www.dal.net/.
[2] EFnet - The Original IRC Network, http://www.efnet.org/.
[3] Gnu libtool - the gnu portable library tool, http://www.gnu.org/software/libtool.
[4] HP openview, http://www.openview.hp.com/.
[5] IBM Tivoli, http://www.ibm.com/software/tivoli/.
[6] Microsoft Operations Manager, http://www.microsoft.com/mom/.
[7] E. Ackerman, "Yahoo's hadoop software transforming the way data is analyzed," http://www.siliconvalley.com/news/ci_10897240?nclick_check=1.
[8] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *2003 SOSP*.
[9] A. Anandkumar, B. Bisdikian, and D. Agrawal, "Tracking in a spaghetti bowl: monitoring transactions using footprints," in *2008 ACM SIGMETRICS*.
[10] Apache, Hadoop, http://lucene.apache.org/hadoop/.
[11] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, "Information and control in gray-box systems," in *2001 SOSP*.
[12] 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 *2007 ACM SIGCOMM*.
[13] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *2004 OSDI*.
[14] D. Carraway, Lookbusy, http://devin.com/lookbusy/.
[15] X. Chen, et al., "Automating network application dependency discovery: experiences, limitations, and new solutions," in *2008 OSDI*.
[16] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *2004 OSDI*.
[17] H. Dreger, et al., "Dynamic application-layer protocol analysis for network intrusion detection," in *2006 USENIX Security Symposium*.
[18] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, X-Trace: a pervasive network tracing framework," in *2007 NSDI*.
[19] A. Fox and E. Brewer, "Path-based failure and evolution management," in *2004 NSDI*.
[20] M. J. Freedman, E. Freudenthal, and D. Mazires, "Democratizing content publication with coral," in *2004 NSDI*.
[21] V. Paxson, "On calibrating measurements of packet transit times," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, pp, 11–21, 1998.
[22] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat, "WAP5: black-box performance debugging for wide-area systems," in *2006 WWW*.
[23] National Lab of Applied Network Research, ftp://ircache.nlanr.net/Traces/.
[24] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "SALSA: analyzing logs as state machines," in *2008 Usenix Workshop on the Analysis of System Logs*.
[25] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie, "High speed and robust event correlation," in *IEEE Commun. Mag.*, 1996.
[26] Y. Zhao, Y. Cao, A/ Goyal, Y. Chen, and M. Zhang, Rake web site http://list.cs.northwestern.edu/Rake/.

Yao Zhao is a software engineer in Site Reliability Engineering of Google. He got his master degree in Computer Science at Tsinghua Univ at 2001, and his Ph.D. in of Electrical Engineering and Computer Science at Northwestern University in 2009. After graduation, he worked for Bell labs and IPD of Alcatel-Lucent for three years. His research interests include network measurement, monitoring and security, wireless ad-hoc and sensor networks.



Yinzhi Cao is a Ph.D. candidate in of Electrical Engineering and Computer Science at Northwestern University. He graduates from Tsinghua University in China with a bachelor degree in electronic engineering. His research focuses on web security, especially client-side browser security and JavaScript language security.

Yan Chen is an Associate Professor in the Department of Electrical Engineering and Computer Science at Northwestern University, Evanston, IL. He got his Ph.D. in Computer Science at University of California at Berkeley in 2003. His research interests include network security, measurement and diagnosis for large scale networks and distributed systems. He won the Department of Energy (DoE) Early CAREER award in 2005, the Department of Defense (DoD) Young Investigator Award in 2007, and the Microsoft Trustworthy Computing Awards in 2004 and 2005 with his colleagues. Based on Google Scholar, his papers have been cited for over 4,000 times.

Ming Zhang has been a researcher in the Networking Research Group at Microsoft Research since 2005. His research interests lie in designing, building, and managing cloud computing infrastructure, data center networks, and mobile systems. He has published over 30 scientific papers in top-tier systems and networking conferences including OSDI, SIGCOMM, NSDI, MobiSys, and Oakland. He received his M.A. and Ph.D. in Computer Science from Princeton University in 2003 and 2005 respectively and his B.S. in Computer Science from Nanjing University in 1999.

Anup Goyal is a seasoned software engineer currently on the search team at Google. Prior to this role he worked as a senior software engineer at Yahoo on various products including search, cloud computing and display advertising. Anup graduated in computer science from Indian Institute of Technology (IIT Kharagpur) and went on complete his masters from Northwestern University.