Rake: Semantics Assisted Network-based Tracing Framework

Yao Zhao, Yinzhi Cao, Anup Goyal, Yan Chen and Ming Zhang†
Northwestern University, Evanston IL, USA
†Microsoft Research, Redmond, USA

***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. In this paper, we present a novel semantics-assisted gray-box tracing approach, called Rake, which can accurately trace individual request from 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 provide necessary semantics of their applications while the core Rake component can be reused. We evaluate Rake using a few popular distributed applications, including web search, distributed computing cluster, content provider network, and online chatting. Our results demonstrate that 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 to reveal several previously unknown issues that may lead to performance degradation, including an IPC (Inter-Process Communication) abusing problem.**

1. INTRODUCTION

Large-scale distributed system and cloud computing have undergone unprecedented growth in recent years. Parallel computing platform, such as Hadoop [8], enables Yahoo to search through the entire Library of Congress in fewer than 30 seconds [5]. 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 [2]–[4, 24] 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 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, the task of 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 in [6, 7, 18, 19, 21]. An ideal approach to extracting task tree should possess the following properties:

- **Accuracy.** It can accurately identify the causality between different messages, hence further locate the problematic nodes.
- **Non-invasiveness.** Existing approaches such as X-Trace [18] and Pinpoint [19] require modifications to the OS, middleware, and/or applications. While these approaches provide accurate tracing results, their invasiveness prevents them from being widely adopted for two reasons. First, since they require source code instrumentation, they are inapplicable to legacy applications or third party applications. Second, even if source code is available, instrumenting a complicated, continually-evolving application can be time-consuming and error-prone.
- **Scalability.** Modern distributed systems such as Web Search may comprise thousands of machines and receive thousands of queries per second. An approach should be able to handle such systems efficiently.
- **Applicability.** It is desirable to be applicable to all applications. Due to various practical trade-off such as model limitation, accuracy requirement and instrumenting or deployment overhead, no single approach can be applicable to all systems. One goal of this paper is actually to expand the family of tracing and

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

diagnosing approaches with a handy approach suitable for many distributed systems.

Most previous approaches for tracing task trees can be classified into either the black-box ones or white-box ones. A white-box approach usually needs to insert certain unique IDs into the messages by instrumenting the application, the middleware, or the OS [18, 19]. In contrast, a black-box approach does not need any instrumentation or understanding of application's internal structure or semantics [6, 7, 21]. Instead, it only relies on temporal correlation between messages. While a black-box approach is non-invasive, it tends to have limited accuracy. For instance, even with extensive training period, [10] may still mistakenly infer certain false dependencies. This motivates us to develop a novel, "gray-box" approach for task tree extraction which is both non-invasive and accurate.

In this paper, we propose Rake, a semantics assisted gray-box approach to understand the distributed system's execution and further locate performance problems and failures. The basic idea stands on the observation that in messages of the same task there are polymorphic "IDs" that can be dug out and utilized to link the messages into a task tree. In order to design Rake as a general performance diagnosis tool, we made the following contributions:

- We propose the novel semantics assisted diagnosis approach which is non-invasive because Rake needs no modification to applications, middleware or the OS. Rake is also a gray-box approach, requiring limited semantics instead of implementation details.
- 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. 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 2, and introduce Rake in Sections 4. Practical issues such as trace collection are discussed in Section 5. We present evaluation results in Section 6 and conclude in Section 7.

2. Related Work

Significant recent research has been done on debugging or troubleshooting service problems in the view of the whole distributed systems. Many of these systems model the dependencies between components with the task tree (which is called either causal path or execution path) [6, 11, 19, 21]. A task tree embodies control flows, resources, and performance characteristics associated with servicing a request.

1. Task Tree Extraction Approaches

1) Black-box approaches: Project 5 [6] attempts to identify execution paths of messages as passively as possible 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 [21] to improve Project 5. WAP5 also uses time correlation between incoming and outgoing messages on a node to link messages with probabilities. A simple exponential function is introduced to estimate the linking probability between two packets, and therefore close messages are linked with high probabilities. Anandkumar *et al.* studied the linking of transaction footprints and reduce the maximum likelihood rule to the minimum weight bipartite matching problem [7]. These black-box based approaches [6, 7, 21] 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.

The most recent research work, Sherlock [10], considers an aggregated dependency graph instead of individual task trees. A dependency graph models dependent relationship among components in the network. Active measurements are then conducted to obtain users' experiences on the latency of different network applications. The follow-up work of Sherlock, Orion [14], uses the delay spike based analysis to further increase the accuracy of discovered dependencies.

2) 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 should be instrumented to invoke X-Trace. However, for a large distributed system using many softwares from different vendors 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.

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

TABLE I Classification of management and diagnosis systems.

3) 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. Probably Magpie [11] is the closest related work to Rake. Magpie 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 [23] is another log-based approach which relies on the application logs to derive state-machine views of the system's execution. In comparison, Rake generally uses network sniffed traffic as the input, while Magpie and SALSA rely on the event logs generated by application and the operating system. Such approaches may suffer from many problems as mentioned in Section 1.

4) Intrusiveness Classification: Table I shows a classification of previous diagnosis and workload extraction systems. Sherlock [10] and Project5 [6] only use network sniffed traces, which has no modification to the OS and applications. Reynolds *et al.* develops their own library to collect OS level traces such as system calls in [21], 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 extremes, either very invasive white-box or not quite 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.

2. *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 [9], Arpaci-Dusseau *et al.* studied how to treat 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 security area, such as in network intrusion system [16] for packet classification.

3. Problem Definition

It is desirable to achieve the accuracy in tracing task trees as white-box approaches (such as X-Trace [18]) while not injecting IDs by patch the whole system. This motivates us to study the possibility to dig out meaningful IDs out network messages. We designed Rake based on the following key observation:

Generally, in distributed system implementations, there are no explicit unique ID 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.

For example, consider the recursive DNS query process. One can take the DNS query target as the ID of the query and response messages, and hence all the DNS messages for the same query task can be easily connected. With the knowledge on the format of the DNS query/response messages and the recursive DNS query process, a DNS task tree can be extracted from mixed network traffic and rebuilt easily.

1. *Mathematical Model*

A task tree is defined as a tree $G(V, E)$. Each node in V is a message and an edge $v_1 \rightarrow v_2$ means message v_2 is the result of processing v_1. In this tree model, a child message is triggered by only one parent message. This simple model works for many applications and adopted by most diagnosis approaches [6, 10, 21]. It will be our future work to extend Rake for more general models.

To extract the task trees out of a mixed set of messages, a few steps are enough:

1) Determine message type by signatures. Assume function $S(m)$ uses signature matching to determine the type of message m. For example, DNS messages can be simply identified by the port number of 53.
2) Extract the ID set of message m. Let set $P_m = f_{S(m)}(m)$, where f_t extracts the IDs of messages of type t. For a DNS response message, we can simply use the host name of the query as the ID.
3) Extract the ID set of following messages which are trigger by message m: $F_m = g_{S(m)}(m)$ and g_t infers the expected IDs of the messages to be triggered by m. For example, F_m of the DNS query can be the query host name, which will be the ID of the DNS response message.
4) Join of ID sets to find causality: linking messages m_1 and m_2 if $P_{m_1} \cap F_{m_2} \neq \emptyset$. If a message has multiple parents, pick the closest one in time. In the DNS example, both the F_m of the DNS query and the P_m of the corresponding DNS response message are the same, so these two messages are linked.

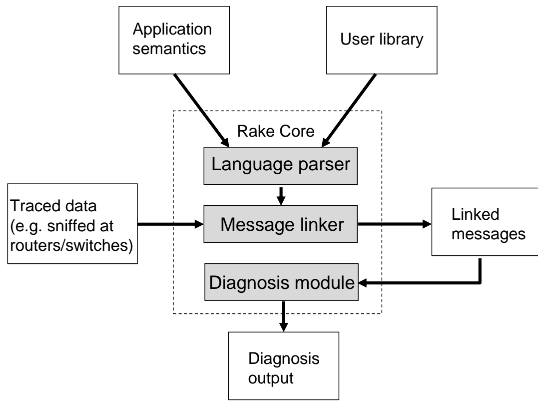

Fig. 1. Architecture of Rake

2. Challenges

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

- The simple model dose not tell us what are the functions f and g to extract IDs out of messages. How can we figure out these functions using the semantics? Furthermore, are there general rules?
- Different applications have different semantics. How can we design Rake to be general and easily adopted by different applications with different semantics?
- How to make Rake accurate and efficient?

4. 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.

1. 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 Section 4.3). 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.

2. Semantics Used in Rake

Given a new application, a natural question to ask is what kind of knowledge in the application is needed? As shown in the mathematic model, only semantics that help us find the functions S, f and g to extract IDs of messages are of interest. So first of all, we need the high-level flow information of the messages through

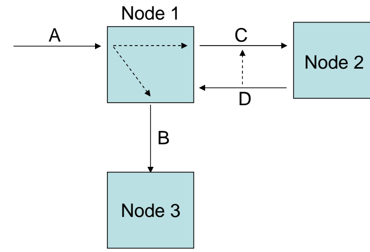

Fig. 2. Example of message linking

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 Rake does not require very 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 transforming*: 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 IPC (inter-process communication) [8] uses a unique ID to match every pair of calls and returns in one communication channel (or socket).

3. 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 the user semantics and using them to link messages.

1) Basic Rake Language: Since XML is a widely used, general-purpose specification for creating custom markup languages, we choose XML to present the Rake language. The Rake language is designed to generalize the linking methods introduced in the above section.

We used IRC as an example to describe the Rake language (See Figure 3). The Rake language is message driven, and it mainly defines the properties of the messages. In this simple example, we are interested in the chat messages, and hence define a message named "IRC PRIVMSG" with the XML tag *<Message name="IRC PRIVMSG">*. There are five basic properties for a message to specify, which specifically define the three functions S, f and g in our mathematic model:

- *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. For example, protocol type, port number and regular expression can be used to classify IRC chat message.
- *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 Link_ID, introduced above. For example, when an IRC server first receives a chat message from the client, the Child_ID is 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, hence these IDs can be set to the "None" type or ignored in the XML file.

```
<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

Note the union of Link_ID and Response_ID makes the g function of the message, and the union of Child_ID and Query_ID is the f function.

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, 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 right 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.

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

```
<Message name="DNS_Response">
  ......
  <Link_ID>
    <Type> User_Function </Type>
    <Libray> dns.so </Libray>
    <Function> Get_DNS_Dest </Function>
  </Link_ID>
  <Child_ID id="0">
    <Type> User_Function </Type>
    <Libray> dns.so </Libray>
    <Function> Child_IDs </Function>
  </Child_ID>
</Message>
```

Fig. 4. Example of DNS XML description

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 signature patterns that are more complex. 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 4.3.3, where we have examples.

Note, all the matching rules defined in a 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.

1) 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.* regular expression mentioned above. For example, in CoralCDN, a HTTP requests triggers the Coral webproxy to query other Coral nodes using DHT packets for caches. A 20-byte ID is generated by hashing the HTTP URL with SHA1 hash function [17]. Therefore, to get the Child_ID of the HTTP request, Rake should call the sha1 hash function. Since we cannot exhaustively predefine the ID extracting functions, we choose to let user supply their own function. In Rake implementation, we utilize the dynamic (or shared) library techniques. Rake specifies the interfaces, defining the input and output of the interfaces and the user just implements the interface accordingly. 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.

2) 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).

3) 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 calls "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 call 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:

```
<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.

4. Generality and Applications of Rake

A key question to answer is that: Does semantics assisted linking work in every application? In this section,

we discuss the potential limitation of Rake, and discuss the popular applications that Rake can work with.

1) Potential Limitations: In logic, if a message triggers another message, they should be related. However, their relation may not be revealed by their content. And this is the potential area that Rake cannot be applied to.

Take Hadoop distributed file system [8] as an example. The master node stores the index of the files (*e.g.* records of (Filename $\rightarrow$ FileID)) and the slave nodes store the files. When a client request a file with filename S, master node translates it to the FileID I, which can be a random number. There is no way to map S to I even if users supply some user functions. There are two potential implementations: 1) $iterative$: the master node retrieves the file through I for client; 2) $recursive$: the master node returns I back to the client and client goes to fetch the file itself. In the first implementation, Rake fails to get the task tree; however, in the second case, Rake can link the query of file S and the response of FileID I using the query/response ID in the messages. Fortunately, Hadoop takes the recursive way because it wants to keep the master node working on indexing only and avoid overload of delivering data. Hence in the real Hadoop implementation, Rake does not have linking problem.

In short, the fundamental limit of Rake is that some related messages may not be able to be linked together because some key mapping happens in the memory of the program and there is lack of alternative semantics for linking. But we find that most of the distributed systems adopt the recursive model to reduce server load. Thus this problem does not happen and Rake is able to identify the task trees for many distributed applications.

2) Rake Applications: Our target is mainly the large scale distributed systems, which usually involve many nodes and complex structures such as hierarchy or DHT.

We studied several applications in different categories:

- *Large-scale web search system*: We investigate a web search system of one of the top search service providers and Section 6.5 describes more details.
- *Distributed Cluster Computing Platform*: We deployed Hadoop [8], which is an open-source implementation of Google's MapReduce and is widely used in industry such as Yahoo!, Amazon and Facebook. We evaluate the diagnosis ability of Rake using some testbed experiments (See Section 6.4).
- *Contend distribution networks*: Specifically, we studied and deployed CoralCDN [20]. Our evaluation in Section 6.3 shows the accuracy of Rake is much higher than the black-box approach WAP5 [21].
- *IRC and messengers*: In our description of Rake, we already use IRC as an example as IRC's semantics is simple and enough for Rake to utilize. More detailed evaluation can be found in [25]. Similarly, we investigate messengers such as MSN and believe the chat content can be easily used to identify the flow of the chat messages in the system.

5. Accuracy of Message Linking

The message linking is based on the IDs extracted from messages using application semantics, and therefore the accuracy depends on the application and we have accuracy analysis for the four applications we studied in Section 6. Here we describe, in high level, the factors that affect the linking accuracy.

For query-based applications such as web search and DNS, the query keyword and its transform are usually used as the IDs to link messages. If the same query keyword is queried by different users multiple times in the same time, this will potentially generate ambiguity problem and the linked task tree may be incorrect and shuffled. For example, in web search, popular search keywords have some chance to be queried at close time. On the other hand, the cache mechanism widely used in the real life helps to solve the ambiguity problem as the duplicated query may be held and not further processed.

6. Complexity of Message Linking

When a message goes into the Rake system, Rake will first use signature matching to determine the type of the message and then call either native or user provided function to extract the IDs of the message. The previous IDs can be stored in the hash table and hence linking messages by IDs is quite obvious. While the complexity really depends on the applications, generally the running time is linear to the number of messages and the size of messages. In our experience, the most time consuming part is actually the signature matching part, which in future can be optimized using techniques from intrusion detection systems.

5. Practical Issues and Diagnosis

In this section, we discuss practical issues on software evolution and trace collection, as well as how to diagnose with task trees discovered.

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 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 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

Generally, Rake takes the sniffed network traffic as the input trace. While sniffing is already a very mature and widely used technique, sniffing and collecting data from a large network is a challenging problem.

Sniffers can be put on hosts 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.

1) Partial Sniffer Deployment: For various reasons, fully sniffing the whole network may be infeasible or too costly. Generally, partial sniffer deployment degrades the power of Rake by causing the diagnosis granularity to be coarser. For example, in Hadoop system, the master servers control all the slave nodes and generally the slave nodes talk less. In this case, sniffing on the relatively few master nodes can still cover most of the task tree and Rake may only miss the latest layer of the tree involving the communication between slave nodes.

2) 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, as most data packets are useless. Therefore, simple preprocessing after sniffing, and sending back only a summary of the packets is desirable. This way, the network overhead introduced in order to collect traces can be neglected. For example, in CoralCDN, we only need to extract 20 bytes from one packet which is a small portion of a large data package.

3) 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.

3. Diagnosis with Task trees

Diagnosing the 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.

1) Diagnosis Using Processing Time: In many time 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 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.

2) 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 6.4), this simple diagnosis approach actually helps us identify the *Slow master node* problem.

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

6. 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.

1. 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 in implemented in around 2000 C++ lines. 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 [1] 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 the 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.

2) Experience of Applying Rake to Applications: Similar to X-Trace which needs programmers to instrument the applications, users need to to instrument Rake with certain semantics. 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 non-designers, in the following two aspects.

1) Task Trees Discovery: For network protocols such as DNS and IRC, we find it is very convenient to simply study the RFCs of them. The RFCs usually clearly describer 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 and focusing on network communication.

For CoralCDN and Hadoop which 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 due to the 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 transform (*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 which 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. The only trick to apply is the ID inheritance when the query target or task ID is not contained in some RPC response messages.

2. 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 but can be found in [25].

We compared our Rake algorithm with previous studies using the black-box approach WAP5 [21]. Since WAP5 does not work for computation intensive applications such Hadoop, as the 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.

3. Evaluation on CoralCDN

1) CoralCDN Background: CoralCDN is a decentralized peer-to-peer web-content distribution network. CoralCDN is built on top of Coral, a key/value indexing infrastructure which uses a distributed sloppy hash table(DSHT [20]). Basically, the web user is redirected by Coral DNS server to another Coral DNS or Coral web server, and the Coral web server fetches the web contents from the cache in the DSHT or from the original source of the web contents. 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.

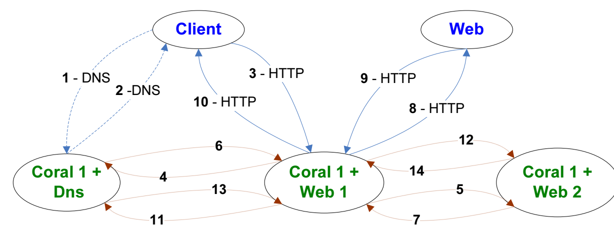

Fig. 5. Coral execution path from Rake.

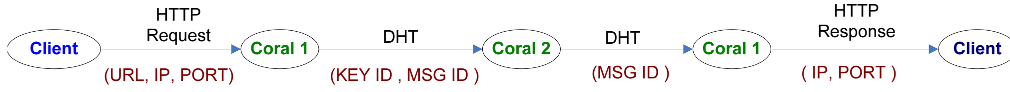

Fig. 6. Semantic information flow in CoralCDN System .

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.

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

- UrlSet1 – The sniffed network traffic of a major university in China. We replayed a total of 21 GB HTTP traces collected from university on coral CDN.
- UrlSet2 – The sanitized access log from [22]. 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 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.

By modifying the CoralCDN source code, 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 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 [21].

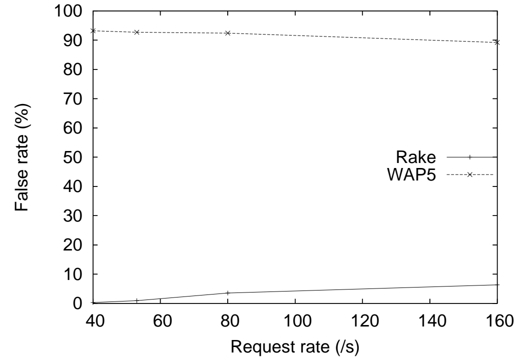

Fig. 7. False rate of WAP5 vs Rake

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.

Originally, we run CoralCDN on multiple PlanetLab

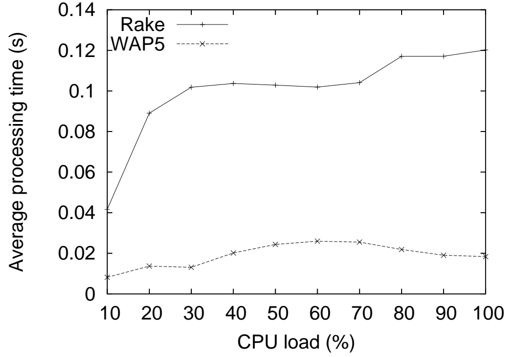

Fig. 8. Processing time of WAP5 *vs* Rake.

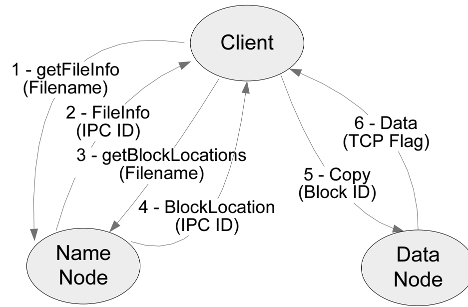

Fig. 9. Semantics of Hadoop DFS - Get operation.

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 [13] 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.

4. Evaluation on Hadoop

In this section, we present an example to use Rake to diagnose Hadoop [8].

1) Hadoop Background: Hadoop [8] is an open-source implementation of Google's MapReduce [15]. 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.* [23]).

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.

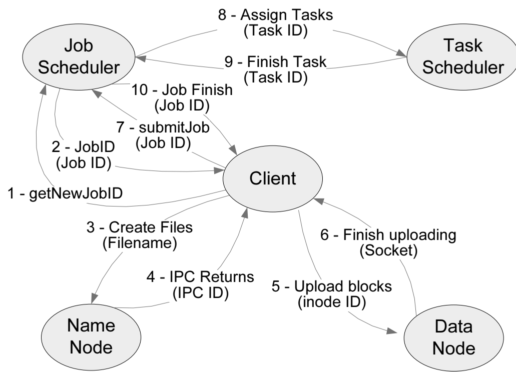

Fig. 10. Semantics of Hadoop - Grep operation.

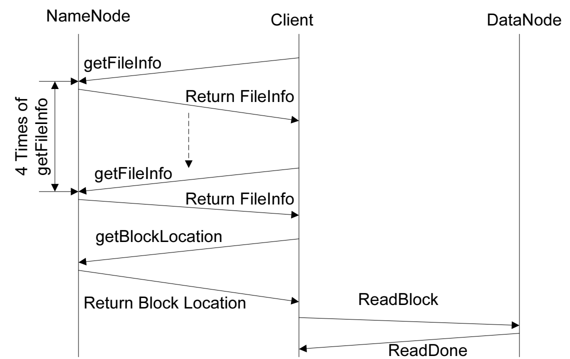

Fig. 11. Abused IPC calls in Hadoop.

1) 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 an 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, IPC calls and returns are matched via the unique IPC call IDs, and the IPC call 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 call IDs and inode IDs, and finally the socket tuples.

2) 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 did in CoralCDN experiments.

4) Evaluation Results:

1) DFS Reader: In this experiment, we use Rake to inspect the *Get* operation of Hadoop DFS. In each single run of the experiment, two Hadoop clients download the same file from the DFS system simultaneously and we conducted the experiments five times.

Accuracy: We manually checked message linking results of Rake and found that Rake can link the messages

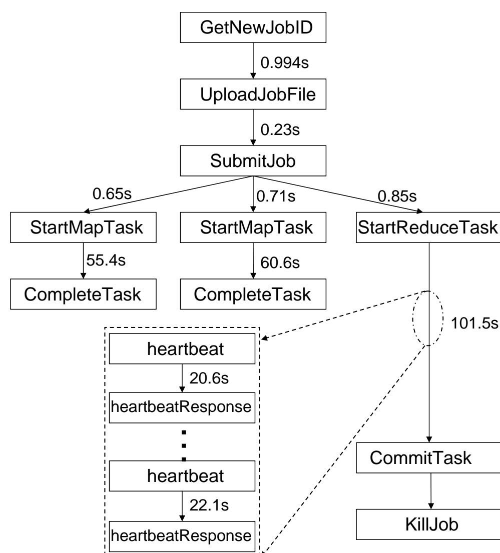

Fig. 12. Running time of Hadoop steps.

without any error. As mentioned in Section 6.4.2, the socket information helps to solve the ambiguity that may potentially caused by downloading the same file simultaneously.

Abused IPC Calls: Figure 11 shows the messages linked by Rake. Surprisingly, it shows that the IPC call "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 IPC calls, and the programmer may not realize that he makes some IPC calls. In this case, the IPC call "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 IPC calls. To the best of our knowledge, we are the first one to find this problem.

2) Hadoop MapReduce Job - Grep: 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.

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 the a small block (*e.g.*the last block of a data file). Further data-ming approaches such as the distMatrix [23] can be used to diagnose more accurately, but this is not our focus in this paper.

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 IPC calls may become very 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 the 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 IPC calls 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 IPC calls and responses and hence is able to disclose the problem in the master node. It is worth mentioning that the Hadoop's own logs may not be able to identify this problem, because Hadoop does not log every heatbeat and their response messages.

5. Web Search System

Web search is one of the most popular service. Take the Google search platform for an example [12], it usually has a tiered structure. The HTTP requests are distributed to web servers and the search keywords are extracted then. Next, the keyword is reformatted into some canonical format and sent to the index servers. The document IDs of the hit documents will be sent back and then the target documents are fetched. Finally, the search results are returned to the clients by the web servers.

1) Semantics for Linking Messages: Confirmed with a researcher from a major Web search provider, we find it is easy to link all the messages related to a search query in the search system. At the beginning, the search keyword and its canonical format serve as the IDs to link the messages from the load balancer to the index servers. The returned document IDs can be linked to the search keyword in a query/response communication between the web server and the index server. Next the document ID can be used to link the messages of web

server to fetch documents. Note the cache feature in the web search helps on the linking accuracy. In the search of same keywords multiple times in a short period, only the first one traverse the complex task tree, while the rest get simple task trees due to cached search results.

2) Message Linking Accuracy Analysis: The inaccuracy of message linking comes from the ambiguity problem. If two identical search keywords come to the same web server at close time (within Δt), they may cause mis-linking, for that they and their following messages have the same IDs. After communicating a major Web search provider, we choose Δt as 180ms, because a search query is usually finished within 60ms.

We obtain *one peak hour of logs* collected at the frontend servers of a major Web search provider. For privacy reasons, we cannot reveal more statistics about the traces. Since the search query is transformed to be a canonical format, and different but similar queries may have the same canonical format, we cannot directly compare the search keywords for the ambiguity rate. Unfortunately the transform function of all popular search systems is unpublished. To bypass the problem, we replay the search queries with Microsoft bing and use the search results to determine if two queries share the same canonical format or not. Two queries with the same canonical format from the same "wget" command return the same (or very similar) results.

The average percentage of similar/ambiguios request is 4.5%. This implies that the average accuracy of the system is about 95.5%. For debugging the system, we need not to find the task tree of every request. Simply ignoring the ambiguous requests and using the task tree of 95.5% of request, we will be able to debug the system. It is worth mentioning that the ambiguity is mainly caused by user repeating their search keywords multiple times in a short period.

7. 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 IDs based on application semantics. Using several popular distributed systems, we demonstrate what semantics Rake utilizes to link messages in a task tree accurately. Meanwhile, to easily adopt Rake to new applications, we designed the XML-based Rake language, which allows clients to reuse the core rake components by only submitting application semantics. For evaluation, we deployed both CoralCDN and Hadoop, and conducted controlled experiments to evaluate the message-linking accuracy. We also demonstrated how Rake can help on diagnosing performance problem in these two systems. Meanwhile, accuracy analysis on the web search system and IRC system also demonstrates the accuracy of Rake in different systems.

References

[1] Gnu libtool - the gnu portable library tool. http://www.gnu.org/software/libtool.

[2] Hp openview. http://www.openview.hp.com/.

[3] Ibm tivoli. http://www.ibm.com/software/tivoli/.

[4] Microsoft operations manager. http://www.microsoft.com/mom/.

[5] Ackerman, E. Yahoo's hadoop software transforming the way data is analyzed. http://www.siliconvalley.com/news/ci_10897240?nclick_check=1.

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

[7] Anandkumar, A., Bisdikian, B., and Agrawal, D. Tracking in a spaghetti bowl: Monitoring transactions using footprints. In *ACM SIGMETRICS* (June 2008).

[8] Apache. Hadoop. http://lucene.apache.org/hadoop/.

[9] Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Information and control in gray-box systems. In *SOSP* (2001).

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

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

[12] Barroso, L. A., Dean, J., and Holzle, U. Web search for a planet: The google cluster architecture. *Micro, IEEE 23*, 2 (2003), 22–28.

[13] Carraway, D. Lookbusy. http://devin.com/lookbusy/.

[14] Chen, X., and et. al. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI* (2008).

[15] Dean, J., and Ghemawat, S. MapReduce: Simplifi ed data processing on large clusters. In *OSDI* (2004).

[16] Dreger, H., and et. al. Dynamic application-layer protocol analysis for network intrusion detection. In *USENIX Security Symposium* (2006).

[17] Eastlake, D., and Jones, P. US Secure Hash Algorithm 1 (SHA1). RFC 3174, 2001.

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

[19] Fox, A., and Brewer, E. Path-based failure and evolution management. In *NSDI* (Apr. 2004).

[20] Freedman, M. J., Freudenthal, E., and Mazires, D. Democratizing content publication with coral. In *NSDI* (2004).

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

[22] National lab of applied network research. ftp://ircache.nlanr.net/Traces/.

[23] Tan, J., Pan, X., Kavulya, S., Gandhi, R., and Narasimhan, P. SALSA: Analyzing logs as state machines. In *Usenix Workshop on the Analysis of System Logs* (2008).

[24] Yemini, S., Kliger, S., Mozes, E., Yemini, Y., and Ohsie, D. High speed and robust event correlation. In *IEEE Communications Magazine* (1996).

[25] Zhao, Y., Cao, Y., Goyal, A., Chen, Y., and Zhang, M. Rake: Semantics assisted network-based tracing framework. Tech. Rep. NWU-EECS-09-14, University of Northwestern, Jun, 2009.