Falcon: A Practical Log-based Analysis Tool for Distributed Systems

Francisco Neves, Nuno Machado and José Pereira
HASLab, INESC TEC and University of Minho
Braga, Portugal
{francisco.t.neves, nuno.a.machado}@inesctec.pt, jop@di.uminho.pt

***Abstract*—Programmers and support engineers typically rely on log data to narrow down the root cause of unexpected behaviors in dependable distributed systems. Unfortunately, the inherently distributed nature and complexity of such distributed executions often leads to multiple independent logs, scattered across different physical machines, with thousands or millions entries poorly correlated in terms of event causality. This renders log-based debugging a tedious, time-consuming, and potentially inconclusive task.**

We present Falcon, a tool aimed at making log-based analysis of distributed systems practical and effective. Falcon's modular architecture, designed as an extensible pipeline, allows it to seamlessly combine several distinct logging sources and generate a coherent space-time diagram of distributed executions. To preserve event causality, even in the presence of logs collected from independent unsynchronized machines, Falcon introduces a novel happens-before symbolic formulation and relies on an off-the-shelf constraint solver to obtain a coherent event schedule.

Our case study with the popular distributed coordination service Apache Zookeeper shows that Falcon eases the log-based analysis of complex distributed protocols and is helpful in bridging the gap between protocol design and implementation.

I. Introduction

Developers of distributed systems cater for recording runtime behavior by judiciously adding log statements to source code [1], [2]. The number of log statements needed, and the detail of the information collected, depends on the complexity of the code. In systems that deal with concurrency and faults, such as fault-tolerant consensus protocols, the resulting effort is substantial. However, when an unexpected outcome is noticed, log files are often the only source of information that programmers can use to debug and fix the problem.

Unfortunately, log analysis in distributed systems still remains a daunting task, which has motivated programmers to ask for more practical ways to understand runtime behavior.[1] First, besides the sheer number of entries, trace files are typically spread across several nodes and generated by distinct logging libraries with heterogeneous formats. Second, although timestamped, interleaved statements or execution on different nodes leads to a wide set of possible event execution flows and intermediate states that have to be considered. Third, the lack of context propagation between nodes hinders the ability to establish the causal relationship between events, i.e., the *happens-before* relationship typically denoted by "$\rightarrow$" [3].

Causality is particularly helpful for debugging distributed executions, as it allows reasoning about the order of distributed events[2]. However, relying solely on log entry timestamps is not enough to establish causality. On the one hand, these timestamps are based on physical clocks and, even if clocks are synchronized on all relevant nodes, log messages are often produced asynchronously after the fact they describe. On the other hand, blindly considering that timestamps induce causality hides the true system logic by flattening history.

Several tracing systems have been proposed in the past to track causality and alleviate the burden of debugging distributed systems [4]–[8]. Nonetheless, they require careful program instrumentation and do not support the analysis of events stemming from distinct, heterogeneous log sources. In contrast, popular operating system utilities such as *strace* and *ltrace* are powerful assets for troubleshooting runtime behavior, as they are language-agnostic and capable of capturing the system calls and signals executed by a program, but fall short when it comes to inferring causality across processes.

In this paper, we aim to achieve the best of both worlds by *enabling the inference of causally-related activity atop commodity monitoring tools*. To this end, we propose Falcon, a practical and effective log-based analysis tool for distributed systems. Falcon does not require custom instrumentation and supports popular tracing and logging utilities (e.g., *log4j*, *strace*, *tshark*), thus being suitable for debugging real-world applications.

Falcon operates as a pipeline: first, it normalizes the events collected by the different logging sources; then, it resorts to symbolic constraint solving to generate a global execution trace that preserves causality; finally, it produces a space-time diagram that enables a visual analysis of the whole execution.

To ensure event causality, Falcon employs a novel approach that models a distributed execution by means of symbolic variables (representing the logical clocks of the events traced) and encodes the happens-before relationships as constraints over those variables. Solving the constraint system with an off-the-shelf solver yields a complete execution schedule that coherently relates the data from the various log files, thus providing a thorough and accurate view of the original production run. Due to its flexible design, Falcon's pipeline can also be extended with additional log libraries and visualization tools.

[1] https://issues.apache.org/jira/browse/ZOOKEEPER-816

[2] We use *event* and *log entry* interchangeably in this paper.

Our case study with the popular coordination service Apache Zookeeper shows that Falcon is efficient and facilitates the understanding of complex distributed executions by relating low-level system calls with user-defined log messages.

The rest of this paper is structured as follows. Section II presents some background concepts and a motivating example. Section III describes the design of Falcon, while Section IV provides its implementation details. Section V presents the case study with Apache Zookeeper. Section VI overviews the most relevant related work and, finally, Section VII concludes the paper by summarizing its main points.

II. Background and Motivation

Low-Level Tracing Overview. *NIX environments nowadays offer various kernel-level tracers that enable powerful troubleshooting capabilities. Moreover, by running at the operating system level, these tracers are programming-language-agnostic and even applicable to programs running on virtual machines, thus being extremely useful for program debugging. Notable examples include *ftrace*, *strace*, *ltrace*, *eBPF*, and *SystemTap*.

At the core of most of these tools are system calls. In computing, a system call, or *syscall*,[3] can be defined as the fundamental interface between an application and the operating system kernel. During the execution of a program, whenever it requires access to open and close files or to establish a connection with a remote program, these intentions are converted into syscalls. For example, the *strace* tool captures the signals received and the system calls invoked by the target program. This is possible because it resorts to the *ptrace* syscall, which allows a process to take control over another process.

The interception of a syscall can be done both when the execution switches between the user mode to the kernel mode (*entry* point) and vice-versa (*exit* point). Intercepting at the former allows accessing the syscall parameters, whereas intercepting at the latter gives the success/failure of the operation.

The time spent between both interception points can vary arbitrarily, as it depends on the state of the operating system, resource contention, and programming decisions, such as signal handling. Also, the two points are not guaranteed to appear contiguously in the trace. In fact, the output of *strace* usually exhibits an interleaving of entry and exit points of different syscalls.

Some of the aforementioned tools can be also used for tracing the messages exchanged in a distributed system, since they allow tracking the socket *read* and *write* syscalls invoked during the execution. Unfortunately, such syscall log is not enough per se to effectively analyze and debug distributed protocols that rely on complex communication patterns (e.g., consensus, fault-tolerance, and replication protocols), which are challenging to design and implement. The reason is that there is no information with respect to the *causality* between the syscalls logged.

[3]http://man7.org/linux/man-pages/man2/syscalls.2.html

```
    # Format: [pid], syscall(parameters) = return_value
3.1 [n3-894] read(n2, "Go go go", 1023) = 8
3.2 [n3-894] write(n1, "Wait guys", 1023) = 9
3.3 [n3-894] write(n2, "Wait guys", 1023) = 9
3.4 [n3-894] read(n2, "Ok", 1023) = 2
2.1 [n2-782] write(n3, "Go go go", 1023) = 8
2.2 [n2-782] write(n1, "Go go go", 1023) = 8
2.3 [n2-782] read(n3, "Wait guys", 1023) = 9
2.4 [n2-782] write(n1, "Ok", 1023) = 2
2.5 [n2-782] write(n3, "Ok", 1023) = 2
1.1 [n1-675] read(n3, "Wait guys", 1023) = 9
1.2 [n1-675] read(n2, "Ok", 1023) = 2
1.3 [n1-675] read(n2, "Go go go", 1023) = 8
```

Fig. 1: Complete trace resulting from merging the partial outputs of *strace* on three distinct nodes. [ni-pid] denotes the global identifier of a process pid running on node i.

Motivating Example. As an example of this limitation, consider a scenario with three participants of an online multiplayer game, represented by three processes on distinct machines connected through TCP sockets. In this scenario, player 2 (corresponding to process **n2-782**) tells his teammates to advance with a message *"Go go go"*. Player 3 (process **n3-894**) disagrees with the suggestion and asks the team to wait by replying *"Wait guys"*. Player 2 consents and writes *"Ok"*. Player 1 (process **n1-675**), in turn, simply receives the instructions given by the other players. The result of this interaction was that player 1 advanced alone, causing the team to later lose the game. Why did player 1 act against the instructions given by the rest of the team?

In Figure 1, we present a possible log obtained by merging the output of running *strace* on each process. The log contains the syscalls executed during a chat conversation between the three players, namely the *reads* and *writes* on each process' socket. For the sake of readability, each entry is identified by the concatenation of the node and process ids and the actual files descriptors were replaced by the node ids.

In order to correctly reason about the runtime behavior from the trace in Figure 1, one must first establish the *happens-before* relationship between the syscalls. As defined by Lamport [3], if an event a causally-precedes an event b in a program execution, then a *happens-before* b (denoted $a \rightarrow b$). A more detailed definition of the happens-before relationship is given in Section III-C.

Causality in distributed systems is typically captured by logical clocks [3] or vector clocks [9]. However, low-level tracing tools such as *strace* are not able to record logical time. Let us then mimic the procedure of manually inferring the happens-before relations present in Figure 1.

The causal order of syscalls within each process' trace is trivial to define because it respects the program order [3]. As such, the main challenge here is to infer the inter-process happens-before relationships.

Note that the first parameter on each *write*/*read* syscall denotes the process that sent/received a message. Considering that a *read* is always preceded by its corresponding *write* and that TCP ensures reliable and ordered delivery, one is then able to causally order the syscalls across the three processes.

```
    # Format: [pid], syscall(parameters) = return_value
2.1 [n2-782] write(n3, "Go go go", 1023) = 8
2.2 [n2-782] write(n1, "Go go go", 1023) = 8
3.1 [n3-894] read(n2, "Go go go", 1023) = 8
3.2 [n3-894] write(n1, "Wait guys", 1023) = 9
3.3 [n3-894] write(n2, "Wait guys", 1023) = 9
1.1 [n1-675] read(n3, "Wait guys", 1023) = 9
2.3 [n2-782] read(n3, "Wait guys", 1023) = 9
2.4 [n2-782] write(n1, "Ok", 1023) = 2
2.5 [n2-782] write(n3, "Ok", 1023) = 2
1.2 [n1-675] read(n2, "Ok", 1023) = 2
3.4 [n3-894] read(n2, "Ok", 1023) = 2
1.3 [n1-675] read(n2, "Go go go", 1023) = 8
```

Fig. 2: Trace resulting from causally reordering the syscalls in Figure 1. [n*i*-*pid*] denotes the global identifier of a process *pid* running on node i.

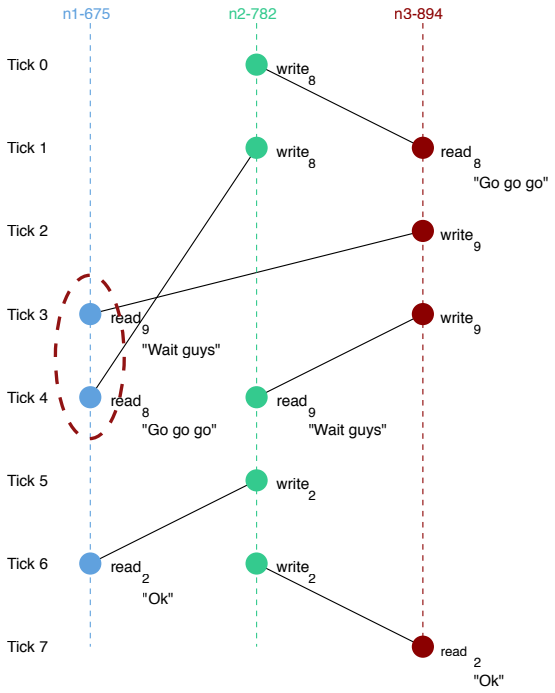

Fig. 3: Space-time diagram of the trace in Figure 2. Each vertical line represents a node, whereas the circles within a line represent the syscalls executed by that node. Solid lines connecting circles indicate a happens-before relationship between the events. The dashed circular area highlights the problematic message race.

In Figure 2, we depict a possible trace resulting from causally reordering the syscalls according to the intra- and inter-node happens-before relationships. To further ease the analysis of the execution, we also convert the ordered trace into a space-time diagram, which is shown in Figure 3. In the diagram, vertical lines are the execution timelines of the processes indicated by the labels, and the circles are the events happening in each process. Each event is associated with a given logical clock "tick". We added the message sizes on each event and the message content on *read* syscalls. Each pair of connected events indicates a happens-before relationship.

Displaying the messages received by each process, one obtains the following chat logs:

n1-675	**n2-782**	**n3-894**
n3: "Wait guys"	n2: "Go go go"	n2: "Go go go"
n2: "Go go go"	n3: "Wait guys"	n3: "Wait guys"
n2: "Ok"	n2: "Ok"	n2: "Ok"

Note that the chat log of process **n1-675** exhibits an inconsistency (highlighted in red) with respect to the actual message history, which explains the reason behind the reckless move by player 1. The dashed circular area in Figure 3 pinpoints the root cause of this inconsistency: a delay in the arrival of the message *"Go go go"* sent by process **n2-782** caused an inversion in the expected chat output. Since the inverted messages are (semantically) causally related, this means that there is a message race bug in the system implementation.

This example illustrates a game scenario that simply caused a team to lose one round. However, in complex distributed systems that require coordination, the consequences may be much more severe (e.g. data loss or corruption). It is thus of paramount importance to devise practical and effective tools to aid the analysis of execution logs.

III. Design

We propose Falcon – a practical and effective log-analysis tool for distributed systems, capable of generating a global execution schedule from multiple independent log files while preserving event causality. This is achieved by means of a novel symbolic constraint model that encodes the *happens-before* relationship between events. Moreover, Falcon automatically generates a causal space-time diagram of the execution, which further eases the analysis of the logs and the understanding of distributed executions. This section describes Falcon's design requirements, architecture, and happens-before model.

A. Design Requirements

Time spent at post-mortem software debugging is directly affected by the amount of useful information captured during production runs. Since logging is an expensive operation, a trade-off must be made between the log's verbosity level and the performance and space overhead imposed at runtime [1]. For that reason, different tracing tools opt for focusing on different aspects and provide distinct features (e.g. printing log statements, sniffing network packets, profiling performance, etc). Nevertheless, one should be able to leverage all those features in order to ease the burden of debugging complex distributed systems. A practical and effective log-analysis tool should thus meet the following design requirements:

- **Support several log sources.** The tool should be able to extract useful knowledge about the execution from multiple data sources, such as logging libraries, network sniffers (e.g. *libpcap-based* tools), and low-level tracing tools (e.g. *ptrace*-based tools).
- **Combine data in a causally consistent way.** The tool should be able to combine all logged events in a seamless

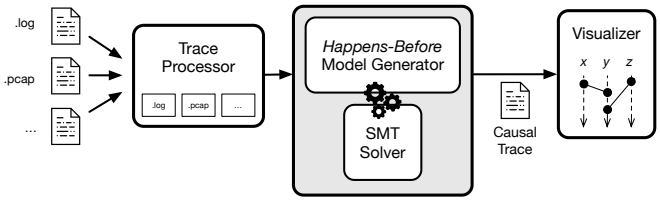

Fig. 4: Falcon's pipeline architecture, which comprises modules for trace processing, model generation, and visualization.

and coherent fashion, even if they were captured at different physical machines with unsynchronized clocks. In practice, this corresponds to ensuring that the *happened-before* relationship between events is established across all log files regardless of their source.

- **Provide a visual representation of the execution.** To obviate complexity due to log verbosity and further help developers to reason about the execution, the tool should be able to display events in a "human-friendly" way. In the particular context of distributed systems, space-time diagrams depicting the inter-process causal dependencies have long been used to aid the understanding of distributed protocols over multiple processes [10].

In the next section, we describe how Falcon meets the aforementioned requirements.

B. Architecture

Falcon is designed with a modular architecture, whose components operate together as a pipeline. In a nutshell, Falcon receives as input log files from multiple data sources and outputs a space-time diagram that preserves event causality. Figure 4 depicts the architecture of Falcon, composed by three main modules: the *trace processor*, the *happens-before model generator*, and the *visualizer*. Each module is described in detail as follows.

Trace Processor. Since the events logged by the different tools can vary in both format and content, Falcon needs to first *normalize* and *merge* the collected data into a global event trace with a common scheme. This procedure is done by the trace processor module. The trace processor is equipped with a dedicated driver for each type of log, responsible for translating the library-specific entries into events that can be processed by Falcon. As such, drivers may range from simple parsers for textual logs (e.g. for *log4j*) to packet unpackers for network sniffers (e.g. *tshark*). In some cases, the trace processor generates events that are the result of merging data from different logs. For example, an event representing the sending of a message can be built by augmenting the information of a *write* syscall with the message payload captured by a network sniffer. The events resulting from Falcon's log normalization and merging are the following:

- **START**$(process)$: a process starting event;
- **END**$(process)$: a process finishing event;
- **FORK**$(parent, child)$: a process creation event, where $child$ denotes the process spawned by process $parent$;
- **JOIN**$(parent, child)$: represents a join event, where process $parent$ waits until the $child$ process finishes;
- **CONNECT**$(process, src, dst)$: represents a new connection, where src and dst denote the addresses (IP and port) of the local and remote processes, respectively;
- **ACCEPT**$(process, src, dst)$: event indicating that a connection was established, where src and dst also denote the local and remote addresses, respectively;
- **SND**$(process, src, dst, msg)$: a message sending event, where msg is the identifier of message being sent from the src address to the dst address;
- **RCV**$(process, src, dst, msg)$: a message receiving event, where msg denotes the identifier of the message sent by the src address and received by the dst address.
- **LOG**$(process, msg)$: a log entry event, where msg is the content of the message logged by process $process$.

The trace processor module also exposes a public API to ease the development of drivers and the integration of additional logging libraries into Falcon.

Happens-Before Model Generator. The complete, normalized event trace is then fed into the happens-before (HB) model generator. This module is responsible for combining all events into a single causally-consistent schedule. To this end, the HB model generator builds a symbolic constraint formulation encoding the happens-before relations between events. For instance, the model encodes a constraint stating that the *send* event of a message must happen-before the corresponding *receive* event. The HB constraints are further described in Section III-C.

Solving the model with an off-the-shelf constraint solver yields a causally-ordered event schedule.

Visualizer. The visualizer finishes the Falcon's pipeline by providing a graphical representation of the causal trace generated in the previous step. In detail, the visualizer generates a "space-time diagram", as introduced by Lamport [3], depicting both the events executed by each process and the inter-process causal relationships between them.

C. Happens-Before Constraint Model

As defined by Lamport [3], there exists a happens-before relationship between two events a and b, denoted $a \rightarrow b$, if:

- a and b belong to the same process[4] and a precedes b in the execution.
- a and b belong to different processes and a represents the sending of a message m and b represents the reception of m.

Distributed executions often comprise other causal relations that should be considered, namely $a \rightarrow b$ also holds if:

- a is the *fork* event of a process q by a process p and b is the first event of q.
- a is the last event of a process q and b the *join* event of q by a process p.
- a is the *connect* event issued by a process p to a process q and b is the *accept* event in q.

[4]We use the term *process* to denote both processes and threads.

Note that the happens-before relation is transitive, irreflexive and antisymmetric. Also, when $a \nrightarrow b$ and $b \nrightarrow a$ holds, then a and b are considered to be *concurrent*.

Falcon casts the problem of combining the events from independent logs into a global, causally-ordered execution schedule as a *maximum satisfiability modulo theories (MaxSMT)* problem. The MaxSMT problem can be seen as an optimization version of the satisfiability problem (for types of variables other than boolean ones) and has the goal of finding a total assignment to variables of a formula that maximizes the number of satisfied clauses. Among the variants of the MaxSMT problem, this paper assumes a *partial* MaxSMT problem where some clauses are considered as *hard* and others are considered as *soft*. The goal is thus to find an assignment to the variables such that all hard constraints are satisfied and the amount of satisfied soft constraints is maximized.

Falcon's causality model comprises *i)* integer symbolic variables that represent the *logical clocks* [3] of the events supported by Falcon (see Section III-B) and *ii)* hard constraints over those variables stating the causal relations between the events. A solution to this model thus assigns a value to each variable such that all happens-before rules are satisfied. In practice, this corresponds to inferring a causally-consistent execution schedule by computing a logical clock per event.

More formally, the constraint model, denoted Φ_{HB}, consists of a MaxSMT formulation defined as the following conjunction of sub-formulae:

$$\Phi_{HB} = \underbrace{\phi_{inter} \wedge \phi_{intra}}_{hard} \wedge \underbrace{GOAL}_{soft} \tag{1}$$

where ϕ_{inter} encodes the inter-process causality constraints, ϕ_{intra} encodes the intra-process happens-before rules due to program order, and $GOAL$ states the soft constraints that allow steering the solving procedure towards a given goal. Falcon currently provides support for generating logical clocks that: *i)* follow the original timestamp order as much as possible (ϕ_{ts}), and *ii)* expose concurrency issues by minimizing the logical time intervals between events (ϕ_{min}). We now describe each sub-set of constraints in more detail.

a) Inter-process HB Constraints (ϕ_{inter}): these constraints represent the causal dependencies due to message exchanges and inter-process synchronization. Following the happens-before rules presented at the beginning of this section, the inter-process HB constraints ϕ_{inter} are written as follows:

$$\begin{aligned} &fork_{p,q} < start_q \\ &end_q < join_{p,q} \\ &connect_p < accept_q \\ &snd_{p,m} < rcv_q \end{aligned}$$

where p and q are distinct processes, m represents a given message, and the variable names correspond to the events described in Section III-B. For instance, for a message m sent by p to q, the constraints encodes that the logical clock of the corresponding event $\textbf{SND}(p, p, q, m)$ in the trace must be smaller than that of the event $\textbf{RCV}(q, p, q, m)$.

b) Intra-process HB Constraints (ϕ_{intra}): these constraints state that events in the same process execute sequentially according to the program order. Let Γ_p denote the event trace of a process p, and let c_i and c_j be the symbolic variables representing the logical clocks of events i and j. The intra-process HB constraints ϕ_{intra} are given by:

$$\forall i, j \in \Gamma_p : (i < j \implies c_i < c_j)$$

c) Timestamp Constraints (ϕ_{ts}): timestamp constraints are soft constraints (see $GOAL$ in Equation 1) that aim at approximating the schedule produced by the solver to the actual event ordering observed during the production run. These constraints state that events should be given logical clocks that follow the order given by timestamps in the log files. However, since two causally-ordered events logged on different machines may exhibit physical timestamps conflicting with their HB relationship, timestamp constraints may be violated in order to satisfy causality.

d) Clock Minimization Constraint (ϕ_{min}): clock minimization constraints are also encoded as $GOAL$ soft clauses and strive to minimize the values assigned to the symbolic variables. The goal is to produce a *compact* schedule capable of exposing event concurrency. For instance, if two distinct **SND** events exhibit the same logical order in the schedule yielded by the solver and their corresponding **RCV** events belong to the same process, then there is a message race.

Let $e \in \Gamma$ be an event in the complete execution trace and c_e the symbolic variable representing its logical clock. The clock minimization constraint ϕ_{min} is written as:

$$\phi_{min} = min \sum_{\forall e \in \Gamma} c_e$$

Solving the Φ_{HB} model generated by Falcon using an off-the-shelf SMT solver yields an execution schedule in which events are guaranteed to be causally ordered.

IV. Implementation

This section discusses some relevant implementation details of our prototype of Falcon. The prototype is publicly available at `https://github.com/fntneves/falcon/`.

The trace processor module is implemented as an extensible Python program that allows the integration of custom drivers for normalizing log files into a pre-defined JSON format. Currently, the trace processor provides three out-of-the-box drivers. The first is a *ptrace-based* tool that collects syscall traces. The second driver handles logs generated by *log4j*, a logging library for Java programs. The third uses *tshark* to extract message payloads from *pcap* files and add them to the corresponding send and receive events.

When tracing syscalls for pairs of events causally related, we intercept the syscall of the first event solely at its *entry* point and the syscall of the second event only at its *exit* point. Since *ptrace-based* tracing utilities do not guarantee that the two interception points of a syscall appear contiguously in the trace, this approach is crucial to correctly infer causality.

The happens-before model generator is written in Java and uses the Z3 solver [11] to solve the model. The causal trace produced by the solver is then output in JSON format.

Falcon's current visualizer is implemented as a JavaScript program that consumes the causal trace and generates a space-time diagram using the SVG.js library. We are currently extending Falcon to use *ShiViz* [10] as the visualization module, as it provides interactive analysis features.

V. Case Study: Apache Zookeeper

Apache Zookeeper [12] is an open-source, scalable and reliable service that enables distributed coordination. Zookeeper poses a good case study for Falcon as its implements complex algorithms and protocols for leader election and atomic broadcast, which are hard to analyze and understand in detail.

In a distributed deployment, Zookeeper runs with several servers, of which one is leader and the others are followers. Both roles are distinguishable in the sense that read requests can be served by the followers while write requests are handled only by the leader. For this case study, using Zookeeper 3.5.0, we analyze a setup containing two Zookeeper nodes that communicate with each other to elect the leader. In particular, our execution scenario consisted of setting up a standalone Zookeeper server and, then, adding a new node to the server quorum.

During the execution, we collected Zookeeper's built-in log file produced with the *log4j* logging library and used our *ptrace*-based tracer tool to record syscalls regarding thread synchronization events, connections, and messages exchanges.

As the output layout generated by *log4j* is configurable, we set the layout parameter to a custom Java class. In order to correctly identifying the thread responsible for logging a given message, we augment each log entry with a unique identifier consisting of the concatenation of both the thread and process ids. However, since the Java Virtual Machine does not allow accessing the native thread identifier from a high-level API, we rely on the Java Native Access to execute the *gettid()* system call and retrieve the thread id directly from the native operating system. The result of the syscall is thus introduced as a parameter in the output layout of *log4j*.

In the following, we show how Falcon can be used to analyze the execution of Zookeeper and evaluate the performance and scalability of the constraint solving procedure.

A. Falcon in Action

Figure 5 depicts the space-time diagram generated by Falcon for the logs collected during our Zookeeper execution scenario. The causal trace was obtained by solving the model with the timestamp soft constraints. The diagram shows that there are two main processes (5598 and 5670) that spawn several threads while running. For brevity, we include just the thread timelines relevant for this example. In other words, we discarded the threads that have only **START**, **END** and **LOG** events. However, they can be useful for conducting a more thorough behavior analysis.

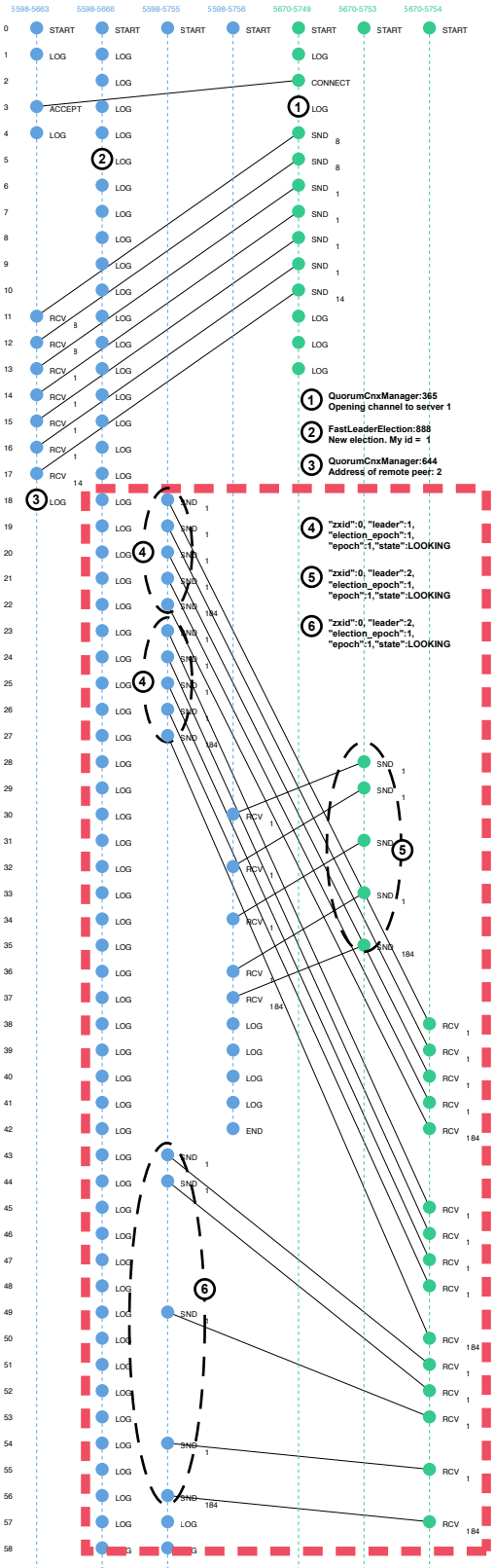

Fig. 5: Space-time diagram of two Zookeeper servers connecting to each other and performing a leader election. Vertical lines are thread timelines and circles denote the executed events on each thread timeline. The dashed and solid rectangles represent different protocol steps, namely the connection establishment between the peers and the leader election.

The **LOG** events correspond to the timestamped entries logged by *log4j*, while the remaining events were collected by the syscall tracer. To ease the understanding of the diagram, we highlight certain events with a circled number and display their message content (for **LOG** events) or payload (for **SND** and **RCV** events).

Figure 5 shows that, after booting, the server joining the quorum starts by connecting to the existing peer. In particular, the **LOG** events identified by ① and ③ reveal that the process 5598 is the node with the server identifier (sid) 1 while process 5670 has $sid = 2$. Note that the events also reveal the lines of code at which the messages were logged, namely lines 365 and 644 of the *QuorumCnxManager* class.

The most interesting part of the diagram is arguably the leader election procedure though, since it is one of the major applications of fault-tolerant consensus protocols. When the two quorum peers are connected, server 1 triggers a new leader election (see event ②). In Zookeeper, each server can be in one of the following states: *LOOKING*, *FOLLOWING*, *LEADING* and *OBSERVING*. At the beginning of the execution, all servers are in the *LOOKING* state and vote in themselves to be the leader by sending notifications to the other servers with the $leader$ field set to their sid (see messages ④, ⑤ and ⑥). If a server receives a notification with a sid higher than its own, then it updates its vote proposal to the higher sid and broadcasts the new vote proposal to the rest of the quorum. Note that, since there are no client requests during this execution, the last seen transaction identifier $zxid$ remains unchanged for both servers. Otherwise the servers would vote for the peer with higher $zxid$.

In a nutshell, the diagram of Figure 5 reveals the following leader election protocol in Zookeeper:

- Server 1 sends the vote message ④ with payload $\{\text{“}leader\text{”} : 1\}$ to server 2;
- Server 2 sends the vote message ⑤ with payload $\{\text{“}leader\text{”} : 2\}$ to server 1;
- Server 1 receives the vote message sent by server 2, updates its vote proposal for the latter because it has higher sid, and sends back the updated vote – message ⑥ with new payload $\{\text{“}leader\text{”} : 2\}$ – to server 2;
- Server 1 and server 2 update their state from *LOOKING* to *FOLLOWING*[5] and *LEADING*[6], respectively, as indicated by the log messages.

A further analysis of this space-time diagram also allows drawing some conclusions regarding the Zookeeper's behavior:

a) ***Notification timeout****:* The message ④ is sent twice due to a timeout that occurs when a server does not receive enough notifications within a given time frame.

b) ***Message partitioning****:* The sending of a message is actually partitioned into several **SND** events, i.e., to *write* syscall executions. Inspecting the Zookeeper's source code[7], we noticed that the messages sent during the leader election protocol are composed by an integer and a buffer. The integer is sent by executing the *write* syscall four times, while sending the buffer requires a single *write* invocation.

[5] *FastLeaderElection:636 - leader=2, ..., my id=1, my state=FOLLOWING*
[6] *FastLeaderElection:636 - leader=2, ..., my id=2, my state=LEADING*
[7] See class *QuorumCnxManager* at line 698.

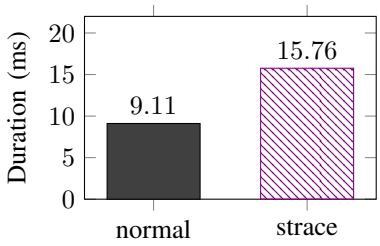

Fig. 6: Performance impact on Zookeeper execution caused by tracing with *strace*.

c) ***Causality****:* The diagram shows that messages ④ and ⑤ are not causally related, because the sender and receiver threads execute concurrently. In contrast, there is a causal relationship between messages ⑤ and ⑥ based on an state change. Upon reception, message ⑤ is added to a queue by the receiver thread. Afterwards, the sender thread dequeues the message, processes it (i.e., updates the vote proposal of server 1), and sends message ⑥ to server 2. Therefore, ⑤ → *receiver thread enqueues message* → *sender thread dequeues message* → *change of vote proposal* → ⑥.

B. Performance Impact of Syscall-level Tracing

We developed a micro-benchmark to evaluate the performance overhead imposed on Zookeeper due to tracing syscalls with *strace*. The micro-benchmark consists of a client issuing requests to two Zookeeper servers. Concretely, the client performs 10K iterations of four operations: *i)* check whether a znode exists, *ii)* create a new znode, *iii)* check again whether the znode exists, and *iv)* delete the created znode.

Figure 6 compares the average duration of an iteration (in milliseconds) between a vanilla execution of Zookeeper and an execution with *strace*. As expected, enabling the tracing affects negatively the runtime performance, causing Zookeeper to be $1.7\times$ slower comparing to the baseline.

C. Scalability of Constraint Solving

Depending on the debugging level, message logs may contain hundreds or thousands of entries. In order to better understand how the constraint solving time varies with an increasing amount of log entries, we ran the same configuration of Zookeeper for INFO and DEBUG logging levels. To further increase the log size, we duplicated the number of entries on both cases log. The resulting logs contained 568 events for the DEBUG level and 342 events for the INFO level.

The time required by Z3 to solve the constraint models for both log files is depicted in Figure 7. The results show that adding the DEBUG-level log events to the model caused the solver to take $3.18\times$ more time to find a solution than with an INFO-level log. Although a trade-off must be made between the duration of the constraint solving and the amount of information logged, we believe that Falcon is useful in practice and much more scalable than manual log analysis.

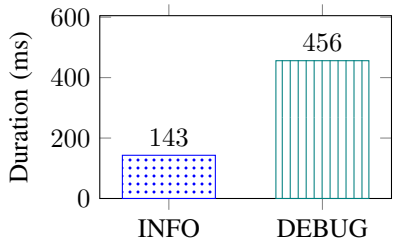

Fig. 7: Time spent on solving the generated models for both levels of debugging: INFO and DEBUG. Solving the model that contains DEBUG-level logs significantly increases the duration of model solving.

VI. Related Work

Similarly to Falcon, prior work has also focused on tracing distributed systems executions for performance and correctness analysis.

Sherlog [2] is a tool that analyses source code and runtime logs in order to isolate the cause of errors occurred at production. However, it assumes that the log is generated on single machine, hence not being suitable for distributed systems.

DTrace [13] and Fay [14] are two dynamic instrumentation systems able to help diagnosing performance degradation issues. Unlike Falcon though, they are not capable of establishing causality relationships between events.

X-Trace [8] is a tracing framework that provides a comprehensive view of distributed systems, using execution trees constructed from propagated metadata within each network protocol. Dapper [7] is a tracing infrastructure developed by Google to trace end-to-end requests in Google services. These traces are generated through RPC instrumentation, with support for annotations. Magpie [6] is an online modeling system that correlates interactions, or events, between components in order to generate a performance model of the system. Pinpoint [15] and Pivot Tracing [5] are similar to Magpie but allow the execution of distributed queries during the analysis process. Canopy [4] is an end-to-end performance tracing tool, developed by Facebook, that records causally-related performance data from an end-to-end execution path. Although these tools and frameworks are designed for diagnosing performance issues, they could be combined with Falcon in that their traces could be part of the Falcon's pipeline. Moreover, as these systems do not provide data visualization features, an integration with Falcon would further improve their usefulness for in-house log analysis and debugging.

VII. Conclusion

In this paper we introduce Falcon, an extensible tool pipeline for combining and visualizing log data from several data sources. The key contribution of this tool is the ability to merge log data from multiple sources and establish happens-before relationships between events, providing a coherent diagram for a visual analysis.

The tool is applied to Zookeeper and demonstrated with a syscall trace and *log4j* log files. The resulting diagram helps in understanding how remote threads interact among themselves and what messages were exchanged to execute a given task. Clearly, the combination of hundreds of events that is ordered by the SMT solver to produce a trace would clearly be unfeasible manually. Additionally, we assess the performance impact on syscall-level tracing in Zookeeper, showing that the impact is tolerable, in line with what is expected from a more detailed log level configuration in *log4j*.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable feedback. This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia as part of project UID/EEA/50014/2013. The research leading to these results has received funding from the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732051.

References

[1] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *SOSP '17*. New York, NY, USA: ACM, 2017.

[2] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from run-time logs," in *ACM SIGARCH Comp. Arch. news*, vol. 38, no. 1. ACM, 2010.

[3] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, 1978.

[4] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, and Y. J. Song, "Canopy: An end-to-end performance tracing and analysis system," in *SOSP '17*. New York, NY, USA: ACM, 2017.

[5] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *SOSP '15*. ACM, 2015.

[6] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: Online modelling and performance-aware systems." in *HotOS '03*, 2003.

[7] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Tech. Rep.

[8] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *NSDI '07*. USENIX Association, 2007.

[9] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distributed Algorithms*. North-Holland, 1988, pp. 215–226.

[10] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems: Challenges and options for validation and debugging," *Communications of the ACM*, vol. 59, no. 8, pp. 32–37, Aug. 2016.

[11] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS '08/ETAPS '08*. Springer-Verlag, 2008.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems."

[13] J. Mauro, *DTrace: Dynamic Tracing in Oracle® Solaris, Mac OS X, and FreeBSD*. Prentice Hall, 2011.

[14] Ú. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz, "Fay: extensible distributed tracing from kernels to clusters," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4, 2012.

[15] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *DSN '02*. IEEE, 2002.