HANSEL: Diagnosing Faults in OpenStack

Dhruv Sharma[*] UC San Diego
Rishabh Poddar[*] UC Berkeley
Kshiteej Mahajan U Wisconsin
Mohan Dhawan IBM Research
Vijay Mann IBM Research

ABSTRACT

With majority of the world's data and computation handled by cloud-based systems, cloud management stacks such as Apache's CloudStack, VMware's vSphere and OpenStack have become an increasingly important component in cloud software. However, like every other complex distributed system, these cloud stacks are susceptible to faults, whose root cause is often hard to diagnose. We present HANSEL, a system that leverages non-intrusive network monitoring to expedite root cause analysis of such faults manifesting in OpenStack operations. HANSEL is fast and accurate, and precise even under conditions of stress.

CCS Concepts

•**Computer systems organization → Cloud computing;** •**Networks → Cloud computing;** *Network monitoring;* •**Software and its engineering → Fault tree analysis;**

Keywords

Network monitoring, OpenStack, REST, RPC.

1. INTRODUCTION

With majority of the world's data and computation handled by cloud-based systems, cloud management stacks (CMSes), such as Apache's CloudStack [2], VMware's vSphere [14] and OpenStack [1], have become an increasingly important component in the cloud software. These CMSes are complex, large scale distributed systems enabling orchestration of tasks, including virtual machine (VM) spawning, termination, migration, etc. However, like every other distributed system, these CMSes are susceptible to a variety of complex faults, most of which are often hard to diagnose even for skilled developers/operators. For example, several faults [8–10] in Rackspace's cloud offering based atop OpenStack took several hours or even days to resolve. In this paper, we focus on the problem of automatically determining the cause of such hard to diagnose faults manifesting in OpenStack operations.

While cloud systems are resilient to hardware failures by design, software and human issues amount to ~87% of all failures in cloud systems [25]. Prior work on troubleshooting cloud systems has focused mostly on one of two approaches—model checking or event analysis. State-of-the-art distributed system model checkers [24, 26, 31, 32, 38, 43] leverage novel reduction policies for managing state space explosion and unearth hard to find bugs. In contrast, cloud and network troubleshooting systems [15,28, 29] mostly focus on fine-grained event analysis, correlating system behavior with event logs. These events vary depending on the system under consideration and may also require extensive instrumentation. However, correct implementation of both these approaches requires extensive knowledge of the entire system. Further, any additions or modifications to the system would require updates to both model checking policies and event analysis mechanism.

We present the design and implementation of HANSEL—a fast, lightweight approach based on network monitoring to diagnose faults in OpenStack. HANSEL systematically analyzes each OpenStack message between components, infers the message context, and uses it to chain together a sequence of events representing discrete actions, such as spawning or terminating VMs. HANSEL is novel because it (a) requires no change even on introduction of new component nodes, and (b) does not require any expensive system level instrumentation or offline log analysis. Further, techniques used by HANSEL are generic and thus potentially applicable to other distributed systems.

Practical fault diagnosis in OpenStack is hard for two main reasons. First, several routine OpenStack operations, such as instance creation, deletion, involve cross-service interaction for successful completion. However, errors occurring along the call chain are inferred differently, and

[*]Both authors contributed equally.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CoNEXT '15 December 1–4, 2015, Heidelberg, Germany
© 2015 ACM. ISBN 123-4567-24-567/08/06. . . $15.00
DOI: 10.475/123_4

thus often incorrectly propagated back, by the disparate components involved in the operation. Second, OpenStack is still under active development. Thus, an effective fault diagnosis solution must ensure compatibility with existing and future deployments. Routine feature upgrades, addition/removal of system components, or scale of deployment must not render the fault diagnosis useless.

Unlike prior work [15, 19, 22, 28, 29] for fault detection and diagnosis, HANSEL only requires network monitoring agents at each node in the distributed system to capture and relay OpenStack REST and RPC messages to a centralized analyzer service. HANSEL relies on a key observation that most distributed systems use unique identifiers in communication messages for identifying resources across components. HANSEL's analyzer service uses this property to systematically combine both REST and RPC messages across different distributed system components, while being agnostic of the API behavior and the message protocol.

HANSEL mines each OpenStack message for unique identifiers, such as 32-character hexadecimal UUIDs, IP addresses, source and destination ports, to create a message context. HANSEL then stitches together these contexts to update temporal states observed at each component in the communication, and generate a control flow graph corresponding to operator actions. This modeling of the actual temporal state along with several other heuristics help HANSEL prune the control flow graph and achieve precision. On detection of an error, HANSEL backtracks from the faulty node in the graph to identify the causal chain of events.

We have built a prototype of HANSEL for OpenStack JUNO. However, HANSEL's use of well defined patterns for detecting unique identifiers, without relying on any particular message format or specific sequencing of protocol messages, makes it applicable to any version of OpenStack, and potentially other cloud orchestration platforms. We evaluated HANSEL on a physical setup with 3 compute nodes, using the Tempest integration test suite [12] with over $1K$ test scenarios, and exercised 369 faults in our setup. We further stress tested HANSEL and observed that it can handle about $1.6K$ REST/RPC messages per second of control traffic.

This paper makes the following contributions:

(1) We present HANSEL—a system that analyzes network messages to model control flow and localize faults in OpenStack. To our knowledge, HANSEL is the first such system that relies only on OpenStack messages and requires no other system modifications.

(2) We list several scenarios (§ 3.1 and § 8.1) where existing mechanisms fail/mislead even skilled developers/operators.

(3) We provide a practical design (§ 5) for HANSEL, which mines unique identifiers from OpenStack messages to stitch together a stateful trail of control flow amongst the component nodes. We also present several heuristics (§ 6) and optimizations to improve HANSEL's precision.

(4) We apply HANSEL on OpenStack (§ 7), and evaluate it (§ 8) to demonstrate its speed, precision, and scalability.

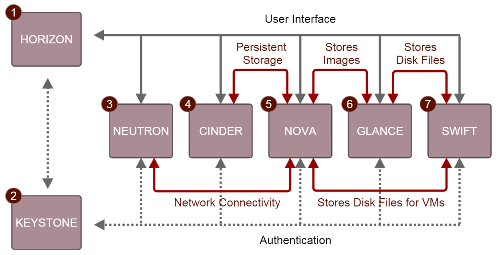


Figure 1: OpenStack architecture.

2. OpenStack BACKGROUND

OpenStack is a state-of-the-art, open source, Python-based CMS used by over 250 leading organizations [7]. OpenStack is a fairly complex system with over 2.5 million LOC. Fig. 1 shows a schematic architecture of a basic OpenStack deployment. Typically, each of these services run on different nodes with distinct IP addresses, and provide a command-line interface (CLI) to enable access to their APIs via REST calls. We now briefly describe these key constituents and their functionality.

(1) Horizon is a Web-based dashboard for OpenStack.

(2) Keystone is OpenStack's identity service, which provides authentication, authorization and service discovery mechanisms primarily for use by other services.

(3) Neutron provides "networking as a service" between virtualized network interfaces managed by other services.

(4) Cinder is a block storage service, which virtualizes pools of block storage devices, and provides end users with APIs to request and consume these resources.

(5) Nova provides a controller for orchestrating cloud computing tasks for OpenStack, and supports a variety of virtualization technologies, including KVM, Xen, etc.

(6) Glance provides image services for OpenStack such as the discovery, registration, and retrieval of VM images using REST calls, along with querying of VM image.

(7) Swift is an object/blob store, which enables creation, modification, and retrieval of objects via REST APIs.

OpenStack has external dependencies over MySQL for managing entries in the database, `oslo` messaging module for RPC communication using RabbitMQ, virtualization solutions like KVM and Xen, and Python-based HTTP clients for REST communication. Additionally, OpenStack mandates that all nodes be NTP synchronized.

2.1 REST/RPC communication

Inter-services communication within OpenStack happens via REST calls, and each OpenStack component has a corresponding HTTP client that enables access to its REST APIs. On the other hand, intra-service communication takes place exclusively through RPCs. Since a service may potentially be distributed across several nodes, all RPC messages are channeled through the RabbitMQ message broker. For example, communication between the Nova controller and Nova agents on the compute nodes happens via RPCs through the RabbitMQ broker.

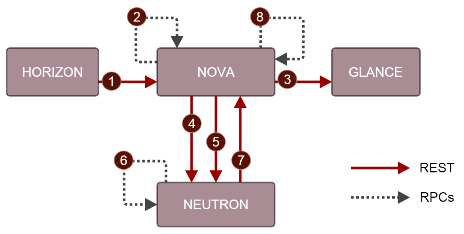


Figure 2: OpenStack workflow for launching a new instance.

To manage state and track progress of each request, all RPC messages share a unique request id, which is a 32-character hexadecimal string prefixed by "req-". This request id is also present in the response header of the REST call that initiated the ensuing RPCs. In OpenStack, however, only RPCs triggered by REST calls contain request ids. Other RPCs neither have request ids nor do they propagate any updated states across components. Thus, from the perspective of an OpenStack operation, where the result must be communicated to the dashboard/CLI via REST calls, RPCs without request ids are not part of user-level activity.

2.2 Example workflow

Fig. 2 shows a schematic workflow to launch a new instance in OpenStack. We omit invocations for authentication for sake of brevity. When a client launches an instance from the dashboard, Horizon leverages NovaClient to issues an HTTP `POST` to Nova to create a VM for the specified tenant (step 1). At this moment, the control migrates from Nova to the NovaCompute service on the compute node, where it initiates RPCs to build the instance (step 2). Nova then invokes the GlanceClient to issue an HTTP `GET` request to Glance to fetch the VM image and start the boot process (step 3).

Simultaneously, Nova invokes NeutronClient to issue a series of `GET` requests to Neutron to determine the existing network, port and security group bindings for the specified tenant (step 4). Nova halts the boot process and invokes Neutron APIs (with the `POST` request body containing the VM and network identifiers) requesting it to create and attach a new port for the VM (step 5). Neutron immediately responds with the newly created port identifier for the VM, which is not yet attached to the instance. Thus, Nova makes the request and blocks the boot while waiting for a callback from Neutron indicating that it has plumbed in the virtual interface. When Neutron has completed port attachment to the VM (step 6), it leverages NovaClient to issue a `POST` to Nova indicating the same (step 7). When all the events have been received, Nova continues the boot process (step 8).

3. MOTIVATION

Incorrect system configuration, resource exhaustion, access to privileged resources/actions, data corruption in the database, etc., may all be causes of faults in OpenStack operations. Further, OpenStack's dependence on third party libraries and utilities, such as RabbitMQ, Python and even system utilities, may trigger faults.

For the purpose of this paper, any event that manifests as an error message in OpenStack is treated as a fault. These error messages are readily identifiable using light-weight regular expression checks.

FAULT LOCALIZATION. A key indicator to initiating root cause analysis is the presence of error notification(s) on the OpenStack dashboard or the command-line interface (CLI). However, such an error may be merely indicative of *some* fault. In several scenarios, this key notification may be misleading, or even absent altogether, forcing developers/operators to mine the system logs to determine the actual source of the fault. The process would be expedited substantially if the fault could be localized to a specific OpenStack component/service.

IDENTIFYING CAUSAL SEQUENCE. Isolating a fault to a particular component is not enough to identify how the fault occurred. Once a fault has been localized, debugging the component effectively would require knowledge of the exact sequence of events that led to the fault. Further, in the presence of multiple, simultaneous administrative tasks in the cloud system, it is imperative for operators to identify which particular task resulted in the fault.

In this section, we illustrate the effectiveness of this two-pronged approach with the help of three specific, yet representative, scenarios that benefit from using HANSEL. We demonstrate how HANSEL's fault diagnosis coupled with causal sequencing of events assists both developers and operators. We also argue why prior work does not suffice for root cause analysis in production environments on the representative OpenStack scenarios discussed.

3.1 Representative Scenarios

We describe below three scenarios to show how HANSEL can expedite root cause analysis by localizing faults and identifying a causal sequence of events. While the scenarios themselves are concrete, they are representative of the typical cross-component communication patterns involving the major OpenStack constituents, as described in § 2.

3.1.1 VM create

DASHBOARD ERROR. In a particular VM create scenario, we observed that OpenStack abruptly terminates the entire process and Horizon simply reflects a "No valid host was found" error on the dashboard, even when there was one fully functional compute node available.

LOG ANALYSIS. Digging deeper into the logs, we observed that the Nova service throws two errors—"An error occurred while refreshing the network cache" and "No valid host found". Presumably, following the first error response, Nova scheduled the request on other compute nodes and failed to find an "enabled" compute node (as indicated by the second error log). Curiously, the Neutron logs on the compute node listed no errors. However, repeating the scenario with `TRACE` level logs enabled for Nova , we observed exceptions that pointed to a buggy NeutronClient [1]. Clearly, the exception on the dashboard and the errors in Nova logs are misleading, as they do not point to the root cause of the fault. Additionally, in a production setting, typically only `ERROR` log level is enabled. Thus, it would take a highly skilled and experienced operator to piece together the chain of events and determine the missing link [2].

HANSEL'S DIAGNOSIS. HANSEL's precise analysis of all causal events in the transaction reported that Neutron RPCs raised a "Connection to Neutron failed" error. Additionally, we also observed that the Nova `POST` request (from the compute node) to Neutron server (step 5, Fig. 2) for creating ports on the compute node was never sent out, indicating that a previous step had failed, i.e., step 4 to fetch network, port and security groups for the instance. This information coupled with failed Nova `POST` request, points to a buggy NeutronClient on the compute node.

3.1.2 VM delete

DASHBOARD ERROR. When deleting an instance, we observed that the dashboard marked the VM state as `ERROR` without explicitly mentioning the error.

LOG ANALYSIS. Analysis of Nova logs on the compute node listed two errors—"Setting instance vm_state to `ERROR`" and "Failed to deallocate network for instance". Strangely, Neutron logs again showed no errors. Enabling `TRACE` level debugging for Nova again pointed towards exceptions in the buggy NeutronClient. With no error on the dashboard, it is hugely frustrating for operators to determine the root cause of such faults.

HANSEL'S DIAGNOSIS. HANSEL reported that the Neutron RPCs raised a "Connection to Neutron failed" error, similar to the earlier scenario. Further, the chain of events showed no `DELETE` message sent out to delete the ports on the VM instance was never sent out. Note that a `DELETE` request must be issued to delete a resource. These two facts together again point to a buggy NeutronClient.

3.1.3 Attach VM to an external network

DASHBOARD ERROR. An instance creation requires plumbing virtual interface to an existing network. However, if the existing tenant network virtualization, e.g., VxLAN, GRE, etc., does not match the network configuration, e.g., flat networks, then OpenStack aborts the action. We replicated this scenario and attempted attaching an instance to an external network when creating it. We observed that the dashboard reflected a "No valid host found" error, even when several compute nodes were available.

LOG ANALYSIS. Examining the logs, we observed that Nova throws an exception about the `libvirt` driver failing to attach the virtual interface to the instance. Subsequently, Nova fails to schedule the VM on a "valid" host. No other errors/exceptions were observed in the logs. However, the real reason for the fault was hidden as only a warning in Neutron logs, which indicated that the virtual interface binding to the external network was not possible. Such erroneous reporting on the dashboard and even the logs significantly increase time for root cause analysis.

HANSEL'S DIAGNOSIS. Using HANSEL, we observed that the Neutron `REST` response for Nova's `POST` to create and bind ports (step 5, Fig. 2) indicated a binding failure of the virtual interface. Subsequently, we observed no `POST` callback (step 7, Fig. 2) from Neutron is not sent to Nova.

The above examples clearly demonstrate the merit in network monitoring and analysis for root cause analysis, in contrast to the log based inference alone. Later in § 8.1, we describe other interesting scenarios where the dashboard and logs do not readily point to the root cause of faults and network monitoring can significantly expedite the process.

3.2 Difficulty of fault diagnosis

Several routine OpenStack operations, such as instance creation and deletion, involve cross-service interaction for successful completion. However, errors occurring along the call chain are inferred differently by the disparate components involved in the operation. For example, as described in § 3.1, errors encountered by Neutron during its operations (such as "VIF binding failed") are escalated by Nova to the dashboard as generic failures in VM creation, like "No valid host found".

Model checkers [24, 26, 31, 32, 38, 43] characterize the behavior of distributed systems and offer guarantees about component behavior. However, due to their static nature, they cannot diagnose faults that arise in a running production system. Other approaches [15,19,22,28,29] have advocated analysis of system events either through logs or instrumentation. Both these approaches, while being useful, have implementation and deployment limitations. Not only do they require complete knowledge of the entire system, their success actually depends on the intrusiveness of the implementation. Effectiveness of log analysis is limited by the comprehensiveness of the system logs and their debug level, while system instrumentation to track faults requires modification of the entire source code, recompilation and subsequent deployment.

4. SEQUENCE STITCHING PROBLEM

The problem of network monitoring for root cause analysis in OpenStack crystallizes to determining the correct sequence of REST and RPC calls for every administrative task that leads to a fault. Below we present the problem as finding the correct transitions in an event sequence (ES) that drive the system from a given state to the error state.

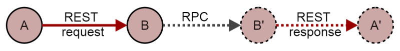
MODEL. We model a transaction in OpenStack as an ES, and each REST and RPC message as a transition that drives the system to a new state. In OpenStack, RPCs update

[1]While buggy clients are not common, they are possible due to incompatibilities with third party Python libraries.

[2]Developers may augment HANSEL's diagnosis with additional logs available in development settings for root cause analysis.

the system state for the same component, while RESTs potentially update states across components (recall § 2). Directives issued from the dashboard or CLI, i.e., REST calls that initiate new administrative tasks (or "transactions"), create new start states in the ES, corresponding to different transactions. All other REST/RPC messages result in intermediate states. An error state is one where the REST/RPC transitions report an error.

A transaction is thus a sequence of temporally related REST and RPC transitions between states. In particular, (a) REST requests issued to components create new states in the ES for those components, and (b) RPCs and REST responses create updated states of components in the ES. Thus, REST requests signify transitions from one component to another, while RPCs update the state of existing components. REST responses update the state of the component initiating the REST request. As an example, the figure below shows the ES for a transaction involving a REST request from a component A to B, followed by an RPC at B, which updates its state to B'. The REST response from B' updates the state of A to A'.



Problem Definition. Given the described model, our problem reduces to answering the following:
• How to accurately model the state of a component based on just network messages, and
• How to identify the correct placement of REST and RPC messages connecting different states.

Challenges. The problem of finding valid transitions gets exacerbated in OpenStack for the following reasons:

C1. A transaction may span several REST/RPC calls, and multiple such transactions occur simultaneously in OpenStack. Thus the model as described above runs the risk of state space explosion.

C2. Occurrences of simultaneous transactions make it difficult to segregate them, and isolate the component states corresponding to different transactions from each other.

C3. Inter-service control flow via REST calls only identifies the destination (based on the well-defined destination port); the source of the REST request is unidentified as ephemeral ports are used for communication.

C4. Control for components like Nova may flow across several compute nodes via RPCs during a transaction. Hence, its state may be distributed across several nodes. Placement of transitions must account for such control flow.

5. Hansel

Key Idea. OpenStack makes extensive use of unique identifiers to describe and locate resources in the system, and the different system nodes maintain internal states corresponding to each of these identifiers. Guided by this observation, Hansel analyzes network messages flowing across different components to mine these unique identifiers, and leverages them to construct an execution trail of events.

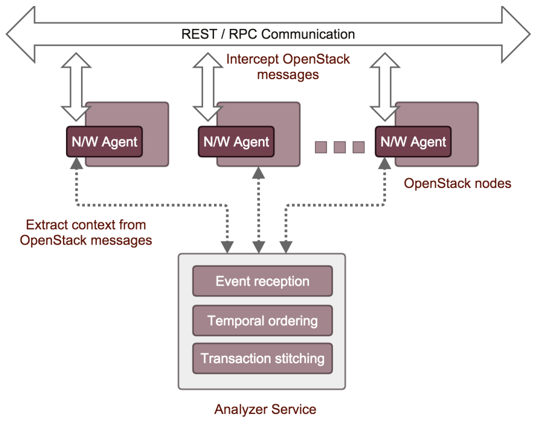


Figure 3: Schematic architecture of Hansel.

On detection of a fault, Hansel backtracks the execution graph to string together the exact sequence of events that possibly lead to the fault. Fig. 3 shows a schematic architecture for Hansel, which comprises of a distributed setup of network monitoring agents and a central analyzer service. Hansel sits transparently underneath the existing deployment, without affecting *safety* and *liveness* of the distributed system. Hansel supports precise fault detection using three key features:

5.1 Distributed network monitoring and context extraction

Hansel requires network monitoring agents at each node in the OpenStack deployment since it must monitor *all* OpenStack messages sent amongst the nodes. These agents mine each network message for unique identifiers, such as the 32-character hexadecimal UUIDs, IP addresses and OpenStack message ids (which are 32-character hexadecimal strings prefixed by "req-"). These identifiers are used to create a message context, which is communicated to a central analyzer service along with additional metadata about the OpenStack message, including the source/destination, protocol (i.e., REST or RPC), and the OpenStack message body. Mining the unique identifiers at the network agents expedites computation of the execution graph at the central analyzer service.

5.2 Execution graph construction

Hansel's centralized analyzer service extracts the message contexts communicated by the network agents to construct an execution graph of network events, which is a representation of the event sequence model described in § 4. The creation of such a graph requires: (a) temporal ordering of messages (containing contexts) received from the network agents, and (b) stitching together related message contexts into a logical transaction.

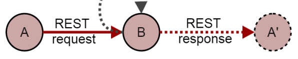
5.2.1 *Temporal ordering*

In spite of the timestamp consistency across OpenStack nodes ensured by NTP synchronization, messages from the network agents may still arrive out of order. Hansel ensures that such out-of-order messages can be temporally ordered, by leveraging a continuous stream (or pipeline) of "time buckets" (or t-buckets), each having a start and end timestamp. Only messages that have timestamps falling within these time-bounds of a t-bucket, can be added to it. Messages within each t-bucket are further ordered using an inexpensive sorting mechanism.

Each t-bucket remains active for a stipulated period of time during which new messages can be added to it. This ensures that even the messages that have timestamps belonging to the given t-bucket but have been received late due to network and buffer delays, can still be processed in a sorted order. The expiry of a t-bucket is marked by the expiry of an associated timer. Messages that are timestamped later than the newest bucket lazily generates a new t-bucket. Messages marked earlier than the oldest active t-bucket in the pipeline are discarded. The above mechanism ensures a reasonable temporal ordering of messages. However, the granularity of the timer and the bucket window length are system dependent and should be based on the traffic observed at the analyzer service. Note that higher expiry times may lead to delayed message processing.

5.2.2 *Transaction stitching*

Determining a complete transaction sequence in our execution graph entails finding the correct transitions in the event sequence as described earlier in § 4. Before we present our methodology of transaction stitching, we first address the problem of state space explosion (C1) that accompanies the ES approach. The number of RPCs in a typical OpenStack deployment significantly outnumber the REST calls. We note that the response header of the REST call that initiated the related RPCs contains the OpenStack request id shared across the RPCs (per § 2). Since, a REST request-response pair can be easily identified from the message metadata, we can also determine the parent REST request that resulted in the ensuing RPCs. We therefore collapse all consecutive RPC transitions with the same request id into the state created by the parent REST request, thereby significantly reducing the states in the execution graph. The states in our graph are now linked by REST transitions only, while RPC transitions are merely self edges. Recalling the example describing our model, transaction $A \to B \to B' \to A'$ reduces to $A \to B \to A'$ as shown below.



Approach. HANSEL processes the messages from the t-buckets as follows. A REST request to a particular component (e.g., from A to B in the above example) results in the creation of a new state in the execution graph for that component (i.e., B), initialized with the context extracted from the incoming REST request. Ensuing RPCs augment the state thus created with their own contexts. The problem of transaction stitching, therefore, reduces to the identification of an already existing state in the execution graph that initiated the REST request, i.e., the identification of state A in the above example.

Subset Inequality. We identify state A as follows. A REST request to B could have originated from a state A only if the context in the REST request is a subset of the contexts available at state A. Further, A acquires its contexts either from an incoming REST request, or computes it using the ensuing RPCs, or both. Thus, for a REST transition from state A to B, the following must hold true:

$$A \to B : \mathbb{C}(A_{\text{REST}_{\text{req}}}) \cup \mathbb{C}(A_{\text{RPC}}) \supseteq \mathbb{C}(B_{\text{REST}_{\text{req}}})$$

where $\mathbb{C}(X)$ represents the set of contexts extracted in X. Subsequently, HANSEL creates a copy A' of state A and augments it with the unique context received in the REST response from state B:

$$\mathbb{C}(A') = \mathbb{C}(A) \cup \mathbb{C}(B_{\text{REST}_{\text{resp}}})$$

All further transition placements from A must account for this state update and HANSEL must place subsequent transitions starting from state A' instead of A, thereby maintaining a causal sequence of transitions.

Example. Fig. 4 depicts our approach with the example transaction from § 2.2, launching a new instance in OpenStack. HANSEL processes the REST request from Horizon (step 1) to create a new start state in the execution graph ($A1$), corresponding to Nova. Ensuing RPCs update $A1$ with their contexts. The subsequent GET request to Glance in step 3 creates a new state B corresponding to Glance. HANSEL stitches the new state B to $A1$ based on our approach, and creates an updated state $A1'$ for Nova using the REST response from Glance. The process continues similarly, resulting in the construction of the graph as shown, where the A states correspond to Nova, B to Glance, and C to Neutron. Note that the REST in step 5 creates a new state $C2$ for Neutron, and does not update $C1$.

Achieving Precision. While the above technique for transaction stitching can identify and chain together a complete transaction, it may result in spurious linkages between states belonging to different transactions occurring simultaneously. This is because the context of a particular state may be entirely composed of generic identifiers, such as tenant ID, subnets, ports, etc., causing HANSEL to identify multiple parents for that state, some of which may belong to other transactions in the execution graph.

For example, an HTTP GET request for ports associated with a specific tenant may get incorrectly attached to several other transactions if the algorithm's subset inequality is satisfied for other states. Note that *all* the extraneous state transitions are valid as they can be potentially reached from the start state via the CLI using REST APIs (recall § 2). However, such transitions are undesirable as they are not part of the actual user transaction, leading to several plausible chains of events, making it hard to determine the actual sequence of states corresponding to the transaction.

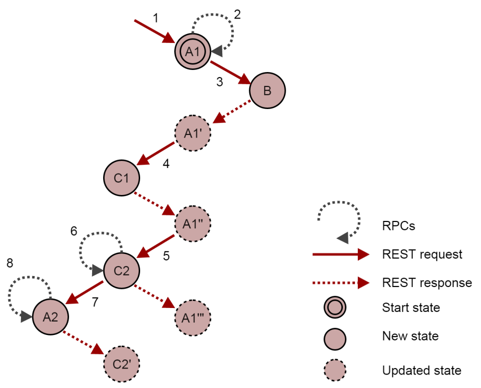


Figure 4: Execution graph for launching a new instance.

We address these issues in § 6, where we describe several heuristics to achieve precision, and overcome the remaining challenges (C2–C4) listed in § 4.

5.3 Fault diagnosis

As a message context is consumed from *t*-buckets to update the execution graph, HANSEL also scans the OpenStack message body (available from the context) to detect the presence of faults. This is straightforward for RESTs, where errors are indicated in the HTTP response header. For RPCs, however, diagnosing faulty messages requires domain-specific knowledge of OpenStack so that error patterns are accurately identified in the message body. To ensure this operation remains lightweight, HANSEL does not parse the JSON formatted message body; it simply uses regular expressions to identify error codes in the message, if any. Once a fault is identified, HANSEL backtracks from the faulty state in the execution graph to the start state, determining the sequence of events that led to the fault.

6. IMPROVING PRECISION

We now describe several heuristics that enable HANSEL to prune the undesirable transitions between states (per § 5.2.2). Note that all these heuristics are generic enough and do not bind HANSEL to any specific version of OpenStack.

6.1 HTTP modifiers

The HTTP/1.1 protocol contains several directives including `GET`, `POST`, etc., that enable clients to operate on resources described in the URI. However, except `POST`, all other directives, i.e., `GET`, `HEAD`, `PUT`, `DELETE`, `OPTIONS`, and `TRACE`, are idempotent in behavior (RFC 2616, Sec 9) [6]. HANSEL implements the following three heuristics for these resource modifiers to remove undesirable linkages:
(**1**) We disregard repeat occurrences of idempotent actions for a specific URI. In other words, HANSEL's algorithm ignores a REST message corresponding to an idempotent method for a specific resource if the exact same request/response has been seen in the transaction's past, since it does not result in any state update. For example, Nova might issue multiple identical `GET` requests to Neutron (say, querying for ports associated with a tenant) in the same transaction while waiting on a status update. HANSEL however, ignores all the intermediate repeat requests until a new, updated status is finally reported to Nova.
(**2**) There are several modifier sequences that are unlikely to occur in a transaction. For example, it is unlikely for a transaction to create a resource and then immediately delete it, i.e., a `POST` immediately followed by a `DELETE` for the same resource. Other such transitions include: `PUT` followed by `POST`, `PUT` followed by `DELETE`, and `DELETE` followed by `PUT`, for the same resource. However, HANSEL's transaction stitching using the context subset inequality may introduce these unlikely (and thus undesirable) transitions as linkages in the execution graph, thereby reducing precision.
(**3**) Several `GET` requests may interleave two modifier requests, such as `PUT`, `POST` or `DELETE`. It is desirable that a subset of the contexts extracted from `GET` responses between the two modifiers be utilized in the second modifier request, assuming the absence of any irrelevant `GET` requests.

6.2 IP/Port distinction

OpenStack setups typically install different services on separate nodes for scalability and fault isolation. HANSEL leverages this observation to further improve the precision.
(**1**) Placing Horizon and CLI on a node with a distinct IP address helps HANSEL accurately determine the starting point for each user-level transaction. All other transactions with source IPs different from the Horizon/CLI node represent intermediate sub-transactions.
(**2**) OpenStack services use REST calls for inter-component communication, while intra-component communication occurs using RPC messages. This observation helps prevent addition of transitions (i.e., REST calls) across states for the same component. For example, Nova does not invoke its own APIs using REST. Thus, there should not be any REST transition between two Nova states in the execution graph.
(**3**) A service may be distributed across multiple nodes in the deployment, and control may migrate from one node to another within the service via RPCs (per § 2). Thus, before stitching a REST request to a possible parent state, HANSEL also analyzes the RPCs at the parent state to identify a list of possible destination IPs to which control could have migrated. The REST request is stitched to the possible parent only if the source IP of the request is either the IP of the parent, or is part of the list of possible destination IPs.

Note that even if services share IP addresses, HANSEL would only require the various OpenStack HTTP clients to report the source service as an HTTP request header to prevent spurious linkages. No further changes would be needed in the core deployment.

6.3 Purging

Long-running transactions in OpenStack see continuous activity, usually in the form of status update requests. We leverage this observation to distinguish between completed transactions and ongoing ones. Specifically, if a particular transaction does not get updated with any new events in a considerable period of time, we deduce that the transaction has been completed. Pruning such completed transactions from the execution graph not only prevents spurious linkages in future, but also reduces the maintained state space.

We, thus, regularly purge completed transactions from HANSEL's data structures by pre-determining a threshold M of the number of messages processed by the analyzer service among which consecutive events in the transaction must be observed. We augment the construction of the execution graph (as described in § 5.2) by maintaining an ordered list of "purge buckets" or (p-buckets). As and when messages are processed to update the execution graph, the created states are also added to the current active p-bucket. Once μ new messages have been processed, a new p-bucket is added to the list of p-buckets in order to receive new states.

This mechanism enables HANSEL to easily identify transactions that have not seen any recent activity. Let $B_{1...n}$ be the list of all p-buckets, with B_n being the latest (the current active) bucket, and B_1 the oldest. Then, all transactions that have not seen any new states in the last $\lceil M/\mu \rceil$ p-buckets are assumed to have been completed, i.e., all transactions that lie completely in buckets $B_1 \ldots B_{n-M/\mu}$ are deemed complete. These transactions appear in the execution graph as disconnected components, and are purged. An alternate approach to state count-based p-buckets would be to similarly have a time-based purge mechanism where transaction that do not have any new events for a threshold period of time are purged from the execution graph. The thresholds in both cases are system-dependent, and need to be empirically determined.

7. IMPLEMENTATION

We have implemented a prototype of HANSEL for OpenStack JUNO based on the design described in § 5 and § 6. We built HANSEL's distributed network monitoring agents atop Bro [3, 34], which we augmented with a custom protocol parser for the RabbitMQ messaging protocol (60 LOC in C++). We leveraged Python-bindings to Bro's communication library (Broccoli [4]) to send events from the monitor nodes to our central analyzer service, written in ~800 LOC in Python. We also implemented an interactive Web-based visualizer tool to validate HANSEL's output.

MULTI-THREADED ARCHITECTURE. The analyzer service is implemented as a multi-threaded application with three primary components—the Event Receiver, the Pre-Processor, and the Transaction Chainer. Each of these components executes as a separate thread and interacts with the others using shared memory and exposed APIs. The Event Receiver's sole responsibility is fast reception of events from the distributed Bro monitoring agents via Broccoli connections. It writes the received events to a buffer shared with the Pre-Processor. The Pre-Processor employs binary search to place these possibly out-of-order events into time buckets, to ensure temporal ordering (per § 5.2.1). It maintains a local timer to periodically purge any expired time buckets from the active pipeline, and makes them available for the Transaction Chainer to consume. The Transaction Chainer processes the events in these expired buckets to construct the execution graph (per § 5.2.2).

OPTIMIZATIONS. We employ a number of optimizations to improve the performance of the analyzer service. First, we implement the shared buffer (co-owned by the Event Receiver and the Pre-Processor) as a circular buffer to minimize contention between the two threads, and avoid any loss of events during the wait. Second, we defer actions such as sorting of events within t-buckets, error detection, etc., till t-bucket timer expiry is triggered and the bucket is waiting to be consumed, thereby reducing processing delays in the active pipeline. The t-bucket sorting uses Python's built-in Timsort algorithm, which is well suited to our use-case.

8. EVALUATION

We now present an evaluation of HANSEL. In § 8.1, we present three case studies highlighting HANSEL's utility in effective fault localization and causal analysis for OpenStack operations. In § 8.2, we use the Tempest integration test suite to evaluate HANSEL for its precision in uniquely identifying the transaction for every fault introduced. In § 8.3, we evaluate the effectiveness of several optimizations on HANSEL's accuracy. In § 8.4, we measure HANSEL's network and system overhead under conditions of stress.

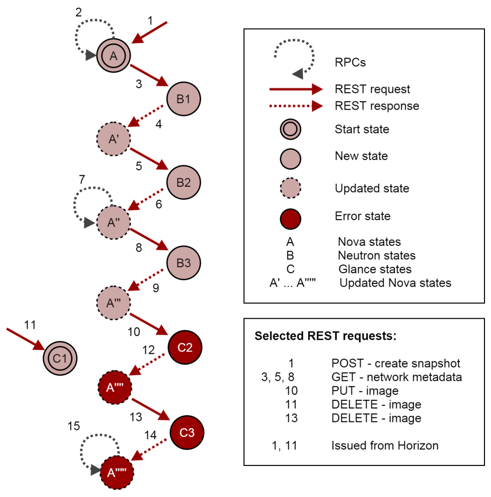
EXPERIMENTAL SETUP. Our physical testbed consists of 7 servers (including 3 compute nodes) connected to 14 switches (IBM RackSwitch G8264) arranged in a three-tiered design with 8 edge, 4 aggregate, and 2 core switches. All of our servers are IBM x3650 M3 machines having 2 Intel Xeon x5675 CPUs with 6 cores each (12 cores in total) at 3.07 GHz, and 128 GB of RAM, running 64-bit Ubuntu v14.04. We installed OpenStack JUNO (v2014.2.2), with each component on a different server. The inter-component OpenStack traffic and Bro-to-analyzer service communication was isolated to avoid any performance penalties. The values of t- and p-buckets were evaluated empirically to be 32s and 500 messages, respectively.

8.1 Accuracy: Case studies

In this section, we present three scenarios highlighting HANSEL's utility in the diagnosis of faults in OpenStack operations. While the list here is far from exhaustive, they are representative of HANSEL's effectiveness in diagnosing other faults as well. The scenarios are also representative of the typical cross-component communication patterns involving the major OpenStack constituents.

8.1.1 Delete VM image while saving snapshot

OpenStack enables operators to create snapshot images from an active instance. This scenario involves interaction amongst multiple services, including dashboard/CLI, Nova,



Selected REST requests:

1	POST - create snapshot
3, 5, 8	GET - network metadata
10	PUT - image
11	DELETE - image
13	DELETE - image
1, 11	Issued from Horizon

Figure 5: Execution graph for case study in § 8.1.1.

and Glance, to create an instance snapshot and upload it to Glance. Once the request is initiated from the dashboard, it issues a `POST` request to Glance for creating a reservation for the image. The image identifier received from the reservation is passed to Nova in a `POST` request to create and eventually upload the image to Glance.

Once the hypervisor (in our case, KVM with `libvirt`) finishes extracting the image, the dashboard reflects this change in status of the operation from `Queued` to `Saving`. However, post the image extraction, the completion of the operation requires two additional steps—(i) writing the extracted image to a temporary location on the system performed by `libvirt`, and (ii) subsequent upload of this image file to the Glance server. While either of these stages are in process, other simultaneous operations can be performed on the image and we briefly discuss them below.

If the image is accidentally deleted from Glance during the first stage of disk write by `libvirt`, no error is reflected on the dashboard. However, Nova logs on the compute node reveal "Exception during message handling: operation failed: domain is no longer running", which is clearly not very informative. Re-running the scenario with `TRACE` debug level, we note a `libvirt` "virDomainManagedSave() failed" error, indicating the real cause of the fault.

If the `DELETE` request is issued during the snapshot upload to Glance, still no error is reflected on the dashboard. The only errors detected in the logs are (i) "Exception during message handling: image may have been deleted during the upload", and (ii) a subsequent failed cleanup attempt by Nova (to delete the failed upload that in turn fails due to absence of the specific image entry on Glance). Glance logs show no errors in either case.

HANSEL'S DIAGNOSIS. As shown above, log analysis may miss out on diagnostic clues due to the log levels and the often vague nature of error messages. In contrast, HANSEL reported all the errors in each of the above scenarios. Unlike, log analysis, HANSEL also detected the failed `PUT` and the subsequent cleanup initiated by Nova. Fig. 5 presents the output of HANSEL's visualizer for this case study. Steps 1–9 involve creating a snapshot of the VM image. The `DELETE` REST call in step 11 halts the `PUT` request (step 10) to upload the VM image on Glance, and subsequently deletes it. As a result, Glance responds with an error (step 12) captured in state $C2$. Following the failed `PUT` call, Nova initiates a cleanup operation (step 13), which also fails as shown in node $C3$. While the *real* cause of the error was the `DELETE` operation (step 11), HANSEL in its present form does not link faults across transactions. Note that 11 is a separate transaction by itself. Thus, HANSEL did not pinpoint the `DELETE` operation as the cause of the fault. We leave such cross-transaction diagnosis for future work.

8.1.2 External dependency crash

OpenStack relies on several third party dependencies for correct functioning. For example, the Neutron agent on the compute node requires presence of a functional OpenvSwitch (OVS) to create port bindings for every VM scheduled for creation on that node. However, if the OVS on the compute node goes down, the Neutron agent crashes immediately. If an operator schedules creation of a VM on such a compute node, the dashboard throws "No valid host found". Interestingly, Neutron logs show no faults.

HANSEL'S DIAGNOSIS. In contrast, HANSEL's analysis of the entire sequence of events indicated (i) Neutron RPCs with failed interface bindings, and (ii) the same being communicated to Nova (i.e., Neutron's response to Nova's `POST` in step 5, Fig. 2), which in the absence of another candidate compute node, ultimately resulted in the dashboard displaying the "No valid host found" error raised by Nova scheduler. While, HANSEL does not detect the root cause of the fault, it does determine the OpenStack component responsible for faulty behavior.

8.1.3 Delete a network with existing VM

In addition to scenarios where fault notifications are either not propagated back to the operator or are misleading in nature, there can be several use-cases where the notifications are not informative enough for the operator to take a corrective action. This scenario presents a case where an operator issues a `DELETE` request for a virtual network from the dashboard but the operation fails. The dashboard notification indicates no definite reason for the failure.

There can be multiple plausible reasons for the failure such as a Neutron controller service component crash, inaccessibility of the database service, or existing ports that are currently connected to that network. Absence of such details may leave the operator clueless about the corrective actions that must be taken to resolve this issue, and leaves no choice other than to look for existing ports on the virtual network, or to mine the log files of related components for any crashes or disconnections.

As part of each network delete operation, any existing subnets of the network must be deleted first. In this scenario,

Category	Tests	Txns	Events				States
			RPC	REST	Error	Drop	
Image (Glance)	100	3.7K	28.0K	16.4K	50	1	2.0K
Mgmt (Nova)	218	5.9K	52.3K	26.6K	125	3	4.8K
N/w (Neutron)	109	2.2K	15.5K	9.0K	45	0	1.2K
Storage (Cinder)	58	1.0K	3.5K	3.9K	25	0	0.4K
VM (Nova)	224	8.2K	87.7K	37.0K	124	173	5.8K
Total	709	21.0K	187.0K	92.7K	369	177	14.2K

Table 1: Characterization of the Tempest test suite.

we had an existing VM instance with a network port attached to one of the subnets of the virtual network. This caused the subnet deletion to fail, consequently leading to the failure of the network delete operation.

HANSEL'S DIAGNOSIS. HANSEL was able to correctly capture and mark the erroneous DELETE request for the subnet along with the encapsulated message in its response stating "Unable to complete operation on subnet. One or more ports have an IP allocation from this subnet.".

8.2 Identification of unique transaction

We define the precision η of identification of unique transaction per error as:

$$\eta = (N - n)/(N - 1)$$

where N is the total number of possible parents, and n is the actual count reported by HANSEL. If HANSEL correctly identifies a fault to just one transaction, i.e., $n = 1$, $\eta = 1$. In the worst case with $n = N$, $\eta = 0$.

REPRESENTATIVE SCENARIOS. We leverage cases from the Tempest OpenStack integration test suite [12] and analyze them with HANSEL to determine its precision when executing several transactions in parallel. Our choice of using Tempest is based on several design goals and principles it follows and are discussed subsequently.

(**1**) Tempest is designed to run against any OpenStack deployment, irrespective of the scale, the environment setting or the underlying services that the deployment uses.

(**2**) Tempest accesses only the public user-level APIs that all OpenStack components expose through their REST clients and CLI. Therefore, all tests that Tempest runs and validates, are user-level transactions that a tenant or administrator is allowed to make in a real-world setting via OpenStack's API.

(**3**) Tempest provides a comprehensive set of functional and integration test cases derived from actual scenarios occurring daily in fully operational OpenStack deployments. These scenarios include actions spanning single or multiple nodes.

(**4**) Since the Tempest test suite can run on large deployments, it provisions running it's testcases in parallel to stress the OpenStack deployment close to the degree that it would endure during peak usage cycles.

TEST SUITE CHARACTERIZATION. The latest Tempest OpenStack integration test suite has $\sim 1K$ tests of which 709 executed successfully on our setup, including both the "positive" and "negative" scenarios. The rest were not applicable for our setup and thus skipped by the test harness.

All tests that included operations such as creation, deletion, updation, migration, etc., of instances, were classified as VM. Tests that update tenant network configuration, ports, routers, etc., were classified under Network. Management tests included updating host lists, key pairs, availability zones, tenant quotas, etc. Tests involving block storage were marked as Storage, while those involving VM images are categorized under Images.

We executed each of the applicable 709 Tempest tests in isolation and observed its network communication using HANSEL. Table 1 reports our findings for each category described above. TESTS lists the number of tests executed in each category. TXNS indicates the sum total number of top level transactions initiated for all tests in the corresponding category. The REST and RPC columns list all the network messages processed by HANSEL. These messages also include periodic updates, heartbeats, etc., which HANSEL prunes before creating execution graph.

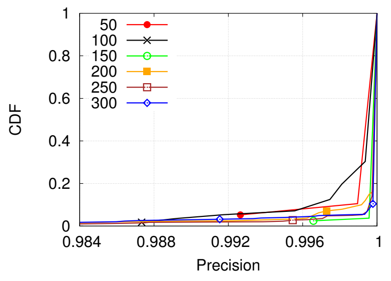
ERRORS indicates the erroneous network messages received across all tests, while DROPS shows the messages received but discarded as their corresponding t-bucket had already expired. The low number of drops with respect to the high count of REST and RPC events indicates the effectiveness of the t-bucket expiry time in handling out-of-order events. STATES shows the sum total of nodes generated per execution graph by HANSEL across all tests in that category. The effectiveness of HANSEL's core algorithm, optimizations and other heuristics is evident from the low number of states in the execution graph considering the high number of REST and RPC messages received.

PARALLEL WORKLOAD. We determine HANSEL's effectiveness, in localizing faults to unique transactions in a more realistic setting, by randomly selecting Tempest tests proportional to their distribution in the test suite and executing them concurrently. The executed tests include a mix of both erroneous and successful tests, as well as the different categories described earlier. Fig. 6a plots HANSEL's precision for varying number of parallel tests, ranging from 50 to 300 in increments of 50. Each test case spans several distinct transactions. We observe that HANSEL's precision (>98%) does not vary significantly even with a high number of concurrent transactions.

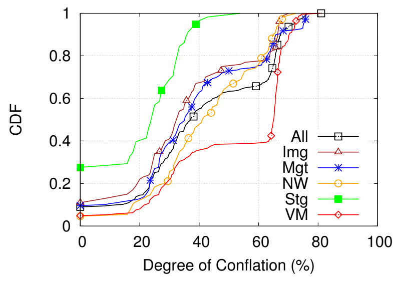
8.3 Effectiveness of optimizations

8.3.1 *Impact of* RPC *state reduction*

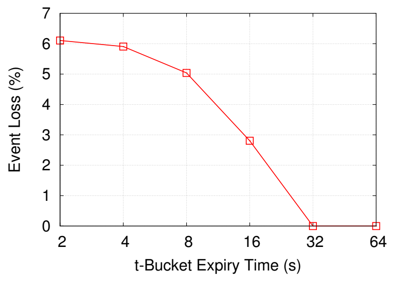
We evaluate the impact of RPC state conflation described in § 5.2.2 using execution results from all 709 Tempest tests to calculate relative savings in state space. Fig. 6b plots the aggregated and individual results for all the categories of tests listed in Table 1. We observe that our optimization shows significant benefits across all the tests, with >30% tests demonstrating >60% reduction in state space. We further observe that tests in the VM category show the highest savings with >60% state space reduction for >60% VM-based tests. Since a large fraction of all OpenStack



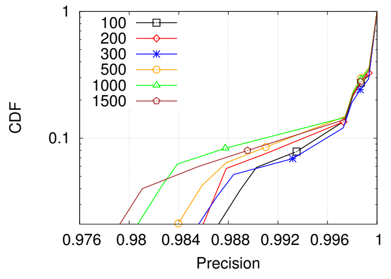
(a) Precision for parallel workload.



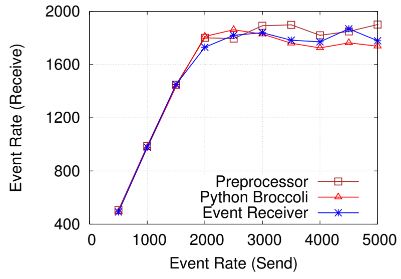
(b) State space savings due to RPC conflation.



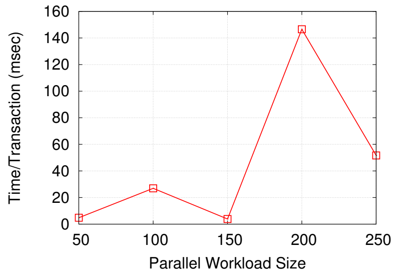
(c) Impact of t-bucket on temporal ordering.



(d) Impact of p-bucket size on precision.



(e) Overhead of communication pipeline.



(f) Time spent within chainer per transaction.

Figure 6: HANSEL's precision and effectiveness of optimizations.

actions also involve VM based operations, this optimization would yield significant savings. In contrast, the Storage tests yield minimum benefits with 50% of them showing <25% reduction in state space. On average, the RPC state conflation optimization saves ~61% of all states managed by HANSEL.

8.3.2 *Impact of t-buckets*

To evaluate the impact of t-buckets for temporal ordering of messages (per § 5.2.1), we selected 30 Tempest tests across all categories proportional to their distribution, and executed them in parallel on our system, with varying timer expiry for the t-buckets. We subsequently measured the count of messages received but discarded (because their corresponding t-bucket had already expired). Fig. 6c plots the results with varying t-bucket expiry times from 2s to 64s incremented exponentially. We observe that at 2s, HANSEL reports drop rates of ~6%, which drop to 0% at 32s. Thus, at lower t-bucket expiry times, HANSEL is more likely to drop events, which may result in incorrect causal sequences.

8.3.3 *Effectiveness of purging*

We evaluate the effectiveness of the purging mechanism (per § 6.3) to measure the improvement in HANSEL's precision as a result of purging older (and therefore possibly unrelated) transactions. We selected 50 Tempest tests across all categories proportional to their distribution, and executed them in parallel with varying p-bucket size. Fig. 6d plots the results, and we observe that the p-bucket size does not significantly alter the precision, even when size changes from 100 to 1.5K messages. This is because: (a) transactions do not cross the p-bucket boundaries, or (b) other heuristics were sufficient to achieve the desired precision.

8.4 Performance

8.4.1 *Overhead of communication pipeline*

In this section, we evaluate the overheads of HANSEL's event transmission pipeline till each message is processed. Specifically, we measure the rate of events sent by the Bro agents running at the different OpenStack nodes, and compare the rate of events processed by the analyzer service. Note that this excludes time to generate the execution graph.

We use tcpreplay [11] to send out HTTP events at varying rates from the Bro agents and observe the rate of events received at the analyzer service at various stages in its pipeline. Fig. 6e plots the variation in events received with event rates starting from 500 till 5K, incremented in steps of 500. We observe that events processed by the Pre-processor match with those received at HANSEL's Event Receiver (with no losses reported) and those sent by Python Broccoli. This trend continues even at higher send rates.

Note that Python Broccoli reports lower event rate after the 1.6K mark. We believe that the Python bindings for Broccoli significantly impacts the processing pipeline, and is thus a bottleneck. Leveraging the C++ Broccoli bindings may improve the observed event rate.

8.4.2 *Overhead of Transaction Chainer*

HANSEL's fault diagnosis also depends on the time taken by the Transaction Chainer to create the execution graph. We randomly selected several Tempest test cases and executed them concurrently. Fig. 6f plots the results with varying number of tests, ranging from 50 to 250. We observe that under all scenarios HANSEL's Transaction Chainer takes

<150ms per transaction. We attribute the variance across the results to the random selection of test cases.

8.4.3 System overhead

We measure the overhead of applying HANSEL to our testbed. We ran 30 Tempest tests in parallel, which initiated >1.5K concurrent transactions, and monitored all nodes to measure CPU and memory consumption by Bro agents and the central analyzer service. These tests took ~15mins to complete. We observed that peak CPU usage for Bro-based monitoring agents was <1%, while their memory usage was ~100 MB. The analyzer service reported peak CPU usage of ~4.35% and memory consumption of ~100 MB.

9. LIMITATIONS & FUTURE WORK

(**1**) The granularity of fault detection and subsequent diagnosis is contingent upon the nature of network messages monitored. Error messages that do not match the regular expression checks may be potentially missed. In comparison, log analysis also suffers from this limitation.
(**2**) HANSEL cannot always detect fault manifestations due to system dependencies. For example, if an OVS agent on a compute node crashed, HANSEL may detect from the nature of communication that the Neutron agent on that compute node is not working. However, HANSEL cannot precisely localize the fault due to the OVS crash itself (unless the error messages reveal it). In such cases, HANSEL together with log analysis may be useful to pinpoint the root cause.
(**3**) HANSEL in its present form does not detect faults that manifest across transaction boundaries, e.g., a DELETE request halting an ongoing operation (per § 8.1.1). While HANSEL can find causal linkages between the two transactions, it may not conclusively determine the cause of the problem with the halted transaction. We leave such cross-transaction fault analytics for future work.
(**4**) Unlike later versions of OpenStack that exclusively use 32-character UUIDs as identifiers, some older versions use either 8 or 32-character identifiers, or their hashed variants. Thus, HANSEL would only require minor format specific changes to its regular expressions to be applicable to other OpenStack versions. Further, while the general technique of stitching transactions based on event sequences can be applied to debug other systems, the direct portability of HANSEL in its current form needs further investigation. For example, while cloud platforms such as CloudStack and vSphere also make use of 32-character UUIDs in their REST and RPC calls, a meaningful application of HANSEL to both of them would require a more careful study of the end-to-end communication patterns and the role of UUIDs in resource management. We defer such exploration to future work.
(**5**) Distributed stream processing systems, such as Spark Streaming [45], may significantly improve HANSEL's scalability and performance. Spark Streaming models incoming streams as a series of micro-batch computations on discrete time intervals (analogous to HANSEL's bucketing mechanism), supports iterative algorithms on the workload, and is thus well-suited for our use case. We defer the development and evaluation of HANSEL's analyzer service as a distributed streaming application to future work.
(**6**) The execution sequences constructed by HANSEL could also be leveraged to diagnose other problems in the system, such as the ability to localize performance bottlenecks. Like prior work [37], HANSEL could also be augmented with this capability by correlating latencies in requests/responses with the utilization levels of system resources.

10. RELATED WORK

10.1 Distributed System Model Checkers

Distributed System Model Checkers (DMCK) are utilized primarily for discovering bugs by exercising all possible sequences of events so as to push the target distributed system into critical situations. They are primarily aimed at preemption of faults by verifying *a priori* the reliability of distributed systems. DMCK, however, suffer from the problem of state-space explosion, which is addressed by state-of-the-art black-box approaches [24, 26, 32, 38, 43] through non-systematic, randomized reduction techniques. These techniques rarely exercise multiple system failures, which limits the effectiveness of bug discovery. This renders them inadequate in addressing the typical complex failures encountered in distributed systems, especially cloud systems. While SAMC [31] addresses the above concerns, unlike HANSEL, it requires complete semantic knowledge of the target distributed system. Further, DMCK cannot be used for diagnosing faults in running production deployments.

10.2 Distributed Systems Tracing

Various techniques have been proposed in prior work that diagnose faults by tracing the sequence of events and interactions among the components of a distributed system.
EXPLICIT METADATA PROPAGATION. In this approach, end-to-end tracing is integrated within the distributed system operations itself. These systems [18–22, 28, 35–37, 39, 41] rely on instrumentation at the level of kernel, applications, middleware, libraries or network messages. In other words, they modify the target system to (a) ensure propagation of unique metadata per request across the system, and (b) get a causal chain of events associated with that request. Pip [35] and Webmon [23] rely on application level tagging, while Pinpoint [19] tags client requests as they travel through the system, via middleware and library modifications. Several popular industry implementations [5, 13, 37], also take a similar approach towards end-to-end tracing. X-trace [22] employs a combination of middleware and network message instrumentation to uniquely tag all network operations resulting from a particular task. Ju *et al.* [28] make modifications to OpenStack to taint RPC and REST traffic for end-to-end OpenStack task tracing. vPath [39] makes kernel modifications to trace all synchronous RPC calls associated

with a particular task. In contrast, HANSEL does not require any such instrumentation, and uses available metadata to stitch together a causal sequence of events.

SCHEMA-BASED. Magpie [16] and ETE [27] rely on the event semantics of distributed systems, and use temporal join-schemas on custom log messages. Such approaches are less scalable than those based on propagation of metadata, since they delay the determination of causal relations until all the logs are collected. In contrast, HANSEL is API-agnostic and does not require knowledge of event semantics. Moreover, HANSEL is based on a fast, online algorithm.

BLACK-BOX TRACING. Some tracing systems [17, 30, 40, 42, 44] rely on log analysis to infer causal relationship among events. Such tools, however, are limited by the verbosity of the logs, and employ probabilistic data mining approaches. FChain [33] and Netmedic [29] use statistical inferences from system metrics (like CPU and network usage) and require a comprehensive training phase. Unlike the deterministic approach taken by HANSEL, these tools are probabilistic and are not exhaustive in their fault detection.

11. CONCLUSION

We present HANSEL, a system that uses non-intrusive network monitoring to expedite root cause analysis for faults manifesting in OpenStack operations. HANSEL examines relevant OpenStack messages to mine unique identifiers, and stitches together a stateful trail of control flow amongst the component nodes. We present several optimizations and heuristics to improve HANSEL's precision. HANSEL is fast and accurate, and precise even under conditions of stress.

12. ACKNOWLEDGEMENTS

We thank our shepherd, Theophilus Benson, and the anonymous reviewers for their valuable comments.

13. REFERENCES

[1] OpenStack. https://www.openstack.org/.
[2] Apache CloudStack. https://goo.gl/1S3K9W.
[3] Bro. https://www.bro.org/.
[4] Broccoli. https://goo.gl/4NUdFi.
[5] Cloudera HTrace. https://goo.gl/Pz3lQu.
[6] HTTP. http://www.ietf.org/rfc/rfc2616.txt.
[7] Openstack customers. https://goo.gl/jFLPrU.
[8] Rackspace Issue 1. https://goo.gl/2tdjHB.
[9] Rackspace Issue 2. https://goo.gl/CnqSTl.
[10] Rackspace Issue 3. https://goo.gl/JVVpX0.
[11] Tcpreplay. http://tcpreplay.synfin.net/.
[12] Tempest. http://goo.gl/OZiXTV.
[13] Twitter Zipkin. https://goo.gl/bHtUKc.
[14] VMware vSphere. http://goo.gl/kNAR0f.
[15] P. Bahl et al. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. In *SIGCOMM'07*.
[16] P. Barham et al. Using Magpie for Request Extraction and Workload Modelling. In *OSDI'04*.
[17] L. Bitincka et al. Optimizing Data Analysis with a Semi-structured Time Series Database. In *SLAML'10*.
[18] A. Chanda et al. Whodunit: Transactional Profiling for Multi-tier Applications. In *SOSP'07*.
[19] M. Y. Chen et al. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *DSN'02*.
[20] Y.-Y. M. Chen et al. Path-based Failure and Evolution Management. In *NSDI'04*.
[21] R. Fonseca et al. Experiences with Tracing Causality in Networked Services. In *INM/WREN'10*.
[22] R. Fonseca et al. X-trace: A Pervasive Network Tracing Framework. In *NSDI'07*.
[23] T. Gschwind et al. Webmon: A Performance Profiler for Web Transactions. In *WECWIS'02*.
[24] R. Guerraoui et al. Model Checking a Networked System Without the Network. In *NSDI'11*.
[25] H. S. Gunawi et al. What Bugs Live in the Cloud?: A Study of 3000+ Issues in Cloud Systems. In *SOCC'14*.
[26] H. Guo et al. Practical Software Model Checking via Dynamic Interface Reduction. In *SOSP'11*.
[27] J. L. Hellerstein et al. ETE: A Customizable Approach to Measuring End-to-end Response Times and their Components in Distributed Systems. In *ICDCS'99*.
[28] X. Ju et al. On Fault Resilience of OpenStack. In *SOCC'13*.
[29] S. Kandula et al. Detailed Diagnosis in Enterprise Networks. In *SIGCOMM'09*.
[30] S. P. Kavulya et al. Draco: Statistical Diagnosis of Chronic Problems in Distributed Systems. In *DSN'12*.
[31] T. Leesatapornwongsa et al. SAMC: Semantic-aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *OSDI'14*.
[32] H. Lin et al. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI'09*.
[33] H. Nguyen et al. FChain: Toward Black-box Online Fault Localization for Cloud Systems. In *ICDCS'13*.
[34] V. Paxson. Bro: A System for Detecting Network Intruders in Real-time. In *USENIX Security'98*.
[35] P. Reynolds et al. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI'06*.
[36] R. R. Sambasivan et al. Diagnosing Performance Changes by Comparing Request Flows. In *NSDI'11*.
[37] B. H. Sigelman et al. Dapper: A Large-scale Distributed Systems Tracing Infrastructure. *Google Research*, 2010.
[38] J. Simsa et al. dBug: Systematic Evaluation of Distributed Systems. In *SSV'10*.
[39] B.-C. Tak et al. vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities. In *ATC'09*.
[40] J. Tan et al. Visual, Log-based Causal Tracing for Performance Debugging of MapReduce Systems. In *ICDCS'10*.
[41] E. Thereska et al. Stardust: Tracking Activity in a Distributed Storage System. In *SIGMETRICS'06*.
[42] W. Xu et al. Detecting Large-scale System Problems by Mining Console Logs. In *SOSP'09*.
[43] M. Yabandeh et al. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI'09*.
[44] D. Yuan et al. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. In *ASPLOS'10*.
[45] M. Zaharia et al. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP'13*.