Control plane systems tracing and debugging – towards implementation

Gleb Peregud, Maria Ganzha
Department of Mathematics and Information Sciences
Warsaw University of Technology
Warsaw, Poland
gleb.peregud@gmail.com, M.Ganzha@mini.pw.edu.pl

Marcin Paprzycki
Systems Research Institute
Polish Academy of Sciences
Warsaw, Poland
marcin.paprzycki@ibspan.waw.pl

***Abstract*—Reasoning about, and debugging, hierarchical control plane systems is hard. Moreover, OpenTracing, the industry adopted tracing model, has problems with tracing activities in presence of coalescing effects, which materialize, among others, in cloud platforms and build systems. In our earlier work we have proposed a novel approach to distributed systems tracing, based on an extension of OpenTracing. The aim of this contribution is to outline how the proposed approach can be implemented.**

***Index Terms*—control plane systems, OpenTracing, coalescing effects, distributed systems**

I. Introduction

In earlier work [1], the persistent problem of debugging hierarchical control plane systems, employing intent-driven actuation, was introduced and an outline of a solution presented. As discussed, understanding behaviour of such systems requires observing their actions and associated state changes across *all* layers, comprising of thousands of services, running across hundreds of machines. Moreover, control plane systems employ intent-driven actuation and exhibit coalescing effects, resulting in arbitrary relationships between incoming requests and individual actuations, performed by the system. Hence, such systems are extremely difficult to debug. In this context, a widely adopted approach to debugging is distributed systems tracing, and the most popular tools follow the OpenTracing model [2], based on Google's Dapper. However, it is also known that Dapper does not handle coalescing effects [2].

As a result of comprehensive analysis of state-of-the-art of academic research and of industrial practices, reported in [1], it was proposed that an innovative mix of distributed systems tracking and provenance tracking may facilitate debugging modern automation systems, and hierarchical control plane systems, in particular. Moreover, a set of requirements, leading to a meta-level architecture, named *Provenance-Enhanced Distributed Systems Tracing; PEDST*, has been formulated.

The aim of this contribution is to go to the next step and outline how the proposed PEDST approach can be implemented. Here, it should be stressed that, due to space limitation, we omit discussion of the state-of-the-art. Interested reader is referred to [1]. However, to the best of our knowledge, no noteworthy contribution has been published in the meantime.

II. Requirements

Let us start with definitions of main concepts and follow with the refined set of requirements for the system that will support debugging control plane systems. For extended definitions of terms, see [1], and references collected there.

- *Hierarchical Control Plane Systems* (HCPS) are API-level systems, which accept requests, manage entities states, and translate those into a series of requests sent to "lower-level systems", which are often HCPS themselves.
- *Coalescing effect* is any form of batching of work units.
- *Intent-based actuation* is an execution of a scheduling policy to reach an intended state, which performs appropriate imperative steps individually.

Based on conducted research, we believe that system satisfying summarized below requirements will be able to trace HCPS that employ intent-based actuation.

1) *Coalescing effects support* – required to support tracing of intent-based actuation logic in control plane systems;
2) *Support for abstract entities* – essential to align with cloud APIs, which support entities at multiple levels of abstractions (clusters, deployments, VMs, etc.);
3) *Support for composite entities* – required to deal with objects like archives, VM images, container images, etc., prevalent in cloud APIs;
4) *Low storage overhead* – necessary for large-scale systems, deployed as control plane systems, in the cloud;
5) *Full coverage* – to ensure that all activities will be tracked and resource state mutations will be recorded (modulo presence of unavoidable infrastructure faults);
6) *Gradual fidelity execution tracing* – to allow developers to selectively apply execution tracing, to trade off tracing accuracy and implementation effort;
7) *Gradual fidelity provenance tracking* – to allow developers to selectively apply provenance tracking, to trade off provenance tracking accuracy and implementation effort;
8) *Low mental burden* – to support adoption in industry;
9) *Cross-host tracking* – for distributed systems tracing;
10) *Multi-layer systems support* – to allow tracking across layers in hierarchical systems;
11) *Asynchronous data intake* – to support data ingestion from traced distributed systems, in presence of unreliable

978-1-6654-3281-8/21/$31.00

network, unpredictable latencies, and lack of ordering guarantees, in multi-host network communication;

12) *Event-based data production* – to deal with compute nodes and/or processes faults, and to avoid buffering in tracing mechanisms;
13) *Flexible control flow support* – for tracing of systems evolving over time and to deal with preexisting complex control flows in control plane systems.

Note that existing solutions, like Dapper or CamFlow [3], satisfy only (different) subsets of these requirements, but *none of them* satisfies all of them at once (see, [1]).

III. Provenance-enhanced distributed systems tracing model

In [1], we have proposed a Provenance-Enhanced Distributed Systems Tracing (PEDST) model, which supports both *execution tracing* and *provenance tracking*. Overall, in this model, actions are captured as *executions*, forming a tree, to represent the control flow. *Operations* capture reads and writes of *incarnations*, and each of them is a version of an *entity*. Pairs of *executions* can interact. Recorded *interactions* consists of *messages*. *Incarnations* can be nested with *sub-incarnations*. A *message* may carry an *incarnation* as a payload. Here, use of OpenTracing model as a foundation addresses requirement 8, while support for *sub-incarnations* satisfies requirement 3.

To underpin the model, we use two schemas: (1) to capture things as they happen – *the local schema* – used for online logging; and (2) provenance-enhanced distributed tracing – *the global schema* – for offline processing. In what follows we define each of them.

A. Global schema

The global schema represents the concepts of PEDST, and represents fully-processed data, ready for querying. Let us now formulate all pertinent definitions.

Definition III.1 (Observed universe). Observed universe U is a tuple (E, I, O, X, P, N), where E is a set of all observed executions, I is a set of all observed incarnations, O is a set of all recorded operations performed by observed executions, X is a set of all recorded interactions between participating executions, P is a set of all registered processes, N is a set of all registered entities.

Definition III.2 (Execution). An execution $e \in E$ is a tuple $(id_e, t_{start}, t_{end}, e_p, c, p, a)$, where:

- id_e is its identifier
- t_{start} is a timestamp of the beginning of this execution
- t_{end} is an end of execution
- e_p is a reference to the parent execution
- e_c is a reference to the creator execution
- p is an optional reference to the process this execution instantiates.
- a is a set of annotations associated with this execution.

Definition III.3 (Incarnation). An incarnation $i \in I$ is a tuple $(id_i, t, tombstone, n)$, where:

- id_i is its identifier of the incarnation
- t is a timestamp of the incarnation creation
- i_{parent} is the incarnation's parent identifier
- $tombstone$ is a boolean flag if this incarnation records end of the current lifetime of this entity
- n is an optional reference to the entity of this incarnation.

An incarnation can represent a version of any type of object managed in the system – regardless of its level abstraction – hence contributing to requirement 2.

Definition III.4 (Operation). An operation $o \in O$ is a tuple $(t, id_o, e_{op}, t_{op}, i_{subj})$:

- t is a timestamp of the operation
- e_{op} is an execution represented by given tuple
- t_{op} is an operation type (*Write* or *Read*)
- i_{subj} is a reference to incarnation (subject of operation).

Definition III.5 (Interaction). An interaction $x \in X$ is a tuple $(id_x, e_{init}, e_{target}, M_x)$, where:

- e_{init} is a reference to execution that initiated interaction
- e_{target} is a reference to execution interacted with
- M is a set of messages exchanged between E_{init} and E_{target}, where each message *m* is a tuple $(t, e_{init}, e_{target}, \textit{Payload})$, where:
  - t is a timestamp of the message
  - $Payload$ is a payload of the message.

Definition III.6 (Process). A process $p \in P$ is a tuple $(id_p, \textit{description}, e_d)$, where:

- id_p is identifier of the process
- $description$ is a human-readable process description
- e_d is an entity reference where a given process is defined.

Definition III.7 (Entity). An entity $n \in N$ is a tuple $(id_n, description)$, where:

- id_n is its identifier of the entity
- $description$ is a human-readable description of the entity

Definition III.8 (Annotation). An annotation is a tuple $(t, Payload)$, where:

- t is a timestamp of the message
- $Payload$ is a payload value.

Graphs: Next, we define a set of graphs over information captured in the observed universe. They will be used to extract information from data, e.g. answer queries. They are also used as a guide for the implementation of the model.

Definition III.9 (Execution graph). An execution graph of U is a directed graph where nodes are elements of E and if e_j is a parent of e_i, there is an edge between e_i and e_j.

The execution graph captures the parent-child relationship of executions. It is equivalent to the forest of trace trees in the OpenTracing model. Let us now call $Edges(U)$, a multiset of tuples (e_{init}, e_{target}) for each element in X of U.

Definition III.10 (Interaction graph). An interaction graph, in U, is a multigraph, where nodes are elements of E and edges belong to $Edges(U)$ and connect elements of E.

Definition III.11 (Effects graph). An effects graph, in U, is a bi-partite directed graph, where executions in E form partition in U, and incarnations in I form partition in V, and each element of O gives rise of a directed edge either from U partition to V partition in case of *Write*, or from V partition to U partition in case of *Read*.

Let us now call $Readers(i_i)$ a set of executions, which read a given incarnation i_i: $Writers(i_i) = \{e_{op} : \forall\ o_i = (e_{op}, t_{op}, i_{subj}) : t_{op} = \text{READ} \wedge i_i = i_{subj}\}$. Let us also call $Writers(i_i)$ a set of executions which wrote a given incarnation i_i: $Writers(i_i) = \{e_{op} : \forall\ o_i = (e_{op}, t_{op}, i_{subj}) : t_{op} = \text{WRITE} \wedge i_i = i_{subj}\}$. If the size of set $Writers(i_i)$ is larger than 1, the observed dataset contains errors.

Definition III.12 (Direct provenance graph). A direct provenance graph P^d of U is a directed hypergraph, corresponding to the effects graph (E, I, O) in U, where each incarnation $i \in I$ is a node in P^d, and all executions $e_i \in E$, incoming-adjacent to i, form a directed hyperedge from corresponding outgoing-adjacent incarnations to i.

Definition III.13 (Indirect provenance graph). An indirect provenance graph P^i is a direct provenance graph P^d, with an additional edge between every two nodes representing incarnations i_1 and $i_2 \in I$ when there is a path between $Reader(i_1)$ and $Writer(i_2)$ in the execution graph in U.

Definition III.14 (Direct provenance set). A direct provenance set $S^d(i)$ for an incarnation $i \in I$ of U, is a transitive closure of incoming adjacent neighbours of $i \in P^d$ of U.

Definition III.15 (Indirect provenance set). An indirect provenance set $S^i(i)$, for incarnation $i \in I$ of U, is a transitive closure of incoming adjacent neighbours of i in P^i of U.

Extensions: allow inferring additional information from the observed dataset, and recovering additional information through over-approximation. These extensions address requirements 6 and 7, by allowing to extract value from minimal tracing instrumentation, by trading off accuracy of execution tracing and provenance tracking.

Definition III.16 (Sub-incarnation provenance extension). For a given universe U, sub-incarnation provenance extension is a universe U' extended with imaginary executions: for each sub-incarnation an execution is implied, which performs a read operation on parent incarnation of a sub-incarnation, and a write operation of the sub-incarnation itself.

Sub-incarnation provenance extension is an optimization mechanism, which allows users to avoid explicitly recording write operations on sub-incarnations while allowing to capture provenance relationship between a writer of a parent incarnation, and a reader of a sub-incarnation.

Definition III.17 (Sub-execution provenance extension). For a given universe U, sub-execution provenance extension is a universe U', extended with imaginary read operations for every pair of execution and incarnation. If there exists a read operation performed by any of execution ancestors on the incarnation, a new read operation is implied between the execution and the incarnation.

This inference mechanism captures a practically probable provenance relationship between an object read by a parent execution, and an object written by a child execution.

Definition III.18 (Message-passing provenance extension). For a given universe U, message-passing provenance extension is a universe U', extended with imaginary executions: for each message sent from e_1 to e_2, in each interaction a unique incarnation i, and two operations op_w and op_r are implied, where e_1 performs op_w as a write on i and e_2 performs op_r as a read on i.

This extension assumes that when (two) processes communicate, they exchange information relevant for provenance tracking, hence implicitly linking their executions in the provenance graph.

B. Local schema

The local schema enables incremental logging of events, and is sufficient to recover provenance-enhanced traces. Events are generated by traced components, and can be of the following types: (1) execution begin, (2) execution end, (3) operation performed, (4) message sent, (5) message received, and (6) execution annotated. Processes and entities are recovered from events implicitly by the system. Here, events logged by all components form stream F. We assume that a sequence of events F is causally ordered. Hence, to construct an *observed universe* we need to perform a *left-fold* on the stream, by starting using `observe` function with an initial empty universe[1]:

$$U^0 = (E^0, I^0, O^0, X^0, P^0, N^0)$$

$$U = \text{leftFold}(\text{observe}, U^0, F)$$

where

$$E^0 = \emptyset, I^0 = \emptyset, O^0 = \emptyset, X^0 = \emptyset, P^0 = \emptyset, N^0 = \emptyset$$

The *observe* function needs to be embellished with a state, allowing it to maintain the intermediate state information, which is not representable in the global data model. The *observe* is defined for each individual event type. For example, *Execution begins* event f is represented as a tuple $(id_e, e_{parent}, e_{creator}, id_p)$ is processed in the following steps (observing other event types is achieved analogously):

- Checks if parent execution has been observed so far.
- Checks if creator execution has been observed so far.
- Notes existence of process id_p.
- Stores this incomplete execution information, in the intermediate state, with the event timestamp as the execution start timestamp.

[1]In industry, this pattern is known as Event Sourcing [4].

C. Logging protocol

Use of the proposed model puts two requirements on event producers. All identifiers need to be constructed consistently, across all components of the traced system. It is the responsibility of an application developer to maintain causal ordering of events. A vast research body has been produced on this topic [5], and it is out of the scope of this work.

IV. Architecture

Let us now outline main aspects of the architecture that, together with the model, satisfies outlined requirements. We have chosen a white-box tracing – allowing to satisfy requirements 6 and 7 - and disclosed provenance approaches for our initial implementation. Overall, the PEDST architecture consists of:

- *Logging protocol* – for event production by traced systems and ingestion by the PEDST solution;
- *Logging data schema* – defining events schema produced by observed components (see, Section III-B);
- *Ingestion pipeline* – which ingests events from event producers and stores them in the storage system;
- *Storage system* to store the events and traces;
- *Events processor* – it builds up traces in *global schema* using incoming events, and stores it in the *storage system*;
- *Global data schema* – to capture the global view of activities recorded from all systems (see, Section III-A);
- *Set of queries*, defined over global data;
- *Visualization tool*, to represent data to an engineer.

Data flow between these elements is represented in Figure 1. Let us now describe in more detail the key components of the architecture.

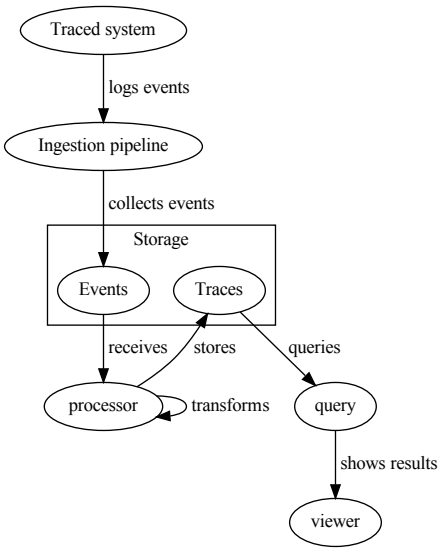

Figure 1: Data flow in the PEDST architecture.

Ingestion pipeline: Events logged by a *traced system* are received by an *ingestion pipeline* and stored in the *storage system*. Depending on the scale of the *traced system*, the *ingestion pipeline* might be as simple as a shared NFS directory, a large Splunk deployment or, for large systems, pipeline similar to Google's Dapper. An asynchronous *ingestion pipeline* satisfies the requirement 11, and contributes to the requirement 12. *Ingestion pipeline* gathers events from all components of the traced systems. Hence, it contributes to requirement 5. A distributed *ingestion pipeline* satisfies requirement 9, by ingesting events from all hosts.

Storage system: Stores both events received from the *ingestion pipeline*, and traces representing the *observed universe* (defined in Section III-A). Additionally, for more efficient querying, it may contain post-processed data, derived from the *observed universe*.

Events processor: Receives events from the *storage system*, processes and deletes them. Processing follows procedure defined in Section III-B. Whenever event processing results in a new (or updated) record, it is persisted in the *storage system*. *Events processor* is optimizing provenance information, to a representation suitable for efficient graph queries.

Queries: Queries executed against data in the *storage system* allow constructing graph representations of the *observed universe* and its *extensions* (see Section III-A).

V. Implementation

Let us now consider how the core aspects of the proposed model and the architecture have been implemented as a Tenmo framework [2]. As far as the main technology stack is concerned, the implementation consists of the following elements:

- A PostgreSQL database, used for the *storage system*.
- Go and C++ logging library, implementing structured *logging protocol* and *ingestion pipeline*.
- Python based *ingestion pipeline*, for logs-based ingestion.
- Python processor, which listens to event notifications from the *storage system*, constructs traces from events, and stores them in the database.
- GraphViz for visualization of query results.

Overall, for each component, the developed solution follows the standard industry implementation practices, and is similar to the LogProv implementation [6]. Let us now describe in some detail implementation of selected components.

A. Data definitions

Data models exist in three equivalent implementations (in Python, Go and PostgreSQL schema). They follow the models defined in Section III-A, with few additional fields needed in each language. They are translated to a chosen language, following standard engineering practices (see, Listing 1).

```
type eventExecutionBeginsJson struct {
  EventUlid ulid.ULID `json:"event_ulid"`;
  Timestamp time.Time `json:"timestamp"`;
  ExecutionId string `json:"execution_id"`;
  ParentId string `json:"parent_id,omitempty"`;
  CreatorId string `json:"creator_id,omitempty"`;
  ProcessId string `json:"process_id,omitempty"`;
  Description string `json:"description,omitempty"`;
}
```

Listing 1: Execution begin event defined in Go. All fields are JSON-serializable, so they can be stored as a `payload jsonb` field, in the *Storage system*.

[2]The code is available at https://github.com/gleber/tenmo

B. Ingestion pipeline

Tenmo implements two ingestion mechanisms. (1) Tenmo Go library connects directly to the Tenmo PostgreSQL instance and appends data to the `events` table. (2) Nix integration (see Section VI-A) uses a log tailing Python script, which iterates over JSON-structured log lines, produced by Nix binaries, and transforms them to Tenmo event schema and inserts to the *storage system*.

Note that, in a production-grade implementation, Tenmo would be powered by a high-performance log ingestion solution like Syslog-ng, Apache Kafka, FluentD, or Loginson [7].

C. Logging library

Tenmo provides a Go client-side library (usage example presented in Listing 2), which allows to register all events in PEDST. For example, to track provenance, the traced component has to record its read and write operations on objects used in the system.

```
parentExecutionId := // Retrieve parent identifier, often from a Context
// Register an execution start.
executionId, ender :=
    tenmo.ExecutionRegistration(tenmo.Execution{
        tenmo.ExecIdRand("worker-step"),
        parentExecutionId,
        "worker step"})
// Make sure that the execution is ended  .
defer ender()
// Perform own work.
PerformOwnWork();
// Pass own execution ID to a child goroutine.
go PerformChildWork(executionId);
```

Listing 2: Example of a typical hierarchical-aware execution tracing using Tenmo Go client library.

D. Storage system

The Tenmo implements a basic non-distributed storage, based on a single PostgreSQL database[3]. The database includes: (1) an events table, and (2) a set of tables capturing the *observed universe*. It directly implements local and global schemata of PEDST model, using standard engineering practices (for example, see Listing 3).

```
create table operations (
  operation_id text primary key,
  stored_at timestamptz not null default now(),
  ts timestamptz not null,
  execution_id text references executions (execution_id),
  op_type char(1), -- operation type 'w' or 'r'
  entity_id text not null references entities (entity_id),
  incarnation_id text not null references incarnations (incarnation_id)
);
```

Listing 3: Example of a storage table definition in Tenmo, specifically the `operations` table.

The events table is populated by the *ingestion pipeline* (appended as is). The set of tables representing the *observed universe* is populated by the *events processor*.

[3]This implementation ignores question of security. A production-grade implementation would require an elaborate permissioning mechanism to protect data stored in Tenmo database.

E. Events processor

The *events processor* consumes events from `events` table and populates other tables in the *storage system*. The processor subscribes to a notification channel, and gets notified about every new event in `events` table, via PostgreSQL's `NOTIFY` event, sent by an appropriate database trigger. The processing logic is as follows (represented as a pseudo-code):

```
for event in fetch_unprocessed_events(order_by=ingestion_timestamp):
    with transaction:
        mark_claimed(event) or continue
    with transaction:
        U = load_universe()
        U', success = observe(event, U)
        if success == true:
            mark_processed(event)
            store_universe(U')
        else:
            mark_unclaimed(event)
```

To avoid event double-processing, *events processor* uses atomic database transactions to mark an event as claimed, followed by the processing logic. If successful, an event is marked as processed. If an event exceeds a limit of processing attempts, it is no longer considered during next processing cycles. Processed events are regularly garbage collected.

F. Tenmo graph

Tenmo graph is stored in `graph` table, in the database, and has fields `source`, `verb` and `target`. All PEDST relationships – including PEDST extensions – are stored denormalized into the triples, forming a composite graph. Table I lists the relationships reflected in the table.

Source	Verb	Target
Operation affecting an incarnation		
incarnation_id	read_by	execution_id
execution_id	reads	incarnation_id
execution_id	writes	incarnation_id
incarnation_id	written_by	execution_id
Child execution and its parent		
execution_id	child_of	parent_id
parent_id	parent_of	execution_id
Executions		
execution_id	created_by	creator_id
creator_id	creator_of	execution_id
Incarnation and its entity		
incarnation_id	instance_of	entity_id
entity_id	entity_of	incarnation_id
Interaction and its messages		
sender	sent_to	target
target	received_from	sender
Sub-incarnations and its parent		
incarnation_id	part_of	parent_id
parent_id	divides_into	incarnation_id
Adjacent pair of incarnations in an entity		
incarnation_id	after	incarnation_id
incarnation_id	before	incarnation_id

Table I: Tenmo graph relationships.

G. Queries

Given the triple form of storage, answering queries, with or without graph extensions, is a matter of traversing the composite graph and analysing found paths. Tenmo implementation uses PostgreSQL's recursive Common Table Expressions (CTE), as

a mechanism to recursively traverse the graph for all queries (see Listing 4 for an example for query implementation).

```
CREATE OR REPLACE FUNCTION get_all_paths_from(start text)
RETURNS TABLE(depth integer, verbs text[], path text[]) AS $$
BEGIN
RETURN QUERY
WITH RECURSIVE search_step(id, link, verb, depth, route, verbs, cycle) AS
↪  (
  SELECT r.source, r.target, r.verb, 1,
         ARRAY[r.source],
         ARRAY[r.verb]::text[],
         false
  FROM graph r where r.source=start
  UNION ALL
  SELECT r.source, r.target, r.verb, sp.depth+1,
         sp.route || r.source,
         sp.verbs || r.verb,
         r.source = ANY(route)
  FROM graph r, search_step sp
  WHERE r.source = sp.link AND NOT cycle
)
SELECT sp.depth, array_append(sp.verbs, '<end>') AS verbs, sp.route ||
↪  sp.link AS path
FROM search_step AS sp
WHERE NOT cycle
ORDER BY depth ASC;
```

Listing 4: The `get_all_paths_from` function is the foundation for queries implemented in Tenmo.

VI. Experimental results

To validate the proposed approach, we have applied Tenmo to a software stack representative of typical HCPS. Specifically, it (1) deploys of a service in a cloud environment, (2) contains imperative-based processing of user requests, (3) contains at least 2 distinct intent-based systems, in the control flow of a typical request, and (3) is linguistically heterogeneous. The complete stack is as follows:

- Nix [8] is a build system and a package manager, focused on reliability, correctness and reproducibility.
- KubeNix [9] is a Kubernetes resource builder, that uses NixOS module system [10, sec. 5] for resource definition, and Nix as its build system.
- Kubectl is a command line tool to interact with a Kuberentes cluster.
- Kuberentes automatizes deployment, scaling, and management of containerized applications.

Figure 4 shows a Tenmo trace of a typical operation of the stack. The top part, outlined with an indigo color rectangle, is an extract from the KubeNix build process for a deployment. The right part, outlined in light green color, is an extract of a record of Kubectl execution. The bottom part is an extract of a trace of the Kubernetes deployments controller performing an intent-based actuation of a resource. Two red circles mark locations where Tenmo traced executions, or data, flow across layers in the system, hence satisfying requirement 10.

A. Build systems – Nix

We have integrated Tenmo into Nix, to show that Tenmo is capable of dealing with build systems (an example of intent-based actuation systems), hence satisfying requirement 1). Nix build logic has been extended with comprehensive execution and object-related logging, using existing JSON logs. Tenmo

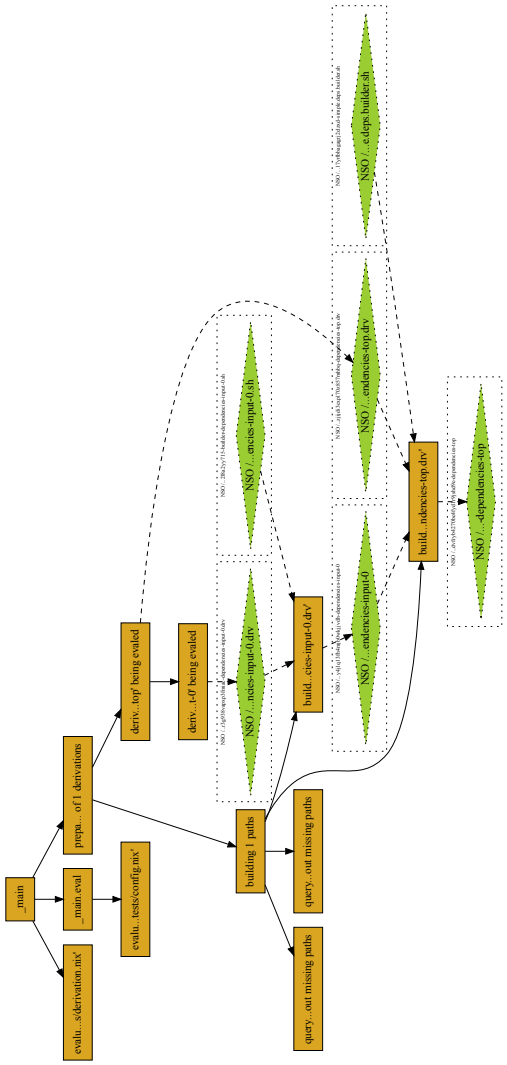

Figure 2: Tenmo trace a Nix build of two derivations `top` and `input0`.

integration for Nix captures:

1) Top-level evaluation process as a whole.
2) Individual `.nix` file evaluations as children of Item 1.
3) Instantiation of derivations required for the build, as children of an artificial execution node, representing instantiation process as a whole.
4) The instantiation from Item 3 execution nodes record writes of `.drv` files as incarnations.
5) Builds of individual derivations based on `.drv` files.
6) Executions capturing builds from Item 5 record their reads of `.drv` incarnations from Item 4, reads of existing and writes of new Nix store paths.

This allows to trace both the execution tree of Nix build process[4] and track provenance of Nix store paths.

Figure 2 shows a simplified Tenmo trace of a Nix build execution using an empty Nix store without any external

[4]The visualized graph clearly shows that Nix is a suspending-type of build system [11].

dependencies, where `top` derivation depends on `input0` derivation and other input files (see Listing 5).

```
tenmo=# select * from provenance_set('i:///.../3hknj...-dependencies-top');
 depth |                                    obj
-------+-----------------------------------------
     2 | i:///.../sh0p2...-dependencies-top.drv
     2 | i:///.../q6ngy...-simple.deps.builder.sh
     2 | i:///.../wfchb...-dependencies-input-0
(3 rows)

tenmo=# select * from provenance_set_indirect(
tenmo(#     'i:///.../3hknj...-dependencies-top');
 depth |                                              obj
-------+-----------------------------------------------------
     2 | i:///.../sh0p2...-dependencies-top.drv
     2 | i:///.../q6ngy...-simple.deps.builder.sh
     2 | i:///.../wfchb...-dependencies-input-0
     4 | i:///.../40zj9...-dependencies-input-0.drv
     4 | i:///.../a2k78...-builder-dependencies-input-0.sh
(5 rows)
```

Listing 5: The direct and indirect provenance sets of a Nix build product `dependencies-top` captured and computed by Tenmo.

B. Cluster deployment

KubeNix: KubeNix produces a `.json` file, an input for `kubectl`. Given that KubeNix is built using Nix, its Tenmo integration is reused. Blue rectangle in Fig. 4 represents a build of a `deployment` with 10 Nginx Kuberentes pod replicas. It contains 958 operations, 877 incarnations, 877 entities, and 1729 executions (see, example in Listing 6).

```
tenmo=> select incarnation_id from incarnations where incarnation_id like
↪  '%kubenix-generated.json' or incarnation_id like '%nginx-conf.json%';
 incarnation_id
------------------------------------------------
 i:///nix/store/ib4k4...-kubenix-generated.json
 i:///nix/store/ryvdj...-nginx-conf.json.drv
(2 row)

tenmo=> select unnest(path), unnest(verbs) from (
select * from shortest_path('i:///.../ryvdj...-nginx-conf.json.drv',
↪  'i:///.../ib4k4...-kubenix-generated.json', 5) limit 1);
                                    unnest |     unnest
-----------------------------------------------+------------
 i:///.../ryvdj...-nginx-conf.json.drv         | written_by
 891549311310 69                               | child_of
 891549311304 15                               | writes
 i:///.../mkfqz...-kubenix-generated.json.drv  | read_by
 609670607667 24                               | writes
 i:///.../ib4k4...-kubenix-generated.json      | <dest>
(6 rows)
```

Listing 6: A path between `kubenix-generated.json` and Nginx configuration as incarnations extracted from a Tenmo trace.

Kubectl: Kubectl accepts a definition of the resource, compares it to the current state of the resource on the Kuberentes cluster and, if different, submits it to the server. We have extended Kubectl to log executions and incarnations.

A sample execution produced the graph captured in Fig. 3. The graph shows that the tool reads the input file, transforms resources into an in-memory representation, compares it with the version of the resources fetched from a Kubernetes cluster, and submits their new versions to the cluster. Listing 7 contains the captured provenance set.

C. Cluster orchestration - Kuberentes

We have also extended Kuberentes to include minimal Tenmo tracing in `deployment` resource controller. The bottom part of Figure 4 shows an extract of its Tenmo trace.

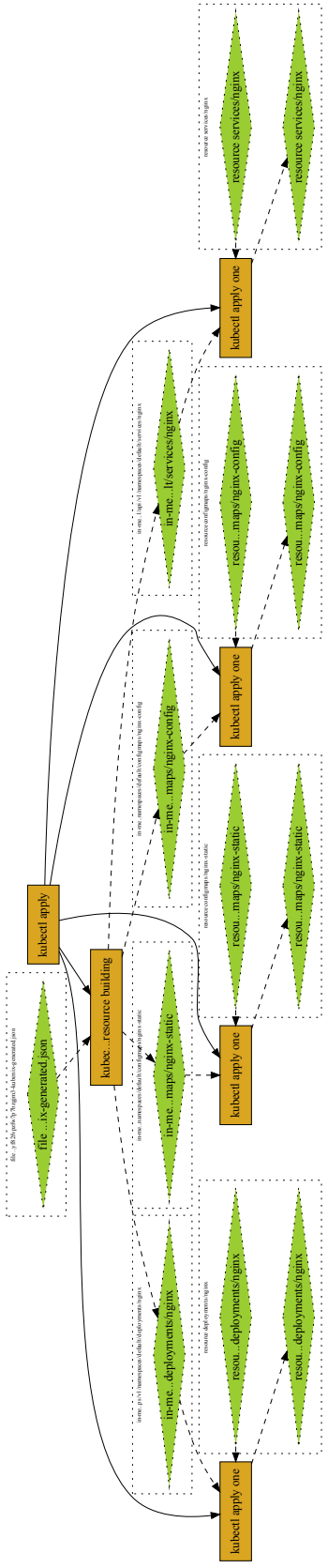

Figure 3: Tenmo trace of a Kubectl execution on a KubeNix-generated resource file.

```
tenmo=> select obj from provenance_set_indirect(
       'i://https://host/api/v1/namespaces/default/services/nginx?ulid=01EJP9R7H');
                                                                                obj
--------------------------------------------------------------------------------
i://in-memory-https://host/api/v1/namespaces/default/services/nginx?ulid=01EJP98PA
i://https://host/api/v1/namespaces/default/services/nginx?ulid=01EJP90V0
i://file:///nix/store/va46w...-kubenix-generated.json?ulid=01EJP9V6C
(3 rows)
```

Listing 7: Tenmo can produce a indirect provenance set for the final deployment resource.

Interested readers can find more details about PEDST model, it's implementation as Tenmo, and it's usage, in [12].

VII. Concluding remarks

The aim of this contribution was to discuss how the approach, proposed in [1], can be implemented. In this context we have introduced the *Provenance-enhanced Distributed Systems Tracing* model, presented the architecture, and discussed key aspects of its implementation, in the form of the Tenmo framework. We have also shown experimentally that Tenmo solves key problems of debugging *Hierarchical Control Plane Systems*. In the future we plan to implement the missing features and run a comprehensive set of tests in real-life, large-scale ecosystems, with the main goal to test scalability of the proposed approach.

References

[1] G. Peregud, M. Ganzha, and M. Paprzycki, "Control plane systems tracing and debugging –existing problems and proposed solution," *MARC*, 2020.

[2] 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," Google, Inc., Tech. Rep., 2010. [Online]. Available: https://research.google.com/archive/papers/dapper-2010-1.pdf

[3] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Symposium on Cloud Computing (SoCC'17)*. ACM, 2017.

[4] "Event Sourcing," 8 2020, [Online; accessed 11. Aug. 2020]. [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html

[5] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[6] R. Wang, D. Sun, G. Li, M. Atif, and S. Nepal, "Logprov: Logging events as provenance of big data analytics pipelines with trustworthiness," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 1402–1411.

[7] C. Vega, P. Roquero, R. Leira, I. Gonzalez, and J. Aracil, "Loginson: a transform and load system for very large-scale log analysis in large it infrastructures," *The Journal of Supercomputing*, vol. 73, no. 9, pp. 3879–3900, 2017.

[8] E. Dolstra, *The purely functional software deployment model*. Utrecht Uni., 2006.

[9] xtruder, "kubenix," Sep 2020, [Online; accessed 10. Sep. 2020]. [Online]. Available: https://github.com/xtruder/kubenix

[10] E. Dolstra, A. LÖh, and N. Pierron, "Nixos: A purely functional linux distribution," *Journal of Functional Programming*, vol. 20, no. 5-6, pp. 577–615, 2010.

[11] A. Mokhov, N. Mitchell, and S. Peyton Jones, "Build systems à la carte: Theory and practice," *Journal of Functional Programming*, vol. 30, 1 2020. [Online]. Available: https://oadoi.org/10.1017/s0956796820000088

[12] G. Peregud, "Novel method for provenance-enhanced tracing in cloud systems," Master's thesis, Zakład Sztucznej Inteligencji i Metod Obliczeniowych, 2020.

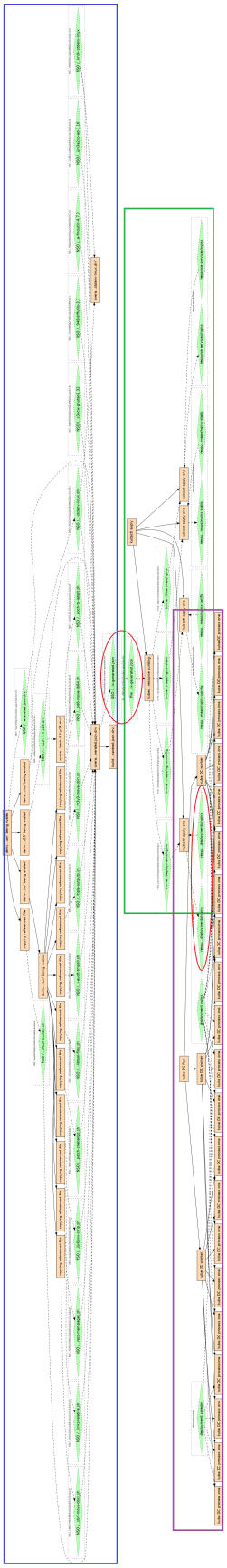

Figure 4: Extract from Tenmo record of a HCPS composed out of Nix / KubeNix, kubectl, and Kubernetes deployment controller.