Warsaw University of Technology

FACULTY OF
MATHEMATICS AND INFORMATION SCIENCE

Master's diploma thesis

in the field of study Computer Science and Information Systems
and specialisation Artificial Intelligence Methods

Novel method for provenance-enhanced tracing in cloud systems

Gleb Peregud

student record book number 198954

thesis supervisor
dr hab. Maria Ganzha, prof. PW

WARSAW 2020

....

supervisor’s signature

....

author’s signature

Abstract

Novel method for provenance-enhanced tracing in cloud systems

Ability to reason about software systems is paramount. Hierarchical control plane systems (HCPS) are hard to debug and reason about because of their particularities, such as prevalence of intent-based actuation. Industry adopted distributed systems tracing model called OpenTracing, which has been tailored for usage in online serving systems. OpenTracing does not handle activity tracing in presence of coalescing effects of request trees. These effects are prevalent in control plane systems – including cloud platforms – and in build systems. Making things worse, control plane systems often employ elements of build systems. Similarly, existing debuggability mechanisms also do not satisfy particularities of HCPS employing intent-based actuation. The goal of this thesis is to provide a solution for reasoning about such systems by creating a novel distributed systems tracing mechanism, based on an extension of OpenTracing model. Related research area of provenance tracking provides a hint for the solution. A good debugging system for HCPS needs to satisfy a specific set of requirements. A novel provenance-enhanced distributed systems tracing model has been proposed, along with a sketch of its graph-theoretical formalization. The model is capable of tracing systems in presence of coalescing effects. A supporting architecture has been defined. The model implemented as a framework called Tenmo following the proposed architecture. It is a activity tracing and object provenance tracking framework based on white-box instrumentation and distributed asynchronous ingestion pipeline. Applicability of Tenmo is proven on a set of software systems representative of HCPS employed by cloud providers. The provenance-enhanced distributed systems tracing solves the debuggability problem for HCPS and occupies an previously empty point in the design space of debugging solutions.

Keywords: Distributed systems tracing. Provenance tracking in hierarchichal control plane systems. Distributed systems, Cloud. PaaS/IaaS. Declarative infrastructure management. Build systems.

Streszczenie

Nowatorska metoda śledzenia w systemach chmurowych z elementami prowieniencji

Zagadnienie rozumowania o systemach komputerowych jest niezmiennie ważne. Rozumowanie o hierarchicznych platformach sterowania z elementami zamiarowego wykonywania jest ponadprzeciętnie trudne. Znaczącym źródłem komplikacji jest użycie modelu zamiarowego wykonywania (ang. intent-based actuation). Popularny mechanizm śledzenia (ang. tracing) zapytań w systemach rozproszonych OpenTracing został stworzony, przede wszystkim, do śledzenia zapytań w systemach serwujących (ang. online serving systems). Jednakże OpenTracing nie obsługuje śledzenia aktywności systemu, w momencie gdy zachodzi efekt łączenia aktywności (ang. coalescing effect) z różnych drzew zapytań (ang. request traces). Takie sytuacje występują często w platformach sterowania systemami komputerowymi, m.in chmurą, oraz w systemach kompilacji (ang. build systems). Platformy sterowania chmurą same często posiadają w sobie elementy systemów kompilacji. Żadne obecnie rozwiązania nie obsługują specyfiki hierarchicznych platform sterowania z elementami zamiarówego wykonywania. Celem pracy jest rozwiązanie tego problemu poprzez stworzenie nowatorskiej metody śledzenia zadań w systemach rozproszonych opartej o nowy model, będący rozszerzeniem modelu OpenTracing. Istniejące badania w tematach pokrewnych m.in. śledzenie prowieniencja (ang. provenance tracking) dostarczają elementów które możemy wykorzystać do rozwiązania problemu. System debugowania dla hierarchicznych platform sterowania musi spełniać specyficzne wymagania. Na ich podstawie został opracowany nowy model śledzenia zdarzeń z elementami śledzenia prowieniencja, wraz ze szkicem jego formalizacji na podstawie teorii grafów. Model ten radzi sobie z problemem łączenia wydarzeń z różnych drzew zapytań. Wraz z modelem powstała architektura spełniająca powyższe wymagania. Na podstawie modelu i architektury zostało stworzone narzędzie Tenmo, oparte o instrumentacje oraz rozproszony i asynchroniczny mechanizm zbierania i agregowania danych śledzenia z systemów rozproszonych. Stworzone narzędzie zostało zweryfikowane poprzez jego zastosowanie do reprezentatywnego zbioru oprogramowania odzwierciedlającego główne elementy składowe hierarhicznych platform sterowania chmurą. Opisana nowatorska metoda rozwiązuje problem śledzenia hierarhicznych platform sterowania z elementami zamiarowego oraz zajmuje wcześniej puste miejsce w przestrzeni rozwiązań debugowania systemów komputerowych.

Słowa kluczone: Metody śledzenia w rozproszonych systemach. Prowieniencja obiektów w platformach sterowania systemów komputerowych. Systemy rozproszone. Chmura. PaaS. IaaS. Deklaratywne zarządzanie infrastrukturą. Systemy kompilacji.

Warsaw, ...

Declaration

I hereby declare that the thesis entitled „Novel method for provenance-enhanced tracing in cloud systems", submitted for the Master degree, supervised by dr hab. Maria Ganzha, prof. PW, is entirely my original work apart from the recognized references.

....

Contents

1. Introduction

Complexity of computer systems has been growing [1], along with ever-increasing usage of computers, in all areas of life. First, computer programs were written in low-level languages, targeting single CPU computers. Moreover, they were often written and used by the same person. Increase in size of developed programs, combined with growing power of computing resources, resulted in individuals losing the ability to fully comprehend computer systems they have been creating/using. Today, some programs are developed by tens of thousands developers [2], [3], both directly and indirectly, due to the reuse of code libraries, services interactions, and use of cloud computing. In large technology corporations, there are services comprising millions of lines of code. They inevitably employ hundreds of different Remote Procedure Call (RPC) interfaces across and between their components [4]. In what follows, systems, which directly serve user requests – e.g. Google Search services – will be called *on-line serving systems*.

Along with the growth of complexity of the systems, software automation has been increasing in complexity. Nowadays, it ranges from simple build systems, like Make [5], through imperative deployment automation systems, like Ansible [6] and Puppet [7], and ending with complex multi-cloud deployment systems, like Terraform [8]. These automation systems, when used in the context of large enough services and software development processes (for example building, testing, and deploying to production in a cloud), are also becoming too complex to understand by an individual developer. For instance, thorough understanding of a sufficiently complex service deployment would require expertise in Terraform, Docker [9], Linux kernel, Kubernetes [10], one or more cloud provider APIs, deployment requirements of the deployed servers and, potentially, in multiple additional areas. Moreover, these automation systems, typically, differ from usual serving systems, which makes it harder to reason about them.

Among the most complex automation systems, out there, are the implementations of cloud services [11]–[13]. These systems serve public APIs to cloud customers and drive all of the infrastructure underpinning the cloud. These advanced automation systems, share a lot of properties of deployment systems – not to mention that some of cloud public APIs actually **are** deployment systems' APIs. The main difficulties with reasoning about systems, which implement cloud services, are discussed in the following sections. Let us start from the intent-driven actuation.

1.1. Intent-driven actuation

Deployment systems – as today's automation systems in general – are shifting from a mostly-imperative execution model (like Ansible), to a declarative-first model (like Terraform). The latter are intent-driven in nature. They follow a scheduling policy to reach an intended state, via execution of a sequence of, usually small and restricted, imperative steps. Such systems will be called *intent-based* and their execution – an *intent-driven actuation*.

For example, increasingly popular [14] Kuberentes – an open-source platform for managing containerized workloads and services that facilitates both declarative configuration and automation – uses a form of intent-based actuation. Kuberentes API objects [15] are used to describe the desired state of the cluster – the user intent. Based on this intent, Kubernetes performs the necessary work (it actuates) to make the current state of the cluster to match the desired state.

Furthermore, select deployment systems – like NixOps [16] – and most modern build systems [17] – like Bazel [18] – are examples of automation systems employing intent-driven actuation most profoundly. In essence, they build a new instance of a target object from the scratch, based on a declared specification, and only the final operation of replacing the current version of the object with the newly created one is an imperative operation.

Decoupling of intent-setting and actuation through a scheduling policy makes it harder to track causality between intent-setting requests and scheduled work. This is due to presence of coalescing effects in these systems.

1.2. Coalescing effects

Systems which employ intent-driven actuation inevitably will exhibit the *coalescing effects* (see fig. 1.1) in their control flow. In general, coalescing occurs when work units, related to multiple incoming requests, are batched over time, before being executed "all-together". Here, batching of disk writes, to achieve higher throughput, is a typical example. In build systems coalescing effect occur in a case of "diamond-shaped dependency graph", since the shared dependency will be built in response to both incoming dependency edges. In these cases, no singular attribution of causality is possible, from a write request to the write of the batch. Each write request contributes, to some extent, to the actual disk write.

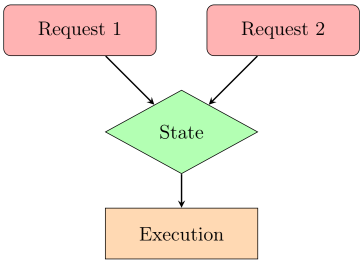


Figure 1.1: Coalescing effect

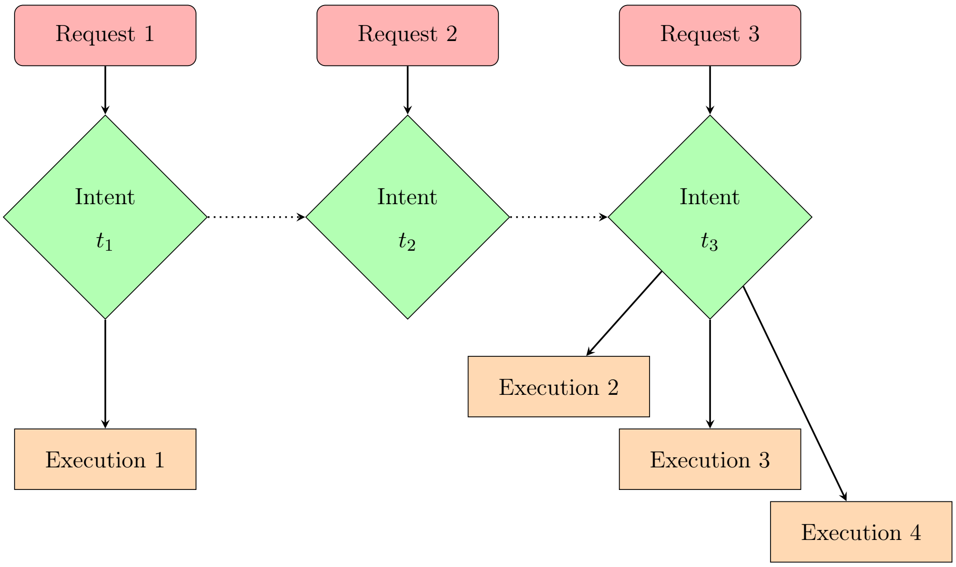


Figure 1.2: Decoupling of requests and execution through state.

In general, presence of coalescing effects changes relationship, between incoming requests and activity performed by the system, from one-to-many to many-to-many, hence making it harder to associate causality between the requests and the activity. Lack of clear association between requests and activities prevents capturing and visualizing causality in the system.

In the case of intent-driven actuation, coalescing effects materialize since actuation is a result of aggregation of intent changes over time, based on multiple user requests. When a certain number of user requests is accumulated, the actuation is executed, according to a scheduling policy. The problem is that the policy can drive actuation over an arbitrary path between current state and the desired state, of the automated system. This decouples handling of individual user requests, and execution of the intent conveyed through these requests. The automation system can choose – based on the scheduling policy, and properties of the system – to batch all user requests and satisfy them all at once, to execute multiple actions based on a single user request, or do “something in-between”. Hence, the relationship between user requests and actuation actions no longer is 1 to M, but can become N to M, instead.

Unsurprisingly, the coalescing behaviours in modern automation systems are complex. Let us illustrate this with a few examples:

- If two consecutive user requests set exactly the same desired state of a service deployment, only one actuation action will be (and should be) performed. The second request causes no change in an intent, hence no action is scheduled.

- If two consecutive user requests modify the same property of a desired state in a quick succession, only the latter actuation is likely to occur, due to the, so called, debouncing, typically enabled in a scheduling policy.

- If three consecutive user requests modify the desired state from A to B, next to C, and next back to B, typically only the actuation from A to B will be executed. Scheduling policy will typically not trigger any competing actions while A to B transformation is being actuated.

- If a single user request changes a desired state, of a complex object, to large-enough extent, the automation system is likely to perform a series of actions over a long period of time, to reach the desired state. For example, if one changes replication factor, for a group of cloud virtual machines, from 1 to 10000, the cloud would not be able to start 9999 VMs instantaneously. Allowing for an instantaneous increase in size would cause a large surge of utilization of internal systems. Additionally allowing all customers to do this would require keeping a very large unutilized reserved capacity, which is not economical. Hence, the process of bringing online additional machines is realized is such a way to not to destabilize the cloud ecosystem.

Observe also that a combination of such behaviours can lead to *arbitrary relationships between user requests and actuation actions* (see fig. 1.2). As mentioned above, these effects make it harder to reason about systems exhibiting them, making many (if not majority of) existing debugging tools ineffective.

In this context, a widely adopted industry approach to aid reasoning about distributed systems is *distributed systems tracing*. Here, the most popular tools are modeled after Google's Dapper [19]. However, the original paper, in which Dapper was introduced, notes that the proposed model does not handle coalescing effects (see, section 2.1.1 "OpenTracing and its implementations"). This fact was one of the reasons for work completed in this thesis.

1.3. Control plane systems

Coalescing effects are often present also in another type of software systems, which we will call *control plane systems*. These systems have additional properties making them harder to reason about.

Control plane systems materialize most often as upper layers of cloud service stacks. They accepts API requests and translate those into a series of requests sent to lower-level systems. These cloud services are handled by a multi-layered software system. Each layer, itself, is a distributed system, often deployed in a microservices architecture. Here, each layer handles some aspects of the API request, e.g. authentication, authorization, request routing, billing, underlying containers management, cluster-level VM machine orchestration, node-level VM management, networking configuration, and so forth.

Cloud APIs can be both imperative and declarative, but in many cases they have elements of both paradigms. Imperative API allows user to perform direct actions on cloud resources. These can range from simple operations like "shut down a virtual machine" to complex long-running and multi-staged operations like "live-migrate a VM to another region".

Declarative API allow users to declare their intent with a single API call. These intents can be arbitrarily complex. An example of such desired intent is a shape of a service deployment, e.g. size and regional distribution of a group of virtual machines. A desired intent can – besides virtual machines – declare resources like networks, VPNs, virtual IPs, load balances, persistent volumes, SQL databases, etc. Declarative API calls typically trigger long-running asynchronous work, performed by the service provider servers, to change the deployment to match the desired state.

Additionally with the advent of Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) solutions, deployment systems are being provided to customers, as services. For example, any PaaS offering exposing Kubernetes API, effectively exposes an intent-based automation solution. It can safely be assumed that PaaS offerings from Amazon Web Services (AWS), Google, or Azure – given size of their customer base – in practice, combine the scale of big serving systems with complex execution model of intent-based systems.

All of the aspects described above, and knowledge gathered from practical use of intent-based actuation, make the cloud control plane systems one of the most complex systems in existence. Hence, they are particularly hard to reason about. Finally, it is worthy noting that one more aspect, which makes some control plane systems hard to reason about, is their hierarchical nature.

1.4. Hierarchical control plane systems

In deployment practice, some cloud service API are implemented on top of other cloud services. For example, Google Cloud Functions (GCF) runs on top of Google Cloud Run (GCR), which runs on top of Google Kuberentes Engine (GKE). In turn, GKE runs on top of Google Cloud Engine (GCE). Hence some control plane system are not only *hierarchical*, but the hierarchy can have quite a complex structure of dependencies.

As discussed in section 1.1, Kuberentes is a system employing intent-driven actuation. This makes GKE – as a Kubernetes implementation – a prominent example of a hierarchical control plane system employing intent-based actuation, combining multiple properties. As it can be easily realized, such structure of the system makes it hard to, holistically, reason about it.

1.5. Build systems in control plane systems

Let us now consider the fact that control plane systems, with declarative APIs, need to be provided with the desired state, to be actuated. Build system outputs are often used as the desired state, provided to the control plane system. For example, Infrastructure-as-Code (IasC) model recommends treating service deployment as any other type of code – to be be built and tested using *build systems*. Here, Bazel build system is being used to define desired shape of Kubernetes deployments [20], [21]. Immutable infrastructure [22] model – a model related to the IasC model – often prescribes use of Docker containers, which are built in Docker's primitive build system [17]. These containers are then provided to a control plane system to be deployed. Build systems typically output composite objects as outputs (e.g. a directory, an archive, or a container image), but cloud management tools typically extract individual components of these composite object then send them to cloud APIs.

Furthermore, *build systems* – or build-like systems – are often used as parts of control plane systems. For example, Terraform Cloud [23] and EPAM Cloud [24] execute Terraform planning process, on behalf of users as part of their API. Terraform planning process is a build process, which builds an artifact describing a set of actions to be performed to reach a desired state, declared in the input files, and the actual state. It is worth noting that the EPAM Cloud product is a multi-cloud orchestration solution, making it a hierarchical control plane system employing the intent-based actuation.

Tools like NixOps [16], Disnix [25] and Kubenix [26] take this approach even further, blurring the lines between the build process and the deployment process. They allow to define a deployment of a cluster of VMs, or containers, end-to-end, starting with individual binaries, through the content of deployed VM, or containers images, all the way to the shape of deployment in a cloud.

With this background, it should be clear that ability to reason about build system actions, and their inputs and outputs, is an important element in the overall ability to reason about modern automation systems in Cloud.

1.6. Data processing pipeliens

As a side note, it can be observed that the complexity of data and data processing pipelines, within cloud infrastructures, in particular, as been growing exponentially. The growth affects both sizes of data repositories and complexity of data processing systems (e.g. workflows). One of the important tools to deal with the growth is tracking provenance of data used in e-science [27]. Provenance has been researched in a diverse set of areas, can be implemented at various levels, and used for a range of applications (see [28] for more details). This adds one more aspect to the overall image of modern IT deployments and will serve as a source of ideas for this work.

1.7. Problem statement

To summarize, the need to be able to reason about software systems is an ever-growing concern in the industry. For the reasons described above, *hierarchical control plane systems employing intent-based actuation* – deployed in large-scale clouds – are especially hard to reason about.

Ability to obtain information about the behavior of hierarchical control plane systems is crucial to be able to reason about them. These systems are usually deployed as layered collections of small servers. Understanding system behavior in this context requires observing related activities across many different programs and machines. An engineer, looking only at the overall behaviour of the system may know there is a problem, but may not be able to guess which service of which layer is at fault, nor why it is behaving poorly. Given that these systems are developed by large organizations comprising of multiple teams, the engineer may not be aware precisely which services are in use; new services and pieces may be added and modified from week to week, both to add user-visible features and to improve other aspects such as performance or security. The engineer will not be an expert on the internals of every service; each one is built and maintained by a different team.

Given the hierarchical nature of some control plane systems a tool providing information about its behaviour needs to be ubiquitously deployed across the layers and services. Moreover services and machines may be shared simultaneously by many different clients, so a performance artifact may be due to the behavior of another application. A control plane system deployed as part of a cloud service is inevitably a multi-tenant system. Behaviour of individual tenants can affect how other tenants are being served by the control plane system, and it can affect health of the overall control plane system. Such interactions are difficult or impossible to reproduce, hence requiring the tools providing information about the control plane system to be always on.

Control plane systems have a unique property as they actions are primarily focus on managing a state. That is their APIs deal with reads and mutations to resources; these APIs change the state of resources and depend on these states; that is to say that they heavily depend on side effects. Debugging such systems requires full coverage of tracked activities and recorded state changes (e.g. sampling technique employed in Dapper is not suitable here).

Debugging tools are most useful during incident response (e.g. finding the root cause of SLA violation for a given tenant). This puts a soft requirement onto a debugging tool to make tracing data available for analysis quickly after it is generated. Availability of fresh information enables faster reaction to production anomalies.

Given the above and author's experience in industry, this work is addressing the needs of a software engineer developing, maintaining and operating hierarchical control plane system employing intent-based actuation in a large-scale cloud deployment. The deployment is structured into hundreds of microservices layered in groups on top of each other, moreover it depends on dozens of infrastructure services providing IaaS-level mechanisms. The system is cumulatively owned by tens of teams. Teams evolve over time, with people joining and leaving, making onboarding time an important business consideration, hence making knowledge transfer between engineers both crucial and sometimes imperfect.

1.7.1. Goal of work

Taking into account what has been said thus far, the following research goal can be formulated.

The aim of undertaken work is to propose a debugging solution, applicable to hierarchical control plane system employing intent-based actuation.

The final purpose of the sought-after solution is support the aforementioned software engineers, more effective at developing and operating hierarchical control plane systems, and allow them to be more efficient throughout the whole software development lifecycle.

1.7.2. Hypothesis

A number of approaches to problem of reasoning about complex software systems have been developed in the industry and researched in the academia. Our hypothesis is that the problem of debugging of hierarchical control plane system employing intent-based actuation can be solved using elements from distributed systems tracing area and elements from provenance tracking area.

1.7.3. Motivating example

This section will describe a motivating example, which will be use throughout this work to evaluate various aspects of the problem and a proposed solution. We will model, formalize, implement and analyze these throughout this thesis.

Buggy deployment

Let us start with the a guestbook service – a classic service in which visitors can leave a message that will then be written to a database and subsequently shown to later visitors of the same site. It is deployed on top of two containers, using `docker run` to run them and using `scp` to deploy configs. Docker containers run on a single remote host. Docker is configured to use a private image repository. Application container contains a guestbook implementation and expects a configuration file provided at `/data/configs` volume. Database container runs MySQL and stored its data on `/data/mysql` volume. A full deployment script for this system looks like this:

```
docker build -f Dockerfile
docker push
ssh remote -- docker pull guestbook-app:alpine
ssh remote -- docker stop --name app
scp ./config/app.conf remote:/opt/guestbook/configs/
ssh remote -- docker run --name app \
    -v /opt/guestbook/configs:/data/configs -d guestbook-app:latest --rm
```

A typical deployment contains updates both to the source code and updates to application configuration. Changes to these are performed by two independent software engineer teams. A DevOps engineer is observing that guestbook has suddenly started returning HTTP 500 error code when accessed. In this scenario the root cause is a new bug in the source code of the application causing application to try to connect to a incorrect database port.

This sample is not a control plane system, but it is useful as it allows to show a major part of the problem using familiar mechanisms on a simple system.

Rollback of source of truth

A second example used in this work is a heavily simplified intent-based deployment system, which has enough elements of a control plane systems to be representative of one. A system consisting of an application is deployed using the Gitops approach[29]. A git repository is used to store code and configuration of the application. The deployment service is responsible to perform a deployment whenever new version of the code is pushed. The deployment process gets informed about a new commits in the repository by a message by a server-side Git repository hook. When notification is received, the server checks out a repository at the most recent commit hash and performs a build of an application in a temporary directory. It compares the resulting binary and the new config with the currently running binary and the currently used config. If they differ, the application is stopped, the new binary and the new config are copied on top of the old ones, and the new binary is started. The deployment process is slow and concurrent runs are not permitted.

In this scenario a problem is observed after a series of commits are pushed to production in quick succession by a team. The bug has been introduced in one of these commits. An engineer is responsible for identifying the root cause and fixing the service by reverting the offending commit.

In a realistic system using infrastructure as code deployments with automatic pushes to production, typically more complex setups are used involving more complex systems like Kubernetes, Bazel, Terraform or similar. Describing in details operations of these complex systems is infeasible in a thesis. Both the system in the example and these complex systems employ intent-based actuation. In the example the intended state is stored in the HEAD a Git branch, and the actuation is the deployment process which does whatever necessary to make the desired state a reality.

1.8. Outline of the thesis

Chapter 2 ”Related work” looks into the research areas relevant to the problem and identifies shortcomings of existing solutions. Chapter 3 ”Requirements” summarizes requirements we have identified for the solution. Chapter 4 ”Provenance-enhanced distributed systems tracing model” introduces a data model for tracking of work, interactions, and objects in control plane systems systems as a set of concepts; provides a sketch of formalization based on graph theory; it defines a set of extensions which allow to infer additional information from a dataset. Chapter 5 ”Architecture” presents an architecture of the software supporting the described model and being required to satisfy the requirements. Chapter 6 ”Implementation” describes select details of the proof-of-concept implementation of the solution. Chapter 7 ”Experimental results” reports how the system was applied to a set of real-life software that is representative of our use case. Chapter 8 ”Concluding remarks” summarizes our research and practical experience of using the proposed model and its implementation.

2. Related work

So far we have outlined the problem, and the area of the proposed research. In this section we will discuss related approaches explored in scientific literature, and in the industry. Overall, this is a well-studied area, with a lot of research, so we will refer only to the works most relevant to the problem of debugging (reasoning about) hierarchical control plane systems, employing intent-driven actuation.

Here, let us note that, in general, *distributed systems tracing* and *provenance tracking* are useful tools that allow developers reason about complex software systems. The former has evolved in the industry, as a pragmatic tool to solve day-to-day needs of companies running complex software systems. Provenance tracking, on the other hand, has been researched extensively in academia, in numerous fields. Both approaches (and developed tools) overlap (e.g. both use the concept of record of work, as a fundamental building block) and can be used to address different aspects of "debuggability". In this work, we combine them into a suitable tool for our use case (section 1.7). The proposed solution is based on a widely adopted industry-standard distributed systems tracing mechanisms, with extensions drawn from the provenance tracking research. Let us now review related works in both areas.

2.1. Tracing in distributed systems

2.1.1. OpenTracing and its implementations

As mentioned in chapter 1, the state of the art solution, in the industry, for debugging (reasoning about) distributed systems, are tracing solutions following the OpenTracing model [30], for example Google's Dapper, Uber's Jaeger, or Twitter's Zipkin. These implementations are based on white-box tracing, where a traced service is instrumented with explicit logging statements in appropriate places in the control flow of the service. Dapper instrumentation is integrated in the common RPC layer of internal Google services. Open-source implementations assume that instrumentation is added into the business logic of the traced service.

This model has been standardized by the W3C standards body, as the W3C Recommendation

Trace Context [31] standard, for tracing HTTP requests. This form of distributed system tracing was originally introduced by Sigelman et al. [19] and the model does not provide a solution for tracing coalescing effects. It is important to stress that none of the existing open source implementations of this model provide a solution to this problem.

Dapper has been optimized for tracing in large scale online serving systems, which are heavily skewed towards scatter-gather read operations. One of the central optimizations in Dapper is a built-in sampling mechanism, which is used to probabilisticly limit number of traces gathered throughout the system. Control plane systems are different, since they are not skewed towards reads as much. Their main use case is management of compute, storage, networking, and other cloud resources. This also means that their query rates are not as high as of online serving systems. Henceforth, debugging control plane systems involves different trade-offs. Ability to trace all activities is highly desirable (since most of them are mutating operations) and achievable (since their query rates are lower).

The approach taken by OpenTracing is called "tainting". Here, all work performed in response to the original incoming request, is tainted with a single unique (or unique-enough random) identifier, called a trace identifier. The identifier is propagated across all systems and activities transitively participating in handling the request.

Although Dapper is built and is being used to trace activity triggered by RPCs, interaction tracking is not a first class citizen of the Dapper model. Dapper's annotations are used to track interactions between processes, but annotations are technically an extensibility mechanism and their payload is not part of the model.

Typical tainting approach supports propagation of a single identifier only, hence its inability of tracing of coalescing effects. It's storage requirements are linear in number of work units. The tainting approach supports only tree-shaped traces. Here, let us notice that a tree-shaped trace is too restrictive for intent-based systems. This is because the coalescing effects are inherently present in intent-based systems, make their control flow to not be representable as a tree. This means that any intent-based actuation performed in a system can be often linked with more than one incoming request, hence **necessitating a more flexible model**.

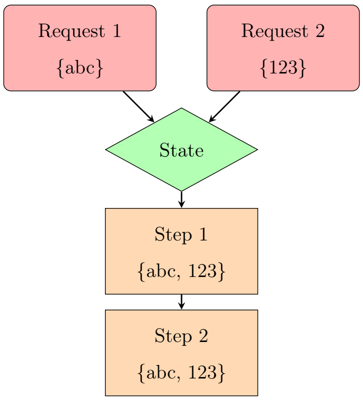


Figure 2.1: Naive extension of the tainting model in presence of coalescing effect

A naive extension to the tainting approach introduced to handle coalescing is to perform propagation of a set of trace identifiers (see fig. 2.1) to child spans. However, this exhibits a number of problems:

- has super-linear storage requirement in presence of coalescing effects, since each recorded work units, caused directly or indirectly by a work unit where a coalescing occurred, needs to store multiple trace identifiers;
- duplicates records of some units of work in multiple trace trees, making it harder to analyze traces;
- does not capture relationship between work units and the state, which was used for coalescing.

Moreover, this model is not capable of dealing with intent-based systems, since causality relationship from an incoming API request to a work unit in intent-based systems is not surjective.

2.1.2. OpenTelemetry and it's observability mechanisms

Other mechanisms, aiding developers in reasoning about their services, are well represented by the OpenTelemetry project, which provides capabilities for capturing metrics, distributed traces, and logs. The OpenTelemetry provides a single set of APIs, libraries, agents, and collector services, to capture distributed traces, metrics and logs from a system.

Distributed tracing of OpenTelemetry follows the OpenTracing model [1], hence inheriting its shortcomings.

Metrics observability mechanism is most suitable to capture aggregated statistics about the behaviour of the system. Metrics can be labeled with additional information, allowing for finer-grained view, but it is still focused on over-time aggregated numbers [32].

Logging observability mechanism, on other hand, is typically not structured, i.e. the content of a log message is a free-form string. OpenTelementry logging mechanisms allows one to associate a log entry with a trace span, providing a more structured information [33].

Neither of the aforementioned mechanisms are suitable to track relationship between individual incoming requests and individual work units performed by a system.

2.2. Provenance and provenance tracking

Provenance tracking is an extensive field of research, which provides useful mental models, to reason about operations of complex systems. Let us review how these relate to our use

[1] Actually OpenTracing project has been merged into OpenTelemetry project while this thesis has been worked on.

case (section 1.7).

2.2.1. Provenance in Cloud

Usage of Cloud brings additional challenges into the area of provenance tracking: clouds are, typically, build in multi-layered architecture of IaaS, PaaS, and SaaS layers. Imran et al [34] highlighted the need for aggregated provenance tracking, across multiple layers of cloud environment. The paper talks about the problem, and sketches a number of properties a solution needs to have. They show that provenance tracking, for individual layers, has been researched extensively, and points out that to track provenance in cloud setting it is necessary to be able to aggregate provenance information across the layers at the query time. They present a use case, where the needed mechanism is implemented.

Our goal is to provide a generic solution to the problem described in this work.

- Cloud service providers often employ both imperative and declarative APIs. Declarative API often allows users to declare their arbitrary complex intent, e.g. a desired shape of a deployment of a set of virtual machines, with a single API call, which results in asynchronous work done by the service provider servers, to change the deployment to match the desired state.

- These systems consist of a large number of servers communicating with each other via network RPC interfaces, where some systems are traditional imperative systems and some systems implement intent-based actuation.

2.2.2. Provenance in network control plane systems

Network control plane systems have been moving towards intent-based paradigm. Intent-based networking (IBN) is an approach for automated and policy-aware network management. IBN extends the software-defined networking (SDN) model, by allowing engineers to specify *what* policies they want their network to implement, rather than *how* their network's underlying mechanisms will implement such policies [35].

ProvIntent [36] is a framework extension for the SDN control plane tools that accounts for intent semantics. It extends the ProvSDN [37], to explicitly incorporate intent evolution as intent state machines into provenance tracking.

2.2.3. OpenProvenance

OpenProvenance defines the W3C PROV data model for provenance interchange on the Web. It is primarily the data model, a number of interchange formats, and a set of libraries to work with these formats. PROV data model is focused on provenance information interchange for Web documents, but it can be used for other types of data. OpenProvenance model (OPM) is sufficiently expressive to deal with the coalescing effects problem, since it is possible to represent non-tree-shaped activities with it and it allows to record objects as triggers for activities.

However, we find PROV model to not be suitable for our use case (section 1.7) for the following reasons.

Storage overhead. Large scale control plane systems are mostly-hierarchical systems, where most of the work is performed in a tree-shaped control flow. PROV Activities are not structurally hierarchical, but one could represent this relationship with consistent usage of *wasStartedBy* and *wasEndedBy* relations. However, the storage overhead of this is considerable: in the OpenTracing model, this relationship is represented by a single identifier stored once per child “span”, while the OPM requires recording two relations, with 1 required and 5 optional fields each, to achieve the same. We conjure that it would preclude its usage in large scale control plane systems of Cloud providers.

Verbosity. "Representing distributed systems using the Open Provenance Model" [38] describes application of the OpenProvenance model to represent some aspects of distributed systems, focusing on representing message passing communication between services. This extension shows how the model can be applicable to our use case (section 1.7). The paper describes that a naive representation of messages passing in OPM requires 8 nodes to represent two messages. The paper introduces D-profile – an OPM 1.1 profile – which acts as an abstraction over the naive OPM graph, reducing the verbosity of the representation. This means that, when using the OPM, we need to trade off verbosity versus mental burden of using the tool. This leads us to the next point.

Complexity. The OPM model is very expressive and supports tracking much more complex interactions. It supports concepts like Alternates, Bundles, Communication, derivations, relations sub-typing and much more. In contrast to this OpenTracing model uses just three concepts (trace, span, annotation) to represent traced activities.

We believe that a tool introduced into a toolbelt of software engineers to aid with reasoning about the system should be easy to reason about itself, or it defeats the purpose. We also believe that the optimal solution should be an incremental extension of the OpenTracing model, to give it a chance of adoption in the industry. Hence, the optimal solution to our use case (section 1.7) lies somewhere in between the OpenTracing and the OpenProvenance models in the design space of debuggability tools.

SPADE

SPADE [39] is a practical implementation of the Open Provenance Model, to capture provenance data in distributed settings, using PROV model. SPADE is focused on gathering low-level information from OS audit logs, network artifacts, LLVM instrumentation, etc. SPADE provides support for tracking provenance in distributed systems, but the focus is on low-level information and on automated provenance gathering mechanisms persisted. SPADE's syscall and library call level instrumentation would not scale for a large scale production system.

2.2.4. Provenance formalisms

Provenance tracking has been also researched from the perspective of formal systems. We will take a look at a few selected works in this area.

π -calculus. Souilah et al, 2009 [40] present a formal provenance in distributed systems, based on the π-calculus. The approach is based on enriching exchanged data with provenance information, similar to the tainting approach. Hence this approach is not suitable for our use case (see section 2.1.1).

Why-across-time provenance. Why-across-time provenance (wat-provenance) [41] provides a mechanism to track data provenance in the realm of state machines, in time-varying stateful distributed systems. It provides a formal model, which could be a foundation to implement a provenance-enhanced distributed systems tracing solution. Wat-provenance requires determinism, while a lot of large scale distributed systems involve a fair share of non-determinism in load balancing, bin-packing, resource allocation, load shedding, etc.

Additionally both these formalisms require a large departure from the well-established OpenTracing paradigm.

2.2.5. Database provenance

Database provenance is typically concerned with tracking what input data has been used to produce a given output data. This is similar to our goal of tracking how a configuration is used through multiple layers of control plane systems to affect the state of a deployment. Hence let us consider it is some detail.

Provenance traces [42] research database provenance at the query time. Approach taken by Cheney et. al. is a bottom-up approach – the traced system needs to be written from scratch, using the formal language defined in the paper. The solution generates very large traces, since all data transformations performed by a database engine is recorded. The data gathered during query execution is later analyzed using a *slice* operator, to extract useful information.

If applied naively, the bottom-up approach and very large traces of "provenance traces" solution make it infeasible to be used in an established large-scale control plane system, since it (1) requires rewrite from scratch, and (2) does not allow developer to fine tune which tracking data is gathered. This makes it much less likely to be adopted in industry.

2.2.6. Provenance in security

Provenance and provenance tracking in security is focused on threat detection and protection areas. This puts a number of constraints on provenance tracking approaches used in this area.

ProTracer [43] uses a mix of logging and tainting approaches, and is focused solely on the Advanced Threat Protection area. It makes use of kernel-level audit logging and syscall interception that is a black-box approach with zero trust towards the processes observed. ProTracer traces usage of OS-level objects like files, IPCs and network sockets. Tracing of network operations allows ProTracer to reconstruct causality across hosts. ProTracer generates a very detailed dataset about an observed process.

ProTracer does not provide a general data model, making it impossible to represent abstract entities, like cloud resources or Kuberentes objects. Simple operations on these entities – e.g. create a disk image – in control plane systems, often involve thousands of kernel-level operations, across multiple hosts. This means that the kernel-level objects provenance tracking does not scale, up to the needs of large scale control plane systems, since the signal-to-noise ratio in the produced dataset will inevitably be low. For these reasons ProTracer approach is not applicable to our use case (section 1.7).

CamFlow [44] is an automated provenance capture system, implemented as a Linux Security Module (LSM), designed for the purpose of system audit. It tracks whole system provenance, by constructing a provenance graph. Here, vertices are states of kernel objects, while edges capture information flow between them. It focuses on a single machine provenance tracking and outputs results in W3C PROV format, but is specialized for the kernel objects. CamFlow allows to integrate application-level provenance into the generated dataset. This is done by configuring an application to log to a pseudo-file provided by the CamFlow kernel module. This allows CamFlow to associate individual log records with a process activity tracked, incorporating them into the CamFlow provenance graph.

ProTracer, CamFlow and other provenance tracking solutions used in security context are taking different trade-offs than are necessary in provenance tracking of control plane systems. Security usage requires zero trust towards the traced system, while our main use case allows for full trust. This forces security provenance solutions to focus on observed provenance, while it is infeasible for large scale control plane systems to use observed provenance due to prevalent use of abstract entities.

2.2.7. Provenance for workflows and data transformation pipelines

One more area, where provenance has been researched extensively, is scientific workflow management systems, and the wider area of data transformation pipelines. Let us summarize relevant research in this area.

RAMP is a data intensive scalable computing (DISC) provenance framework [45]. This system is restricted to the domain of data intensive computations, over static data. It is implemented as a set of wrappers around Hadoop primitives, which produce provenance data, as data items are acted upon by these primitives, in a form of out-of-bounds metadata. These wrappers are automatically applied to a Hadoop flow. The control flow captured by the provenance metadata reflexes that of the executed flow itself. Therefore, it cannot be applied to other distributed systems components, like storage systems, coordination services, load balancers, etc. where control flow graphs are not fixed, they evolve as the system evolves, and data passes through the system via a large set of mechanisms (RPCs, databases, pubsub systems, etc).

LogProv is a provenance logging system, implemented for Apache Pig and Apache Hadoop [46]. It supports dynamically shaped big data workflows and pipelines. It does not store provenance information directly into a graph database. Instead, it is using structured logging into the ElasticSearch. LogProv data model can be represented with the OpenProvenance data model only if inspected workflows and pipelines are fixed. LogProv allows to maintain low overhead (which is important in our use case (section 1.7)), and keep flexibility to handle frequently changing pipelines. LogProv is focused on DISC provenance and is not built for distributed systems tracing.

Overall, RAMP is a very specialized framework for DISC provenance tracking, while LogProv is an example of a less specialized and more flexible solution to the same problem. LogProv still focuses on DISC provenance, but we believe it can be extended to handle other use cases. LogProv's approach of requiring explicit use of provenance-enhanced logging entries in Pig Latin script that flexibility.

2.3. Build systems

As discussed in chapter 1, build systems are directly related to the research topic of this thesis. Additionally, tools developed for the needs of reasoning about build systems, combine elements of tracing and provenance tracking.

2.3.1. Build systems tracing

Ability to reason about actions of build systems is important for optimization of a software development lifecycle in any company. Build systems are inherently intent-based, since a build target is described declaratively, and the build systems perform necessary steps to create the target. Build systems, typically [17], rely heavily on coalescing effects. Memorization and incremental recomputation are coalescing effects, and are used to achieve a minimality property. The early cutoff optimization is a coalescing effect, which is used to avoid rebuilding downstream build targets, if changes in a given target are inconsequential.

Some build systems, like Bazel [47], Nix [48, p. 176], or redo [49] provide mechanisms to reason about, debug and optimize their operations. These trace mechanisms are implementation-specific, informal, and do not guarantee composability with other provenance sources, when aggregated. Hence they do not address our use case (section 1.7).

Tracing build processes have been used to uncover licence compliance inconsistencies [50], which is a form of provenance tracing operating in a build system context. This approach does not address hierarchical nature of control plane systems, and captures only two specialized levels of provenance for build tasks and files used in builds.

In section 7.1.1 ”Build systems – Nix” we will, however, describe an application of the same approach as CBDG build trace collection and graph construction to capture hierarchical relationship between package-level build process and individual build processes.

2.3.2. Dependency tracking

Finally, let us note that build system dependency tracking can be considered a provenance tracking technique, for build targets based on their inputs. It falls into the disclosed provenance category [51]. However, at best, it works at the level of the build graph. An observed build-time provenance tracking mechanism would be necessary to achieve better granularity. Each dependency between packages, in a build system, is equivalent to one or more usages of one or more files from a package A, when building a package B, where execution of a build is performed by a process (e.g. “gcc” or a similar file-based compilation tool). In cases where build systems are operating as a part of control plane system, an ability to track relationship between build system outputs and actions of a control plane component is necessary.

2.4. Summary

We were unable to find a solution to the thesis problem in the literature, as sown in the overview of the related work. There are multiple works which address various aspects of the problem, but none addresses all of the needs of the use case this thesis is focused on (see section 1.7). The next section will explicitly list requirements for the sought-after solution.

3. Requirements

Based on the problem statement (see section 1.7) and overview of related work (chapter 2) we will now outline the requirements towards a system that is going to address the described use case.

We believe that a mix of distributed systems tracing and provenance tracking is the right solution to the problem of debugging modern automation systems, and hierarchical control plane systems, specifically. We are looking for a solution, which has the following properties:

1. *Coalescing effects support* – required to support tracing of intent-based actuation logic;

2. *Support for abstract entities* – essential to align well with cloud APIs, which support entities at many levels of abstractions (VMs, containers, clusters, deployments, applications, functions, etc);

3. *Support for composite entities* – required to support objects like archives, VM images, container images, etc. which are prevalent in cloud APIs;

4. *Low storage overhead* – necessary for large-scale systems, deployed as control plane systems at cloud providers;

5. *Full coverage* – ensures that best effort is taken for all activities are tracked and resource state mutations to be recorded (modulo presence of unavoidable infrastructure faults);

6. *Gradual fidelity execution tracing* – allows developers to selectively apply execution tracing, to trade off tracing accuracy and implementation effort;

7. *Gradual fidelity provenance tracking* – allows developers to selectively apply provenance tracking, to trade off provenance tracking accuracy and implementation effort;

8. *Minimal mental burden* – to allow for change of adoption in industry;

9. *Cross-host tracking* – for distributed systems tracing support;

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

12. *Event-based data production* – to deal with compute nodes and Unix processes faults and to avoid buffering in tracing mechanisms in the application;

13. *Flexible control flow support* – accept tracing of systems as they evolve over time and to be able to deal with pre-existing complex control flows in control plane systems.

Based on the problem statement, and analysis of the related work, we can also identify properties, which we are **not** pursuing in our solution:

1. *Support for operating in a limited trust environments* – our use case allows for full trust for the traced system, e.g. allows us to employ white-box instrumentation.

2. *Support for systems with ultra-high query rates* – debugging of control plane systems is mostly concerned with state mutations performed by the system, hence we are focused on faithfully recording all operations relevant to these mutations. State mutations in control plane systems are also inherently not ultra-high query rate, hence supporting such query rates is not a requirement.[1]

We believe that a system, which satisfies these requirements, will address our use case. That is, it will be able to (1) trace non-ultra-large on-line serving systems, and (2) trace hierarchical control plane systems employing intent-based actuation.

Note that this is a full set of requirements towards such a system, and existing solutions like Dapper or CamFlow each satisfy a subset of these requirements, but – as shown in chapter 2 ”Related work” none of them satisfies all of these requirements (see section 8.1 for more discussion).

We will address the requirements outlined above as we propose, analyze, develop and test the solution.

[1] If a given request is not mutating any observable state, the model still can be use to trace it. Moreover such requests can be sampled to support ultra-high query rates, but these will obviously not satisfy “full coverage” requirement.

4. Provenance-enhanced distributed systems tracing model

In chapter 3 ”Requirements”, we have discussed the features and properties that we would like to see in a debuggability solution applicable to modern automation systems, including hierarchical cloud control plane systems employing intent-based actuation. In this chapter, we propose a model satisfying these requirements.

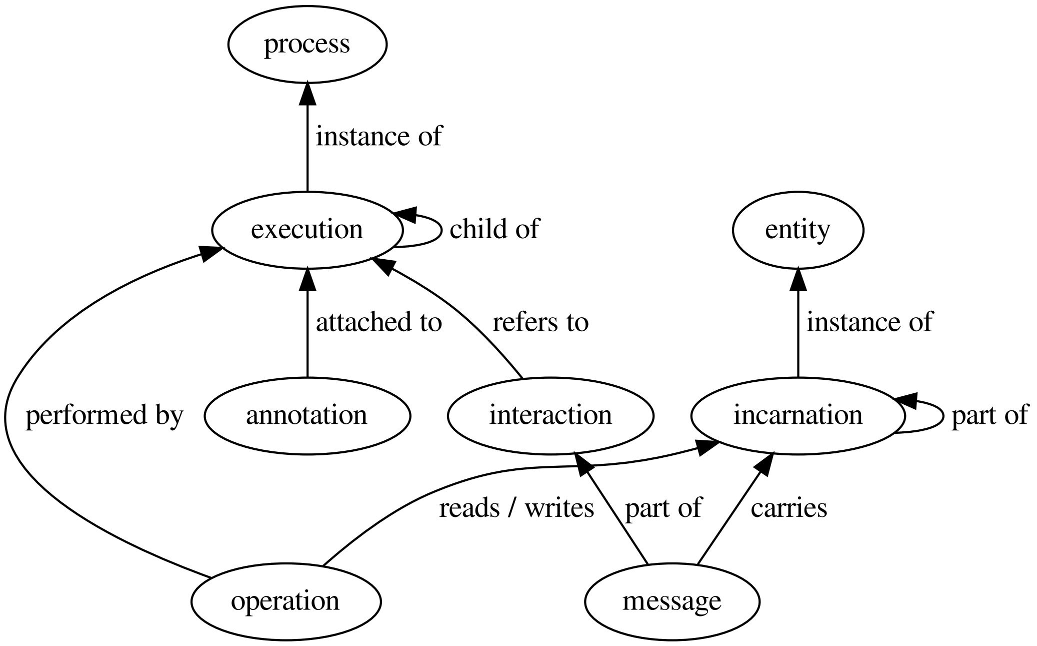
We present Provenance-Enhanced Distributed Systems Tracing (PEDST) model as a foundation for the debuggability solution. PEDST extends OpenTracing model with minimal provenance-tracking capabilities, to address the use case (section 1.7) of tracing hierarchical control plane systems employing intent-based actuation. The extension is focusing on ability to record interactions between logical work performed by the system and objects the system interacts with. As we extend it, we maintain the fundamentals of distributed systems tracing aspects (e.g. activities used as the main tracing vehicle).

4.1. Data model

We propose the following set of concepts (see fig. 4.1) used in PEDST model. These concepts are an evolution of the OpenTracing model concepts with provenance information incorporated to satisfy properties we chose in chapter 3 ”Requirements”.

We introduce a concept of *execution* to track units of work (similar to the OpenTracing “span” concept), whose *operations* on *entities* are explicitly recorded. An execution is typically nested in a parent execution. *Entity* is any kind of thing, which is important-enough to track, in a given system. Read and write *operations*, performed by an *execution*, allow to track provenance of objects they operate on. Each write *operation*, on an entity, gives rise to a new *incarnation* – an immutable revision of an *entity*. Entities are mutable. *Executions* of the same logic are grouped by their association with a *process*, which describes a predefined procedure, such as recipe, tutorial, business logic, workflow, or instructions. A pair of *executions* can interact with each other and is recorded as an *interaction*, which consist of a number of *messages*. A *message* may carry an *incarnation* as a payload.

Figure 4.1: Model concepts.



The proposed PEDST model supports both activity tracing and provenance tracking. The main vehicles of tracing are executions and their parent-child relationships [19]. Correspondingly, the main vehicle for provenance tracking are read and write operations, performed by executions on individual incarnations. This places the provenance tracking capability of this model in the "data provenance" type [28]. The capability to dynamically track provenance of objects (e.g. files, configs, resources, collections, etc) addresses contributes to the "support for abstract entities".

Concepts of executions, incarnations, and operations, are sufficient to perform both execution tracing and provenance tracking and thus such model satisfies our requirements. Other concepts are useful for the model, as based on our real-world experience: *process* and *entity* concepts allow us to improve the usability of tools built on top of this model. *Annotations* are necessary for the model to be a superset of OpenTracing model. *Interactions* and *messages* are necessary to track provenance propagation through RPCs and other message passing mechanisms present in distributed systems.

Furthermore, use of OpenTracing model as the foundation allows us to partially satisfy "minimal mental burden" requirement due to familiarity of general audience of industry software development with this model. Nevertheless PEDST model is more complex than OpenTracing model, but we consider this complexity inherent for the problem. We believe that *entities* concept, their versioning mechanism with *incarnations* and *operations* as a link between activity tracking and object provenance tracking are the minimal extension on top of OpenTracing model, which would make a model comply with other requirements.

Let us now describe in more detail each of the elements of the PEDST model.

4.1.1. Execution

Execution records a unit of work performed by a component of the system, at the granularity chosen by the developer. Execution can be thought of as an instantiation of a *process* (see, section 4.1.4), similarly as an instance is an instantiation of a class, in Object Oriented Programming. Similarly to the OpenTracing "span" objects [30], executions are represented hierarchically, as a tree, via parent-child relationship. Executions have a beginning and an end in time. As execution records a unit of work in progress, or finished, as soon as the work finishes the corresponding execution gets an end timestamp and becomes immutable. Good candidates for being recorded as executions include:

- work performed by an RPC handler of an application server for a specific RPC request;
- compilation phase performed by a language compiler when compiling a specific executable;
- execution of a user-provided code for handling a GET request in a HTTP server like Apache for a specific RPC request;
- specific execution of a workflow step in a workflow engine;
- main loop of a network server responsible for accepting network connections in a specific Unix process;
- query execution of a specific SQL statement on a specific database engine at specific time sent by a specific user.

An execution tree – an execution with its children – represents a logical chunk of work. It may be a simple record of a primitive operation with no children, or it can be a large and complex chunk of work, e.g. record of a large scale database migration. This hierarchical representation of executions allows to gradually improve fidelity of execution tracing – starting with tracing "large chunk" executions covering large and complex work units, and later on gradually adding more fine-grained executions covering elements of the large work units. This satisfies the "gradual fidelity execution tracing“ requirement.

An execution can have a creator execution associated with it (see fig. 4.2). For a given execution, a creator execution and a parent execution can differ. For example an execution

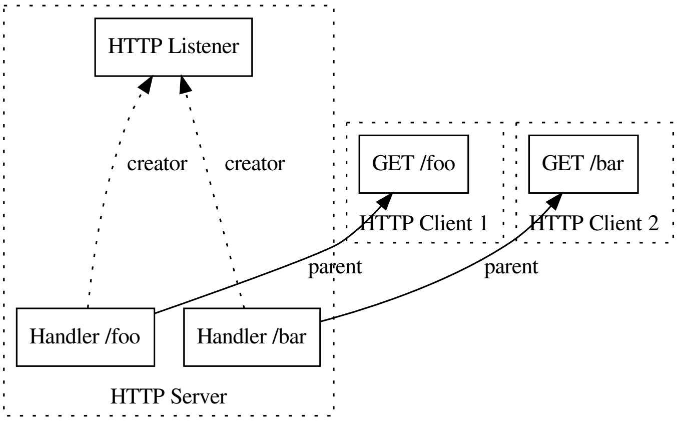


Figure 4.2: Unowned executions example.

recording a thread performing an RPC service listen loop will be a creator a number of individual executions, while these executions record work performed by each individual RPC handler. In this case, the creator execution is not the same as parent execution, since the parent execution will be an execution recording work performed by a networking client, which opened a socket.

A pair of executions can interact (see section 4.1.5). An execution can also perform operations (see section 4.1.3).

4.1.2. Incarnation

Incarnation represent a specific state of an entity (see section 4.1.6) in time (see fig. 4.3). Hence, it has a creation timestamp and it is immutable. When an entity is modified, a new incarnation is recorded. In case of entity deletion, the incarnation will be marked as a tombstone, thus marking the represented entity as no-longer existent. A sequence of incarnations, of a given entity, gives rise to the object timeline. Incarnations are the vehicles of provenance tracking in the model (that is they are nodes in the provenance graph; see, section 4.3).

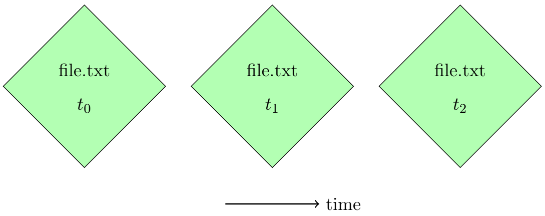


Figure 4.3: Incarnations of `file.txt`.

One can conceptualize entities and incarnations through the lens of version control systems (VCS). In our model, a resource – usually a file – versioned by the VCS is tracked as an entity, while each revision of a given resource stored in a VCS is tracked as an incarnation of this entity.

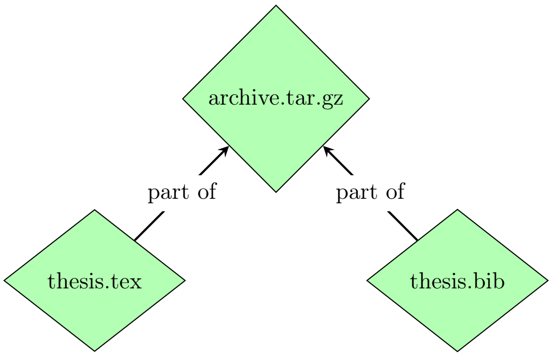


Figure 4.4: Representation of result of `tar -czf archive.tar.gz thesis.tex thesis.bib` as incarnations in our model.

Sub-incarnations. An incarnation can be a part of another incarnation (see fig. 4.4). In such relationship, a parent incarnation consists of multiple sub-incarnations. Sub-incarnations are indivisible parts of the parent incarnation. Whenever a new incarnation is created for the parent incarnation's entity, new incarnations are created for all sub-incarnations. An example: a single file – a sub-incarnation – in an archive – a parent incarnation – being an output of a build system step. This relationship does propagate provenance, hence if a sub-incarnation is read by an execution, the parent incarnation becomes part of the provenance set of the said execution.

Sub-incarnations mechanism satisfies the "gradual fidelity provenance tracking adoption" requirement by allowing developers to initially track objects at coarse level (e.g. a Docker image as a whole), and gradually improve implementation to track sub-incarnations of these objects for finer grained tracking (e.g. track individual files in said Docker image). Additionally it directly addresses the "support for composite entities" requirement.

Implementations of the model are allow to apply the following optimizations without the loss of correctness:

- If an ought-to-be-created child incarnation does not differ from the previous incarnation – as an optimization – no new incarnation has to be created and the parent can point to an already-existing child incarnation. Think about sharing in functional data structure updates.

- Writing all of the sub-incarnation into a datastore can be expensive, so an implementation of this model can use implicit sub-incarnation creation optimization, where a sub-incarnation existence is recorded only when it is referred to.

It is worth noting that this is different from an aggregation relationship, where an aggregating incarnation refers to a number of entities, which are related via some form of a hierarchical relation, but their lifetimes are independent (i.e. imperative updates). Such aggregation relationships do not propagate provenance and are not considered in this work.

4.1.3. Operation

Operations are used to record reads and writes, performed by an execution on incarnations. The operation concept connects activity tracing with object provenance tracking. Write operation on an entity creates a new incarnation of the entity.

For example, an execution recording work of `cp fileA fileB` shell command would have two operations associated with it (see, fig. 4.5):

- Read operation on an incarnation of file named "fileA"
- Write operation of an newly-created incarnation of file named "fileB"

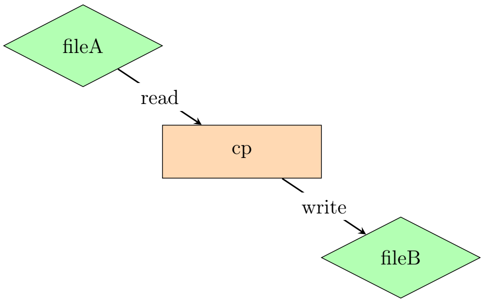


Figure 4.5: `cp fileA fileB` representation in our model.

This representation captures only the semantics of `cp` – that is copying files – and completely ignores work performed by Linux kernel when loading `cp` binary into a Unix process. It is up to a software developer using the model to decide on the level of details they need (see section 6.3.5 for discussion).

If a given incarnation is written in an operation of an execution, all incarnations read by all operations of the same execution are considered to be part of the provenance set of the given incarnation. An incarnation's indirect provenance set is defined by read operations of the given execution, and all parents of the execution. The indirect provenance set is an over-approximation of a provenance set, but it provides a very useful, and easy to implement provenance tracking technique (see section 4.3.1). As an instrumentation coverage in a given system improves, indirect provenance can be used less and less, in favor of direct provenance, hence satisfying the "gradual fidelity provenance tracking" requirement of the system.

The ability to track relationships between *executions* and *incarnations* using *operations* addresses the "coalescing effects support" requirement. When these relationships are tracked, a debugging solution is capable of capturing knowledge that evolution of an intent entity can be related to a number of executions. This allows to track a causality chain between an incoming requesting updating an intent and any executions performed while adapting the state of the managed objects to match the intent.

4.1.4. Process

A *process* describes a predefined procedure, such as recipe, instruction, business logic, or workflow, typically with a human-consumable description (equivalent to OPM's "Plan" [52]). *Processes* are instantiated as *executions* [1]. A *process* can be executed multiple times, causing multiple *executions* to be recorded. A *process* has no timestamps, and may evolve over time, leading to multiple *executions* being recorded for it with varying shape or content. A concept of a *process* can be thought of as a way to group a set of recorded *executions* into a historically related collection. For example, a code stored in a binary file can be recorded as a *process*, when this code is executed it gets recorded an *execution*.

Another use for processes is to aggregate a large number of parallel executions of virtually the same unit of work (e.g. same data processing workflow step operating on different data rows). These are useful for cutting down the noise, when visually representing a large number of executions to a user, e.g. represent them as a single node in the visualization of an execution graph.

If the model is to be used to track interactions triggered by humans in a computer system (e.g. for audit logging), humans could be represented as "processes" , and their individual interactions with a system can be represented as "executions".

Let us specify few examples of processes:

- implementation of "zip" archive creation procedure stored in a 7zip binary file;
- logic of Unix tool "cat";
- abstract query execution as implemented in a database engine;
- implementation of a game main rendering loop stored in an executable of a game;
- RPC handler for creation of a VM in a cloud API;
- implementation of a scientific workflow in a Apache Pig.

4.1.5. Interactions and messages

Interaction is used to record any form of interactions between two concurrent executions. For example, a remote procedure call, performed from unix process A to Unix process B, e.g. a gRPC method call, is an interaction between two executions, which record the work done by these Unix processes. An interaction consists of a sequence of messages.

[1] One can conceptualize *process* as a class in OPP and an *execution* as an instance of a class.

Message captures a one-way information transfer, between two executions, as a part of an interaction. For instance, a synchronous RPC interaction would have two messages – a request from the client to the server and a reply in the other direction. In case of streaming RPCs, e.g. gRPC bidirectional streaming, a single interaction representing a single bidirectional stream, can have an arbitrary number of messages recorded.

4.1.6. Entity

Entity is any kind of thing, which is important-enough to be tracked in a given system. Entity is a long-living identifiable object. Entities are mutable, and any change to an entity gives rise to an incarnation. Entity does not have a timestamp on its own, since its life-cycle is recorded by its incarnations. Entity, typically, has a human description and has to have a computer-consumable identifier, which can be used to address it. At any given timestamp, an entity may exist or may not exist (see fig. 4.6).

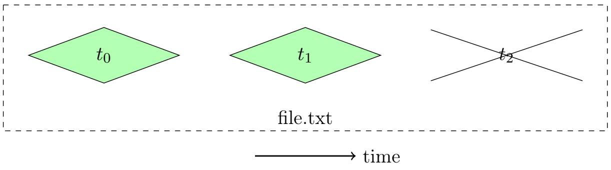


Figure 4.6: Timeline of `file.txt` entity.

Examples of entities:

- file at a specific location, e.g. Unix file path or a Uniform Resource Identifier (URI)
- intent stored in an automation system for a specific resource, e.g. Kubernetes object “spec” field
- state of a resource stored in an underlying system, e.g. Kuberentes object “status” field
- file in a Git repository.

It is often the case that the same resource is represented at different resolution/abstraction levels as multiple entities, and there is an automation process recorded as an execution, which translates between the representations. For example, a compiler creating an object file out of a source code file.

It is worth noting that an entity addressing scheme is heavily dependent on a particular use case. However, in general, entity address should be as global as possible in a given system, so that an address is sufficient to identify, locate and access an entity, with no other pre-existing knowledge about this entity. The URI standard is a suitable encoding for entity identifiers.

4.1.7. Annotations

Annotations – similarly as Dapper's annotations – are used to enrich executions with additional information. Annotation are an extensibility mechanism, allowing tool- and application-specific information to be attached to executions. It is useful to store information which does not fit the model, but is useful for a specific application. It allows custom tooling to be developed on top of the model. Annotations are time-stamped and can hold payload. The payload is an arbitrary value (e.g. using Protocol Buffers *Any* type), but an implementation of the model is allowed to impose limits (e.g. upper limit on size of annotations). Annotations are also useful to associate unstructured line-oriented log entries to executions (see section 7.1.1 ”Build systems – Nix” for example).

4.2. Logging data model

The model, above, describes the data resulting from provenance-enhanced distributed tracing. We call this “the global data model”. This model is not suitable for recording things “as they happen” (e.g. an execution represents a long-running activity, and end timestamp is not available until the end of the activity). To record things as they happen, we propose “the local data model”, which is a structured logging data model.

The local model is incremental, and is sufficient to recover data in the global model. The logging data model is event-based, with a single event being generated by a producer, whenever anything relevant happens. Each event can be recorded immediately, without a need for buffering. For example, the start of an execution and the end of an execution are logged separately, so that long-running executions can be accurately logged in presence of failures. Here is the list of events supported:

- Execution begin
- Execution end
- Operation performed
- Message sent
- Message received
- Execution annotated

Please note that no events directly related to processes, interactions and entities are included, since they are registered in the system implicitly whenever an execution mentions a new process identifier or an interaction identifier in “message received”, or an incarnation mentions a new entity identifier.

Special care needs to be taken when choosing identifiers, since they are used to compose and aggregate tracing and provenance data originating from different components in the system. The identifiers, used in the system, need to conform to the following requirements:

- Incarnation identifier needs to be recoverable from an object itself (e.g. a database row representing an entity needs to have an "id" column and "update time" column). If the underlying storage system is versioned, its internal revision numbers can be used (e.g. version control systems for files). This is to ensure that all services, interacting with a given entity, are capable of logging their interactions with coherent identifiers for entity incarnations [2]

- Identifiers of incarnations belonging to a given entity need to be in monotonically increasing order aligned with the time flow in the system. If a system is eventually consistent, it is the responsibility of the business logic of the system to conjure correctly timestamped identifiers.

- The same requirement applies to identifiers of executions of a given process.

The above requires an execution interacting with another execution (e.g. via RPC messages), to provide the other side of the interaction with its own identifier. This ensures that both sides of the interaction are capable of logging coherent data, when projected to the global data model. This is different from the tainting model, where a single identifier is propagated throughout the whole system. Section 6.3 provides more guidance and examples of how logging should be performed for common scenarios.

4.3. Theoretical foundations

In this section, we sketch how the model can be formalized using graph-theoretical terms. This sketch will help us reason about the model, how logging data model can be used to recover a global view at the traced system. It will give us a set of graph-theoretical tools to perform queries over the data and extend observed data with inferred data.

[2] This requirement will likely cause a need to re-architecture addressing schemes and access mechanisms for entities in the system, e.g. add at least a rudimentary versioning mechanism (e.g. an "last updated time" field in a DB schema). This is a good engineering practice nevertheless, so we accept this restriction of the model. As a future work, we believe it is going to be possible to define a degraded mode for this model, where this requirement is relaxed at the expense of accuracy of tracking of entity versioning.

Here, we will use “observed” and “recorded” interchangeably. We will start with defining a global model for recorded objects, which reflect knowledge about everything recorded so far in a PEDST model store. Next, we will discuss the local model, which is to be used by individual components at the time of recording of its locally-observable bits of information. We will show how the aggregating all events in the local model is enough to correctly reconstruct a global view on data recorded in PEDST model.

4.3.1. Global model

In our approach we use the following data model to record information gathered about the system under tracing. In this model U is an observed universe. We use words *reference* and *identifier* interchangeably here, with a bias towards using reference, if a given field is meant to merely refer to another object in the universe.

To satisfy requirements of supporting scalable recording of write operations and of allowing over-approximation of provenance for objects in presence of hierarchical executions, we support implicit relationships in the observed universe. The universe type can represent exactly what is being recorded, and any extension of the recorded data, with implicit relationships.

Tuples

First we will provide a set of definitions of all constituent parts of a universe.

Definition 4.1 (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 the observed executions, X is a set of all recorded interactions between participating executions, P is a set of all registered processes in the universe, N is a set of all registered entities in the universe.

Each of the fields of the universe is a set of tuples.

Definition 4.2 (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} – an end
- e_p is an reference to the parent execution

- e_c is an 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 4.3 (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
- $tombstone$ 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.

Definition 4.4 (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} - an execution whose operation given tuple represents
- t_{op} - an operation type, which can be either Write or Read
- i_{subj} - reference to the incarnation which is a subject of the operation.

Definition 4.5 (Interaction). An interaction $x \in X$ is a tuple $(id_x, e_{init}, e_{target}, M_x)$:

- e_{init} - an execution reference which has initiated the interaction.
- e_{target} - an execution reference which is being interacted with.
- M - a set of messages exchanged between E_{init} and E_{target}, where each message m is a tuple $(t, e_{init}, e_{target}, Payload)$, which are:
  - t is a timestamp of the message.
  - $Payload$ is a payload value.

Definition 4.6 (Process). A process $p \in P$ is a tuple $(id_p, description, e_d)$, where

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

Definition 4.7 (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 4.8 (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 an observed universe. These will be useful to extract information, e.g. answer queries, from the gathered data. We use this as a guide for the, to-be-presented, implementation for the model.

Definition 4.9 (Execution graph). An execution graph of U is a directed graph where nodes are elements of E and there is an edge between e_i and e_j if e_j is a parent of e_i.

The execution graph captures the parent-child relationship of executions. It is equivalent to the forest of trace trees in the OpenTracing model.

Let's call $Edges(U)$ is a multiset of tuples (e_{init}, e_{target}) for each element in X of U.

Definition 4.10 (Interaction graph). An interaction graph of U is a multigraph, where nodes are elements of E and edges are $Edges(U)$ each connecting two elements of E.

Definition 4.11 (Effects graph). An effects graph of U is a bi-partite directed graph, where executions in E form the U partition and incarnations in I form the V partition, and each element of O gives rise of a directed edge:

- from U partition element to V partition element in case of write operations
- from V partition element to U partition element in case of read operations.

Let's call $Reads(e_i)$ a set of all incarnations read by a given execution e_i. Let's call $Writes(e_i)$ a set of all incarnations read by a given execution e_i.

$Writer(i_i) = \{e_{op} : \forall\ o_i = (e_{op}, t_{op}, i_{subj}) : t_{op} = \text{WRITE} \wedge i_i = i_{subj}\}$.

If the size of set $Writer(i_i)$ is larger than 1, it means that the observed dataset contains errors.

Definition 4.12 (Direct provenance graph). A direct provenance graph P^d of U is a directed hypergraph corresponding to the effects graph $(E,\ I,\ O)$ of 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 incarations to i.

Definition 4.13 (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 $i_2 \in I$ whenever there is a path between $Reader(i_1)$ and $Writer(i_2)$ in the execution graph of U.

Definition 4.14 (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 4.15 (Indirect provenance set). An indirect provenance set $S^i(i)$ for an incarnation $i \in I$ of U is a transitive closure of incoming adjacent neighbours of i in P^i of U.

Extensions

Finally next we will define a set of procedures, to extend an universe with additional implicit information. These universe extensions are mechanisms to infer additional information from the observed dataset. They allow to recover additional information through over-approximation.

These extensions address the “supports incremental implementation“ requirement by allowing to extract value from minimal tracing instrumentation by trading off accuracy of execution tracing and provenance tracking. The use of the extensions can be reduced (hence improving accuracy) as more and more instrumentation is implemented in the system under tracing.

Definition 4.16 (Sub-incarnation provenance extension). For a given universe U, sub-incarnation provenance extension is a universe U' extended with additional 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 of the PEDST model to avoid explicitly recording write operations on all sub-incarnations. This optimization is especially useful if the execution represents a some sort of packing operation, where an operation creates a large blob, which has a lot of individual elements embedded. For example an execution of “tar cfz” command line fits. This extension allows to maintain a provenance relationship between a writer of a parent incarnation and a reader of a sub-incarnation.

Another example is a case where an execution records a black-box transformation of an composite incarnation A without looking inside it (e.g. file copying) and writes a new incarnation B. Next another tool consumes entity B (e.g. the destination file), and extracts a fine-grained information k_B from it (e.g. extracts a single file of the copied archive). Sub-incarnation provenance extension ensures that a provenance relationship between the incarnation A and the fine-grained information k_B is maintained (e.g. a relationship between the original archive and the extracted single file).

Definition 4.17 (Sub-execution provenance extension). For a given universe U, sub-execution provenance extension is a universe U' extended with additional imaginary read operations for every pair of a execution and an 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.

Sub-execution provenance extension is an inference mechanism to capture a likely provenance relationship between an object read by a parent execution and an object written by a child execution. It is useful if instrumentation of the traced system is imperfect. It is based on the assumption that a parent process which spawned a child process is likely to have passed information to the child process which was used by the child process when writing an object.

Definition 4.18 (Inter-incarnation provenance extension). For a given universe U, inter-incarnation provenance extension is a universe U' extended with additional imaginary executions: for each adjacent sub-incarnation a execution is implied which performs a read operation the earlier incarnation and a write operation of the later incarnation if no path exists between these sub-incarnations in a direct provenance graph of U.

It is likely that whenever two consecutive incarnations of an entity are produced, they have been produced by a repeating process which produces the latter incarnation using the former as an input. This is obviously an incorrect assumption for a number of systems, but this extension is a useful tool to explore a provenance graph of a traced system where the instrumentation is imperfect.

Definition 4.19 (Message-passing provenance extension). For a given universe U, message-passing provenance extension is a universe U' extended with additional imaginary executions: for each message sent from execution e_1 to e_2 in each interaction a new 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 is based on the assumption that whenever two processes are communicating, they exchange information, which is relevant for provenance tracking. With it we assume that whenever one execution send a message to another an implicit unit of information is passed, linking two executions in their provenance graph.

4.3.2. Local model

As described in section 4.2 ”Logging data model”, data logged by components is a stream of events called F. Hence to construct an *observed universe* we need to perform a *left-fold* operation on the sequence of events starting with an initial universe:

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

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

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

Alternatively, for each event f^k in F, we have

$$U^{k+1} = \text{observe}(U^k, f^k)$$

In industry, this computation pattern is known as Event Sourcing [53].

We assume that a sequence of events F is causally ordered, which means that every event in it is placed after any event it depends on, for example:

- execution end event, for execution e, is after execution start event for execution e;
- read operation for incarnation i is after a write operation, which creates incarnation i;
- interaction begin targeting execution e is after execution start event for execution e;
- and so forth.

Note that the full list of requirements is embedded into the definition of *observe* procedure as checks. In an implementation of the model, it is the responsibility of an application developer to maintain causal ordering of produced events. A vast research body has been produced on this topic [54] and it is out of the scope for this work.

observe procedure needs to be embellished with a state, allowing it to maintain the intermediate state information, which is not representable in the global data model. Each event is fed into the *observe* function together with its production timestamp. The *observe* has the following semantics, depending on the type of event F.

Execution begins. Event f is a tuple $(id_e, e_{parent}, e_{creator}, id_p)$. Observing an event of this type does:

- 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 event timestamp as the execution start timestamp.

Execution ends. Event f has the shape of (id_e). Observing this events does:

- Checks if an execution with id_e has been stored in the intermediate state and retrieves it.
- Appends $(id_e, start, finish, e_{parent}, e_{creator}, id_p)$ to E in U, where finish is the event timestamp.

Operation performed. Event f is a tuple $(t, id_o, e_{op}, t_{op}, i_{subj}, n_{subj})$. Observing an event of this type does:

- Checks if execution e_{op} has been observed so far.
- Notes existence of entity n_{subj}.
- Appends $(t, id_o, e_{op}, t_{op}, i_{subj})$ to I in U.
- Appends $(t, id_o, e_{op}, t_{op}, i_{subj})$ to I in U.

Message sent. Event f has the shape of $(id_m, e_{init}, e_{target}, id_x, \text{Payload})$. Observing this events does:

- Checks if executions with id_{init} and id_{target} has been observed so far and retrieves it.
- Adds interaction $id_x, e_{init}, e_{target}, \emptyset$ if id_x is not yet present in X of U.
- Appends $(id_m, \text{timestamp}, e_{init}, e_{target}, \text{Payload})$ to M_x of the interaction.

Execution annotation added. Event f has the shape of $(id_e, t, id_x, \text{Payload})$. Observing this events does:

- Checks if an execution with id_e has been observed so far and retrieves it.
- Appends $(t, \text{Payload})$ to M_x of the a of the execution.

4.4. Motivating examples in PEDST model

The set of concepts described in section 4.1 and their formalization described in section 4.3 allow us to define the answer to the motivating examples from section 1.7.3 in a well-defined way.

4.4.1. Buggy deployment

Figure 4.7 contains a manually crafted record of a single hypothetical deployment from the example from section 1.7.3.

Here it can be seen how executions (orange rectangles; see section 4.1.1) of scripts and tools form an *execution graph* (see section 4.3.1), which represents their hierarchical relationships. For example from reading the graph it can be said that `docker run` has been executed by the third **ssh remote** execution (see section 1.7.3).

In this representation dotted arrows between executions represent a creation relationship (if parent is not a creator), hence we can observe that a *remote docker daemon* is a creator for *app container @ remote*. The latter is an *unowned execution* (see section 4.1.1).

Dashed arrows between executions and incarnations (teal diamonds; see section 4.1.2) represent operations (see section 4.1.3). These arrows form an *effects graph* (see section 4.3.1), where arrows from incarnations to executions represent edges from V to U – read operations – and arrows from executions to incarnations represent edges from U to V – write operations.

Provenance graphs can be computed by using all incarnations as nodes of the graph, and forming a hyperedge between each two incarnations from a set of sections each consisting of a dashed arrow from the source incarnation to a execution and a dashed arrow from the execution to the target incarnation. For example, we can see that there is a directed path in the provenance graph consisting of edges:

1. from `app sources 1` to `docker image app:1` corresponding to `build app container` execution;
2. from `docker image app:1` to `docker image app:1 @ registry` corresponding to execution `docker push app:1`.

The representation can be used to manually compute a *indirect provenance set* for an incarnation, by noting all incarnations reachable via dashed arrows traversed in reverse. A *direct provenance set* is the same, but limiting traversal depth to two arrows.

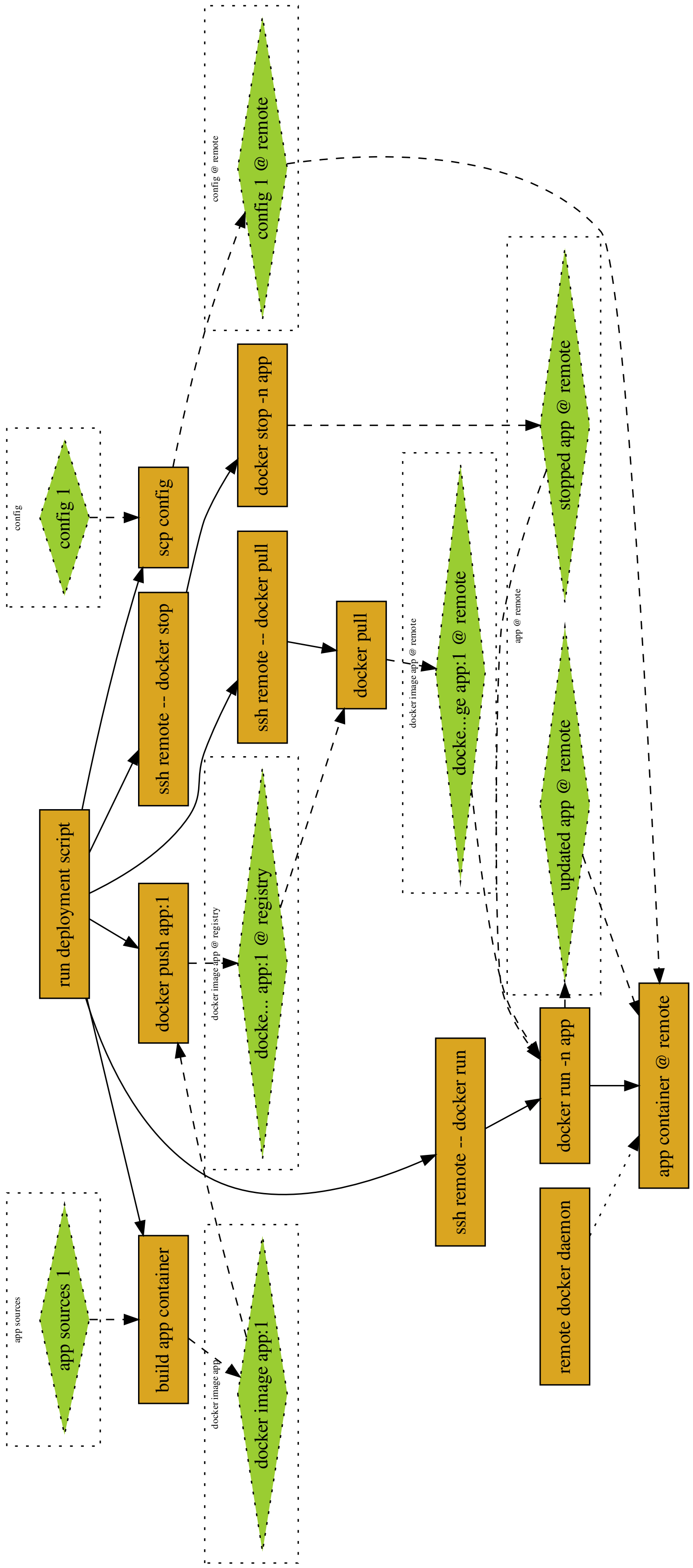


Figure 4.7: Record of a script-based deployment of a simple docker-based service described in section 1.7.3

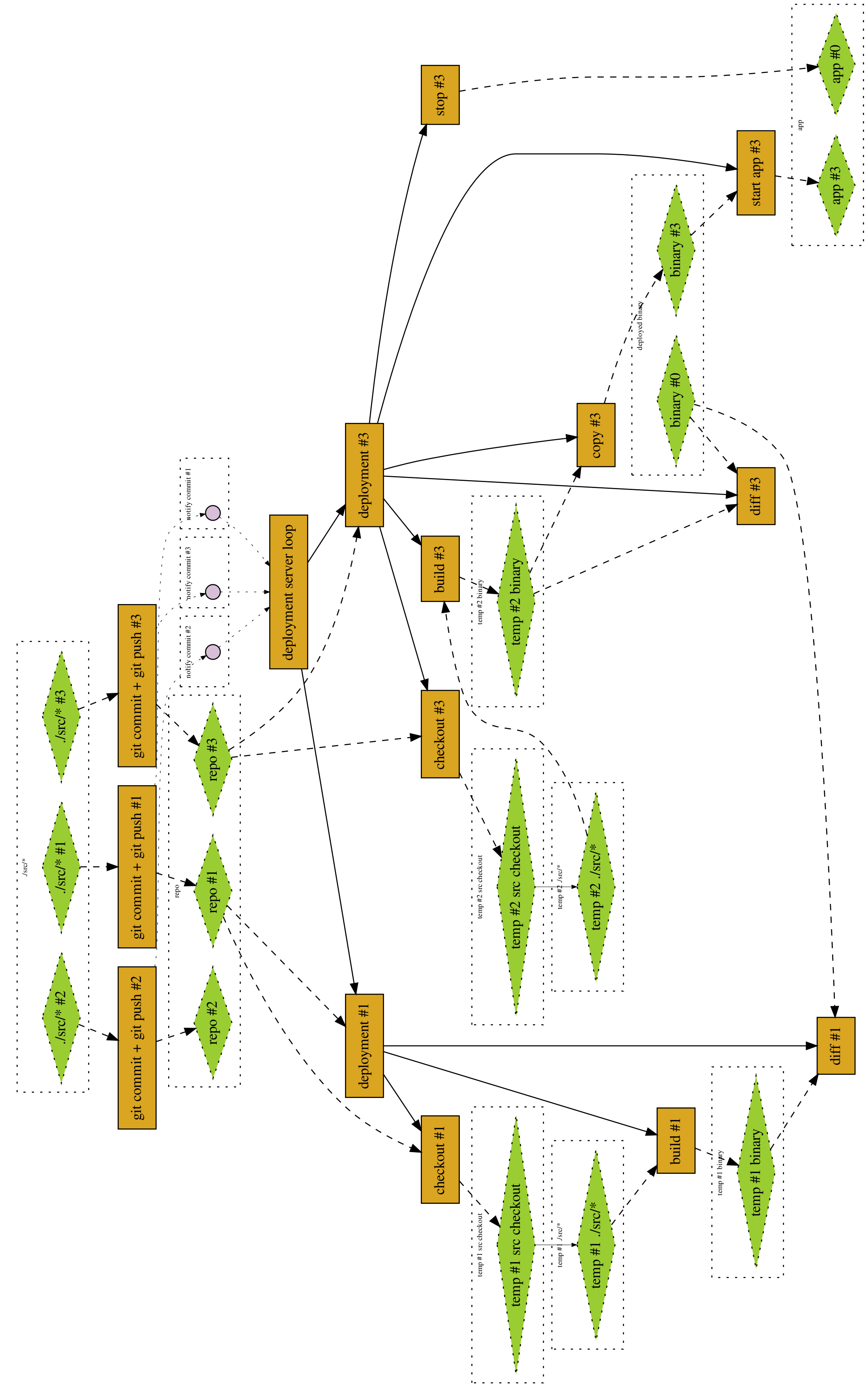


Figure 4.8: Record of a deployment with intent-based actuation based on Gitops approach described in section 1.7.3

4.4.2. Rollback of source of truth.

Figure 4.8 shows what happens in a system when three consecutive git pushes are performed. One can see that a git commit and push operation takes local source code and pushes it to repository, creating new instances of the repository. It can be seen that the `deployment server loop` performs two pushes. The first push attempt does not proceed beyond a diff operation, after doing a checkout and a build in a temporary directory. The second push proceeds further: it copies the newly built binary into the well-known location, stops the old version of the application and starts a new version of the application.

This diagram has a new shape of a light purple circle. These represent messages passed between two executions. One can see that a message has been sent from each of `git commit + git push` executions to `deployment server loop`. These are notifications sent by the Git hook to trigger a deployment. This is the only direct relation between Git operations and a deployment process (note that the individual deployment operations are owned by `deployment server loop`!). Moreover there is no provenance relationship between `app #3` and any of the source nodes `./src/*` using just the definitions of indirect provenance set (definition 4.13).

To track a relationship between `app #3` and `./src/* #3` one need to make use of one of the graph extensions (section 4.3.1). Please note that the process of building of the binary does not perform a read operation on an incarnation of the checked out repo, but instead it reads a sub-incarnation of said repo. Source files stored in the repo checkout are tracked as a sub-incarnation `temp #2 ./src/*`. Use of "sub-incarnation provenance extension" extension allows to track provenance relationship between them due to the aforementioned sub-incarnation relationship.

5. Architecture

The model proposed in chapter 4 is not sufficient on its own to solve the problem stated in section 1.7. The model has to be supported by a solution architecture which satisfies the requirements. In this section we describe such architecture, which is Provenance-Enhanced Distributed Systems Tracing architecture (see fig. 5.1).

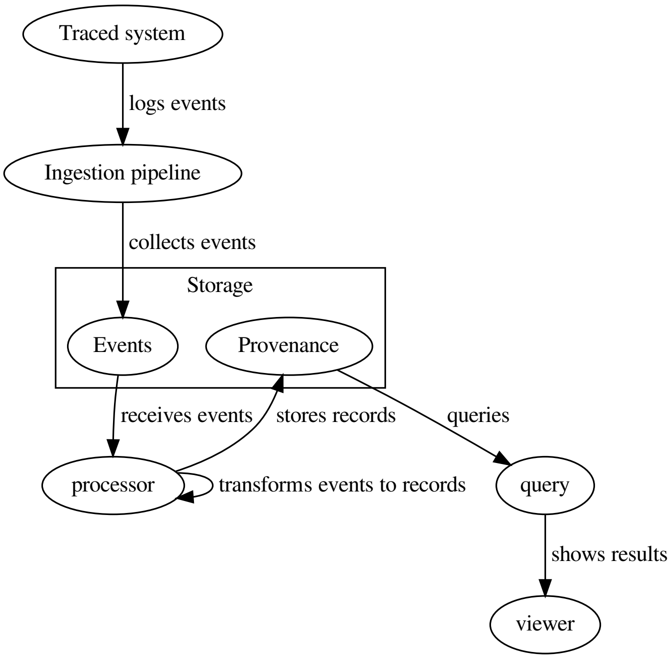


Figure 5.1: Data flow in the Provenance-Enhanced Distributed Systems Tracing architecture.

The PEDST architecture consists of:

- Global data model schema to capture the global view of what has been recorded among all systems participating in the data gathering process (see, section 4.3.1).
- Logging data model schema for structured logging to be performed by each individual component participating in the data gathering process (see, section 4.3.2).
- Storage system to store events (using logging schema) and records (using global data

schema).

- Logging protocol for event ingestion.
- Ingestion pipeline which ingests events from event producers and stores it in the storage system.
- Events processor which uses events to build up global data based on it, and stores it in the storage system.
- Set of queries defined over global data.
- Visualization tool using of the queries' results of the global data.

The model and the architecture are still not a solution for the problem at hand. The data produced by a traced system has to follow a structured logging protocol and data schema described in section 4.2 "Logging data model". Even if the protocol is followed, the PEDST model allows for a considerable freedom how data is logged, hence a set of recommendations how to perform logging from traced application is presented in section 6.3 "Patterns".

To satisfy "Supports granular fidelity implementation", "Supports incremental implementation" and "Low storage overhead" requirements (see, chapter 3), PEDST architecture assumes a white-box tracing approach. This means that tracing and logging instrumentation is actively engineered into the software whose actions are being inspected. This means that we need to provide a protocol to send events (see, section 4.2) from the traced system to a PEDST implementation deployment. Additionally, we have to provide a logging library, to simplify usage of the protocol. The white-box tracing focus puts this work in a "disclosed provenance" category [51].

As per "Limited trust" non-goal, the proposed system is designed to be used in the confines of a single stakeholder's infrastructure. Hence, we are not aiming for a tamper-resistant system and are trusting events logged by and within a traced system.

Ingestion pipeline. Events logged by a traced system are received by an ingestion pipeline (see, section 6.2.4 for details), which gathers events from all components of the system and stores them in the event storage. The main goal of the ingestion pipeline is a durable delivery of events from all components of the, usually distributed, system under tracing. Hence, it contributes to the "Full coverage" requirement. Depending on the requirements, and the scale of the traced system, the ingestion system might be as simple as a shared NFS directory for structured log files with JSON entries, a large Splunk ingestion deployment, or as complex as an ingestion pipeline, used by Google, for Dapper traces. An asynchronous ingestion pipeline satisfies the

“Asynchronous data intake” requirement and contributes to the “Event-based data production” requirement.

The distributed ingestion pipeline stores the events gathered from the traced system in an event store. The event store is essentially a queue for a processor to act upon. Presence of a distributed ingestion pipeline satisfies the “cross-host tracking“ requirement by ingesting events from all hosts participating in the traced system.

Events processor. The processor takes out an event from the events store, processes it, and deletes it. In case of a failure to process an event (e.g. if the event depends on something which is not yet recorded), it should be – after a delay – stored back in the queue. If it is impossible to process an event after a specified time, the event is discarded.

Processing of events happens according to the procedure defined in section 4.3.2 ”Local model”. Whenever processing of an event needs new records to be stored, they are added to the records store.

Events processor may be responsible to transform incoming provenance information into a more convenient or efficient representation and store it in the storage system. The representation should be suitable for efficient graph queries.

Storage. Records store stores tuples, which represent an observed universe, defined in section 4.3.1 ”Tuples”. The storage should also be used to store a query-optimized representation of the same data.

Queries. Queries over the record store allow to construct any of the graph representations of the observed universe (see section 4.3.1) and their extensions (see section 4.3.1).

In chapter 6 ”Implementation” we present a proof of concept implementation of the PEDST model and architecture.

6. Implementation

To show that the Provenance-Enhanced Distributed Systems Tracing model (see chapter 4) and architecture (see chapter 5) solves the use case this thesis is focused on (see section 1.7) they have been implemented as Tenmo framework[1]. In this chapter we present overview of Tenmo framework and discuss selected details of how it is implemented. Finally we discuss recommended usage patterns around production of Tenmo traces and how such traces can be consumed via queries.

Chapter 7 describes the practical use of Tenmo framework in a couple of software systems.

6.1. Overview

The Tenmo framework implements the PEDST model and follows the PEDST architecture. The implementation has grown organically during the implementations of instrumentation in the software (see chapter 7), hence little focus has been placed at reusability and cross-language support. At the same time the neither the model, the architecture, nor the current implementation are in any way restrictive – cross-language support with use of technologies like Protocol Buffers is possible and desired.

Tenmo consists of the following elements:

- A PostgreSQL-based implementation of data storage.
- Go and C++ logging library to enable structured logging as per the logging protocol.
- A Python ingestion pipeline which ingests individual structured logging streams.
- A Python processor, which listens to notifications coming from the PostgreSQL storage and constructs global data based on it, and stores it in a database.
- GraphViz visualization of query results of the global data.[2]

[1] The code attached to the thesis and available at https://github.com/gleber/tenmo repository.

[2] Diagrams representing Tenmo traces in this thesis are generated by this tool.

Overall our solution has similarities with the LogProv approach [46], where individual data operations are logged as tuples into an provenance events ingestion pipeline. As briefly discussed in chapter 2, LogProv uses ElasticSearch to store the tuples, and the graph representation of provenance is recovered from the tuples at query time. LogProv does not deal with failures, since a data processing pipeline abstracts individual failures away. Tenmo, on other hand, has to deal with failures, hence Tenmo is using more granular event-based logging (e.g. separates execution beginning and end events).

6.2. Selected details

In this section, selected details of the implementation will be discussed to give the reader a feel of Tenmo and how it can be used.

6.2.1. Data definitions

In the current implementation global and logging data models are not implemented as separate artifacts. They exist in 3 mirror implementations in Python, Go and PostgreSQL schema. These exactly follows the models defined in section 4.3.1 ”Tuples”, with few additional fields necessary to represent the model in a given language.

For example execution in the global model is defined in Go as:

```
type Execution struct {
        ExecutionId ExecutionId;
        ParentId ExecutionId;
        CreatorId ExecutionId;
        ProcessId ProcessId;
        Description string;
}
```

Here, identifier types are aliases of `string` type:

```
type ExecutionId string
type IncarnationId string
type ProcessId string
type EntityId string
type OperationId string
```

For example execution begin event in the logging model is defined in Go as:

```
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"`;
}
```

Here, all of the fields are JSON-serializable to allow this structure be stored as a `payload jsonb` field of the Tenmo storage system (see section 6.2.3 for details).

6.2.2. Logging library

Tenmo provides a client-side library, written in Go, and it's typical execution tracing looks like:

```
parentExecutionId := /* Retrieve parent identifier, often from a Context object. */;
// Register an execution begin, so it is sent to the Tenmo ingestion pipeline
executionId, ender := tenmo.ExecutionRegistration(tenmo.Execution{
    tenmo.ExecIdRand("worker-step"),
    parentExecutionId,
    tenmo.None,
    tenmo.None,
    "worker step"})
// Make sure that the execution is ended at the end of this scope.
defer ender()
// Perform own work.
PerformOwnWork();
// Pass own execution ID to a child goroutine.
go PerformChildWork(executionId);
```

Here, an execution is registered using an identifier of a parent execution. In Go language, identifier of the current execution is often stored in `Context` object, which is a typical way to propagate context information around projects. It so happens that Kubernetes project has not yet adopted this pattern, hence Tenmo logging library does not depend on it. Own execution identifier needs to be passed down to any subroutines which are performing any work to be traced or any operations to be tracked.

Go language pattern of using `defer` for finalizers is suitable here for any executions whose logical lifetime corresponds to a code scope.[3] In case of long-running operations – as in the intent-based actuation use case – current execution identifier should be stored in a long-living object.

Next, below, presented is an example of how the read and write operations are recorded. It is

[3] For example a C++ implementation of the library would make use of the RAII pattern, to track the lifetime of executions for most cases.

worth noting that the incarnation identifier is meant to be recoverable from the object managed by the system. Here, a simple implementation would combine the object's primary key in a database and the value of its last-modified-timestamp column.

```
    var executionId tenmo.ExecutionId = /* Fetch execution identifier, usually from Context */;
    MyResource myResource = fetchResource(/* A given resource identifier */);
    entityId := myResource.GetTenmoEntityId();
    currentResourceIncarnationId := myResource.GetTenmoIncarnationId();
    tenmo.OperationRegistration(tenmo.Operation{
                executionId,
                tenmo.OpRead,
                entityId,
                currentResourceIncarnationId,
                "resource Foo", "resource Foo before update"})
    myResource.UpdateAndSave();
    updatedResourceIncarnationId := myResource.GetTenmoIncarnationId();
        tenmo.OperationRegistration(tenmo.Operation{
                executionId,
                tenmo.OpWrite,
                entityId,
                updatedResourceIncarnationId,
                "resource Foo", "resource Foo after update"})
```

Here, a process fetches and updates a resource. The process has an existing execution identified by `executionId` associated with it. The execution records its two operations of reading and writing of the resource.

Below is an example of how annotations can be attached to executions:

```
annotationPayload := /* a JSON-serializable struct stored in an annotation */;
var executionId tenmo.ExecutionId = /* Fetch execution identifier, usually from Context */;
annotationJson, _ := json.Marshal(annotationPayload)
tenmo.AnnotateExecution(tenmo.Annotation{executionId, annotationJson})
```

Specifically, in current implementation annotations are essentially JSON objects attached to an execution at a specific timestamp. By default the system timestamp is taken.

Registration of other objects in the PEDST model using the Tenmo Go logging library is analogous.

In the current implementation of the Tenmo Go logging library, events are directly inserted to the `events` table of the Tenmo PostgreSQL-based storage.

6.2.3. Storage

The Tenmo implements a basic non-distributed storage based on a single PostgreSQL database. The database consists of two main sections:

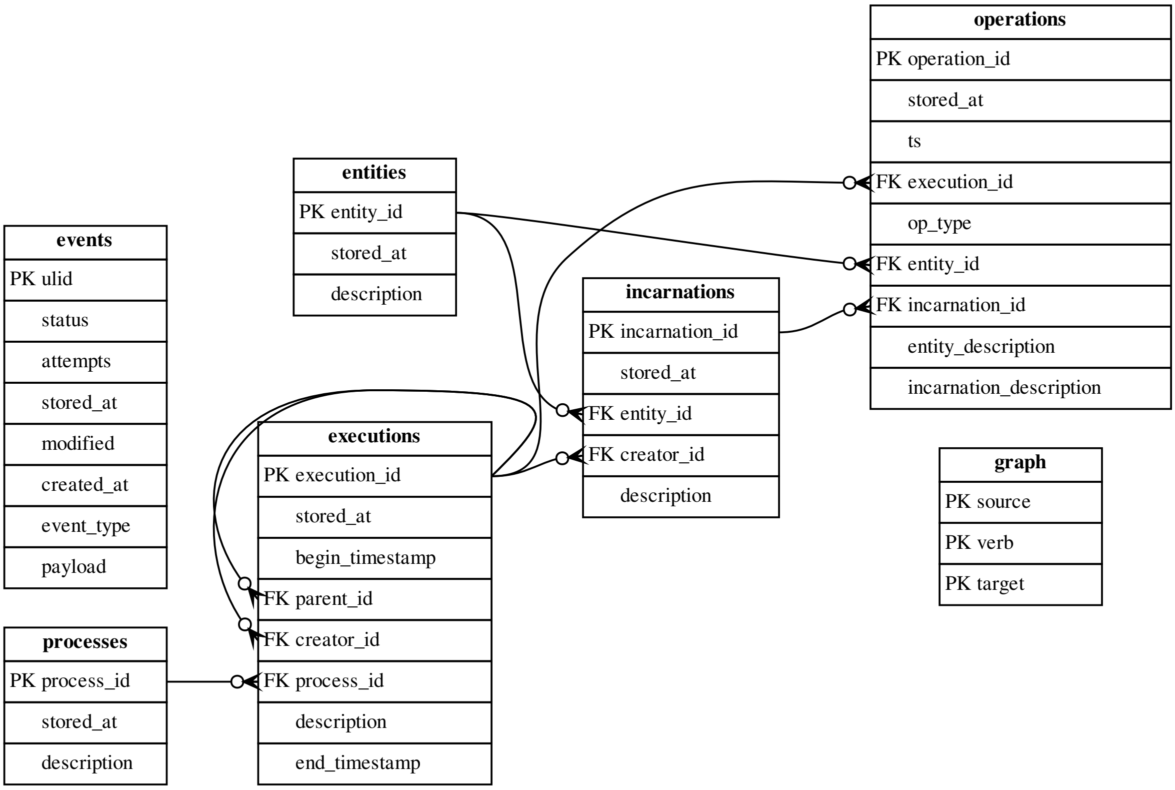


Figure 6.1: Tenmo storage diagram in PostgreSQL.

- an event stream table, and
- a set of tables capturing the observed universe.

Both types of tables follow the model described in section 4.1 very accurately as depicted in fig. 6.1. For example, the definition of `operations` table is the following:

```
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), -- 'w' or 'r'
  entity_id text not null references entities (entity_id),
  incarnation_id text not null references incarnations (incarnation_id),
  entity_description text default '',
  incarnation_description text default ''
);
```

The event stream table is populated by the ingestion pipeline and appended to this table as is. The set of tables representing the observed universe is populated by the ingestion pipeline based on the event stream table. Events in the stream are marked as processed or not. Processed events are regularly garbage collected. Schema of the tables closely follows the global data model.

The `graph` table is discussed in section 6.2.6 ”Tenmo graph”.

6.2.4. Ingestion pipeline

The ingestion pipeline receives data from a instrumentation embedded in the traced system, and delivers it to Tenmo's PostgreSQL database in a form of events stored in the `events` table. The Tenmo proof-of-concept implementation uses two ingestion mechanisms:

- Tenmo Go logging client library connects directly to the Tenmo PostgreSQL instance and appends them to the `events` table.
- Nix integration (see section 7.1.1 uses a log tailing Python script `nix-jsonlog-to-tenmo.py`, which iterates over JSON-structured log lines produced by Nix binaries provided with `--log-format internal-json` flag, transforms them to Tenmo event schema and inserts them directly to the PostgreSQL instance.

The fact that Tenmo Go logging client library inserts events directly into the target events storage system is an implementation detail. In a production-grade implementation of Tenmo it would be powered by a distributed high-performance log ingestion solution like Syslog-ng, Apache Kafka, FluentD, or Loginson [55].

6.2.5. Events processor

Events processor is responsible for implementing an `observe` function, which would consume events from `events` storage table and populate all other tables in the storage.

Whenever a new event is appended to `events` table, a NOTIFY event will be sent via a trigger. Tenmo events processor subscribes to the appropriate channel, hence gets notified about a new unprocessed event. The processing logic is as follows (pseudo-code):

```
events = fetch_all_unprocessed_events(order_by=ingestion_timestamp)
for event in events:
    with transaction:
        mark_claimed(event)
    with transaction:
        U = load_universe()
        U’, success = observe(event, U)
        if success == true:
            mark_processed(event)
            store_universe(U)
        else:
            mark_unclaimed(event)
```

To ensure that events are not double-processes, event processing starts with claiming the event in an atomic database transaction, followed by the event processing logic proper.

If events are not inserted in their causal order (as per requirements defined in section 4.3.2), some events might be left unprocessed until a future processing cycle happens. This could lead to a situation when a backlog of unprocessed events grows indefinitely, e.g. in case of lost events. In our use case – given the distributed nature of the traced systems – lost events are inevitable due to failures in the network, infrastructure, or system itself. The processor will increase a per-event counter every time it gets claimed. When an event crosses an attempts count threshold, the event is no longer being considered for processing. A garbage collection mechanism for such messages is yet to be implemented.

In our implementation of the observe mechanism, to support granular fidelity of provenance information, we always over-approximate the provenance according to the closure definitions in section 4.3.1 "Tuples" and section 4.3.1 "Extensions". Each resulting implicit relationship stored in the database with an appropriate mark. This allows us to query the data at a selected provenance fidelity level, by simply varying a filter applied to a given SQL query (see section 6.2.7 for more details). The resulting graph is described below.

6.2.6. Tenmo graph

Whenever a new event is processed, all relationships between PEDST concepts are materialized into a triple-structured table `graph`. The table stores three fields `source`, `verb` and `target`. Table 6.1 lists the relationships which are being materialized in the `graph` table.

Here, all relationships captured in the model (see fig. 6.1) are denormalized into a triples form. This allows us to treat all relationships in the model as a composite graph[4], and traverse it as such. Each graph defined in the section 4.3.1 and each extension defined in section 4.3.1 are contained in this structure. This composite structure allows to implement queries as graph queries over a triples data, while selecting which relationships to traverse on per query basis using filters of verbs.

It should be noted that all of the relationships are represented bi-directionally, allowing us to traverse the graph in both directions of a relationship depending on the need.

[4] Current proof-of-concept implementation assumes that identifiers of all objects referenced in the `graph` table are globally unique.

Moreover this data representation allows to make full use of extensive research in knowledge retrieval in Resource Description Framework (RDF) [56] and Semantic Web [57] to evolve usability of data gathered using Tenmo. Tenmo's triples store using PostgreSQL is a naive implementation of this concept. Future research is necessary to scale it to large use cases.

Queries for Tenmo implementation over this graph representation are described in section 6.2.7 ”Queries”.

6.2.7. Queries

Given the triple form chosen for storing the PEDST graph for querying, answering queries – with or without graph extensions – is a matter of crawling the composite graph, and analysing found paths.

Current proof-of-concept Tenmo implementation uses PostgreSQL's recursive Common Table Expressions (CTE) as a mechanism to crawl a graph. The following building blocks are used to answer queries in PEDST model.

```
 1 CREATE OR REPLACE FUNCTION get_all_paths_from(start text)
 2 RETURNS TABLE(depth integer, verbs text[], path text[]) AS $$
 3 BEGIN
 4 RETURN QUERY
 5 WITH RECURSIVE search_step(id, link, verb, depth, route, verbs, cycle) AS (
 6   SELECT r.source, r.target, r.verb, 1,
 7          ARRAY[r.source],
 8          ARRAY[r.verb]::text[],
 9          false
10   FROM graph r where r.source=start
11   UNION ALL
12   SELECT r.source, r.target, r.verb, sp.depth+1,
13          sp.route || r.source,
14          sp.verbs || r.verb,
15          r.source = ANY(route)
16   FROM graph r, search_step sp
17   WHERE r.source = sp.link AND NOT cycle
18 )
19 SELECT sp.depth, array_append(sp.verbs, '<end>') AS verbs, sp.route || sp.link AS path
20 FROM search_step AS sp
21 WHERE NOT cycle
22 ORDER BY depth ASC;
```

This implementation follows PostgreSQL manual on CTE [58] to implement a search for all paths in the graph. The search starts with the node `start`, given as an argument to the function (line 1). The non-recursive term (lines 6-10) of the recursive CTE identifies all rows which represent outgoing edges from the selected object identified with `start`. The recursive term of the CTE (lines 12-17) then traverses the graph, identifies cycles (line 15) and records traversal paths (line 13) and verbs encountered on each path (line 14). The final `SELECT` statement (lines 19-22) filters out irrelevant paths – e.g. paths containing cycles – and finalizes the resulting multiset. The result contains a single path per row, with path lengths information in `depth` column, all traversed objects on the path in `path` column and all verbs traversed verbs on the path in `verbs` column.

All other queries are implemented using the same principle. They are described by examples below.

“Buggy deployment” queries

The use case from section 1.7.3 ”Buggy deployment” as modelled in section 4.4.1 is used below for example.

For example, there are 3900 paths coming from the execution representing the final container running the guestbook application:

```
tenmo=> select count(*) from get_all_paths_from('remote-app-container');
 count
-------
  3900
(1 row)
```

Selecting a couple of shortest paths about the execution gives us immediate useful information:

```
tenmo=> select * from get_all_paths_from('remote-app-container') limit 6;
 depth |           verbs            |                          path
-------+----------------------------+-------------------------------------------------------
     1 | {child_of,<end>}           | {remote-app-container,ssh-remote-docker-run-1}
     1 | {created_by,<end>}         | {remote-app-container,remote-docker-daemon}
     1 | {reads,<end>}              | {remote-app-container,remote-app-2}
     1 | {reads,<end>}              | {remote-app-container,remote-config-1}
     2 | {reads,written_by,<end>}   | {remote-app-container,remote-config-1,scp1}
     2 | {reads,instance_of,<end>}  | {remote-app-container,remote-config-1,remote-config}
(6 rows)
```

Here, we can see that the application has been launched due to a `ssh remote - docker run` command and the docker container running the application has been created by Docker daemon running on the `remote` host. Next we can observe that the container reads the second incarnation of the app container image on host `remote` (from `reads` of `remote-app-2`). We can see that it also reads a config stored on host `remote` (from `reads` of `remote-config-1`) and that this config is an instance of an entity `remote-config`.

Let us try doing activity tracing and identify all ancestors of the execution representing the application running in the container after the deployment:

```
tenmo=> select * from trace('remote-app-container');
 depth |           obj
-------+------------------------
```

```
     1 | ssh-remote-docker-run-1
     2 | ssh-remote-3
     3 | deploy-script.sh-run-1
(3 rows)
```

Here we can see that four ancestors were correctly identified, and we know that the current instance of the application has been started by a `docker run` command, executed by `ssh remote`, finally executed by the deployment script.

The `trace` function is implemented in the following way:

```
CREATE OR REPLACE FUNCTION trace(start text)
RETURNS TABLE(depth integer, obj text) AS $$
BEGIN
RETURN QUERY
select * from get_closure_from_by_verbs(start, ARRAY['child_of']::text[]) as t;
END;
$$ LANGUAGE plpgsql;
```

Here the `get_closure_from_by_verbs` function is used with `child_of` verbs filter. It means that this function will return a closure of all objects in the graph which are reachable by traversing only the edges marked with the given verb. Please refer to "Appendix A" for implementation of the `get_closure_from_by_verbs` and other query SQL functions.

If one would like to traverse the graph to find all paths denoting only reads and writes, one could start exploring with:

```
tenmo=> select count(*) from get_all_paths_from_by_verbs('remote-app-container', '{reads,written_by}');
 count
-------
    14
(1 row)

tenmo=> select max(depth) from get_all_paths_from_by_verbs('remote-app-container', '{reads,written_by}');
 max
-----
   9
(1 row)
```

This tells us that there are 14 distinct paths through the theorem 4.11 and the longest path through the graph is of length 9.

Let us list just the immediate `reads` edges from the `remote-app-container`:

```
tenmo=> select * from get_all_paths_from_by_verbs('remote-app-container', '{reads}') where depth <= 1;
 depth |     verbs     |                   path
-------+---------------+---------------------------------------
     1 | {reads,<end>} | {remote-app-container,remote-app-2}
     1 | {reads,<end>} | {remote-app-container,remote-config-1}
(2 rows)
```

Here we can see that this execution did read two incarnations. Let us see what are the paths of length up to two only through `written_by` – “which execution wrote a given incarnation?” – and `reads` – which incarnations did a given execution read?” – edges starting from the remote config incarnation node.

```
tenmo=> select * from get_all_paths_from_by_verbs('remote-app-container', '{reads,written_by}')
                                                        where depth <= 2;
 depth |          verbs           |                            path
-------+--------------------------+-------------------------------------------------------------
     1 | {reads,<end>}            | {remote-app-container,remote-app-2}
     1 | {reads,<end>}            | {remote-app-container,remote-config-1}
     2 | {reads,written_by,<end>} | {remote-app-container,remote-config-1,scp1}
     2 | {reads,written_by,<end>} | {remote-app-container,remote-app-2,ssh-remote-docker-run-1}
(4 rows)
```

This gives us two paths, one to an execution of `scp` tool and one to a local config incarnation.

From the above one can deduce that crawling the graph over `written_by` and `reads` allows to identify processes and incarnations which were involved directly or indirectly in the provenance of a given object. Such incarnations form a provenance set. A full closure gives us an indirect provenance set, and restricting ourselves only to paths of lenth up to 2 gives us direct provenance set. Tenmo provides two helper function to compute these sets:

```
tenmo=> select * from provenance_set('remote-app-container');
 depth |       obj
-------+-----------------
     1 | remote-app-2
     1 | remote-config-1
(2 rows)

tenmo=> select * from provenance_set_indirect('remote-app-container');
 depth |             obj
-------+-----------------------------
     1 | remote-app-2
     1 | remote-config-1
     3 | config-1
     3 | remote-app-1
     3 | remote-docker-image-app-1
     5 | registry-docker-image-app-1
     7 | docker-image-app-1
     9 | cwd-1
(8 rows)
```

This example does not sport intent-based actuation, hence does not show how Tenmo can be used to query across the actuation barrier. The following examples focus on it.

“Rollback of source of truth” queries

The following examples will use scenario from section 1.7.3 ”Rollback of source of truth” as modelled in section 4.4.2 is used below for example.

As discussed in section 4.4.2, given the intent-based nature of the deployment service, a traditional tracing is not capable of associating git operations with the application being started:

```
tenmo=> select * from trace('start-3');
 depth |        obj
-------+-------------------
     1 | deployment-3
     2 | deployment-server
(2 rows)
```

Use of the indirect provenance closure definition – it traverses verbs (`reads`, `written_by` – does not reveal this connection:

```
tenmo=> select * from provenance_set_indirect('app-3');
 depth |    obj
-------+-----------
     2 | bin-3
     4 | tmp-bin-3
     6 | tmp-src-3
(3 rows)
```

It is necessary to extend types of edges traversed in Tenmo graph to reveal the relationship. There are multiple verbs which give us the desired result.

Extending the graph traversal with verb `child_of` to include execution child-to-parent relationships into traversable edges gives us the path:

```
tenmo=> select unnest(path) as obj, unnest(verbs) as verb from (select * from
↪  get_all_paths_from_by_verbs('app-3', ARRAY['reads', 'written_by', 'child_of']) where 'src-3' =
↪  ANY(path) limit 1) t;
          obj          |    verb
-----------------------+------------
 app-3                 | written_by
 start-3               | child_of
 deployment-3          | reads
 repo-3                | written_by
 git-commit-and-push-3 | reads
 src-3                 | <end>
(6 rows)
```

This is an example of usage of “sub-execution provenance extension”.

The desired result can be also achieved by inclusion of traversal of sub-incarnation relationships:

```
tenmo=> select unnest(path) as obj, unnest(verbs) as verb from (select * from
↪  get_all_paths_from_by_verbs('app-3', ARRAY['reads','written_by', 'part_of']) where 'src-3' =
↪  ANY(path) limit 1) t;
          obj          |    verb
-----------------------+------------
 app-3                 | written_by
 start-3               | reads
 bin-3                 | written_by
 copy-3                | reads
 tmp-bin-3             | written_by
 build-3               | reads
 tmp-src-3             | part_of
 tmp-store-3           | written_by
 checkout-3            | reads
 repo-3                | written_by
 git-commit-and-push-3 | reads
 src-3                 | <end>
(12 rows)
```

This is an example of usage of “sub-incarnation provenance extension”.

Use of a messaging information along with the activity tracing information in the Tenmo graph gives the result as well:

```
tenmo=> select unnest(path) as obj, unnest(verbs) as verb from (select * from
↪  get_all_paths_from_by_verbs('app-3', ARRAY['reads','written_by', 'child_of', 'received_from']) where
↪  'src-3' = ANY(path) AND 'received_from' = ANY(verbs) limit 1) t;
          obj          |     verb
-----------------------+---------------
 app-3                 | written_by
 start-3               | child_of
 deployment-3          | child_of
 deployment-server     | received_from
 git-commit-and-push-3 | reads
 src-3                 | <end>
(6 rows)
```

This makes use of the “sub-execution provenance extension” extension.

Using these techniques an engineer can easily identify that the bug could have been deployed based on `src-2` or `src-3`, which would have not been possible with an OpenTracing model. Presented mechanisms allow to track this relationship in a scalable way and accurate way.

Summary

These examples show that Tenmo can answer useful questions about a system under trace using a PEDST model, its graphs and graph extensions. They show that it covers both activity tracing and provenance tracking.

A full list of queries implemented over the Tenmo graph is attached in the appendix A.

6.3. Patterns

PEDST is a rather flexible model, which can be used for a large variety of applications, fidelity levels and use cases. In this section we will discuss what is the correct usage of the Tenmo framework, which aspects of the model are optional and how to achieve gradual integration. Also, as specified in section 4.2 ”Logging data model”, a crucial requirement of the logging protocol is ability to recover, extract, reconstruct, or pass along execution, incarnation, and interaction identifiers. This section provides a couple of examples here how typical interactions and control flows in control plane systems can be modeled in PEDST.

6.3.1. Execution trees

Distributed systems tracing is the foundation of our model. As seen in section 4.1.3, execution tracing together with operation tracking provide a mechanism to track provenance of incarnations affected by executions in an execution tree. Below we describe two common patterns used to track

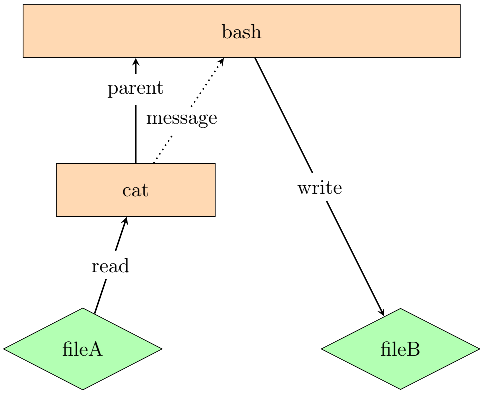
RPC flows Recording execution trees is to be done the same way it is done in the OpenTracing model, with the following differences:

- there is no global trace identifier, hence
- the parent needs to pass its identifier to any child it creates, so that
- each execution is able to record its parent identifier, unless it is the root execution
- each logical process is responsible for recording its own execution, interaction and messages

Execution tracing in the Tenmo implementation does not differ much from an OpenTracing-based instrumentation. To correctly implement execution tracing, in an RPC handler, it should use the bracket pattern [59] (or in C++ the RAII pattern [60]) to record a units of work, performed in the handler. As an input for the recording, a parent execution identifier should be retrieved from the incoming RPC metadata. Whenever an RPC is sent, it’s own execution identifier should be included in the outgoing RPC metadata. The same instrumentation points as originally proposed in the Dapper tech report are applicable.

Independently of the presented logical work units relationship tracing, RPC interactions between executions should be tracked as incarnations and messages (see section 4.1.5).

Unix process flows Execution tracing for Unix processes (see fig. 6.2) is analogous to the RPC handler use case. The same information needs to be passed between parent and child processes, for example, in environmental variables or command line arguments. Analogous recordings should be performed using a bracket pattern.



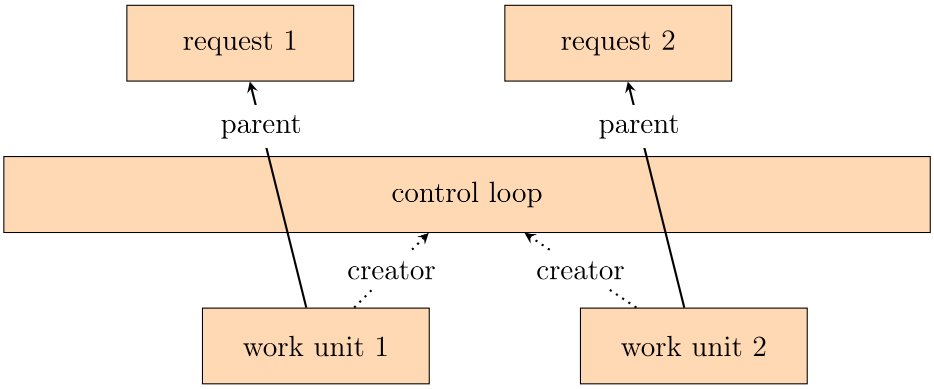


Figure 6.3: Representation of a typical control loop in in PEDST model.

Data transfer between processes using file descriptors – e.g. stdin, stdout or named pipes – can be represented as interactions and messages. This maintains provenance relationship between “fileA“ and “fileB“ as per “message-passing provenance extension” (see section 4.3.1 ”Extensions” for details).

Control loops Most servers include continuously-running control loops in them, be it a socket listening loop or a work queue processing loop. These loops should, usually, be recorded as two levels of executions: one long-running parent execution and a number of short-lived child executions dispatching incoming events (see fig. 6.3). Please note that the dispatch executions are usually creators of the event handler executions, but not their parents.

6.3.2. Intent-based actuation

As seen in section 4.1.3, execution tracing together with operation tracking provide a mechanism to track provenance of incarnations affected by executions in an execution tree.

Intent-based actuation usually consists of two levels of continuously running control loops:

- user request logic, which modified a mutable object which represents intent;
- actuation loop, which gets unsuspended whenever intent changes to update a resource under management to match the intent (i.e. actuation).

Each of these loops is handled as such, and they are interfacing with each other via write and read operations, of the intent entity. This approach allows us to handle both synchronous and asynchronous triggering of actuation.

6.3.3. Intent-based actuation with difference checks

Some automation systems with intent-based actuation will have an additional mechanism of performing periodic difference checks between desired intent and current state of the world to ensure that they match. In this case the actuation loop can be triggered not only by incoming user requests, but also by the detected difference.

This means that a difference check should also be recorded as a child execution of the intent actuation loop execution, which would perform a read operation on the underlying managed object.

6.3.4. Intent-based actuation with intent and status split

In case of a system which has an explicit tracking of status of resources under intent-based actuation (like in Kuberenetes [15]), additional entity for status needs to be tracked. Status represents the current state of the managed resource, and it is used both for returning it through an API and for internal pre-actuation checks. For example Kubernetes maintains a generation number for Deployments as part of status and uses it as part of completeness checks. The explicit split is useful for API consumers, since they can check if the generation number observed by a control plane service corresponds to the desired state of that object.

In the presented model, intent and status of the object should be tracked separately as shown on fig. 6.4. This means that API handler execution would perform a write operation on an intent entity. An execution recording a single actuation cycle would typically perform a read of the current intent incarnation, a read of the current status incarnation, and a write on the intent entity.

6.3.5. Process definitions as inputs

When an execution is recorded, the definition of the procedure, the execution represents, is also a source of provenance information. For example let's take a build process done by `make` tool using the following rules stored in file `Makefile`:

```
a.out:
    gcc a.cc -o a.out
```

If we want to maintain highest fidelity level, up until internal logic of ‘make‘ tool, the following provenance entries will be recorded:

- An execution which represents “make” Unix process lifetime with the following operations recorded

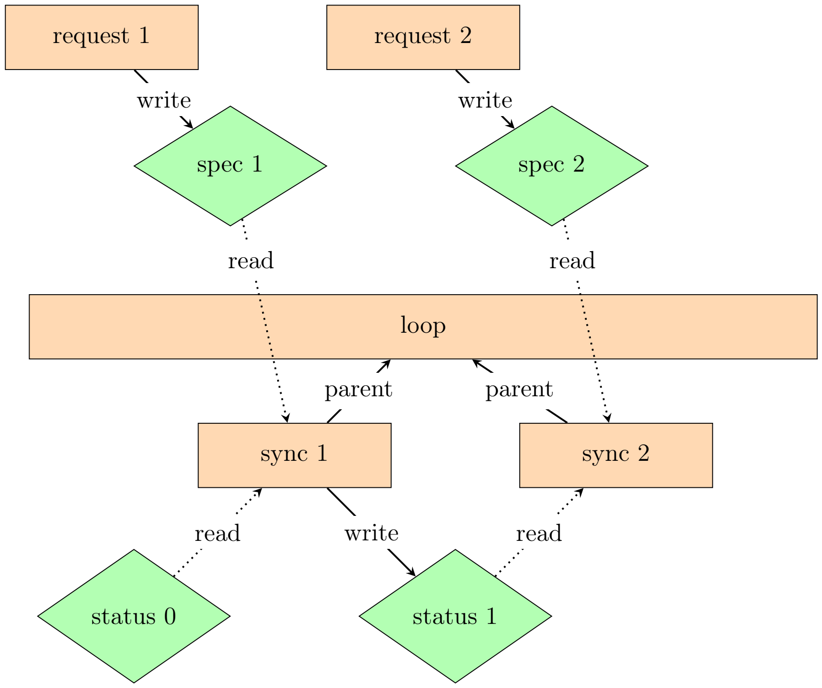


Figure 6.4: Representation of an intent-based controller with explicit split in PEDST model.

  - Read of file “Makefile” to load the build graph
- An execution representing the build process of “a.out” with the following operations recorded
  - Read of file “a.out” to check if it up to date
- An execution representing `gcc a.cc -o a.out` Unix process lifetime with the following operations recorded:
  - Read of binary “gcc” file and all of its dynamic libraries
  - Read of file “a.cc”
  - Write of file “a.out”

This will cause a provenance set for the latest incarnation of the entity “a.out” to be { ‘a.cc‘, ‘gcc‘, ‘libc‘, . . . , ‘Makefile‘} . This is an accurate representation of a provenance relationship between objects in a system, but it might not be practically useful for a user of the system. The fact that the build process has “gcc” and “libc” as inputs is, in most cases, not interesting for the user of the “make“ tool, when debugging a build process. Usefulness of “Makefile” presence in the provenance set is also situational, if a user is debugging a specific build attempt it is not relevant, but it would be relevant if a user is debugging an evolution of the build process over time.

Both Unix binaries and build definition files – from the perspective of provenance tracking – are inputs for the build process and they represent records of process definition of an execution.

At the same time most of the time in practice these inputs are not relevant for day-to-day development and operations of HCPS. Including these provenance relations will make data much noisier, since event a typical simple binary has multiple dynamically linked libraries.[5]

Tenmo framework is not prescriptive here and leaves this up to the integrator to decide if process definitions should be part of provenance tracking in their system for their use case.

6.3.6. Sub-incarnations

It is typical for some processes to produce large bundles of objects, as an output of their processing. Few examples are a build target, which is an archive in any build system and a fully-evaluated target state of a cluster in any declarative cluster deployment tool. The Tenmo protocol does not require the writer of the parent incarnation to also record creation of all of the sub-incarnations, since this approach does not scale well for large build targets. Instead, the sub-incarnation provenance propagation mechanism will ensure that provenance is properly tracked between a parent incarnation and its sub-incarnations.

Let's take the following 'Makefile' rule as an example:

```
archive.zip: foo.txt baz.bin
    zip $@ $^
```

In this case, an execution representing the execution of 'archive.zip' build rule in directory '/opt/foo' will have two file inputs '/opt/foo/foo.txt' and '/opt/foo/baz.bin' and a single output of '/opt/foo/archive.zip' file. As we know, the resulting archive will contain copies of the input files, which we could record as entities under names '/opt/foo/archive.zip:foo.txt' and '/opt/foo/archive.zip:bar.bin'. A hypothetical Tenmo instrumentation of the 'zip' executable should, in this case, record three incarnations:

- '/opt/foo/archive.zip' file incarnation
- '/opt/foo/archive.zip:foo.txt' sub-incarnation of the archive incarnation
- '/opt/foo/archive.zip:bar.bin' sub-incarnation of the archive incarnation

Accordingly whenever another process reads a file, out of the archive, they should record a fact of reading of a sub-incarnation representing an archived file they read.

[5] On author's machine "cat" has 7, "gcc" and "make" have 4 each.

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 6.1: Tenmo graph relationships.

7. Experimental results

In this chapter we analyze how Tenmo framework applies to modern multi-layered and hierarchical control plane systems of large-scale service providers (e.g. cloud providers).

7.1. Applications

Tenmo framework has been applied to the following software stack, representative of typical control plane systems. We want to ensure that the selected stack covers the problem domain well, hence we applied the following requirements towards the stack:

- It has to control deployments of a service in a cloud environment.
- It has to contain imperative-based processing of user requests.
- It has to contain at least 2 distinct intent-based systems in the control flow of a typical request.
- It has to be linguistically heterogeneous, i.e. elements of the stack has to be implemented in a diverse set of programming languages.

The selected stack is the following:

- Nix is a powerful package manager for Linux and other Unix systems that makes package management reliable and reproducible. It provides atomic upgrades and rollbacks, side-by-side installation of multiple versions of a package, multi-user package management and easy setup of build environments.

- KubeNix [26] is a Kubernetes resource builder, that uses NixOS module system [61, sec. 5] for definition of Kubernetes resources and Nix build system for building Kubernetes resources.

- Kubectl is a command line tool to interact with a Kuberentes cluster.

- Kuberentes is an open-source system for automating deployment, scaling, and management of containerized applications.

We first describe the application of Tenmo to the selected software[1], and next we present results of a set of experiments, to show how Tenmo performs in practical scenarios.

7.1.1. Build systems – Nix

We have integrated Tenmo into Nix to show that Tenmo is capable of dealing with build systems, which are a prime example of intent-based actuation systems. As any other declarative build system, it actually is layered on top of a set of imperative building blocks (e.g. executing a compiler or a shell script).

Nix build process (local non-daemon builds only) has been extended to include more comprehensive execution tracing and provenance tracking information as part of the existing JSON logging protocol. Nix language evaluation can be also provenance-tracked, to additionally track provenance of derivation definitions as evaluated from Nix source files. During evaluation instantiation of `.drv` files is tracked as writes. Process of evaluation of individual `.nix` file is recorded as executions. Each execution of an individual build process for a derivation records store paths which were consumed and produced (at store path granularity-level).

Nix implementation provides a concept of `Activity` and `PushActivity` [62]. This allows tracking execution of processes inside of the Nix code base. We made use of this mechanism, since it maps well onto the Tenmo Execution and parent-child executions relationship. Nix's `cpp`Activity.result mechanism was extended and used to register consumed and produced files by executions. Current proof-of-concept Tenmo integration into Nix captures only[2]:

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 artifical 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 Nix store paths (e.g. for a derivation builder scripts), and writes of Nix store paths for their outputs

This allows us to both trace execution structure of Nix build process[3] and track provenance of Nix store paths.

[1] Source code for these integrations is attached to the thesis and is available at https://github.com/gleber/tenmo-thesis.

[2] It also captures other existing Nix `Activity`, but they are not interesting here

[3] The visualized graph clearly shows that Nix is a suspending-type of build system [17].

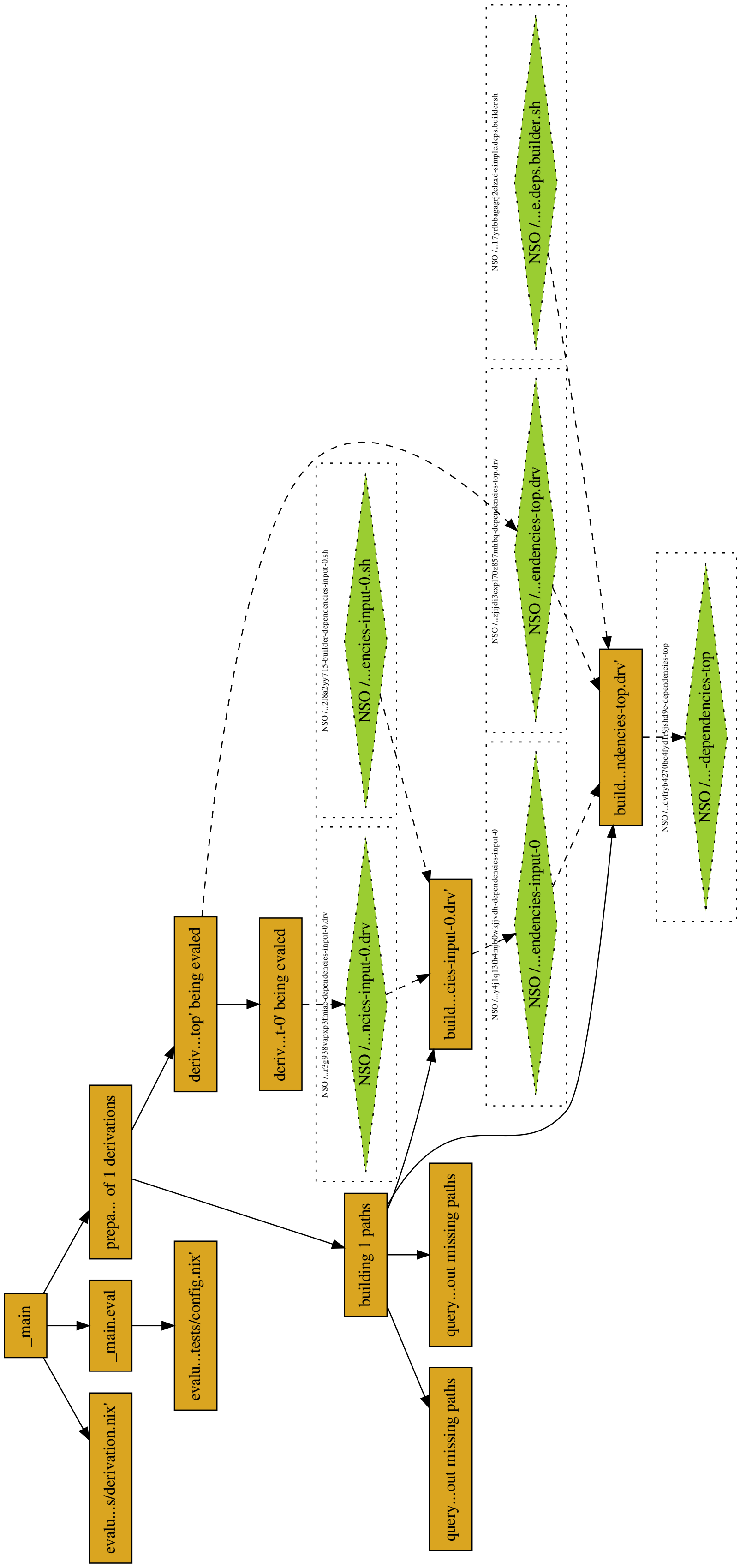

_main
evalu...s/derivation.nix'
_main.eval
prepa... of 1 derivations
evalu...tests/config.nix'
deriv...top' being evaled
deriv...t-0' being evaled
building 1 paths
query...out missing paths
query...out missing paths
NSO /...r3g938vapxp3fmiac-dependencies-input-0.drv
NSO /...ncies-input-0.drv
NSO /...218a2yy715-builder-dependencies-input-0.sh
NSO /...encies-input-0.sh
build...cies-input-0.drv'
NSO /...y4j1q13fh4mjb0wkjjvdh-dependencies-input-0
NSO /...endencies-input-0
NSO /...zijdi3cxpl70z857mhbq-dependencies-top.drv
NSO /...endencies-top.drv
NSO /...17yrlbbagagrj2clzxd-simple.deps.builder.sh
NSO /...e.deps.builder.sh
build...ndencies-top.drv'
NSO /...dvfryb4270bc4fyd1r9jshd9c-dependencies-top
NSO /...-dependencies-top

Figure 7.1: Record of a Nix build of two derivations `top` and `input0`.

Figure 7.1) represents a simple Tenmo record of build execution using a pristine Nix store without any external dependencies (not event `nixpkgs`). In this build definition `top` derivation depends on `input0` derivation. Below we will use names of Nix store paths shortened for brevity by removing `/nix/store/<hash>` part of it. We could follow this graph and compute direct provenance set of the final build product `dependencies-top` to be

$$\{\texttt{dependencies-top.drv}, \texttt{dependencies-input-0}, \texttt{simple.deps.builder.sh}\}$$

Its indirect provenance set can be manually computed to be

$$\{\texttt{dependencies-top.drv}, \texttt{dependencies-input-0}, \texttt{simple.deps.builder.sh}, \\ \texttt{dependencies-input-0.drv}, \texttt{builder-dependencies-input-0.sh}\}$$

The same results can be obtained with Tenmo's SQLs (full paths are elided in the output for brevity):

```
tenmo=# select * from provenance_set(
tenmo(#     'i:///.../store/3hknjp8dvfryb4270bc4fyd1r9jshd9c-dependencies-top');
 depth |                                                  obj
-------+--------------------------------------------------------------------------------
     2 | i:///.../store/sh0p2kw8ylqzjijdi3cxpl70z857mhbq-dependencies-top.drv
     2 | i:///.../store/q6ngyanhbcyjr17yrlbbagagrj2clzxd-simple.deps.builder.sh
     2 | i:///.../store/wfchbfd39qcy4j1q13fh4mjb0wkjjvdh-dependencies-input-0
(3 rows)

tenmo=# select * from provenance_set_indirect(
tenmo(#     'i:///.../store/3hknjp8dvfryb4270bc4fyd1r9jshd9c-dependencies-top');
 depth |                                                       obj
-------+--------------------------------------------------------------------------------
     2 | i:///.../store/sh0p2kw8ylqzjijdi3cxpl70z857mhbq-dependencies-top.drv
     2 | i:///.../store/q6ngyanhbcyjr17yrlbbagagrj2clzxd-simple.deps.builder.sh
     2 | i:///.../store/wfchbfd39qcy4j1q13fh4mjb0wkjjvdh-dependencies-input-0
     4 | i:///.../store/40zj9p2w3lkxkpfr3g938vapxp3fmiac-dependencies-input-0.drv
     4 | i:///.../store/a2k781ggfk1syl2an5y2gx218a2yy715-builder-dependencies-input-0.sh
(5 rows)
```

Implementation of `provenance_set` and `provenance_set_indirect` these SQL stored procedures is described in section 6.2.7.

Future work Unstructured line-oriented logs produced by Nix binaries and Nix derivation builds could be attached to the executions as annotations. This would allow to capture a link between build logs and Tenmo execution traces. This mechanism would have a drawback of generating a lot more data to be ingested by the Tenmo pipeline.

We could extend execution of Nix derivation builder with a `strace`-based tracing mechanism, closely following the CBDG implementation. This would have expanded the results with the ability to capture relationship between Nix derivation build executions and process tree executions happening inside a derivation builder (usually with a shell process acting as a parent, and a compiler as the most prominent child). This gives us execution tracking. File-level tracing of reads and writes would allow to accurately track which files each builder process is using. This gives us operations tracking. Given that Nix model provides a way to map from the file paths to the Nix packages, we can reconstruct accurate entity-subentity provenance graph between files from Nix store and Nix packages. All of the above would allow to have a comprehensive hierarchical tracing and provenance tracking across Nix build process and individual build processes happening inside each Nix derivation build execution.

7.1.2. Cluster deployment

KubeNix

KubeNix as a Kubernetes resource builder is unique compared to other resource builders, since it combines Nix to build resource definitions and Nixpkgs as a package repository to build deployed containers. KubeNix produces a `.json` which can be consumed by `kubectl`. Given that KubeNix is built using Nix, the Tenmo integration with Nix is automatically available for KubeNix operations. An example of a Tenmo trace for a resource builds of `deployment` with 10 Nginx pod replicas is available in file `tenmo-thesis/traces/` the thesis code[4]. It includes a trace for a build with full caches present and just a Nginx configuration being changed. A rebuild operation, which produces no new results.

The trace contains the following number of observed objects:

```
tenmo=> select 'entities', count(entity_id) from entities
union select 'incarnations', count(incarnation_id) from incarnations
union select 'processes', count(process_id) from processes
union select 'executions', count('execution_id') from executions
union select 'interactions', count(interaction_id) from interactions
union select 'messages', count(message_id) from messages
union select 'operations', count(operation_id) from operations;
   ?column?   | count
--------------+-------
 operations   |   958
 processes    |     0
 interactions |     0
 incarnations |   877
 entities     |   877
 messages     |     0
 executions   |  1729
(7 rows)
```

It can be seen that the integration does not cover all types of Tenmo objects. Interactions

[4] The code is also available at https://github.com/gleber/tenmo-thesis repository.

and messages are not crucial for a tool like KubeNix and build system like Nix. One can observe that there is the same number of incarnations as entities, which is a testament to Nix, which immutably stores every Nix store object in the content-addressable manner. That's why we see that very entity is unique and and has a single incarnation associated with it.

When KubeNix is used, the final build result it produced is named `kubenix-generated.json` and we can observe it's Nix store name recorded by Tenmo:

```
tenmo=> select incarnation_id from incarnations where incarnation_id like '%kubenix-generated.json';
                             incarnation_id
------------------------------------------------------------------------
 i:///nix/store/ib4k4scpagnidi7f0imsplfa6zw39vrq-kubenix-generated.json
(1 row)
```

It is known that the example deployment resource refers to a a Nginx configuration, hence we can locate an execution which built it:

```
tenmo=> select incarnation_id from incarnations where incarnation_id like '%nginx-conf.json%';
                           incarnation_id
---------------------------------------------------------------------
 i:///nix/store/ryvdjcnb2b1qyzh4fa9nfyp4h1r3063c-nginx-conf.json.drv
(1 row)
```

Given these two identifiers it is possible to explore how they relate in the build process:

```
tenmo=> select unnest(path), unnest(verbs) from (select * from
↪  get_shortest_path('i:///nix/store/ryvdjcnb2b1qyzh4fa9nfyp4h1r3063c-nginx-conf.json.drv',
↪  'i:///nix/store/ib4k4scpagnidi7f0imsplfa6zw39vrq-kubenix-generated.json', 5) limit 1) t;
                                   unnest                                   |    unnest
----------------------------------------------------------------------------+---------------
 i:///nix/store/ryvdjcnb2b1qyzh4fa9nfyp4h1r3063c-nginx-conf.json.drv        | written_by
 89154931131069                                                             | child_of
 89154931130415                                                             | writes
 i:///nix/store/mkfqzlb86s6c1vr0llagsm2dsqiz36vp-kubenix-generated.json.drv | read_by
 60967060766724                                                             | writes
 i:///nix/store/ib4k4scpagnidi7f0imsplfa6zw39vrq-kubenix-generated.json     | <destination>
(6 rows)
```

Here we search for a single path between these two objects with `get_shortest_path` function with traversal depth limit of 5, and next we unnest the resulting arrays for easier viewing. We can observe that the final generated `kubenix-generated.json` has been created by an execution with identifier `60967060766724`, which previously read a `.drv` file (which is essentially a build recipe). This file was written by an execution `89154931130415`, which was a child of `89154931131069`. The latter is the execution, which has instantiated the `nginx-conf.json.drv` build recipe.

These executions can be looked up to be:

```
tenmo=> select description, execution_id from executions where execution_id =
↪  ANY('{89154931131069,89154931130415,60967060766724}');
                                        description                                      |
                                        ↪  execution_id
-------------------------------------------------------------------------------------------+----------------
 overall building '/nix/store/mkfqzlb86s6c1vr0llagsm2dsqiz36vp-kubenix-generated.json.drv' |
 ↪  60967060766724
 derivation 'kubenix-generated.json' being evaled                                          |
 ↪  89154931130415
 derivation 'nginx-conf.json' being evaled                                                 |
 ↪  89154931131069
(3 rows)
```

This shows that Tenmo instrumentation of KubeNix – through Nix – provides a useful mechanism for exploring the multifaceted build trace. We are able to analyze various types of relationships between executions, incarnations and operations.

Kubectl

Kubectl is a traditional command line tool, which accepts a definition of the resource as a file (or from standard input), parses it, compares it to the current state of the resource on the Kuberentes cluster, and – if different – submits it to the server. Kubectl operates on various kinds of resources, a deployment being our focus. Deployment consists of a replica set, which consists of a number of pods. When Kubectl operates on a deployment, it operates on all of the underlying resources recursively.

We have extended Kubectl to log executions and incarations information about produced resources in the resulting resource description YAML file.[5]

A sample execution of

```
kubectl apply -f /nix/store/va46ws49g08xvyf82fvpzfw7p7bzgjm3-kubenix-generated.json
```

produced the graph captured in fig. 7.2. The graph clearly shows that the tool reads the file, transforms resources into an in-memory representation, compares it with the version of the resources stored on Kubernetes cluster, and submits a new version of the resource to the cluster. For example Tenmo can trivially produce a indirect provenance set for the final deployment resource:

```
tenmo=> select * from provenance_set_indirect(

                  ↪  'i://https://192.168.1.1/api/v1/namespaces/default/services/nginx?ulid=01EJP9REP7AB85EW3554GWPR7H');
 depth |                                                     obj
-------+-----------------------------------------------------------------------------------------------------------
     2 |
↪  i://in-memory-https://192.168.1.1/api/v1/namespaces/default/services/nginx?ulid=01EJP9RDNKD3K51RJACSTS28PA
```

[5] Every JSON file is a valid YAML file and that's what KubeNix actually produced.

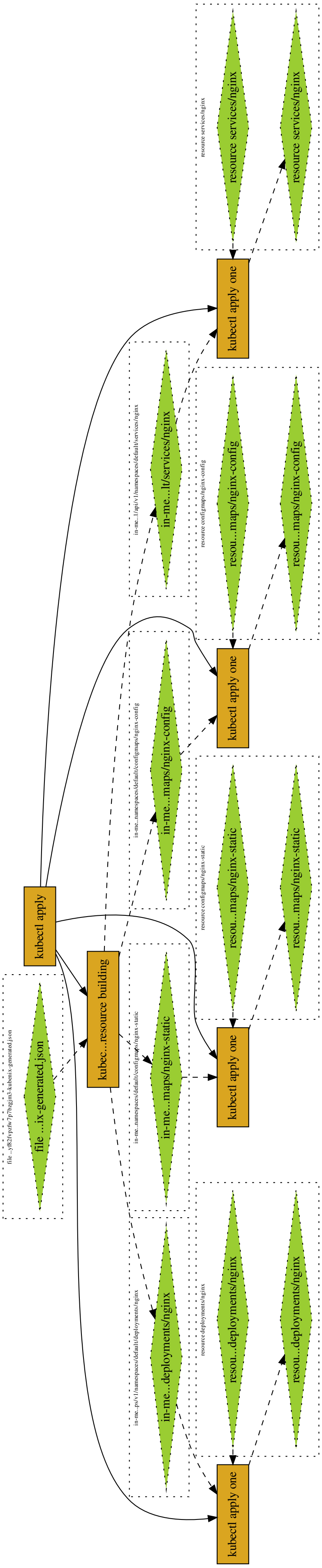

file ...yf82fvpzfw7p7bzgjm3-kubenix-generated.json
file ...ix-generated.json
kubectl apply
kubec...resource building
in-me...1/api/v1/namespaces/default/services/nginx
in-me...lt/services/nginx
in-me...namespaces/default/configmaps/nginx-config
in-me...maps/nginx-config
in-me...namespaces/default/configmaps/nginx-static
in-me...maps/nginx-static
in-me...ps/v1/namespaces/default/deployments/nginx
in-me...deployments/nginx
kubectl apply one
kubectl apply one
kubectl apply one
kubectl apply one
resource services/nginx
resource services/nginx
resource services/nginx
resource configmaps/nginx-config
resou...maps/nginx-config
resou...maps/nginx-config
resource configmaps/nginx-static
resou...maps/nginx-static
resou...maps/nginx-static
resource deployments/nginx
resou...deployments/nginx
resou...deployments/nginx

Figure 7.2: Record of a Kubectl execution on a KubeNix-generated resource file.

```
     2 | i://https://192.168.1.1/api/v1/namespaces/default/services/nginx?ulid=01EJP9REKAW44WKNW8CC1A80V0
     4 |
↪  i://file:///nix/store/va46ws49g08xvyf82fvpzfw7p7bzgjm3-kubenix-generated.json?ulid=01EJP9RCQ95JT85A4AJNP67V6C
(3 rows)
```

An worthwhile aspect of this Tenmo trace is the use of ephemeral incarnations for in-memory objects. This is a fine grained technique of tracking provenance inside of a Unix process.

7.1.3. Cluster orchestration - Kuberentes

We have extended Kuberentes to include minimalistic Tenmo tracing in `deployment` resource controller. A Kuberentes resource controller operates on events sent from `kube-apiserver` whenever any relevant resources change. A controller subscribes for resource changed events per type. For example the `deployment_controller.go` subscribes for changes in:

- `Deployments` - to get notified about new intent provided by a user store in the `spec` field of a resource object;
- `ReplicaSets` - to get notified about state changes of replica sets, which are the entities used to deploy pods defined in a deployment's `spec`;
- `Pods` - to get notified about state changes of pods, which actually run the containers requested by a user;

Deployment controller has been instrumented with Tenmo tracing using the pattern described in the section 6.3.4. Figure 7.3 shows a subset of the Tenmo graph, focusing on the intent-based actuation loop.

7.2. Hierarchical Control Plane System

We believe that the applications of Tenmo in Nix, Kubenix, kubectl and Kubernetes controller collectively shows that Tenmo fulfills the desired functional properties of a hierarchical cloud control plane system.

Figure 7.4 represents a Hierarchical Control Plane System composed out of these components and tracked by Tenmo. The top part outlined with indigo color rectangle is an extract from the KubeNix build process for a deployment. The right part outline in light green color is an extract of a record of Kubectl execution (as described in ??. The bottom part is an extract of a Kubernetes deployments controller performing an intent-based actuation of a deployment

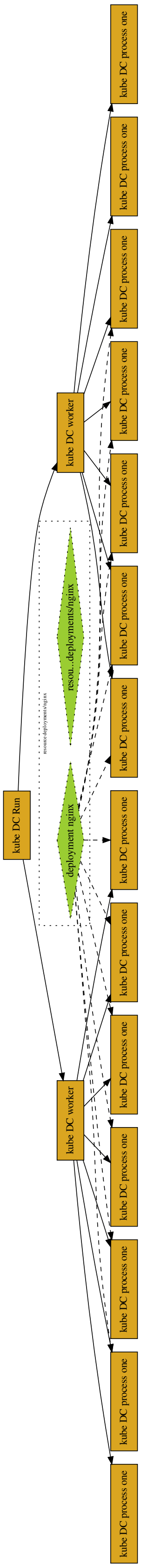


Figure 7.3: Extract from Tenmo record of Kubernetes actuation of a deployment resource.

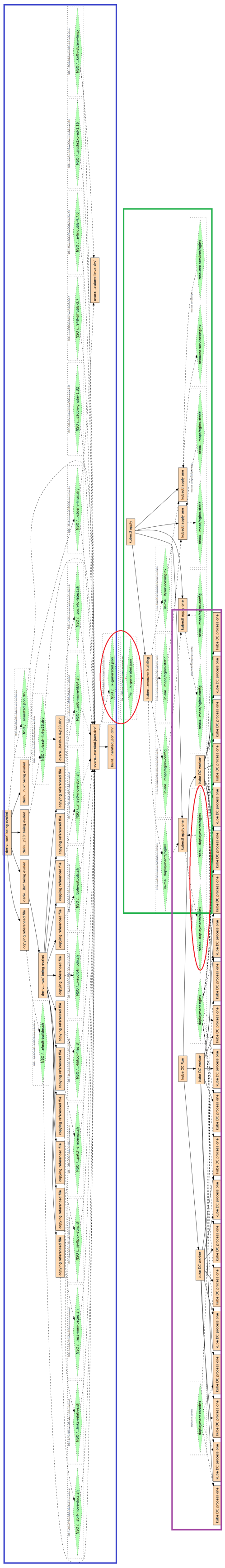


Figure 7.4: Extract from Tenmo record of a HCPS composed out of Nix / KubeNix, kubectl, and Kubernetes deployment controller.

resource (as described in section 7.1.3). Full Tenmo traces would not be possible to visualize in a thesis using existing Graphviz-based tooling.

The Kubnix build and Kubectl execution are related via the file `kubenix-generated.json` built by Kubenix and consumed by Kubectl. Kubectl and Kuberentes deployments controller are linked via the deployment resource representation submitted to Kubernetes by Kubectl tool. These are marked in red circles in fig. 7.4.

This shows that Tenmo is able to track activity and trace provenance of objects across multiple levels of a hierarchical system. Although this is a simplified example, covers the peculiarities of HCPS systems described in chapter 1 ”Introduction”.

8. Concluding remarks

As shown in chapter 7, Tenmo can be applied to a variety of software packages. Tenmo has been shown to be suitable for debugging of build systems, cluster deployment tooling, and cluster orchestration services. We believe that these use cases are representative of a hierarchical control plane system employing intent-based actuation.

8.1. System comparison

Below, we provide a comparison between Tenmo, Dapper [19] and ProTracer [43]. Dapper and ProTracer are two systems, which we believe are the closest match to our use case. The Table 8.1 on page 91 presents a comparison of these systems. As can be seen, they occupy distinct points, in the design space for debuggability solutions.

Section "HCSP support" of table 8.1 summarizes how these systems compare to the requirements defined in chapter 3. We have shown that Tenmo satisfies requirements for a debuggability solution for hierarchical control plane systems employing intent-based actuation.

8.2. Model comparison

Below we discuss how the PEDST model compares to the OpenTracing and to the OpenProvenance models. These comparisons are not meant to be comprehensive, and have introductory nature. However, they provide some birds-eye overview which cna be of value to the reader. Specifically, we show that the Tenmo model is a "middle-ground" between the practicality of the OpenTracing, and the high complexity of the OpenProvenance model. Our model is an extension of the OpenTracing model, and can be reduced to a subset of the PROV data model.

8.2.1. OpenTracing model

In the OpenTracing model the main concept is a trace tree, which is a tree of spans, which represent units of work. Edges in the tree indicate a parent-child relationship between spans.

	Dapper	ProTracer	Tenmo
Focus use case	Distributed serving systems	Advanced Persistent Threat	Control plane systems
Model			
Execution tracing	Yes	Yes	Yes
Object tracking	No	Yes	Yes
Object versioning	– [1]	No	Yes
Interaction tracking	No [2]	Yes	Yes
HCSP support			
Coalescing effects support	–	Yes	Yes
Support for abstract entities	–	No	Yes
Support for composite entities	–	Yes	Yes
Low storage overhead	Yes	No	Yes
Full coverage	No	Yes	Yes
Gradual fidelity execution tracing	Yes	No	Yes
Gradual fidelity provenance tracking	–	Yes	Yes
Minimal mental burden	Yes	No	Yes
Cross-host tracking	Yes	Yes	Yes
Multi-layer systems support	Yes	No	Yes
Asynchronous data intake	No	No	Yes
Event-based data production	Yes	No	Yes
Flexible control flow support	Yes	Yes	Yes
Application			
Mode	RPC Instrumentation	Auto	Opt-in
Trust	Trusted	Untrusted	Trusted
Other			
Trace sampling	Yes	No	No
Maturity	Production	Academic	Academic

[1] Not applicable due to lack of capabilities to track objects.

[2] Dapper does record RPCs, but they are not a first class citizen of the Dapper model.

Table 8.1: Summary system comparison.

An individual “span is also a log of timestamped records, which encode the span’s start and end time, any RPC timing data, and zero or more application-specific annotations” [19, p.3].

The main difference of our model is that our model allows not only for the execution tree to be represented, but also allows us to relate execution trees to each other, whenever they interact with each other via read and write operations on entities in the system. Additionally, interactions between executions are modeled more explicitly and independently of the execution of the parent-child relationship. This allows us to recover more than just a trace tree, out of recorded data, but also a causality graph, interaction graph, effects graph and provenance graph (see section 4.3.1 ”Graphs” below for definitions).

In implementations of the OpenTracing model, all RPC messages are recorded as payload in annotations, but RPC interactions are not explicitly part of the model. Our model explicitly records messages, since they are often used as a vehicle for propagating information and, hence, are relevant from the perspective of provenance tracking (see section 4.3.1 for details on how messages relate to provenance tracking in PEDST model).

Data for a single OpenTracing span is populated by two processes – both RPC client and server. In the PEDST model, each side is represented by an *execution*, and their RPC exchange is represented as a single interaction with an appropriate number of messages (usually two for a single synchronous RPC call).

On top of execution tracing, PEDST model enables capturing interactions with objects, in a system, via *operations* and *incarnation*, allowing it to handle coalescing effects and, henceforth, control plane systems with the intent-based actuation.

In this work we have shown that PEDST model is a superset of OpenTracing model, capable of implementing OpenTracing activity tracing and more. Major difference between Tenmo – as an implementation of the PEDST model – and all implementations of OpenTracing is lack of sampling, which allows OpenTracing implementations to scale to ultra-large online serving systems at the cost of tracing coverage. Tenmo – being focused on control plane systems – is built to provide full coverage of all – usually mutating – activities.

8.2.2. OpenProvenance

The OpenProvenance’s PROV data model is a very elaborate model to track provenance, accommodating three different uses of provenance – agent-centered provenance, object-centered provenance, and process-centered provenance. Our provenance-enhanced distributed systems tracing model has the most similarities with the process-centered provenance. An example of the difference is that our model does not allow to record provenance of an object without capturing a process, which transforms an input object into a given object.

PEDST incarnations can be represented by PROV entities. In PROV, the result of each revision of a thing, in a system, is a new entity. PEDST *entity concept* can be translated to PROV's *specialization concept.* Alternatively PEDST entity concept can or by marking a group of PROV entities with a description that one was a revision of another. The PROV model allows to explicitly track "wasDerivedFrom", relationship between its entities, while PEDST model treats this relationship to be implicit, whenever an execution reads one incarnation and writes another. PEDST's sub-incarnations can be represented with PROV's collections.

PEDST's executions can be represented by PROV activities. PROV *Start* and *End* can represent beginning and end of an execution. Hierarchical nature of PEDST executions can be represented by meticulous use of PROV "wasStartedBy" and "wasEndedBy" relationships, between activities. PEDST's processes can be represented as PROV Plans.

PEDST's interaction can be represented by PROV Communication concept. PROV has no concept which can represent PEDST messages, but those could be, with some loss of information, be packed into attributes of "wasInformedBy" relation.

PEDST's read and write operations can be represented as PROV Usage and Generation concepts, which ties up the PEDST to PROV translation.

PROV contains many other concepts, like agents, responsibility, influence, delegation, bundles, alternate, collections, etc., which are not representable in the Tenmo model. We have not found these concepts to be particularly useful for our use case (section 1.7).

8.3. Outside of the scope of the work

Debuggability of software systems is a vast research area. While working on this thesis we have identified a set of related topics, which we did not pursue as part of this research. We will briefly discuss the relation, why we have not addressed these areas and, in some cases. We'll discuss future development in a next section.

8.3.1. Automated provenance gathering

Although the PEDST model is suitable for use in automated provenance gathering mechanisms, via compiler instrumentation, syscall interception, etc., we believe that automated provenance gathering, if applied to HCPS, will not be successful. Signal-to-noise ratio, in the gathered data, will suffer due to the mismatch between low-level objects, typically suitable for automated provenance gathering, and high-level entities, which HCPS typically operate on. Additionally, it is not obvious how automated provenance gathering will deal with identifying long-living entities and processes, as identifying both is rather subjective, and heavily dependent on the use cases. Moreover, an explosion of data gathered by the system will require additional work to scale the PEDST architecture and the Tenmo framework.

An alternative research avenue, for automated provenance gathering, is through database instrumentation, which would allow us to track how objects evolve over time, in a system. Kubernetes ecosystem could be suitable for such research, but it is not yet clear how database-based provenance gathering would work in a system using `etcd` (a NoSQL database). Traditional database provenance research, typically, focuses on relational databases, with built-in query engines, and NoSQL database systems typically do not have these properties.

8.3.2. Tamper-resistance

Tamper-resistance is a non-goal of this work. Control plane systems are typically owned and operated by a single entity, a cloud provider. Hence, there is a reasonable trust for individuals operating the system. These enterprise deployments would, typically, have other mechanisms to ensure safety, privacy and security of any internal systems (e.g. access control, audit logging, etc).

8.3.3. Audit logging

Although there is a large overlap between what audit logging and provenance-enhanced distributed tracing are aiming to record, the user experience and requirements towards security of the systems are vastly different. Audit logging research often focuses on security and tamper-resistance of the gathered information.

8.3.4. Blockchain provenance

Blockchain provenance is inherent in the construction of most blockchain systems. Additionally recent blockchain-based systems employing smart contracts have some properties similar to the control plane systems (e.g. interleaving of execution- and object-based interactions). Blockchain systems are inherently based on tracking relationships between blocks – a property very similar to provenance tracking. This typically happens at a coarse grain of whole blocks. Granular fidelity data provenance tracking could be useful to trace things at smart contract evaluation level, including sub-block data granularity.

8.4. Future development

This work was focused on proving that a proposed model works and is viable for HCPS. The included implementation is a proof-of-concept. Additional work is necessary both to extend the applicability area for PEDST model and to improve its implementation.

8.4.1. Implementation scalability

We exclude the problem of large-scale deployments, be it sampling, ingestion, processing and querying of the data generated by the tracing system. We expect that a Dapper-inspired implementation, to deal with the volume of generated data, to be sufficient. Additional work is going to be necessary to analyze effects of sampling on the provenance tracking aspects of Tenmo. Given a more expressive data model, additional work is necessary to understand more complex graph-based, and potentially recursive, queries over the data. Additional implementation-specific limitations (e.g. annotation payload size limit) would be necessary for Tenmo framework to scale to large-scale systems.

8.4.2. Library-level instrumentation

The Tenmo model is suitable for the library-level instrumentation, e.g. in RPC libraries, and context propagation libraries, and should be able to follow distributed control paths, with near-zero intervention from application developers. Similarly, as any OpenTracing implementation, distributed tracing can be provided out of the box to application developers, who use Tenmo-instrumented libraries.

We found that there is no single RPC library, which is adopted widely enough in the open-source ecosystems related to HCPS, which would allow us to implement such library-level instrumentation and analyse results.

8.4.3. Security

We believe that the PEDST model, proposed in this thesis, is suitable as a foundation for Advanced Persistent Threat attack detection and investigation solutions (see, [63]). This could be, one more, direction for future work. The PEDST architecture would have to be adapted to act in a limited trust environment, and will not be able to depend on the white-box instrumentation approach. Clearly, additional work is required to understand implications of switching to a black-box approach.

8.5. Contributions

The main contribution of this thesis is the development of a provenance-enhanced distributed systems tracing model, which we implemented in the Tenmo tracing system. In this model, we combine industry-standard tracing concepts, with provenance tracking mechanisms, as researched in a wide range of provenance research.

We show that the proposed model is a general solution suitable for debugging modern hierarchical control plane systems, which include heterogeneous components acting both in imperative and declarative paradigm, including use of intent-driven actuation. We show that our model is suitable for implementation and practical usage.

In this thesis, we have researched, designed and proven a useful solution in the debuggability landscape. It occupies a unique point in the design space of debugging systems.

This said, main contributions of this thesis can be summarized as follows:

- Identified distinguishing characteristics of hierarchical control plane systems (HCPS) with intent-based actuation, making the their debuggability an unsolved problem (section 1.7).
- Applied provenance research results, in a practical software engineering problem of tracing in HCPS.
- Identified necessary properties of a debuggability solution, capable of dealing with HCPS (chapter 3).
- Defines a provenance-enhanced distributed systems tracing model, extending the OpenTracing distributed systems tracing model with provenance tracking capabilities (chapter 4).
- To track provenance, in a way compared to the Dapper model, read and write operations performed by each component in a system have been recorded explicitly and treated as first-class citizens of a model (section 4.1.2).
- Presented an architecture, required to perform provenance-enhanced distributed systems tracing for HCPS (chapter 5).
- Showed an implementation, capable of reconstructing a full provenance and execution graphs, given the logs are gathered according to the logging protocol (chapter 5, chapter 6).
- Showed that if the logging is done according to the protocol, a global view on the provenance graph can be recovered (section 4.2).

- Showed that the proposed provenance-enhanced distributed systems tracing model is practical (chapter 7).
- Showed that provenance-enhanced distributed systems tracing data can be used to infer additional information through over-approximation mechanisms (section 4.3.1).

8.6. Discussion

Tenmo has been shown to be able to model, gather and analyse provenance-enhanced tracing data. We have shown that Tenmo is applicable to a number of software systems, and can answer practical debugging questions.

In its current implementation Tenmo tracing is either fully enabled or disabled. There is no mechanism of granular control over tracing. Such control could be implemented and made to work akin to standard logging verbosity runtime settings. This mechanism would allow to trade-off tracing coverage and fidelity versus performance penalty of tracing. Additional work is required to understand impact of such runtime controls over correctness of execution tracing and provenance tracking.

Given that Tenmo is a white-box tracing solution dependent on instrumentation inside of the system under tracing, adoption costs needs to be considered. In our limited experience implementing Tenmo for Nix and Kubernetes, we have found that the process is mostly straightforward and adding logging statements is obvious and cheap. On the other hand we have found that the following hurdles needs to be overcame.

- Propagation of execution identifiers across layers of a single system. Instrumentation of a Go project which pervasively uses Go's Context would also have been easier.
- Propagation of a parent execution identifier from the caller to the callee across technologies. A RPC-level instrumentation would simplify integration of Tenmo into a system. W3C Trace Context could be used for HTTP interactions.
- Entity incarnation versioning is not always obvious, can be hard to added to the system under tracing and be of a can be not important for the system itself. For example reading individual files from disk provides no information about their versioning, besides a modification time. Using it as a incarnation version is possible, but requires use of "inter-incarnation provenance extension" extension to maintain provenance relationship.
- Identifiers coherence across systems. It has been hard to maintain coherent identifiers for executions and incarnations across the systems. This should be doable in a single

organization via a policy, but is harder to achieve across organizations and/or in open source world.

Additional research is necessary to address these issues.

During work on integration of Tenmo in services, we have found that it is useful to manually add human-inferred data for the needs of analysis. For example, a human can infer from the context that two incarnations, produced by two different systems, each with unique identifier, are actually provenance-related. It might be useful to allow a human to "assert" such relationships into a "observed universe", constructed based on a given dataset. Such assertion, for example, could be structured as an imaginary execution, with read-write operations pair similarly as "inter-incarnation provenance extension" (section 4.3.1 to make sure that provenance relationship is maintained. This would help developers to manually "connect the dots", during interactive development sessions in presence of instrumentation imperfections in the traced system.

Concepts of entities, processes, sub-incarnations and sub-executions allow to aggregate data in various ways. Various data aggregations, in the global model, can be useful to visualize the data to a user in abstracted form, while preserving correctness and improving usefulness. These aggregations should help navigating large provenance and execution graph,s generated in the PEDST model. We expect these to be especially useful for exploratory debugging by software developers, not yet knowledgeable about a system under tracing. Provenance segmentation research [64] provides additional avenues for improving signal-to-noise ratio levels of gathered data.

Another aspect of this system, which has not been investigated, and seems to have potential value, is temporal analysis of Tenmo traces. An example is ability to compute a difference between two (or more) historical executions of the same complex process, at different times, in somehow different contexts. This could address a practical problem of "why did it suddenly stop working today although it was working yesterday?". Additional research is necessary to formulate this problem in a clear manner, and propose a solution.

Moreover the graph data model, obtained during work on this problem, closely resembles the Resource Description Framework (RDF) triples [56]. We believe that the existing extensive research on knowledge extraction and data retrieval, in the area of RDF and Semantic Web [57], could be used here to drive further usability of the solution [65].

8.7. Conclusion

This research aimed at proposing a debugging solution, applicable to hierarchical control plane system employing intent-based actuation. Based on requirements analysis, use case modelling, sketch of model formalization, and empirical usage prototypes, of the proposed solution in a software stack, representative of hierarchical control plane systems deployed at cloud providers, it can be concluded that the proposed solution is successfully applicable to the focus use case of this work. The results indicate that a large scale debuggability solution, for hierarchical cloud control plane systems, is possible and practical.

Based on these conclusions, practitioners should consider looking into provenance and provenance tracking research to adopt ideas for the next generation of observability solutions.

To better understand the usability of data gathered using the proposed solution, future studies could address problem of efficient knowledge retrieval from the large amounts of data the proposed system would acquire. Further research is needed to determine the best ways to query the data for visualizations in debugging tooling, interactively explore the data, annotate the data manually and, overall, help software engineers use the proposed solution to reason about the distributed systems they own.

Existing debuggability solutions used in industry, and proposed in academia, do not address the specific use case of hierarchical control plane systems. This research addresses this gap, by combining two areas of research: distributed systems tracing and provenance tracking.

■

Bibliography

[1] D. Dvorak, „NASA study on flight software complexity", in *AIAA Infotech@ Aerospace Conference and AIAA Unmanned... Unlimited Conference*, 2009, p. 1882.

[2] R. Potvin and J. Levenberg, „Why Google Stores Billions of Lines of Code in a Single Repository", *Commun. ACM*, vol. 59, no. 7, pp. 78–87, Jun. 2016, ISSN: 0001-0782. DOI: 10.1145/2854146. [Online]. Available: https://doi.org/10.1145/2854146.

[3] F. Engineering. (2014). 9.9 million lines of code and still moving fast - Facebook open source in 2014, [Online]. Available: https://engineering.fb.com/core-data/9-9-million-lines-of-code-and-still-moving-fast-facebook-open-source-in-2014/ (visited on 08/31/2020).

[4] L. A. Barroso and P. Ranganathan, „Datacenter-scale Computing", *IEEE Micro*, vol. 30, pp. 6–7, 2010, Special issue of the IEEE Micro Magazine. [Online]. Available: http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.63.

[5] G. Fowler, „A case for make", *Software: Practice and Experience*, vol. 20, no. S1, S35–S46, 1990. DOI: 10.1002/spe.4380201305. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380201305. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380201305.

[6] D. Hall, *Ansible configuration management*. Packt Publishing Ltd, 2013.

[7] J. Loope, *Managing infrastructure with puppet: configuration management at scale*. " O'Reilly Media, Inc.", 2011.

[8] Y. Brikman, *Terraform: Up & Running: Writing Infrastructure as Code*. O'Reilly Media, 2019.

[9] D. Merkel, „Docker: lightweight linux containers for consistent development and deployment", *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[10] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, „Borg, omega, and kubernetes", *Queue*, vol. 14, no. 1, pp. 70–93, 2016.

[11] C. Low, Y. Chen, and M. Wu, „Understanding the determinants of cloud computing adoption", *Industrial management & data systems*, 2011.

[12] K. Wood and M. Anderson, „Understanding the complexity surrounding multitenancy in cloud computing", in *2011 IEEE 8th International Conference on e-Business Engineering*, IEEE, 2011, pp. 119–124.

[13] G. Commentary, „Complexity: Cloud Computing's Achilles Heel", *InformationWeek*, Sep. 2019. [Online]. Available: https://www.informationweek.com/cloud/complexity-cloud-computings-achilles-heel/a/d-id/1335816?.

[14] *OpenShift, Kubernetes - Explore - Google Trends.* [Online]. Available: https://trends.google.com/trends/explore?date=today%205-y&q=%2Fm%2F0j9pqc0,%2Fg%2F11b7lxp79d (visited on 08/31/2020).

[15] The Kubernetes Authors, „Understanding Kubernetes Objects", Accessed on 2020-05-13: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/, 2020.

[16] E. Dolstra, R. Vermaas, and S. Levy, „Charon: Declarative provisioning and deployment", in *2013 1st International Workshop on Release Engineering (RELENG)*, IEEE, 2013, pp. 17–20.

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

[18] P. J. McNerney, *Beginning Bazel.* Apress, Berkeley, CA, 2020, ISBN: 978-1-4842-5193-5. DOI: 10.1007/978-1-4842-5194-2.

[19] 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.

[20] The Bazel Authors, „This repository contains rules for interacting with Kubernetes configurations / clusters.", Accessed on 2020-05-30: https://github.com/bazelbuild/rules_k8s, 2020.

[21] The Skycfg Authors, „Skycfg is an extension library for the Starlark language that adds support for constructing Protocol Buffer messages.", Accessed on 2020-05-30: https://github.com/stripe/skycfg/tree/master/_examples/k8s, 2020.

[22] A. Mikkelsen, T.-M. Grønli, and R. Kazman, „Immutable Infrastructure Calls for Immutable Architecture", in *Proceedings of the 52nd Hawaii International Conference on System Sciences*, 2019.

[23] *Terraform*, [Online; accessed 1. Sep. 2020], Aug. 2020. [Online]. Available: https://www.terraform.io/docs/cloud/index.html.

[24] The EMAP Authors, „Terraform As A Service(TAS)", Accessed on 2020-05-30: https://cloud.epam.com/site/competency_center/e=p=c_services/terraform_as_a_service(=t=a=s), 2020.

[25] S. Van Der Burg and E. Dolstra, „Disnix: A toolset for distributed deployment", *Science of Computer Programming*, vol. 79, pp. 52–69, 2014.

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

[27] Y. L. Simmhan, B. Plale, and D. Gannon, „A survey of data provenance in e-science", *ACM Sigmod Record*, vol. 34, no. 3, pp. 31–36, 2005.

[28] M. Herschel, R. Diestelkämper, and H. B. Lahmar, „A survey on provenance: What for? What form? What from?", *The VLDB Journal*, vol. 26, no. 6, pp. 881–906, 2017.

[29] T. A. Limoncelli, „GitOps: a path to more self-service IT", *Communications of the ACM*, vol. 61, no. 9, pp. 38–42, 2018.

[30] *The OpenTracing project*, [Online; accessed 7. Jun. 2020], Jan. 2020. [Online]. Available: https://opentracing.io.

[31] *Trace Context*, [Online; accessed 1. Sep. 2020], Feb. 2020. [Online]. Available: https://www.w3.org/TR/trace-context.

[32] J. Turnbull, *Monitoring with Prometheus.* Turnbull Press, 2018.

[33] Y. Grinshteyn, „Integrating Tracing and Logging with OpenTelemetry and Stackdriver", *Medium*, Feb. 2020. [Online]. Available: https://medium.com/google-cloud/integrating-tracing-and-logging-with-opentelemetry-and-stackdriver-a5396fbc3e78.

[34] M. Imran, H. Hlavacs, F. A. Khan, S. Jabeen, F. G. Khan, S. Shah, and M. Alharbi, „Aggregated provenance and its implications in clouds", *Future Generation Computer Systems*, vol. 81, pp. 348–358, 2018.

[35] K. Sivakumar and M. Chandramouli, „Concepts of Network Intent", *Internet Research Task Force, Internet Draft, Oct*, 2017.

[36] B. E. Ujcich, A. Bates, and W. H. Sanders, „Provenance for Intent-Based Networking", in *Proceedings of the IEEE Conference on Network Softwarization*, 2020.

[37] B. E. Ujcich, S. Jero, A. Edmundson, Q. Wang, R. Skowyra, J. Landry, A. Bates, W. H. Sanders, C. Nita-Rotaru, and H. Okhravi, „Cross-app poisoning in software-defined networking”, in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 648–663.

[38] P. Groth and L. Moreau, „Representing distributed systems using the Open Provenance Model”, *Future Gener. Comput. Syst.*, vol. 27, no. 6, pp. 757–765, Jun. 2011, ISSN: 0167-739X. DOI: 10.1016/j.future.2010.10.001.

[39] A. Gehani and D. Tariq, „SPADE: support for provenance auditing in distributed environments”, in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, Springer, 2012, pp. 101–120.

[40] I. Souilah, A. Francalanza, and V. Sassone, „A Formal Model of Provenance in Distributed Systems.”, in *Workshop on the Theory and Practice of Provenance*, 2009, pp. 1–11.

[41] M. Whittaker, C. Teodoropol, P. Alvaro, and J. M. Hellerstein, „Debugging distributed systems with why-across-time provenance”, in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 333–346.

[42] J. Cheney, U. Acar, and A. Ahmed, „Provenance traces”, *arXiv preprint arXiv:0812.0564*, 2008.

[43] S. Ma, X. Zhang, and D. Xu, „Protracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting.”, 2016.

[44] 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.

[45] R. Ikeda, H. Park, and J. Widom, „Provenance for generalized map and reduce workflows”, 2011.

[46] 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.

[47] The Bazel Authors. (2020). Optimizing performance - Bazel, [Online]. Available: https://docs.bazel.build/versions/master/skylark/performance.html (visited on 05/30/2020).

[48] E. Dolstra, *The purely functional software deployment model.* Utrecht University, 2006.

[49] apenwarr, „redo, buildroot, and serializing parallel logs”, Accessed on 2020-05-30: https://apenwarr.ca/log/20181106, 2020.

[50] S. van der Burg, E. Dolstra, S. McIntosh, J. Davies, D. M. German, and A. Hemel, „Tracing software build processes to uncover license compliance inconsistencies", *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 731–742, Sep. 2014. DOI: 10.1145/2642937.2643013.

[51] U. Braun, S. Garfinkel, D. A. Holland, K.-K. Muniswamy-Reddy, and M. I. Seltzer, „Issues in automatic provenance collection", in *International Provenance and Annotation Workshop*, Springer, 2006, pp. 171–183.

[52] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. [Bussche], „The Open Provenance Model core specification (v1.1)", *Future Generation Computer Systems*, vol. 27, no. 6, pp. 743–756, 2011, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2010.07.005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X10001275.

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

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

[55] 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.

[56] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, „Dbpedia: A nucleus for a web of open data", in *The semantic web*, Springer, 2007, pp. 722–735.

[57] N. Shadbolt, T. Berners-Lee, and W. Hall, „The semantic web revisited", *IEEE intelligent systems*, vol. 21, no. 3, pp. 96–101, 2006.

[58] *PostgreSQL: Documentation: 12: 7.8. WITH Queries (Common Table Expressions)*. [Online]. Available: https://www.postgresql.org/docs/12/queries-with.html (visited on 09/19/2020).

[59] *Bracket pattern - HaskellWiki*, [Online; accessed 11. Aug. 2020], Apr. 2019. [Online]. Available: https://wiki.haskell.org/Bracket_pattern.

[60] *RAII - cppreference.com*, [Online; accessed 11. Aug. 2020], May 2020. [Online]. Available: https://en.cppreference.com/w/cpp/language/raii.

[61] 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.

[62] E. Dolstra and N. authors. (2020). nix/logging.hh at master · NixOS/nix, [Online]. Available: https : / / github . com / NixOS / nix / blob / 4d77513d97775c05b19b0676a76fcdd3d00bea8b / src / libutil / logging . hh (visited on 08/22/2020).

[63] M. K. Daly, „Advanced persistent threat”, *Usenix, Nov*, vol. 4, no. 4, pp. 2013–2016, 2009.

[64] R. Abreu, D. Archer, E. Chapman, J. Cheney, H. Eldardiry, and A. Gascón, „Provenance Segmentation.”, in *TaPP*, 2016.

[65] S. Powers, *Practical RDF: solving problems with the resource description framework.* " O'Reilly Media, Inc.", 2003.

List of Figures

Appendicies

Appendix A

get_all_paths_from function.

```
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;
```

get_all_paths_from_by_verbs function.

```
CREATE OR REPLACE FUNCTION get_all_paths_from_by_verbs(start text, crawl_verbs 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 and r.verb = ANY(crawl_verbs)

   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 and r.verb = ANY(crawl_verbs)
)
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;
```

get_closure_from function.

```
create or replace function get_closure_from(start text)
returns table(depth integer, obj text) as $$
begin
return query
select t.depth, t.path[array_upper(t.path,1)] from get_all_paths_from(start) as t;
end;
$$ language plpgsql;
```

get_closure_from_filtered function.

```
create or replace function get_closure_from_filtered(start text, filter_verbs text[])
returns table(depth integer, obj text) as $$
begin
return query
select t.depth, t.path[array_upper(t.path,1)] from get_all_paths_from(start) as t where
↪  t.verbs[array_upper(t.verbs,1)-1] = ANY(filter_verbs);
end;
$$ language plpgsql;
```

get_closure_from_by_verbs function.

```
CREATE OR REPLACE FUNCTION get_closure_from_by_verbs(start text, crawl_verbs text[])
RETURNS TABLE(depth integer, obj text) AS $$
BEGIN
RETURN QUERY
select t.depth, t.path[array_upper(t.path,1)] from get_all_paths_from_by_verbs(start, crawl_verbs) as t;
END;
$$ LANGUAGE plpgsql;
```

get_closure_from_by_verbs_filtered function.

```
CREATE OR REPLACE FUNCTION get_closure_from_by_verbs_filtered(start text, crawl_verbs text[],
↪  filter_verbs text[])
RETURNS TABLE(depth integer, obj text) AS $$
BEGIN
RETURN QUERY
select t.depth, t.path[array_upper(t.path,1)] from get_all_paths_from_by_verbs(start, crawl_verbs) as t
↪  where t.verbs[array_upper(t.verbs,1)-1] = ANY(filter_verbs);
END;
$$ LANGUAGE plpgsql;
```

provenance_set function.

```sql
CREATE OR REPLACE FUNCTION provenance_set(start text)
RETURNS TABLE(depth integer, obj text) AS $$
BEGIN
RETURN QUERY
select * from get_closure_from_by_verbs_filtered(start, ARRAY['written_by','reads']::text[],
↪  '{reads}'::text[]) as t where t.depth <= 2;
END;
$$ LANGUAGE plpgsql;
```

provenance_set_indirect function.

```sql
CREATE OR REPLACE FUNCTION provenance_set_indirect(start text)
RETURNS TABLE(depth integer, obj text) AS $$
BEGIN
RETURN QUERY
select * from get_closure_from_by_verbs_filtered(start, ARRAY['written_by','reads']::text[],
↪  '{reads}'::text[]) as t;
END;
$$ LANGUAGE plpgsql;
```

trace function.

```sql
CREATE OR REPLACE FUNCTION trace(start text)
RETURNS TABLE(depth integer, obj text) AS $$
BEGIN
RETURN QUERY
select * from get_closure_from_by_verbs(start, ARRAY['child_of']::text[]) as t;
END;
$$ LANGUAGE plpgsql;
```