TECHNISCHE UNIVERSITÄT BERLIN

ISEngineering

Wirtschaftsinformatik - Information Systems Engineering

CLOUD PROTOTYPING PROJECT

Darshan Mukund Hingu
Kevin Hudemann
Gerrit Janßen
Mukrram Ur Rahman
Muhammad Talal Saleem
Vinothkumar Nagasayanan
Ron Wierzchowski

supervised by
Dominik ERNST

March 14, 2018

Abstract

Contents

Chapter 1

Introduction

1.1 Basics of Provenance Systems

Provenance is simply data quality. Provenance is the relationship among all elements that contributed to the existence of a piece of data.The provenance of data focuses on the history of changes and movement of data. The history of data changes can include subsetting, formatting, transformation, semantic transformation, syntactic transformation and ingesting new data to the existing data. To maintain or proved the quality of data the lineage of the data needs to collected throughout the data transformational process. The metadata to ensure the quality of the data is to be generated in systematic function. All the information about the elements and the relationship among all the elements (sources, processing steps contextual information and dependencies) should include in the definition of provenance capturing.

There are clear differences between process provenance (focusing on workflow execution and the execution environment) and data provenance (focusing on creation and transformation of data). Understanding the type of provenance information of interest and how it will be used can help inform the decision for how to capture the provenance information. The provenance of data can often be collected automatically by saving system logs for events, system inputs, and system outputs but this kind of the provenance does not capture the insight and the rational decision made by the system.

In practical situations, it is not easy to completely capture all types of provenance information. Provenance information can be captured with varying levels of detail. Provenance granularity describes the level of detail of the captured information. The level of provenance granularity is motivated by the use of the particular system.In this project, we are focused towards the data provenance mechanism and technique used to capture in a distributed setup.

1.1.1 Purpose

Capturing the process of formation of data in the IoT ecosystem is like three-folding the already data being generated by the sensor. So, there must a concrete reason for doing this. One simple answer to this is that the data in possession will support the ongoing analysis or some-what enhances the result once the results are completed.

Recall of the analytic process is one of the common purposes to capture provenance. Recall enables awareness and understanding of what processing went through and what task are pending. Recalling is important for analytic clarity and efficiency especially when the processing step is complex and interconnected to another component in the architecture.

Another common purpose of provenance information is to reproduce previously obtain the result. This help to verify that the result is correct and can be trusted. Difference between the results after toggling environment and context parameter can be clearly seen resulting in choosing the best suitable parameter for the upcoming deployment.

Another purpose for provenance involves reviewing the processing step itself and the environment it was run on. This Meta-analysis of processes makes it possible to review and evaluate the step and decision made by the component for the creation of the data. Meta-analysis help to extract patterns and help to optimize performance and suggestion on the changes in real time if need be.

1.1.2 Provenance in IoT

Data in IoT ecosystem is produced in large velocity, volume, and variety. In order to examine and maintain the correctness of the data, provenance data systems are introduced to increase the authenticity. Due to the complicated structure of IoT pipeline in which data comes from distributed nodes across the architecture extremely hard to track manually provenance system is necessary for any sort of insurance on the existing data properties.

Numerous jobs applying complex operation on them are performed which is then propagated to produce some sort of insight from the data or maybe feed into a machine learning algorithm. Finding the reason for an anomaly or outlier from the huge chunk for the data scientist resulting in unexpected results can be a daunting task without capturing of provenance information.

Debugging unexpected results can be narrow done to the components through which the data has gone through saving a lot of time.This help makes analytic decision which was unable to make without the help of the lineage of the data captured. In case of a distributed system where the data is digitally transformed and derived from numerous sources by applying complex function in various context can produce different results. The advantage of using provenance system with the existing system gives the ability to use the data produced by giving the user transparent view of the whole process and the underlying mechanism used to collect it across a different component of a data-processing workflow so the user knows how the data is molded before it reached to the endpoint.

Capturing provenance information in a distributed system gives insight not only to the data-dependencies but also for fault tolerance and usage statistics. Collection of provenance information is useful in many contexts, such as verifying result and explaining the existing of the item. Capturing provenance information for any specific workflow or process make the user of the data to easily follow the origin of the data.

1.1.3 Challenges

Collecting provenance for any given system is a difficult task at hand and requires a deep understanding of the underlying architecture of the system to implement an efficient and useful provenance system. This includes taking into account that every architectural design in every situation is motivated by some core values behind that system which need to be taken into account when building a provenance system which tracks each and every movement of the system overall. There is a lot unanswered question provenance architect has to decide on before building a model which suit the system and fulfill the sole purpose it is built for. Typically, when talking about provenance in IoT domain where the addition of one component can scale the data generated exponential can cause disastrous effect if the decision is not made after suitable consideration and trade-off in mind.

Provenance collecting for this kind of system must be able to scale to both large volumes of data and numerous operation to avoid being a bottleneck. Distributed pipeline are difficult to manage and control especially when there is a great chance of failure of the system and avoid running the provenance system when some component is not working correctly. This might corrupt the data and will produce gibberish data not useful for anyone.

Provenance system should consider the finer granularity in which it captures the transformation of data. Some component of the system is transparent in term of how the data is manipulated. On the other hand, the black-box operator is not transparent enough and does not know how the transformation is taking place. Producing highly accurate provenance for this kind of operation requires specific techniques which incur time overhead to capture and there is a direct trade-off between the performance of the system.

1.2 Related Work

1.3 Project Organization

The project was started as part of the Cloud Prototyping course of TU Berlin in October 2017. With a team of 7 students that did not know each other before, it was necessary to organize the distribution of tasks and use of tools internally. This organization was not set beforehand, but evolved continuously over the course of this semester.

1.3.1 Task distribution

When starting this project as a team of equal students, the distribution of tasks was not trivial as we first had to come up with a problem, use-case and a general vision of what we wanted to achieve. Therefore initially everybody had to figure that out for themselves, before discussing it in the group and coming to an agreement.

After making these high-level decisions, tasked were started to be distributed based on personal preferences. The granularity of tasks was evolving over time, though quite differently for different tasks. For example, it became clear at the beginning that we needed a simulation of a smart grid on which we could test our prototype, which resulted in rather precise implementation tasks early on. However, the design of the data model took more research and was also revised repeatedly, so that the first fine grained implementation tasks were started some weeks later.

After the first five weeks a scrumboard was introduced [**zenhub**]. Before that, regular github issues were used to define task distribution. However, with ever more fine grained and interconnected implementation tasks, it became harder to keep the overview. The scrumboard therefore helped to keep a better overview and also subjectively increased the teams performance, especially once we started to set deadlines on a weekly basis.

The Following table 1.1 was set before the midterm presentation to define responsibilities, especially when it comes to questions regarding the components from within the team as well as from outsiders.

Responsibility	Name
IoT Pipeline	Kevin
Data Model + Provenance Collector	Mukrram
Backend	Vinoth, Mukrram
Databases	Talal
Frontend	Darshan
Deployment & Testing	Gerrit
Project Management	Ron

Table 1.1: Distribution of Responsibilities

1.3.2 Tools

In order to work effectively and potentially simultaneously as a team of 7 members, the use of modern organization tools was necessary. The use of GitHub issues and the scrumboard was mentioned already above. The use of Slack for internal communication and GitHub for code repositories is further explained below. Further tools that aided in code development such Travis CI for continuous integration can be found in the implementation chapter.

1.3.2.1 Slack

Slack had been chosen as the main tool for communication, since it is a very commonly used tool for team organization and all team members were already familiar with it.

Initially only a "general" channel for discussions and a "links" channel for an overview of important links were created. Next integrations for GitHub were added, which allowed to track commits and issues for each repository in a separate channel. Also over time, additional channels were created for more specific tasks, that allowed to keep conversations about different topics (e.g. organization of meetings vs. implementation issues with a certain component) separate, leading to a better overview and less "noise" for members not involved in a specific issue.

1.3.2.2 GitHub

GitHub was chosen as the tool for distributed version control for our coding efforts, again because of the familiarity of all team members. Different components of our software project were developed in different repositories, in order to keep a better overview and establish clear boundaries.

The first repository that was created was *Krymnos/IDP* which functioned as the main repository for GitHub issues and for the first component that was developed: the pipeline implementation needed for deploying and testing our system. Next the repository for the provenance daemon and API was created. Then the repositories for the Backend and Frontend services were created and lastly a repository for archiving the tools and setups we used to facilitate our benchmarks. The links to these repositories are listed in the order they were mentioned below. Details on the components that were developed and the benchmarks conducted can be found in Chapter 2 and 3 respectively.

- https://github.com/Krymnos/IDP
- https://github.com/Krymnos/Provenance-System-for-IoT
- https://github.com/Krymnos/IDP-backend
- https://github.com/Krymnos/IDP-frontend
- https://github.com/Krymnos/idp-benchmark

1.3.3 This Report

This final report was written by all members of the team. The distribution of writing tasks was again done by personal preferences. The Following table 1.2 lists the authors of each section.

1.4 Use Case

A smart grid is an electrical grid which includes a variety of operational and energy measures including smart meters, smart appliances, renewable energy resources, and energy efficient resources. Electronic power conditioning and control of the production and distribution of electricity are important aspects of the smart grid. Providing reliability in smart grids are done using electronic control, metering, and monitoring. To motivate our design decisions we are looking at how a potential user would interact with our system. In this chapter we are looking at the Administrators of a small to medium sized smart grid.

Their main goals are to find failures in near-realtime and to be able to analyze these failures manually. Furthermore they want to have data available

1. Introduction	
* Basics of Provenance Systems	Talal
* Related Work	Mukrram
* Project Organization	Ron
* Use Case	Ron
2. Approach / Implementation	
* Implementation Overview/Approach	Mukrram
* Data Model	Mukrram
* Pipeline Implementation	Kevin
* Provenance API	Vinoth
* Cassandra as Provenance DB	Talal
* User Interface	Darshan
* Backend	Vinoth
* Frontend	Darshan
* Testing & Deployment	Gerrit
3. Benchmarks	
* Introduction - Motivation	Ron
* Benchmark Overhead (Data Volume)	Gerrit
* Benchmark Overhead (Latency)	Talal
* Failure Benchmark	Vinoth
4. Conclusion	Kevin

Table 1.2: Authors of Sections

for offline analysis to compare overall and individual component performances over time. The components that are most relevant for this purpose are the gateway nodes that relay and potentially alter the messages produced by the sensors.

The system administrators have various tools available for monitoring their system. To detect failures heartbeat messages can be implemented, to trace latencies of messages existing tracing systems can be deployed and debugging and logging tools can be used to capture potentially relevant information for manual as well as automatic analysis. However, a combined and more specialized solution has the potential to provide more value.

1.4.1 Problems

The administrators are facing the following main challenges:

- Information on node health should be available as fast as possible.
- Data for debugging should be available at one location independent of the grid.
- Not all Members of the team have the same computer science Background. A simplified interface for manual tasks is required.
- Hardware Resources on the Grid are limited.
- The administrators need to know, how much Overhead additional monitoring tools will introduce on the gateways.

1.4.2 Example Workflows

The following examples are written as user stories and intended to define what the system administrator team is expecting from the system.

1.4.2.1 Installation

- The provenance daemon has to be installed on all nodes that are to be tracked
- For context parameters such as "Line of Code" the existing code running on the gateways needs to be altered to expose such information via the provenance api.
- the Database, Backend and Frontend Services can be installed on any server

1.4.2.2 Monitoring

- The user has a visual overview of all nodes that are part of the grid.
- The overview provides information on node health based on heartbeat messages and send/receive rates
- Changes in a node's health are signaled through changes in colors:
  - Green: Good health is inferred from recent messages.
  - Yellow: Possible problem when node takes longer to respond to messages (Default 5 sec)
  - Red: A failure is confirmed or the node has not responded after >10 sec.

1.4.2.3 Failure Scenarios

- Node Failure: If a node does not react to messages from its neighbours, including heartbeat messages, it is assumed that the whole node has failed.
- Channel Failure: If heartbeat messages reveal that two nodes are healthy, but messages are not delivered between them, a problem with the network link can be inferred.
- Pipeline Daemon: If the process responsible for relaying sensor messages is unresponsive, this information is propagated through heartbeat messages.
- Provenance Daemon: If the process responsible for capturing and sending the provenance information is unresponsive, it is captured in the next heartbeat message.

1.4.2.4 Analysis

- Through the UI the user can click on a node to see messages that have passed recently.
- for each message, provenance data is visible in the UI or as a JSON download.
- when looking for specific information the user can also use the UI query tool (e.g. find message based on ID).

- when a failure occurs the user can look at the information of the last messages that passed
- for larger queries like when comparing system wide performance for a given time period, the database should be queried directly.

Once an issue is identified, it is assumed that the users will use their own tools and possibly physical access to solve them.

Chapter 2

Approach / Implementation

To explain the implementation decisions that went into this project, we will first give an overview of what we aimed to accomplish at the beginning of the project, before explaining how each component got implemented in detail.

2.1 Implementation Overview

TODO

2.2 Pipeline

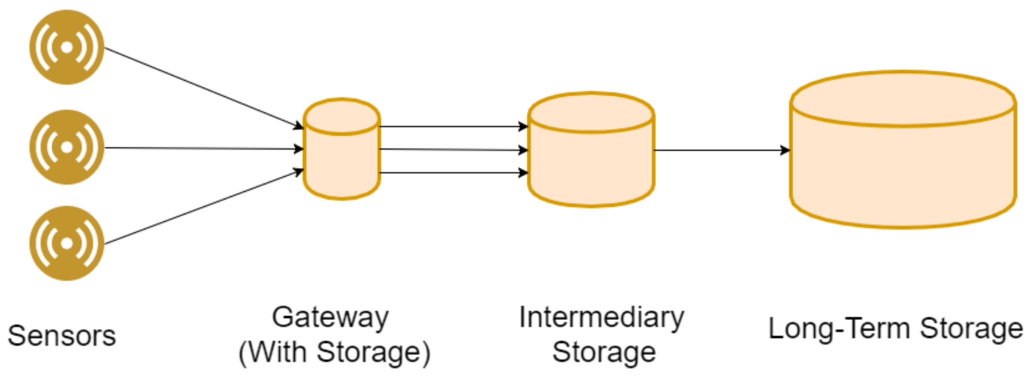


Figure 2.1: Pipeline architecture

The following section provides details on the design and implementation of the proposed IoT data delivery pipeline.

In the context of a smart grid it is common to have a setup in which sensors are not directly connected to a data center to deliver the generated sensor data but connected to a gateway, that is close to the site of where the sensors are deployed. Also the gateway can then be connected to another intermediary hop, forming a so called "pipeline" to deliver the data to the data center. In such a scenario a sensor will send its measurements to the first gateway of the pipeline. The gateway will receive the message, maybe perform transformations to the data like aggregation, and forward the measurements to the next intermediary node in the pipeline. The intermediary node might also transform the data and forward it to the next hop. This process will repeat until the endpoint of the pipeline, the data center, will be reached.

For that reason an implementation for such a pipeline has to be applicable not only to be run in data center but also on devices with limited resources available. This generates requirements like, being able to run on limited resources, as well as also performing well on limited resources. More over limitations in bandwidth, of the links between the different gateways in the pipeline should be considered. Figure 2.1 provides an overview on the given pipeline architecture. Moreover should the gateways and intermediary nodes provide a suitable amount of storage to be able to deal with outages of a subset of the nodes in the smart grid. This should prevent data loss in a big scale and ensure that the pipeline and especially the smart grid, can remain operational in case of any partial outages. For that purpose also different and limited storage capabilities have to be taken into account when designing and choosing the database for the different types of nodes throughout the pipeline. The following will therefore provide a description of the technologies that are used for the first prototype implementation to meet all the different requirements and provide a description on how the components are implemented.

2.2.1 gRPC

GRPC is an open source remote procedure call(RPC) framework, that is built on top of HTTP2 connections and using Google Protocol Buffers for serializing the data that is transmitted and as interface description language. It provides an abstraction of the interfaces that have to be implemented and enables users to use different programming languages to implement their services. Many popular programming languages are supported, like Python, Java and C++. It also supports different platforms and can run in different environments, including data centers but also smaller and mobile devices. In addition to the afore mentioned gRPC provides support for bi-directional and client streaming RPCs and promises high performance. All the mentioned points make gRPC a suitable framework to choose for our pipeline implementation, since it promises to provide high performance in terms of throughput as well as latency, ease of use to the support of many different programming languages and the use of Google protocol buffers. The following section describes Google protocol buffers in general, how the developed message protocol for the pipeline implementation looks like and how the services are defined [3].

2.2.1.1 Google Protocol Buffers

Google Protocol Buffers are a language and platform independent method to serialize structured data and generate the needed code for many different programming languages. It supports Python, Java, C++ and others. The message format has to be defined in a .proto file following the Protocol Buffer Language. After the .proto file is created, the code in the desired language can be generated. Google Protocol Buffers promise to be fast and small in size, and compared to XML offer 20 to 100 times faster serialization and 3 to 10 times less size [2]. The following will provide an overview on how the developed message format for the given task looks like.

2.2.1.2 Message Format

In order to be able to make use of the sensory data for the services needed to be deveolped for the given task, a message format had to be designed. The available parameters generated by each sensor in the smart grid was given. These parameters are:

- Meter ID, an unique ID of a sensor
- Metric ID, an unique ID to define the metric of a value
- Timestamp, a timestamp with the time the sensor value got created
- Value, a sensor measurement value

To reflect the parameters generated by the sensors, the measurement message format is created in the .proto file as follows:

```
message measurement_message {
  string meter_id = 1;
  string metric_id = 2;
  int64 timestamp = 3;
  double value = 4;
}
```

With that all the needed parameters generated by a sensor are reflected in our message format. In addition to that two further parameters are needed, taking the proposed Provenance API into account. The additional parameters are:

- Provenance ID, an unique ID to identify a specific provenance datapoint
- Context Parameters, a string containing all enabled context parameters

To reflect these parameters as well, the following message format was created:

```
message Grid_data {
  measurement_message measurement = 1;
  string prov_id = 2;
  string context = 3;
}
```

So the Grid_data message contains a message following the measurement_message format and the needed parameters for the Provenance API. Note that each field in the message is optional by default, so not every field in a message have to set. In addition to the messages defined, a reply message was defined, that only contains a response code that follows common HTTP Status codes.

```
message reply {
  string response_code=1;
}
```

With the defined messages, all the needed parameters originating from the sensors and the parameters of the Provenance API can be reflected.

2.2.1.3 gRPC Service Definition

In addition to the message format, a service definition, so an interface for the pipeline components has to be defined. To provide a preferably adaptable interface for all types of gateways between the sensors and the Long-Term Storage, one service was defined as follows.

```
service gateway {
  rpc push_data(stream Grid_data) returns (reply) {}
}
```

With that the gateway service is defined, which exposes one RPC called push_data. The push_data RPC expects an input stream of messages of type Grid_data and responds with a message with type reply. This means that the service is receiving a stream of messages, so several messages of type

Grid_data at once. The decision to implement it in this way was made due to the fact that otherwise the reply would have to be sent after each received message, which in return would increase network overhead. The benefit of using gRPC for the definition of the message format and the interface is that with the defined .proto file, the needed serialization code and code stubs to implement the RPC service can be generated in the programming language one desires. It was also the basis for the sensor data emulator described in the following section.

2.2.1.4 Sensor data emulator

Based on the already described message format and the service definition, we were provided with an emulator to simulate the generation of sensory data. It was used to simulate the sensors in a pipeline scenario.

2.2.2 Local Storage

As already mentioned, a way to store measurement messages on each node of the pipeline locally, is needed. The chosen database has to be able to run on all the different types of nodes, so it has to be able to run on smaller and less powerful gateway nodes and more performance offering nodes as small servers or in data centers. More over it has to provide a high write performance to be able to deal with a high load of receiving messages and therefore write operations. It also has to be scalable, lightweight and quick to respond. In addition to that and to be able to deal with the use case specific problem of outages in a smart grid, the database also has to offer persistent storage.

2.2.2.1 Redis

Taking into account the data model of a measurement message, the decision was made to use a key-value storage database. In addition to that all the listed requirements have to be met, and especially keeping in mind the need for a database coping with different hardware configurations, the key-value in-memory database Redis was finalized as database to be used. Redis meets all the requirements such as high performance on write operations. Even though it is an in-memory database it also provides the possibility of being persistent. Snapshots and write operation logs, can be used to provide persistence [4].

The following section will provide an overview on the implementation and explains the most important parts of it in detail.

2.2.3 Implementation

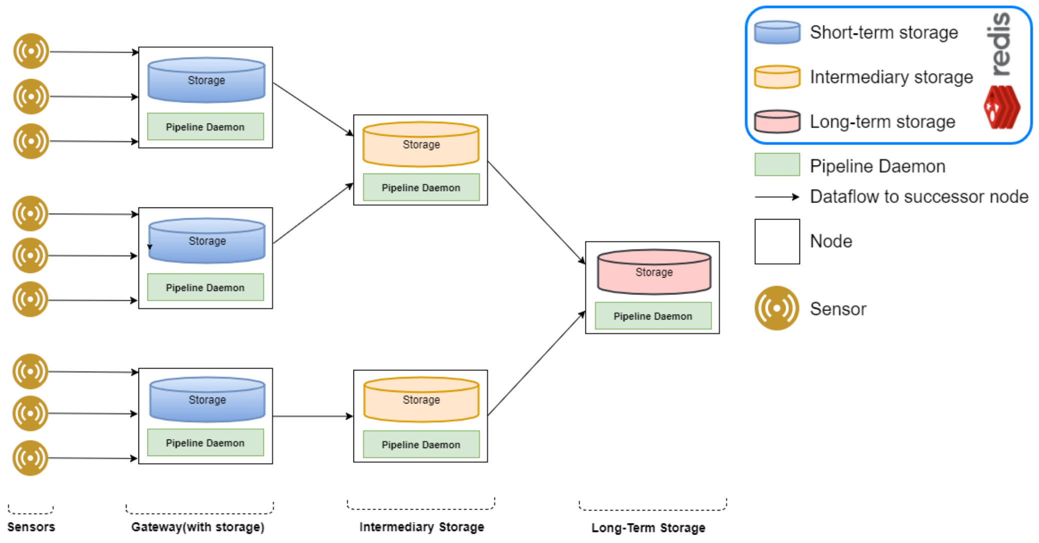


Figure 2.2: Pipeline architecture

In this section a detailed description and explanation of the pipeline implementation is presented. The pipeline is implemented in the programming language Java. For the build management, Apache Maven is used. The Java project is organized as follows. The code for the pipeline component resides in the PipelineComponent class. In addition to that the PipelineInterfaces.proto file provides the already described service and message definitions used to generate the serialization and service stub code.

First of all, the system architecture is visualized in figure 2.2. One node in the pipeline consists of a database to provide local storage and a Pipeline Daemon. The Pipeline Daemon is used to implement the already described service gateway.

2.2.3.1 Pipeline Daemon

The generated gRPC code was used to implement the gateway service and its RPC push_data. The different types of nodes, that can be part of the pipeline, can be created when starting a component, by the use of different run configurations. A Pipeline Daemon can either be run as a gateway or as an endpoint. For that purpose a set of configuration parameters are provided. The following parameters are available:

- port, the port used by the server of this Daemon

- port_next, the port of the next hop in the pipeline
- host_next, the hostname or IP of the next hop in the pipeline
- location, a label for the current physical location of the node
- storageTime, an integer value representing the time messages are stored in seconds
- propertiesfile, specifying the path to the Provenance Daemons properties file
- no_prov, flag that can be set to turn of the collection of provenance data and the use of the provenance system

If the parameters host_next and port_next are set, the Pipeline Daemon is recognized as a gateway. On the other hand if these parameters would not have been set, the Pipeline Daemon is recognized as endpoint of the pipeline, since there is no successor. In addition to implementing the service gateway, the Pipeline Daemon implements a client, based on the push_data interface definition, to be able to make use of the gateway services of other components in the pipeline. This is the case if the Pipeline Daemon is started as gateway. Then it has to be able to receive the messages from another clients and forward them to the next hop in the pipeline.

The gateway service provides the push_data service, as mentioned. The push_data service is responsible of processing the messages a client sent. The client can send messages one by one, or send them as a stream of messages, so the client can send many messages at once. If the client sends a message and the push_data service is called, it will process the message. The push_data service is responsible for collecting the needed context information to feed to the Provenance Daemon and provide information, e.g. how many messages are received and sent per second. In case of receiving a stream of messages the push_data service will cache the received messages and the generated context information. If the stream exceeds the number of 10 messages, the cached context information will be already forwarded to the Provenance Daemon and the measurement messages will be sent to the next hop, while still receiving the stream of messages. If the stream the client sends contains less then 10 messages, the push_data service waits for the stream to be finished, before forwarding the context information to the Provenance Daemon and forwarding the messages to the next hop in the pipeline. This behavior was implemented due to the fact, that it is possible to use one single stream to send a large number of messages. If the Pipeline Daemon would wait until all messages arrived it is possible that additional delay is introduced and the latency increases. More over it would also result in irregular peak workload on the network, which is prevented when splitting up the incoming stream.

Additionally there is the possibility that the no_prov flag was set on startup of the Pipeline Daemon. In this case the push_data service will not collect any context information. It will just cache and forward the measurement messages. The behavior of caching messages if the stream exceeds the number of 10 messages, remains the same. More over the provenance daemon will not be instantiated.

For the purpose of providing information like the number of messages per second to the developed user interface and to capture the health of the pipeline component daemon, additional methods are implemented by the push_data service. A counter in combination with a timer is used to capture the number of messages. The advertisement of the values to the user interface is handled by the Provenance API. The node health is for one implicitly set when interacting with the provenance API, and for the other it is set by the use of a timer if the push_data services is not interacting with the Provenance API.

2.2.3.2 Local Storage

For the use of the local storage, so the Redis database, a Redis client is implemented with the Pipeline Daemon. The push_data service will, while processing the messages, save the measurement messages in the local Redis database. If the Provenance Daemon is active the Provenance IDs, that are received after the created context datapoints are handed to the Provenance Daemon, will be used as the keys for the measurement messages. This will also ensure, that in case of a failure the original messages can be retrieved from the local database and linked to the provenance entries in the Provenance DB. In contrast to that the current system time in nanoseconds is used if the no_prov flag has been set. The measurement messages are saved as hashes, so with one key a whole measurement message can be saved.

The implementation of a Pipeline Daemon and the gateway service that is used for all different types of nodes, remains the same. From service implementation point of view it will only be differentiated between gateways and endpoints of the pipeline, so if messages will be forwarded to another hop in the pipeline or not. Only the amount of local storage provides a differentiation between different nodes. To be able to reflect this in the implementation, the storageTime configuration parameter is used. The parameter is used to simulate the different amount of storage available on the different machines that are part of the pipeline. A timer is used along with the specified time of the storageTime parameter so that after the timer expires the database will be flushed. With this it is ensured to reflect different storage capabilities with the Pipeline Daemon prototype implementation.

2.3 Data Provenance Model

In this section, we shortly explain some of the existing provenance data models and the need for a new data model, then we explain a new data model specialized in the IOT context. Our data model is inspired by the Data Provenance Model[1].

2.3.1 Existing Data Models

2.3.2 New Data Model

We define a data point (DP) as a uniquely identifiable and addressable piece of data (i.e., a value) in the context of the smart grid. Examples for DPs in the context of smart grid includes sensor readings such as 3-phase electric currents, complex analytics results derived from sensor readings etc. The unique identifier is composed of three blocks.

- Unique Identifier - A data point distinguishes itself specifically from other data flowing in the Data Provenance Model for smart grid (e.g., bulk sensor readings that are of no further interest, ephemeral intermediary analytics results, etc.) in that it is addressable, i.e., it has an ID that is unique in the context of the smart grid. In order to ensure the uniqueness of the identifier, we came up with an identifier generation strategy, which generates a unique ten-byte identifier and ensures uniqueness across the system. Data point identifier comprised of:

  - 3-byte node identifier - Guarantees its uniqueness across machines/nodes and processes.
  - 4-byte value representing the seconds since the Unix epoch - Ensures uniqueness in relation to a single second.
  - 3-byte counter - Provides uniqueness within a single second in a single process.

[1]`https://link.springer.com/content/pdf/10.1007%2F978-3-319-68136-8.pdf`

Properties

Our unique identifier mechanism along with its simplicity brought some other advantages and reduced the need to store machine/node identifier separately and also ease some time-based queries (e.g., sorting based on generation timestamp).

- Example: 5a7b91370003c6badfb2

- Input Data Points - A data point may be based on other data points that have contributed to its creation or modification. We refer to these related data points as input data points(IDP). A data point's IDP is a list of unique identifiers of all those data points, which contributed to its creation or transformation. It also stores information about the contribution type.

  - Example:

```
[{
    "average": [    // Contribution type
        "5a81c07800031ddaf123",
        "5a81c093000a1d341fab"
    ]
}]
```

- Context - Specific context for provenance may vary for different IoT applications. We propose a data model for the context in typical IoT environments comprising the concepts of Agents, Execution Context, as well as Time and Location information (cf. 2.3). An Agent is an entity that creates and/or modifies data points (e.g., sensor, device, software agent, etc.). It is recursively defined in such a way that an agent can contain other agents (e.g., a device containing several sensors). This recursion allows for defining agents in a hierarchy and may be used as fine-grained as required. For instance, an agent hierarchy may span from the concept of a particular function in a software library running over a virtualization container on a particular device to a particular IoT network. Execution Context provides information related to the provenance event at runtimes, such as events or data points that triggered the creation/modification of the data point. Time and Location information is also added to the provenance information. We chose two to three specific metrics for each of the above-mentioned context parameters based on our use-case and these metrics are listed below (cf. 2.4):

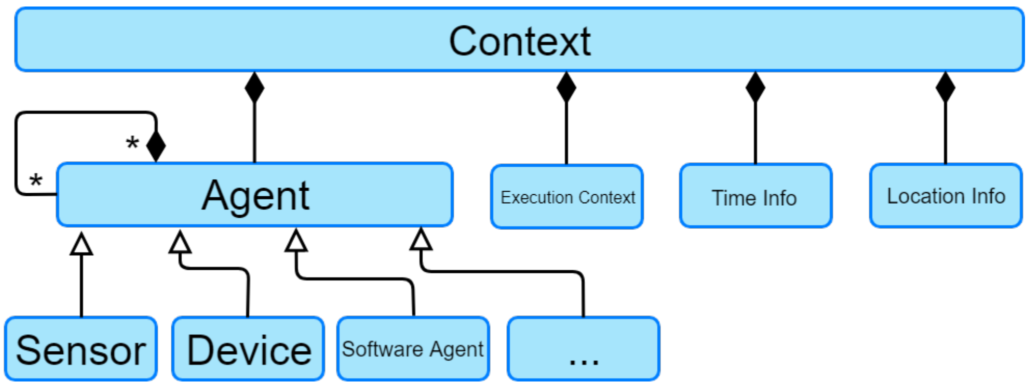


Figure 2.3: Data Model Context

- Node/Machine identifier
- DP creation time
- DP send time
- DP receive time
- Application name
- Class name
- Location
- Line of code
- health status
  - ∗ Node/Machiene
  - ∗ Channel
  - ∗ Provenance Daemon
  - ∗ Pipeline Daemon

We categorize these metrics into two main categories: core metrics and use-case specific metrics. Node/Machine identifier, DP creation time, DP send time, DP receive time and Location are the core metrics of our provenance system. All other metrics are use-case specific: Application name, class name and code line number are for debugging and tracing use-case while health status is for error detection use-case.

TODO

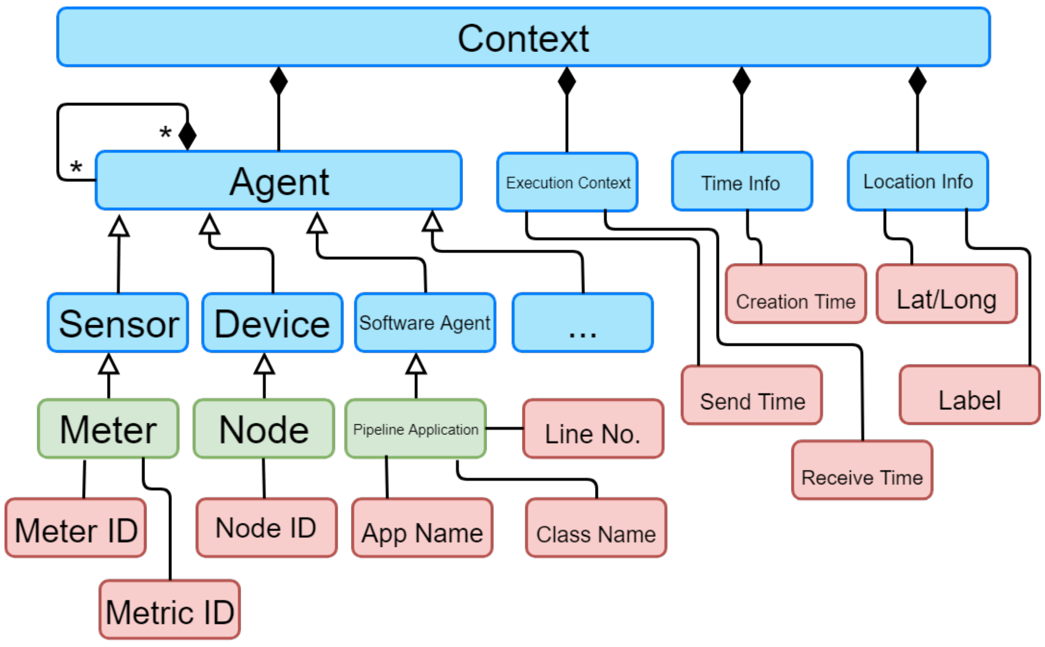


Figure 2.4: Use-case specific Data Model Context

2.4 Provenance API endpoints

In this backend section, we will be explaining about the Provenance API and the technologies used to develop the same. Provenance API is developed mainly as an intermediary between the provenance Database and the frontend. The frontend will be issuing the GET requests to the API endpoints defined by our Provenance API backend.

2.4.1 Technologies Used

- Developed in Java
- Maven project
- Spring-Boot
- Configuration location using environment variables
- Cassandra integration

2.4.2 API Endpoints

We will be explaining the endpoints and the information provided by them,

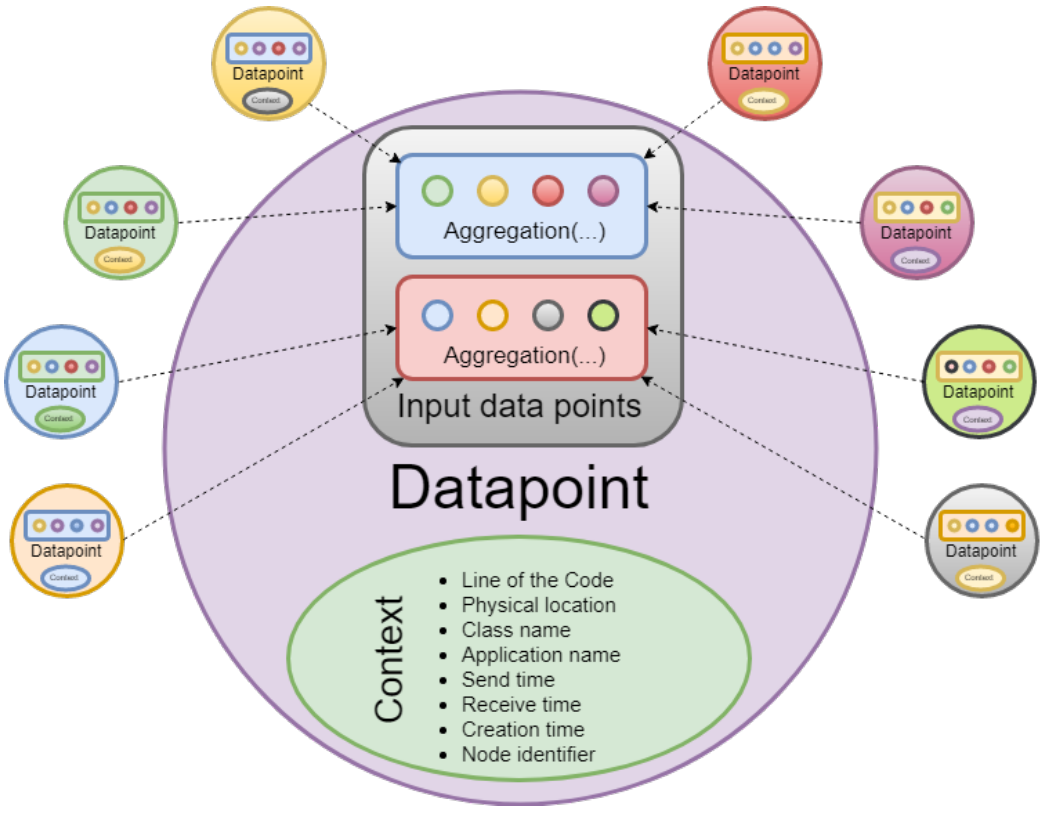


Figure 2.5: Data Model

2.4.2.1 /cluster/stats

To present the visualization of the current running system to the end users, the front end will be querying this /cluster/stats endpoint and get the real time status about each nodes and also the health rate about the nodes.

2.4.2.2 /cluster/topology

The end users will also be presented with the graph tree like structure of the running provenance system. This visualization will show the each node and also the edges pointing to the next successor node. This endpoint provides a list nodes and their successor in JSON format to the querying frontend.

2.4.2.3 /cluster/nodeId/provenance

This endpoint provides a JSON response about all the provenance datapoints that are specific to a particular node in the provenance system. In the frontend, the end user will be clicking over the node presented in the visualized topology, and a table will be presented below with the list of provenance data pertaining to that particular node.

2.4.2.4 /provenance/id

This endpoint provides the all datapoints that contributed to the creation of this particular datapoint. The field inputDatapoint provides that information. JSON response contains a nested loop like structure which embeds all the datapoints related to this datapoint.

2.4.2.5 /provenance/id/static

This endpoint provides the all datapoints that contributed to the creation of this particular datapoint, but in a downloadable JSON format.

2.4.3 Swagger API documentation

After we have defined all the endpoints in the backend, we have used the Swagger API to provide a easily communicable way of explanation about our endpoints.

Moreover, every change in the API should be simultaneously updated in the Swagger API documentation. Accomplishing this manually is a tedious process, so automation of the process was inevitable. To provide this automation we have directly integrated swagger configuration to the spring boot application.

We have used the Docket plugin for the spring boot and defined the configuration by creating a SwaggerConfig.java file and once building this changes will enable a new endpoint /swagger-ui.html , this file provides required Swagger API documentation.

2.5 Provenance Database

Database is at the heart of every architecture especially when there a focus on the data. Choosing the right database goes a long way and the effect of this will be shown in each and every component of the architecture. In IoT domain where the data ingestion is huge and the data needs to query in real-time having a particular set of queries from where the user can query the dataset makes the decision more difficult.

In order to come up with the suitable database we underlying few basic factors which play an important role in finding the perfect match for the our system.

Size of data to be stored This factor considers amount of data stored and retrieved. So we choose the database keeping in mind the overall volume of data generated by the application at any given time and also the size of the data which is retrieved can handle efficiently.

Speed and Scalability This factor considers the speed of reading data from the database and writing data from the database. Some database focuses more on the read-heavy application, while others are designed to cope with the write-heavy operation on the application. This factor is important to our case as we need to consider the more writes of the database when adding new resources resulting in scaling the database into multiple instances.

Accessibility of data Accessibility of data is important as the user of the system need to access the information as soon as possible in it stored for querying. The database needs to handle concurrent writes by numerous component in the architecture connected to the database. The effect of a huge load of writes should be minimum and there should be a mechanism in place which handles fault tolerance in the database. This will ensure all the provenance information is safe and is not lost forever.

Data modeling The structure is the core component in choosing the right database for the application. As the data from the sensors and the provenance information vary a lot. The data model is not specifically fixed in hard boundaries but can be capped under major categories. The database must be able to handle changes in the data model and adapt to the structure quickly efficiently.

Safety and security of data Storing information about the data from the sensor deployed on the ground capture various metrics which are confidential and contains information which is critical. In order to preserve the confidentiality and secrecy, the database must handle some level of security. The safety measures implemented by the database in case of any system crash or failure is quite a significant factor to keep in mind while choosing a database especially in our case where data lost means permanent loss of information.

2.5.1 Apache Cassandra as Provenance Database

After thoroughly considering a numerous number of databases from the pool we decided to use Cassandra as the Provenance Database for our system. Apache Cassandra is an open source, distributed, massively scalable

NoSQL database. It is designed to handle large volumes of structured, semi-structured and unstructured data across multiple data centers, and it supports the cloud deployment. Cassandra offers capabilities like continuous availability, linear scalability and operational simplicity across many commodity servers with no single point of failure. Its powerful, dynamic data model is designed for maximum flexibility and fast response times.

Apache Cassandra supports configured consistency levels to manage availability versus data accuracy for writes-heavy demand in IoT domain. Data is compressed up to 80 percent without any performance overhead this can lead to storing more volume of data in less amount of space. Data is distributed across the cluster and there is no master node in the system so each node can service any request. The distributed architecture is perfect for disaster recovery, redundancy and failover as the data are different to create from the start.It can handle massive data sizes and scale out to large clusters.

Apache Cassandra offering continuous availability, high scalability and performance, strong security, and operational simplicity. It has flexible data storage which easily accommodates the data in various formats and structure. Changes can be made dynamically to the data structure as per requirement.

Apache Cassandra outperformed other NoSQL database in the benchmark using YSCB and was also run on the sample dataset of our data model and performed fairly in term of writes. Apache Cassandra is the best choice for our scenario as it fulfills all the of providing a high write performance with a high load of receiving a message from a different component of the pipeline and the data is quickly available for the user to query. For IoT domain scenario, most of the nodes are distributed in numerous location so deployment of Apache Cassandra at different geographically distributed region could help in increasing the performance and also for faster retrieval of data.

Apache Cassandra Query Language The Cassandra Query Language (CQL) allows you to query Cassandra using queries similar to SQL.CQL commands include data definition queries (e.g., create table), data manipulation queries (e.g., insert and select for rows), and basic authentication queries to create database users and grant them permissions.This fulfills the requirement of the user querying the provenance database for retrieval of provenance information.

TODO

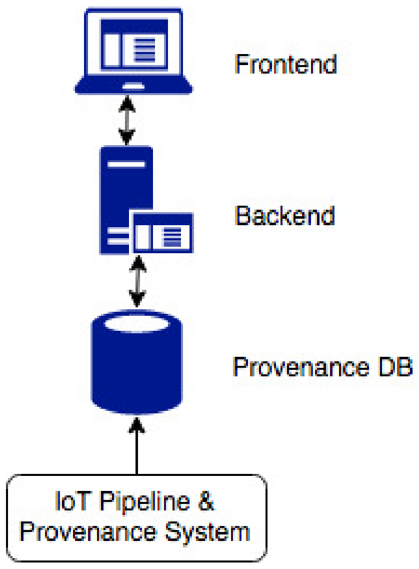


Figure 2.6: This is three-tier architecture.

2.6 User Interface

Today most IS systems are based upon a Client-Server architecture, a three-tiered approach. The server performs the grunt of all operations where as the client side is more concerned with how to present the data. The processes that go on in these tiers can be represented by many more tiers, for instance a database tier can be added, where all data is stored can be represented by the diagram below. However it is important to note that more than one tier can be stored on one server, for instance both the web server and database servers each represent there own tiers in the diagram below, but in reality they could be seen as one.

To implement a web application for Error detecting and monitoring system, client-server architecture is required. The most popular client-server architectures are the two-tier and the three-tier architecture. The choice of architecture affects the development time and the future flexibility and maintenance of the application. While selecting the architecture most suitable for an application, many factors including the complexity of the application, the number of users and their geographical dispersion are considered. This system is designed based on a traditional three-tier architecture used by many web applications. Three-tier architecture includes a presentation layer, business rules/ logic layer, and the data layer (provenance database). The three-tier architecture is shown in Figure 2.6.

Three-tier architecture gives an enhanced security, through the implementation of several layers, enhances the data security on a service-by-service. As client do not interact with the database directly, it provides less risk and conflicts with unauthorized data. Also it improves the data integrity as data corruption through client application can be eliminated as the data is passed in the middle tier for database updates ensures its validity. Due to distributed deployment of application servers, scalability of the system is enhances since a separate connection from each client is not required whereas connections from few applications servers are sufficient.

The three-tier architecture is generally used when an effective distributed client/server design is needed that provides

- increased performance
- flexibility
- maintainability
- re-usability and
- scalability

This model hides the complexity of distributed processing from the user. These features have made the three-tier architecture a popular choice over the two-tier architecture for Internet applications. The three layers are discussed below.

The **Data layer** is responsible for data storage. Primarily this tier (layer) consists of one or more relational databases and/or file systems.

The **Business Rules/Logic layer** is the middleman between the presentation layer and the data layer. This middle tier was introduced to overcome the deployment limitation (whenever the application logic changed the application had to be redistributed at each and every client) in the two-tier architecture. The middle tier provides process management where business logic and rules are executed and can accommodate hundreds of users.

The **Presentation Layer**, also called the Client tier, is responsible for the presentation of data, receiving user events, and controlling the user interface. The user interaction with the system is entirely through this layer.

The three-tier architecture for Smart-grid IoT provenance system's

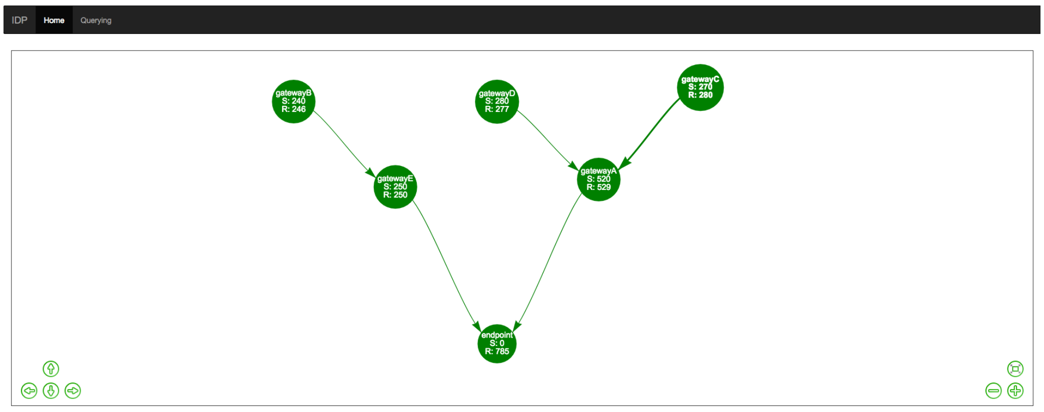


Figure 2.7: Dashboard

Error detection use-case. The data layer in this architecture comprises of Provenance Database of the system, in this case here we have used Cassandra. The logic layer/business layer is defined by Java's Spring boot framework and provides REST component to the Presentation layer and the messages are communicated by JSON. And the Presentation layer where the data is represented to the end-user is developed in Node.js Express framework with Jade view template engine.

2.6.1 Frontend

The requirement to have Frontend according to the usecase was questionable, but the system being able to provide the error detection and also monitoring of the system emphasized us to have user interactive application for monitoring, error identification and also provenance system data visualization. Thus, we developed an tailored frontend system satisfying actors like system administrator, analysts, developers, etc. Frontend represents the client-tier from the three-tier architecture. Its a loosely coupled service from the system giving enhanced security to the security and scalability.

The frontend application gives user a very interactive dashboard. The above figure 2.7 shows Dashboard page when the end user access the system. The dashboard is very informative and helps the user to visualize the components in the provenance system pipeline. It gives a detail view of nodes and all related information about that node.

2.6.1.1 Technology Used

The frontend was developed with in-demand market technology like Javascript scripting language, Node.js, Express framework.

Javascript It was originally named Mocha is a "multi-paradigm language, supporting object-oriented, imperative, and functional programming styles." [1] JavaScript started on web browsers as language to dynamically interact with the user and control the components of the page. JavaScript's language architecture is quite unique from other languages which is especially handy when developing web application that rely on non-blocking operation.

JavaScript architecture combines object-oriented, functional and scripting languages paradigms. Syntax wise, JavaScript is based on C's structured layout, with subroutines, block structures and loops. While it is object -oriented, JavaScript does not have classes, but instead rely on object prototypes for inheritance. This means that the methods and fields of the objects' prototypes may be dynamically changed for future construction of these objects. Functions in JavaScript are also objects, and can be sent as arguments to other functions as the case in functional languages; functions can also have methods and properties of their own. JavaScript also provides dynamic typing as well as static typing. Variables could be defined with a specific type such as Number of String, or casted dynamically by initializing it as a var. Moreover, JavaScript, like scripting languages, allows the use of variadic functions (number of arguments is not defined), and supports Regular expressions such as in Perl. Finally, JavaScript's support for associative arrays is the basis for the construction of JSON data formats. Using a single language throughout the stack enables the reuse of resources such as JSON objects which can be manipulated the same way by any part of the stack.

There were several attempts aiming to place JavaScript on the server side ever since the mid-1990s. But, none of these solutions were as successful as Node.js which is pioneering a new way of server programming focused mainly around asynchronous operations. There are three main reasons that made JavaScript the language of choice of Node.js

1. Google's V8 is an open-sourced high-performance JavaScript execution engine built for Chrome. It complies JavaScript to native machine code instead of interpreting it in real time. V8 and other JavaScript runtime engines also provide a concurrency model using a message queue and an event-loop which allows JavaScript to stack operations and their callbacks and execute the callback when the operation is done.
2. JavaScript is built around an event-driven interaction model because it depends on user actions, which makes asynchronous, non-blocking and callbacks natural, as they are event driven as well.
3. JavaScript is an interpreted language which makes it platform independent.
4. JSON (JavaScript Object Notation). JavaScript has been the language of choice to control the web for a long time; and since the old days, data had to be marshelled into JSON objects when sent to the web. Because of the high dependence on JSON objects, a new 7 kind of JSON like databases were created, enabling the easy exchange of data between back-end and front-end.

Node.js Node JS is a JavaScript platform for building fast, scalable, network applications built on Google's V8 Engine. Node is single threaded and built around the paradigm of none-blocking IO. With Node.js each incoming request by the user is handled by one single thread in opposition to the multi-threaded techniques used by PHP to scale the operations. Each request handled by this thread is coupled with a callback function that is called upon completion of the task. This is possible due to the fundamental support of JavaScript for events, Asynchronous operations, and callbacks; and Node.js puts JavaScript on the server side. Node has several advantag es, some of which are:

- RESTful API. As Node can build an HTTP server out of the box, it can communicate with all other components through HTTP methods for CRUD operations based on the RESTful paradigm.
- Single Threaded. Since it is not blocking I/O operat ions, Node can handle all user requests using a single thread, instead of allocation of new thread for each request, which has a large memory footprint. Nonetheless, Node does use a thread pool at the kernel level to guarantee that the operations are being executed asynchronously without blocking the event-loop. This is necessary because the kernel does not support all operations asynchronously.

NPM he Node Package Manager is based on JavaScript's npm. A built-in module that supports package management, it can be used to easily download and install modules for a Node application. Moreover, Node already has many packages and libraries developed to work on top of it; all of which confine to the asynchronous nature of Node.js.

Express Express is a fast, un-opinionated, minimalist web application framework for Node.js. It is designed for building web applications and is the de facto standard server framework for Node.js. Early 2016 the Express.js project was brought into the Node.js foundation as an incubating project. Express was itself a project in node generation which was split into three Github organizations called expressjs, pillarjs and jshttp. Express is a simple routing plus a sugar layer built on top of the actual base Node.js HTTP server that helps manage a server and the routes. It gives declarative routing without making s witch statements or i f statements or big functions, into a basic middleware pattern. [2]

One should already be familiar with JavaScript syntax and have Node.js installed to get started. There are many reasons why Express and Node are great choice for server side development.One major reason is that since Express is built on top of Node it is the perfect framework for ultra-fast input and output. Node.js is both asynchronous and single threaded- many requests can be made simultaneously without incurring of bottleneck that would slow down processing. The robust API that ships with express allows us to easily configure routes to send and receive requests from a front-end and connect to a database. [2]

Bootstrap Styling Framework This framework has become one of the most popular framework and used by popular website. This framework is a lightweight and modular front-end framework [3]. It provides a powerful web interfaces for a fast development.

Vis.js A dynamic, browser based visualization library. The library is designed to be easy to use, to handle large amounts of dynamic data, and to enable manipulation of and interaction with the data. This library enabled use to visualize the nodes in the pipeline. and it was represented as DAG (Directed Arrow Graph). This library is licenses under Apache 4.0 and MIT [4].

2.6.1.2 Design

In this section we would discuss the design of the frontend. We used bootstrap with handlebars as our main UI design kit. Pug is a very powerful templating framework, alongside with bootstrap it gives a lot of flexibility with design. The styling and the javascript component integration made it the best choice for us to use it in our project. Bootstraps modular structure and less complicated stylesheet function, made it quite simpler for us to work on. Below are the list of pages and its corresponding functionalities: 2.7

- **Home Page**: All the nodes are visualize in network topology using DAG (Directed-Arrow-Graph). Each node in the topology is has a predecessor and a successor node. The nodes in the topology shows relevant information for that particulate node and its running status. The nodes are represented as a circular design which encapsulate further details of the node. When user clicks on one of the nodes it shows last 50 provenance data generated in a tabular form. Navigation bar in home page only shows two options (i) Home (ii)Querying

- **HomePage/Node Click**: Clicking on a particular node with retrieve last 50 provenance data points generated during provenance capture.

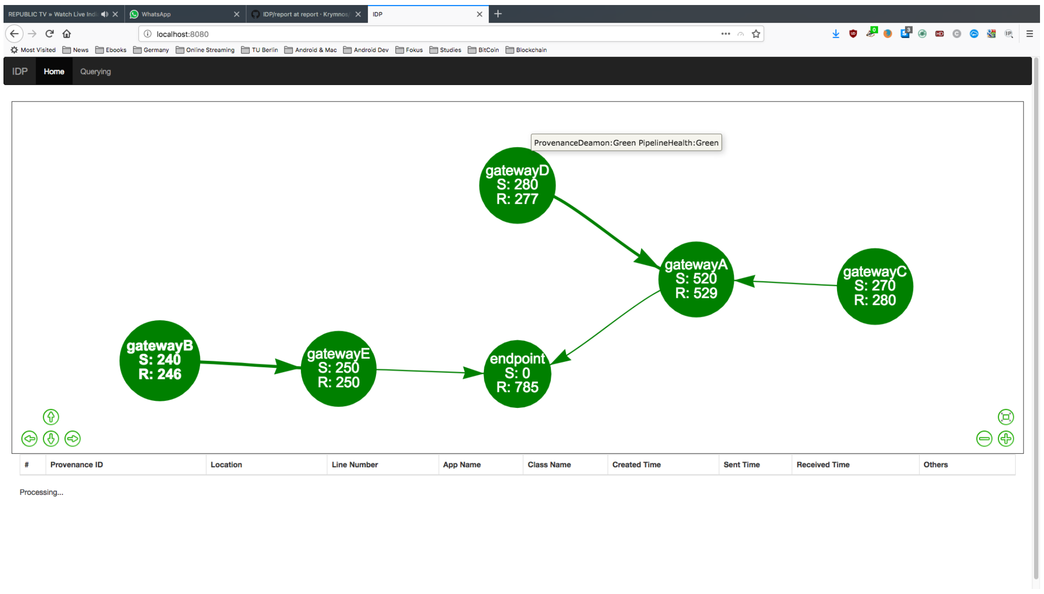
- **HomePage/Visualize Provenance Lineage**: When the user performs click action on the button, it redirects user to a new page where the provenance data is plotted as a DAG(Directed-Arrow-Graph) along with all relevant context information.

- **Querying**: This page displays an 'cql' editor which is a Cassandra Query script where an analyst or an system administrator can write queries to our Provenance Database according to his needs. The result are visualized in 2 forms (i) A tabular form (ii) JSON format.

- **Querying/Submit Query**: Once the end-user writes the query in the cql editor and presses the Submit button, then user is presented with the query result in the form of table.

- **Querying/JSON Query**: Once the end-user writes the query in the cql editor and presses the Submit JSON button, then user is presented with the query result in the form of JSON format.

Node Visualization In the figure 2.8, it shows visualization of the nodes in the form of network topology in our Smart-Grid system. The node defines many relevant information like its preceding and succeeding nodes. The



#	Provenance ID	Location	Line Number	App Name	Class Name	Created Time	Sent Time	Received Time	Others

Processing...

Figure 2.8: Node Topology Visualization

inward arrow to the node defines its preceding node and the outward arrow from that node shows the succeeding node. Each node visualize in the circular form, and encapsulates node details like:

1. node name,
2. message send rate,
3. message receive rate.

Nodes are further more defined by colors. Node colors can be of 2 types (i) Green and (ii) Red. The Green color of the node denotes the health of the node which means the node is working as it should be. Red color denotes that the particular node is not working which helps the end user to come to a conclusion that there are some issues with node and its not working as it it supposed to. The directed edges also denotes by the color Green and Red. It shows the status of Pipeline Health giving further more granular detail about error in the pipeline system.
Many action can be performed on the node visualization. On clicking of particular node it further retrieves provenance data that were generated. The provenance data is displayed in the form of table with all context information like Provenance Id, Location, ClassName, LineNo etc. The table houses 3 buttons, (i) Visualizing the provenance data generated along the pipeline, (ii) downloading the provenance lineage in JSON format, (iii) downloading

provenance data recursively in JSON format.
Further on clicking on the Visualize Provenance data button, a the control redirects to a new page which visualizes the provenance data across the pipeline and showing all relevant context parameters generated during provenance data capture.

2.6.1.3 Frontend Requirements and Setup

TODO

2.7 Testing and Deployment

As discussed in the organization section, we decided to assign responsibilities to team members for different parts of the implementation. This leads to an independent implementation workflow for every component with the need to put them all together at some time - not only for a final deployment, but also for development of the dependent components (for instance the pipeline component that use the Provenance API). To realize a fast development of dependent parts and produce executable artifacts at any time (as it is usual for a Scrum schedule), we decided to implement a Continuous Integration Workflow based on the commits that were made to the respective Repositories. Each commit was automatically tested (at least whether it is buildable) and, if it was desired, deployed to a Docker or Maven Repository. The current build state of each branch was evaluated immediately after a push so that the code health could be checked by everyone at any time. On this way, other parts of the project could include the artifacts by using explicit version tags or *LATEST*-version of the repositories. In the first part of this section, we will describe in detail, how we implemented our CI workflow. In the second part we describe how we used these artifacts for a deployment on AWS[2] and which additional adjustments were necessary to get a larger deployment with individual sensor workloads.

2.7.1 Continuous Integration & Delivery

Travis CI We used Travis CI[3] as Continuous Integration Solution that is free for Open-Source Projects. Travis builds were triggered on changes for every branch of the repository that contain a valid *travis.yaml*. The testing configuration was set inside of the *travis.yaml* that was present in each

[2]We used the AWS deployment also for our benchmarks (**??**).
[3]`https://travis-ci.org/Krymnos/IDP`

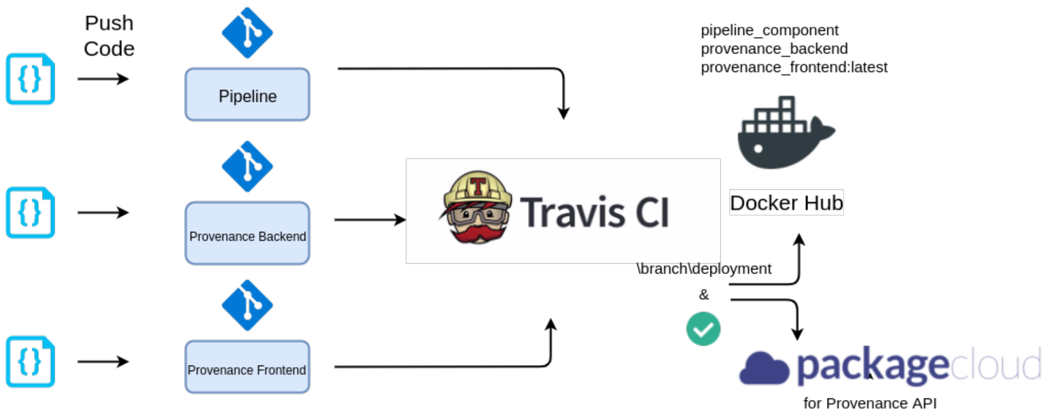


Figure 2.9: Deployment Pipeline

repository. Some components needed additional configuration to get builds and tests running. For instance, the Pipeline Component needs the protobuf executables to generate the communication interfaces during a build. After the compilation is finished, the unit tests of the Pipeline requires a running *Redis*-Database as local storage. All these configurations could be made by the `before-script` and `before-install` sections inside the `travis.yaml` by using the built-in services of travis[4]. As an example, the travis.yaml for the pipeline component can be found in the appendix 1. At the same file a *deployment*-section was defined and only executed if the branchname is equal to "deployment". For the components that are delivered as docker builds, the respective docker deployment script was executed 2, for the Provenance API that is included by the Pipeline Component, the artifacts were pushed to our Maven Repository[5]. The docker deployment script pushed the docker images to *Docker Hub*-Repository of our project[6].

Docker Images We decided to use Docker for the deployment of our components because a docker container runs platform independent, isolated, lightweight and can be interconnect with other services in a very simple way [7]. The Docker engine (in combination with docker-compose or Docker Swarm) also supports a good functionality to bring a deployment to a large scale. That suites well to our plans for a fast iterative development and the possibility to test our system on different Topologies.

[4]`https://docs.travis-ci.com/user/database-setup/`
[5]`https://packagecloud.io/gerritja/IDP`
[6]`https://hub.docker.com/r/cloudproto/`
[7]`https://docs.docker.com/engine/docker-overview/#docker-engine`

To deploy our components as Docker images, we added a *Dockerfile* to the Sensor, Pipeline, Backend and Frontend components. A Dockerfile[8] contains the instructions that are executed during a build of an image. As well it's defined in a Dockerfile which commands has to be executed on an instantiation of a Container. For the Pipeline Component, for instance, we specified that every instance has a Redis Database running that was started inside the docker container as a local daemon.

To make the Docker Container configurable, we introduced environment variables for the Provenance API and further settings. Also delaying the runtime of the Pipeline Component by setting `STARTUP_DELAY`) is possible because we noticed that we had undesired failures at startup because dependent components were not available due to a longer instantiation phase[9]. As an example, the Dockerfile for the pipeline component can be found in the appendix (3).

Setup Topologies with docker-compose Files For manual testing of our System and running benchmarks (**??**), we defined the individual components and the connection between each other inside a compose file for Docker [10]. The compose files could be therefore used as specification for a deployment with Docker or Docker Swarm.

A simple example that defines a simple pipeline with a workload generator, a gateway and an endpoint can be found in the appendix (4). The topology can be started on any machine that has Docker installed and a docker-compose and/or docker swarm extension.

- command for docker swarm:`docker stack deploy -f <composefile> <stackname>`
- command for docker-compose:`docker-compose -f <composefile> up`

2.7.2 Docker Swarm on AWS

Due to the machine in-dependability of docker containers we're able to deploy our topology locally but also along distributed machines. As *docker-compose* is sufficient for a local deployment on one node, *Docker Swarm* is used to

[8] `https://docs.docker.com/engine/reference/builder/`

[9] for instance, the cassandra database takes usually more time to be available than a lightweight pipeline component

[10] `https://docs.docker.com/compose/compose-file`

distribute the orchestration of the components over multiple machines which are organized within a cluster and running on *swarm mode* [11].

To setup such a cluster on AWS we used a predefined Cluster template for *AWS Cloud Formation*[12] that is provisioned by Docker [13]. A detailed readme, how a topology can be deployed is available on the IDP repository on `deployment/howto.md`[14]

Scaling of Sensors/Workload

For our project we got circa 6 GB of real sensor data with measurements of a power grid. The data contains folders for every day in a year. Each folder contains usually 10 data items (csv files), each is representing a sensor. We wanted to be able to test more than 10 sensors in our deployment.

data preparation We modified the origin structure of the sensor data in a way that one folder contains all files to simulate many more sensors for only one day. The workload generator that sends the data to our pipeline components determines a sensor id by the filename (e.g. `31400010000000000.csv`). We had to rename the files due to the issue of having many duplicate filenames. We numbered the files consecutively to get increasing sensor ids (from 31400010000000000 to 31400010000003473) so that we're able to simulate up to 3474 unique sensors[15].

data inclusion To have the ability to run a sensor container completely without external sensor data, we stored a small dataset (few megabytes) inside the sensor container-image. For our plans to scale sensors with unique sensor data it was obviously not an option to put 6 GB into a docker image. Also the creation of thousands of sensor images (with few megabytes for a unique sensor) makes no sense. Therefore, we stored the sensor data to a *Amazon S3* Bucket and implemented a docker image for the sensor that has the skill to mount bucket inside the local filesystem of the container[16]. An example that can be used inside a docker compose file can be found in the appendix **??**.

[11]`https://docs.docker.com/engine/swarm/key-concepts/`

[12]`https://editions-us-east-1.s3.amazonaws.com/aws/stable/Docker.tmpl`

[13]`https://docs.docker.com/docker-for-aws`

[14]`https://github.com/Krymnos/IDP/blob/master/deployment/howto.md`

[15]`https://s3.console.aws.amazon.com/s3/buckets/provenancesensordata/data/oneday` note: the access to the sensor data is restricted due to privacy reasons

[16]We forked an existing base image to add this functionality `https://github.com/serioja90/docker-goofys`

This approach worked fine for a local deployment on one node by using *docker-compose*. Unfortunately, we had to realize, that this image doesn't work on *Docker Swarm*, because the privileged execution of the docker container is mandatory to mount the S3 Bucket inside. The privileged mode is supported by *docker-compose* but not by *Docker Swarm*[17]. To get a working access to the S3 bucket, we defined an external *docker volume*[18] in the compose file in combination with a docker plugin that links the S3 bucket with the docker volume [19].

scalable sensor groups At this point, we were able to mount the complete sensor data in an efficient way. We still had the problem, that a definition of unique sensors was only possible by adding a service entry for every single sensor to the compose file, because the sensor was explicitly expecting the datapath to the measurement file. Docker provides a scale command [20] that can be used to scale up services of identical instances. To define sensor groups of unique sensor by using this command we implemented another service for sensor-id coordination. This service implements a simple counter that gives the current id as an response of a REST-Request and increments it. We modified the sensor code in a way, that a sensor calls the REST endpoint of the coordinator and adds the received id to the basepath of the sensordata folder on the mounted data[21].

The appendix contains a simple example of mounting the S3 bucket and the id coordinator approach6

Deployment of the complete stack

We created different versions of compose files with different Topologies and all components. The files can be found in the IDP-Repository at: `compose-files`[22]. In this folder exists also a readme that describes, how to instantiate a topology locally with *docker-compose*[23].

Figure 2.10 shows a the general scheme for a deployment on AWS.

[17]a discussion/open issue about that can be found here: https://github.com/moby/moby/issues/24862

[18]`https://docs.docker.com/storage/volumes`

[19]https://hub.docker.com/r/rexray/s3fs

[20]`https://docs.docker.com/compose/reference/scale/`

[21]The implementation of the coordinator and the modifications in the sensor implementation can be found here: `https://gitlab.tubit.tu-berlin.de/gerrit.janssen/smemu`

[22]https://github.com/Krymnos/IDP/tree/master/compose_files

[23] Some compose files containing special constraints for AWS, for instance to schedule nodes in different availability zones.

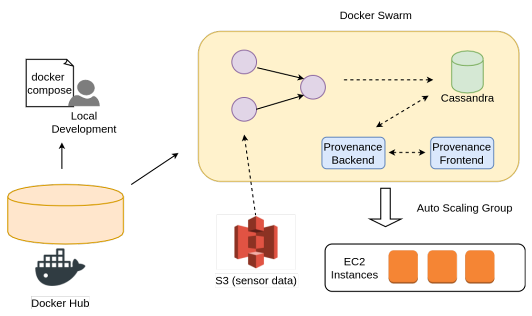


Figure 2.10: deployment scheme of the complete stack on AWS

Chapter 3

Benchmarks

In this chapter we describe the Benchmarks we performed to evaluate our system. The goals we had set beforehand are described in the Motivation section. Then the implementation of each Benchmark, their results and our evalution are discussed seperately.

3.1 Motivation

Based on the use-case described in the first chapter, we decided to evaluate our system firstly by looking at the overhead it adds to the bare pipeline implementation. Since this is one of the major concerns

3.2 Data Overhead

Data Overhead is besides the latency one of the most important metrics especially for a geo-distributed system like a smart grid due to usually strong bandwidth limitations. Therefore it's very important to know which additional "message costs" are related to the use of our provenance system.

We performed several experiments with different provenance configurations as well experiments completely without provenance to determine how big the impact for the use of the provenance system is. We wanted also to determine which setting has more impact of the overhead and which has less to give a recommendation on a possible setup dependent on available bandwidth. We did the benchmarks for different topologies that stand for usual patterns in real topologies. For each topology we will describe our assumptions and then evaluate the results of that.

3.2.1 Test Setup

To run the equal testconfiguration on every topology, we wrote a script[1] that goes through the steps shown in figure 3.1. The benchmark script runs a 120 seconds long benchmark run on each topology with all combinations of the following configurations:

- metrics configuration :

  We executed runs with full provenance (`meterid,metricid,loc,line,class,app,ctime,stime,rtime`) and made runs with each individual metric together with `ctime`, because this metric is mandatory for the correct functioning of the system.

- buffer sizes : `1,5,10`

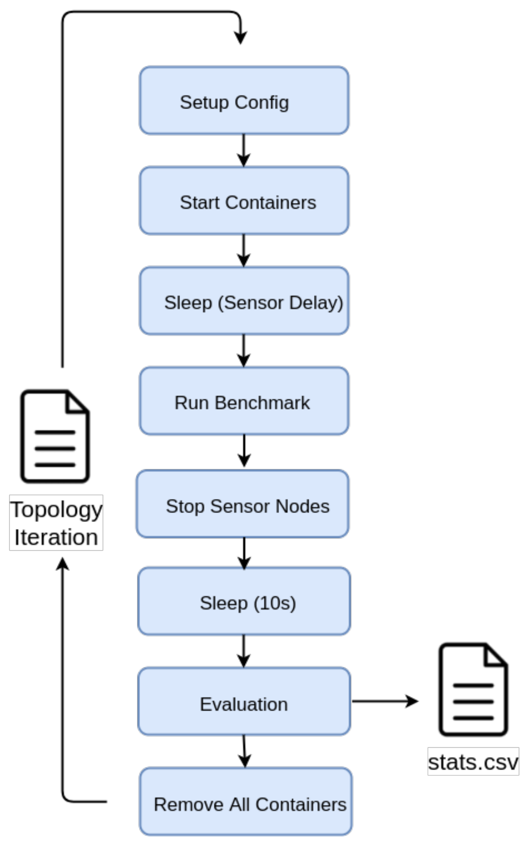


Figure 3.1: Benchmark Procedure

[1]available in the repository `/benchmarks/overhead/benchmark.sh`

The benchmark runs this combinations for each topology file that is defined inside the script. On the first step the respective metrics and buffer configurations are replaced in the compose file that contains the topology. The the containers are started by *docker-compose*. In the script is a fixed startup-delay for the sensors defined. This is needed to ensure that all containers are running at the beginning of the benchmark. On the "Run Benchmark" step, the script waits until the defined benchmark time of 120 seconds are expired. Then only the workload generator nodes are switched of. Another waiting time (10 seconds) is used to ensure that the sensor nodes are terminated and no pending messages are on the pipeline. Then the *docker stats*[2] command is executed to get the `Net I/O` metrics for each node. In addition to the docker stats, all local storages of the pipeline nodes are queried to get the number of messages that passed through the node. The retrieved stats are stored to a csv file that is created at the beginning of the script. After "Evaluation" all containers are stopped and removed to guarantee that no run is influenced by an preceding run by already "used" containers.

The fields of the measurements are:

- TOPOLOGY - the topology file that was used for the run
- PROV_METRICS - provenance metrics that was inserted to the config
- BUFFER_CAPACITY - provenance buffer config
- COMPONENT - component of the pipeline
- NET_IN - NET_IN stat from docker stats
- NET_IN_TYPE - format of the NET_IN value (B,kB,MB)
- NET_OUT - NET_OUT stat from docker stats
- NET_OUT_TYPE - format of the NET_OUT value (B,kB,MB)
- MSG_NUMBER - number of messages that passed the component
- NORMALIZED_MSG_SIZE_IN - NET_IN divided by MSG_NUMBER
- NORMALIZED_MSG_SIZE_OUT - NET_OUT divided by MSG_NUMBER
- NORMALIZED_MSG_SIZE_TYPE - format (B,kb,MB)
- BENCHMARK_RUNTIME - runtime in seconds for the benchmark

[2]`https://docs.docker.com/engine/reference/commandline/stats/`

3.2.2 Test Cases

Single-Node Topology The purpose of the Single-Node Topology is to get the pure overhead that is produced only for one data item without any relaying of messages to other pipeline components.

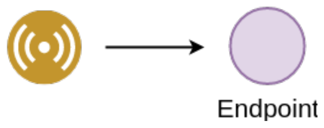


Figure 3.2: One-Node Topology

We evaluated the proportion of the respective metrics by running each metric in a different run and divided the network output by the number of messages that was passed received by the component. Because the metric is a mandatory metric for every run, we substracted the run where only "ctime" was measured from all other runs. "ctime + base" is the run where only ctime was chosen as a metric. Using the network traffic to measure the impact of a metric shows not the real size of a single message. In this measurements are also the communication traffic (e.g. communication between node and cassandra db or traffic between node and docker). Even if the sizes are not the real message sizes, the diagram shows nevertheless the impact to the network traffic by choosing different metrics compared to the no-provenance case.

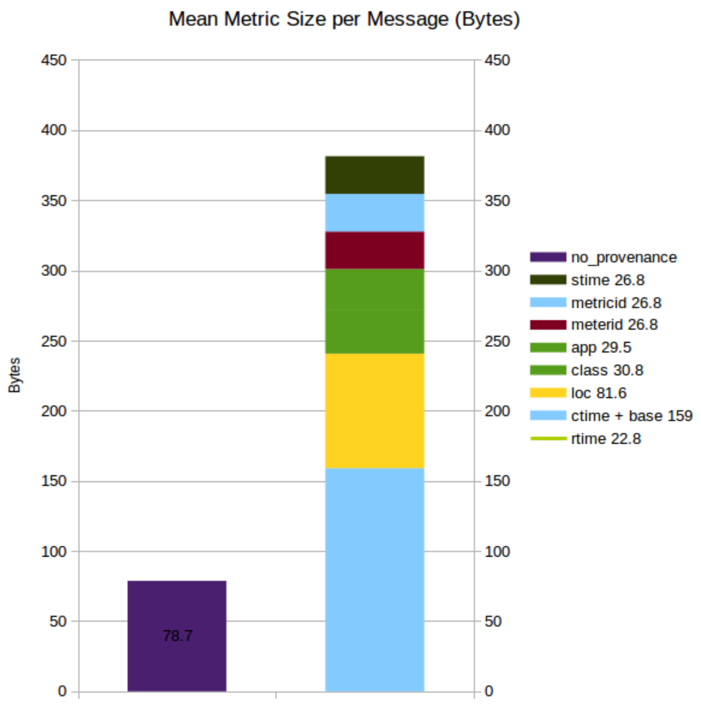


Figure 3.3: Size Distribution for Metrics per Message

Do determine the message size of the measurement, we used "NORMALIZED_MSG_SIZE_IN" stats of the endpoint for the "no_provenance" case.

The diagramm 3.3 shows, that the basic metrics and the location ("loc") are the largest parts of the message. The whole traffic is approximately 5 times bigger when the provenance system is used.

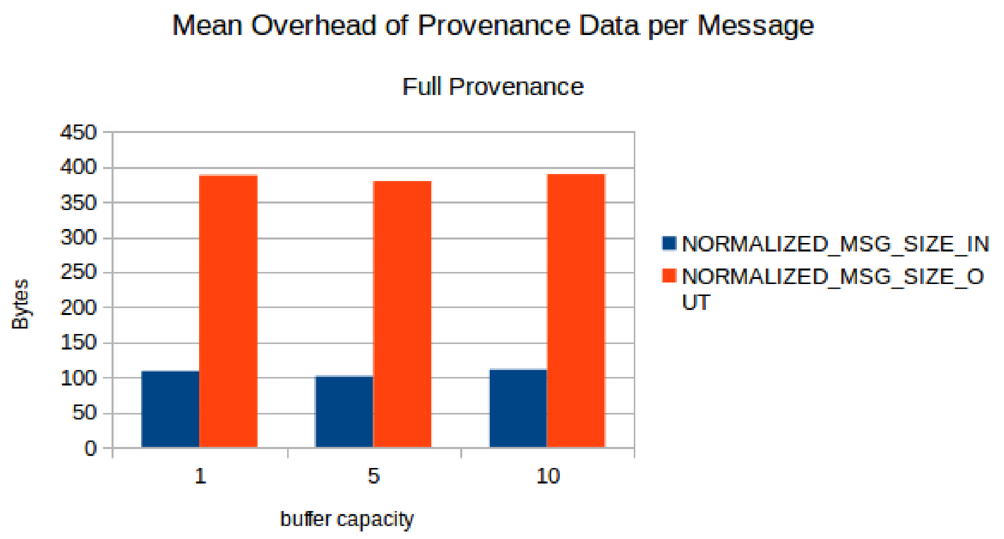


Figure 3.4: Different Buffer Sizes (Topology 0)

By using different buffersizes (1,5,10) no improvement can be observed on diagram 3.4. This option may not improve the used bandwidth due to already existing buffer mechanisms in the the used technologies for the system.

Two-Node Topology Running the benchmark on the Two-Node topology serves to get information if the overhead on a next node level differs from the preceding node level. This topology consists of one sensor and two pipeline nodes (Gateway, Endpoint) which produce provenance data.

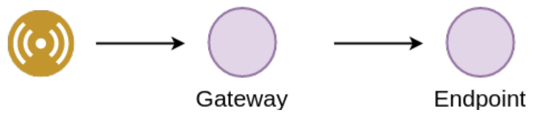


Figure 3.5: Two-Node Topology

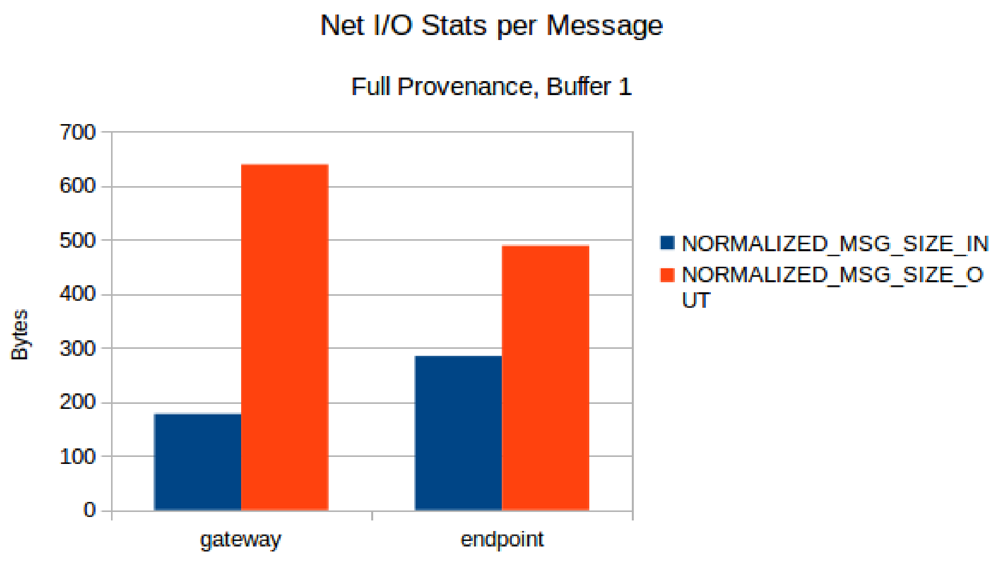


Figure 3.6: Mean Net I/O per Message (Topology 1)

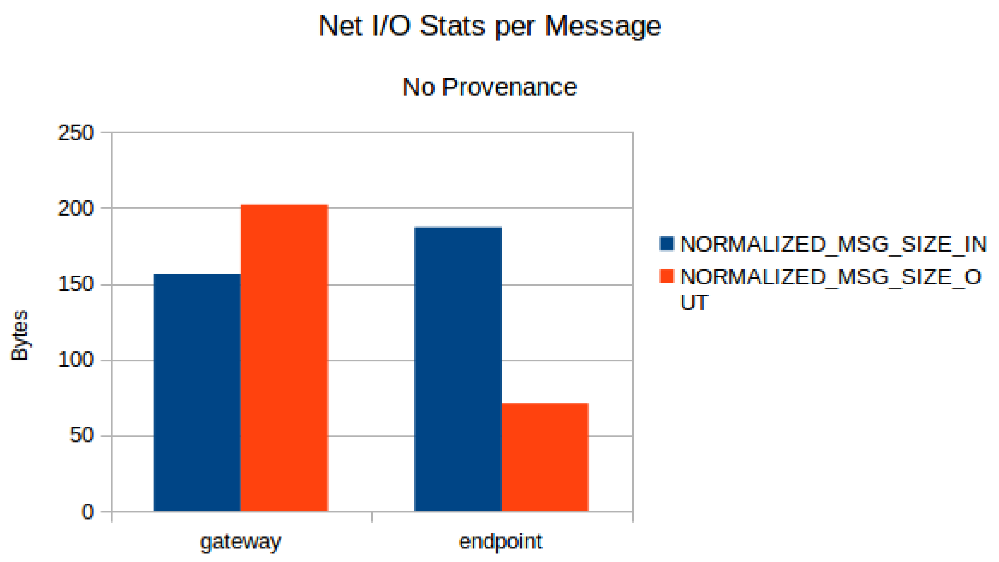


Figure 3.7: Absolute Net I/O for 7480 Messages (Topology 1)

The diagram 3.6 shows, that the per message size of the gateway node is approximately 100 bytes larger than for the endpoint. That suites well to our measurement in diagram 3.4[3]. The difference between the NORMALIZED_MSG_SIZE_IN value of gateway and endpoint can be interpreted as the overhead if we assume that the "IN" value of the gateway stands for a measurement of sensor without provenance. The overhead in that case would be therefore round 100 bytes of data that is pushed to the next node (besides the provenance data that is also being sent to the provenance db). Here as well, it needs to be taken into account that these values are not pure message overheads but an overall impact of the whole system. For example the I/O stats also contain the Heartbeat-Messages that were send between gateway and endpoint.

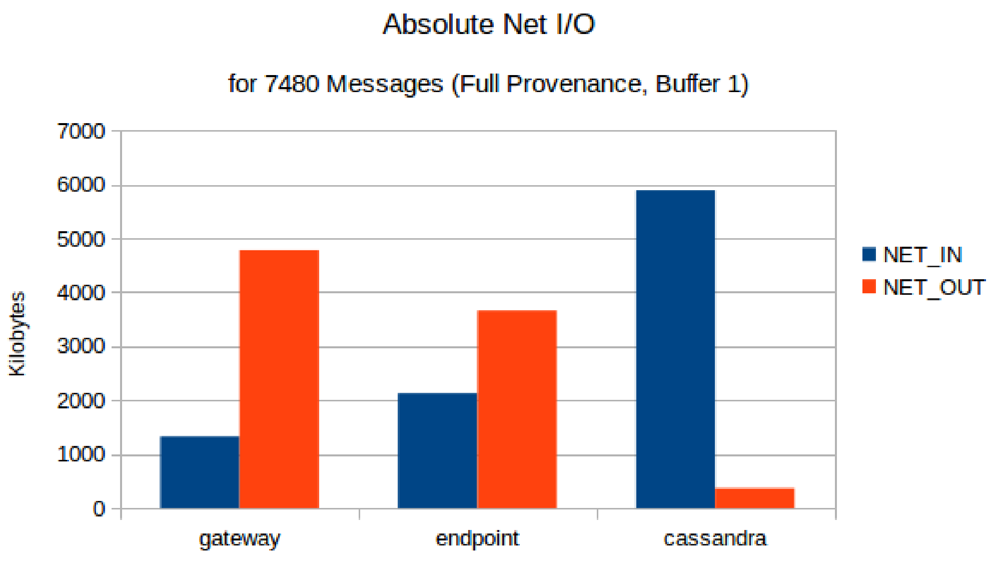


Figure 3.8: Absolute Net I/O for 7480 Messages (Topology 1)

[3]The NORMALIZED_MSG_SIZE_IN in 3.4 is around 100 Bytes large. This value can be interpreted as the data size of a single smart grid measurement

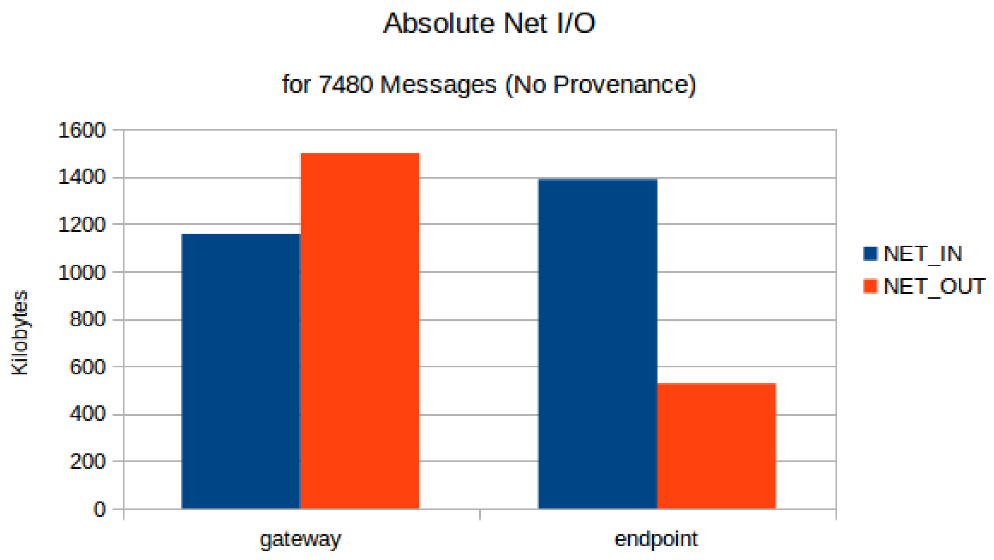


Figure 3.9: Absolute Net I/O for 7480 Messages (Topology 1)

Fork-Node Topology The Fork-Node topology consists of two sensor nodes which sends the measurements to two different gateways. These gateways (Gateway A,B) push the data to a common Gateway C. This Gateway sends the data further to another endpoint node.

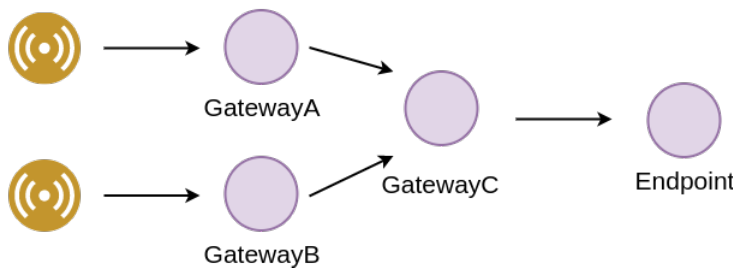


Figure 3.10: Fork-Node Topology

On this run of the benchmark we want answer the question whether the data overhead is growing on common gateways. As it can be observed in diagram **??** is the `NORMALIZED_MSG_SIZE_IN` approximately 100 bytes larger than for gatewayA and gatewayB. This matches our observations in the previous benchmarks. If we compare diagram **??** (Full Provenance) and **??**, we can observe that the Net I/O for gatewayC grows. It's difficult to get an explanation why the mean data is bigger. One would expect that there is no difference between different gateways at least for the non-provenance case. However, we can assume that this growth of data traffic for gatewayC is not from the provenance system itself but from the pipeline implementation.

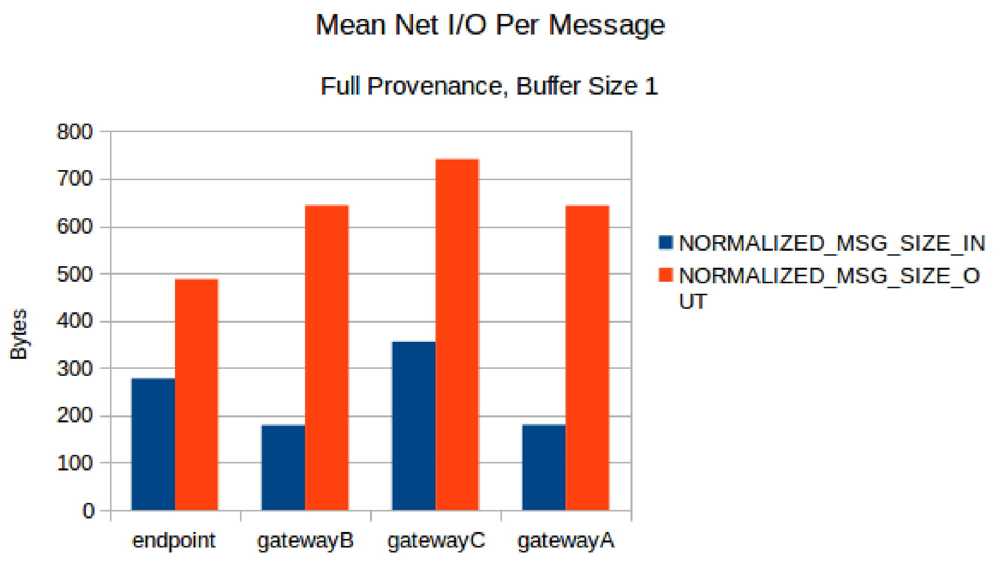


Figure 3.11: Mean Net I/O per Message (Topolgy 2)

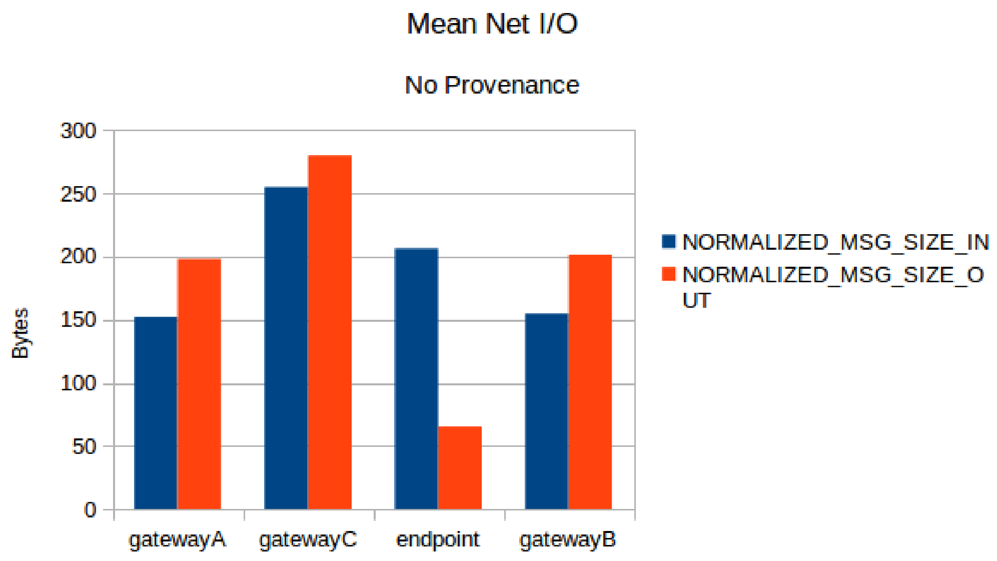


Figure 3.12: Mean Net I/O per Message (Topology 2)

3.2.3 Conclusion

The data overhead was in our tests up to 5 times bigger than the real measurment data. This was also the case, when we used the minimal possible configuration for metrics 3.3 (2 times bigger). This has to be take into account if this system ought to be used in bandwidth limited environments. Nevertheless we can assume when would have larger measurement messages, the provenance data will not be larger than in our measurements because the size of our metrics does not grow with larger measurements. As already mentioned in the evaluation steps, the calculated mean values for the Net I/O are representing the full impact of the provenance system to the network interfaces and does not reflect the real size of messages.

3.3 Latency Overhead

Network latency is the delay from the input of the system to the desired output of the system. For our case, latency is the time for the data point to propagate from one node to the next node. Latency greatly affects how usable the system is. This also brings in the question whether the system can handle a large amount of sending and receiving messages while having minimum latency so that the data is available much more quickly for querying. This is significant for the user to know the limitation of the system and the latency overhead the system encounter so that they could be anticipated before deploying it in their own setup.

We performed several experiments with different provenance configurations and experiments completely without provenance while keeping the sensor parameter constant to determine how big the impact of the provenance system is on the architecture. We also wanted to determine which setting has more impact on the latency and which has less to give a recommendation on a possible best setup. We did the benchmarks for different topologies that stand for usual patterns in real-world topologies. For each topology we will describe our assumptions and then evaluate the results of that.

3.3.1 Test Setup and Configuration

To run the benchmarks we used Amazon as the Cloud provider, which is an industry standard for hosting scalable services. We use Amazon Elastic Container Service which is a highly scalable, high-performance container orchestration service that supports Docker containers and allows you to easily run and scale containerized applications on AWS. In order to minimize the effect of AWS CPU and I/O variability, we performed each test three times on three different periods of different days. New instances of t2.large[4] EC2 instances, using the already configured auto-scaling group was used to further reduce the impact of the lame instance and noisy neighbor on the benchmark results.

Executing the benchmark To run the benchmark on each topology, we wrote a script[5] which takes two parameters: the topology to run in the docker-compose file and the internet protocol address of the EC2 instance.

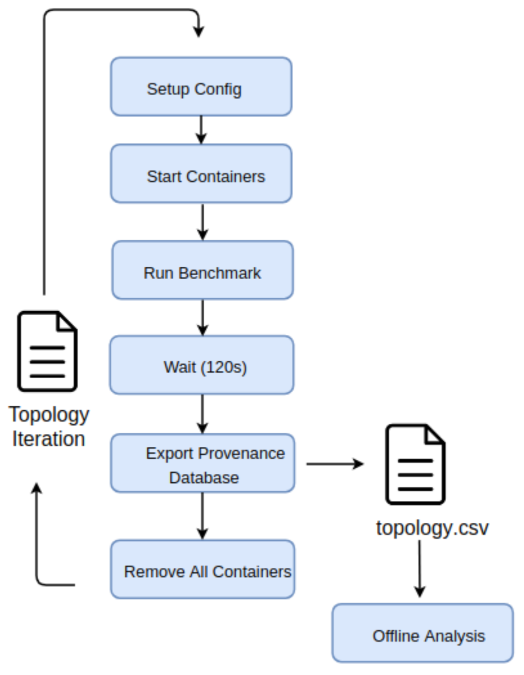


Figure 3.13: Benchmark Procedure

The benchmark runs the above steps for each combination of topologies described in the docker-compose file. The container is started by the docker-compose. After the "Run Benchmark" step the script waits for 120 seconds until it starts to "Export Database(Provenance Database)". There is no need to stop and start the sensor as we are measuring the time for each particular data point propagated through the pipeline. The retrieved benchmarks results are stored in the CSV file for further evaluation. After "Exporting the

[4]Hardware specification : `https://aws.amazon.com/ec2/instance-types/t2/`
[5]Link:`https://github.com/Krymnos/idp-benchmark/blob/benchmark-latency/benchmark_template.sh`

Database" all containers are stopped and removed to guarantee that no run is influenced by a preceding run by already "used" containers.

The field of measurement for the benchmark are:

- TOPOLOGY - the topology file that was used for the run
- PROV_METRICS - provenance metrics that was inserted to the config
- BUFFER_CAPACITY - provenance buffer config (kept constant at 10)
- BENCHMARK_RUNTIME - runtime in seconds for the benchmark (120 seconds for each run)
- TIME-TAKEN-FOR-ONE-HOP - Time taken by the data point to propagate from one node to the next one.
- TIME-TAKEN-TO-TRAVERSE-THE-PIPELINE - Time taken for the data point to propagate from the starting node to the end node.
- MEAN LATENCY FOR HOP - (TIME-TAKEN-FOR-ONE-HOP/ TOTAL DATAPOINT)
- MEAN LATENCY FOR PIPELINE - (TIME-TAKEN-TO-TRAVERSE-THE-PIPELINE/ TOTAL DATAPOINT)

For every topology we run the benchmark procedure 3.13 while changing only the number of context parameters at a time as shown below:

- FULL CONTEXT - Provenance meterics used in the setting are : meterid,metricid,loc,line,class,app,ctime,stime,rtime
- SIX CONTEXT - Provenance meterics used in the setting are : meterid,metricid,app,ctime,stime,rtime
- FOUR CONTEXT - Provenance meterics used in the setting are : metricid,ctime,stime,rtime

3.3.2 Two-Node Topology

This is the base case for the latency overhead benchmark as we need to know the latency overhead of the system when we are running with bare minimal requirements to conduct the benchmark. This will provide us the threshold so that we could differentiate when running with different configuration of the system and topologies. This topology consists of one sensor and two pipeline nodes (Gateway, Endpoint) which produce provenance data.

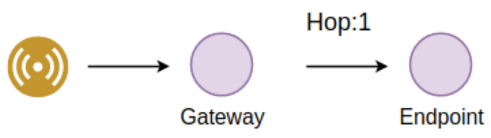


Figure 3.14: Two-Node Topology

We run the topology with full context, six context parameters, four and with no provenance. We evaluated the results by plotting the mean node latency for a collection of data points to travel from Gateway to the Endpoint labeled as Hop:1.

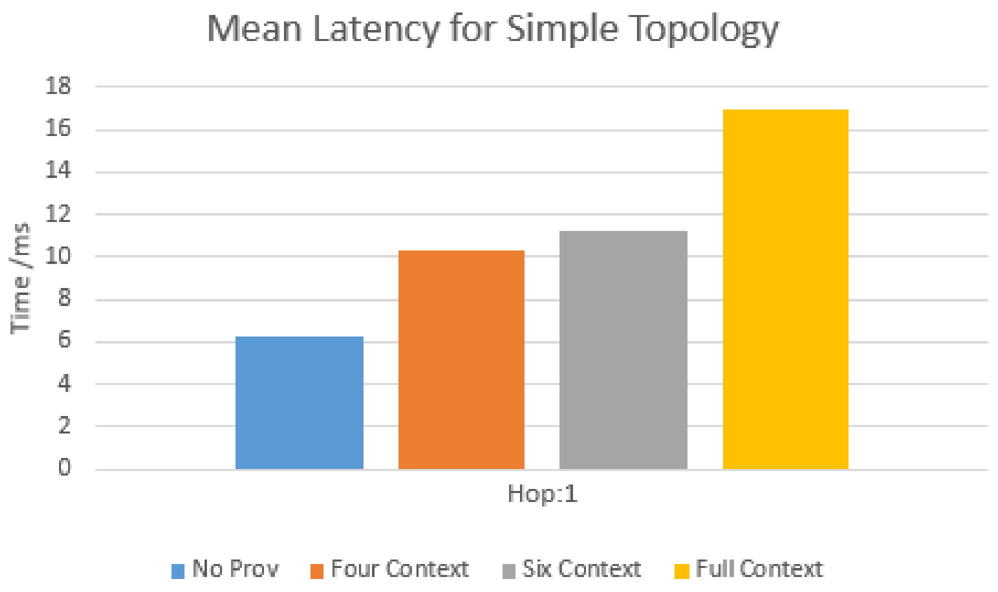


Figure 3.15: Mean Latency for collected Data-points (Two Node Topology)

From the diagram 3.15, its is clearly shown that the message propagating without provenance information has the smallest latency. Furthermore, we can see a clear difference in the latency when using full context compare to using six or four context parameters for running the provenance system. Mean latency for Two Node Topology3.14 using six contexts or four context does not depict a large difference. Possible reasons for this could be the sizes of messages, as well as the amount of computations needed to collect and package the information.

Triangle Topology

The idea behind using this topology for the next benchmark is to check if there is any fluctuation or changes in the reading when adding another gateway to send data point to the endpoint. This will help us to check how the provenance system performs in case of adding nodes vertically to the architecture. The Triangle topology consists of two sensor nodes which send the measurements to two different gateways. These gateways (Gateway A, B) push the data to a common Endpoint.

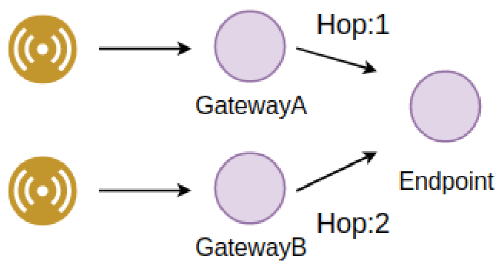


Figure 3.16: Triangle-Node Topology

We run the topology with full context, six context,four context and with no provenance. We evaluated the results by plotting the mean node latency for a collected of data points to travel from Gateway (A,B) to the Endpoint labeled as Hop:1 and Hop:2.

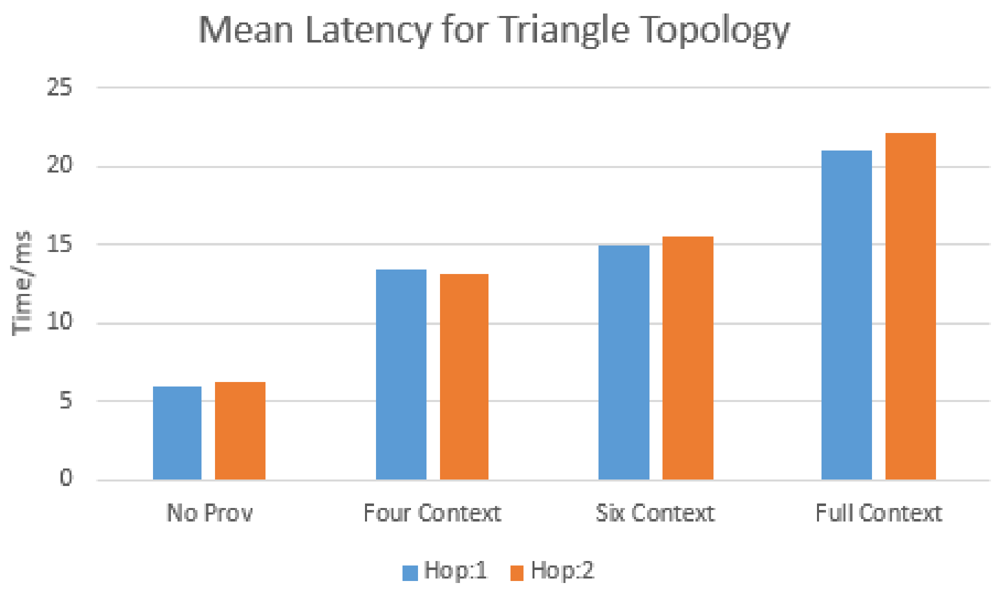


Figure 3.17: Mean Latency for collected Data-points (Triangle Node Topology)

From the diagram 3.17, the mean latency per both hops are similar. There is not much difference in mean time to travel from the Gateways(A, B) to Endpoint but there is a clear difference to the mean latency compare to the figure 3.15 that the mean latency increase for the all four experiment. The reason behind this might be that the Endpoint is receiving messages from two nodes instead of one node thus there is a visible difference in the measurement which is around 8 percent average increase in mean latency. There is visible performance degradation when a single node is receiving messages from two nodes as seen from the benchmark results.

3.3.3 Line Topology

The idea behind using this topology for the next benchmark is to check if there is any visible impact on latency when adding another gateway between the endpoint and the first gateway. This will check how the provenance system performs in case of increasing node horizontally to the architecture.The Line topology consists of one sensor node which send the measurements to next gatewayA, which propagates the data to gatewayB, which then push the data to Endpoint.

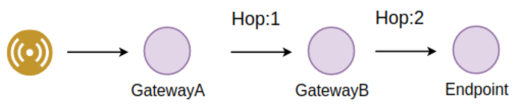


Figure 3.18: Line-Node Topology

We run the topology with full context, six context,four context and with no provenance. We evaluated the results by plotting the mean node latency for a collected of data points to travel from GatewayA to the GatewayB labeled as Hop:1 and from GatewayB to Endpoint labeled as Hop:2.

The benchmark will also give a view how the latency of the node changes when it is receiving and sending messages concurrently to the next node and the provenance database.

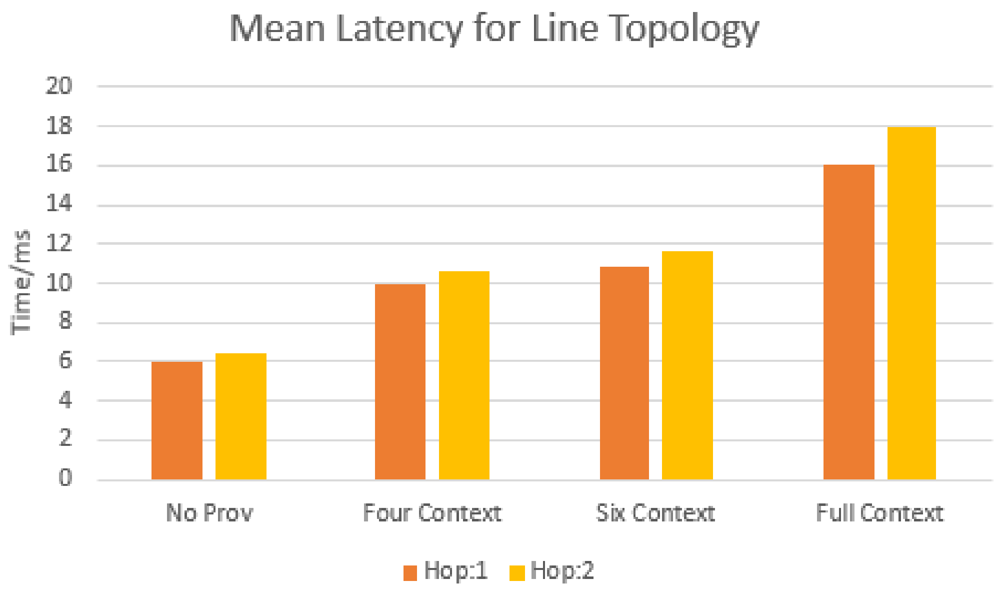


Figure 3.19: Mean Latency for collected Data-points (Line Node Topology)

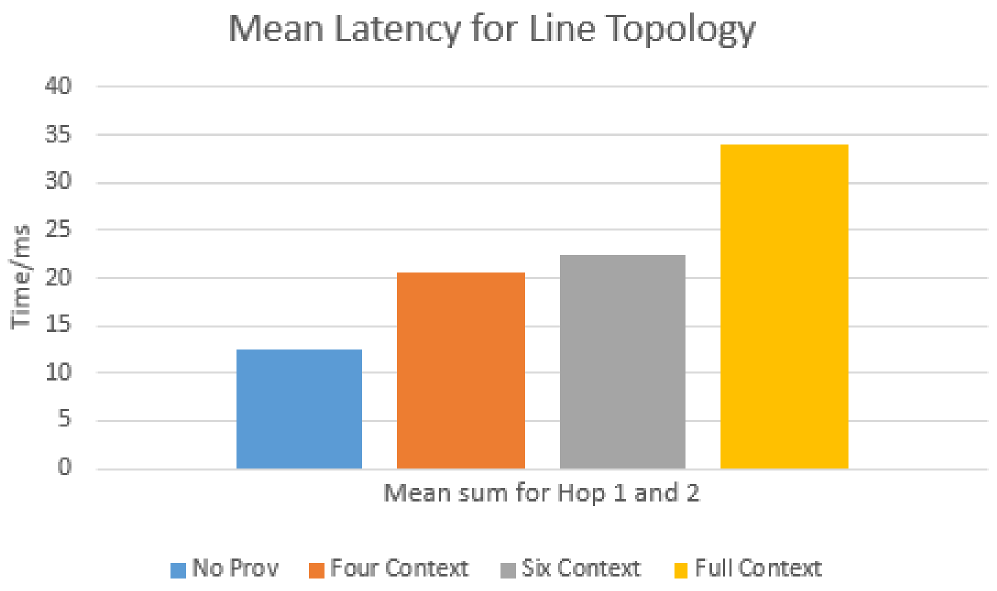


Figure 3.20: Summation of Mean Latency for collected Data-points (Line Node Topology)

The diagram 3.19 shows a visible increase in mean latency for both the hops compared to the diagram in 3.15 where the node was only responsible for receiving the data point. Also seen from the figure 3.19 the mean latency is always greater than for all the experiment for Hop:2. This is due to the GatewayB3.18 is responsible for receiving and sending the message across the pipeline explains the increased latency for both cases discussed.

Figure 3.20 shows that due to small increase in latency on the gatewayB there is a visible difference when changing the contexts parameters from six to full especially in the case when the node is responsible of sending and receiving data-points along the node.

3.3.4 Fork Node Topology

The aim for this benchmark is to see the combined effect of increasing the nodes horizontally and vertically in the architecture and evaluate the latency overhead for the pipeline. The Fork-Node topology consists of two sensor nodes which send the measurements to two different gateways. These gateways (Gateway A,B) push the data to a common GatewayC. This Gateway C sends the data further to another endpoint node.

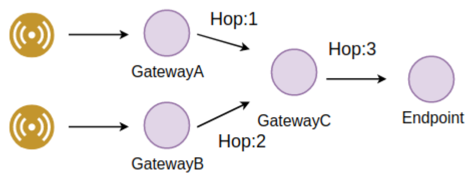


Figure 3.21: Fork Node Topology

We ran the topology with full context, six context,four context and with no provenance. We evaluated the results by plotting the mean node latency for a collected of data points to travel from Gateway (A,B) to the Gateway C labeled as Hop:1 and Hop:2. And Also from Gateway C to Endpoint labeled as Hop:3

Assumption: The pipeline design is in such as way that the gatewayC receives data from two nodes and is sending the data to the endpoint. In this particular case, the majority difference of latency will be caused by this node due to congestion in network traffic.

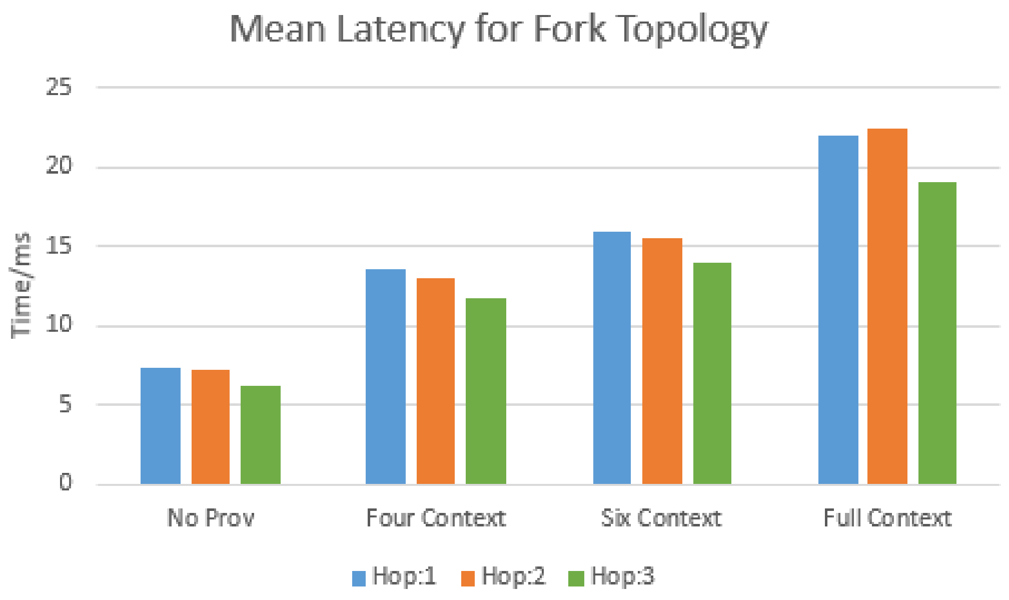


Figure 3.22: Mean Latency per Data point (Two Node Topology)

The diagram 3.22 shows that the mean latency for Hop:1 and Hop:2 are similar to the triangle case 3.16 which a a bit higher then the base case ??.

There is a decrease in the mean latency for Hop:3 as the endpoint is only receiving data from the gatewayC which explains the node have the most number of connection with the provenance database and the next node will be the bottleneck in the system.

Conclusion After looking at the execution of Benchmarks for different settings, it is clear that using less context parameters with the provenance system result in less latency overhead. The end user should keep these results in mind when considering which parameters are relevant to track. However, for the use case of the grid administrators where humans are reacting to failure notifications, it seems fair to say that using all 10 context parameters is feasible, since one or two seconds faster response time will not make much of a difference. It can also be seen from the benchmarks results that the mean node latency for each hop remains consistent with no drastic changes when increasing the hops in the pipeline design, suggesting a good scalability of the system, at least for a small scale deployment such as we had the resources to test.

Changing the Buffer Capacity After analyzing the effect of changing context parameter setting of the provenance system we need to see the effect of buffer capacity on the system in term of latency so that we know the capacity at which the system ingest data with minimal overhead. In this we want to know that if the data ingestion rate of the provenance system effect the node latency or not. Everything was kept same as in the test setup 3.3.1 beside the Buffer Capacity : (10,50,100) running only on Fork Node Topology3.21.

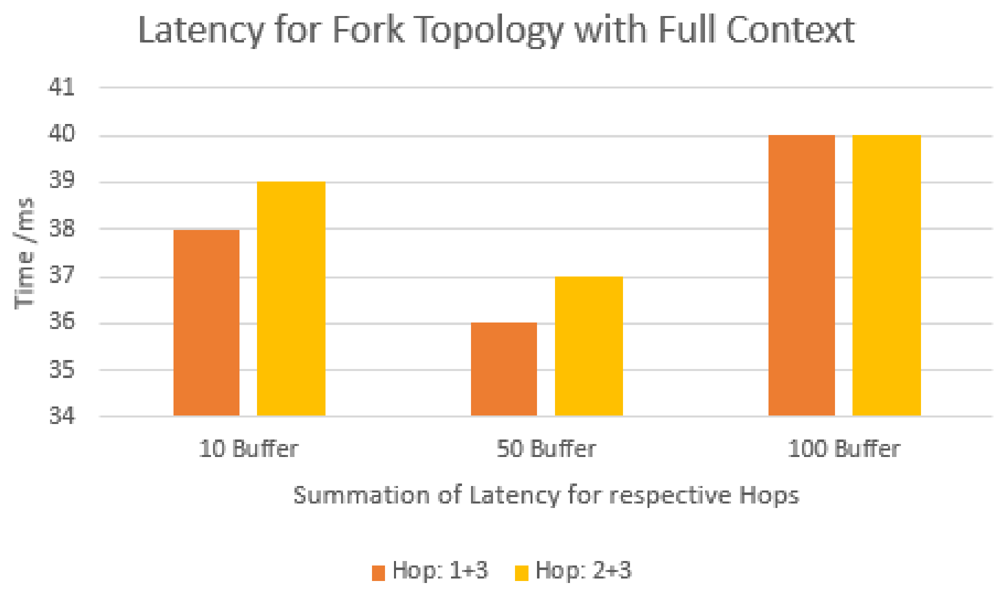


Figure 3.23: Latency for Fork Topology with Full Context

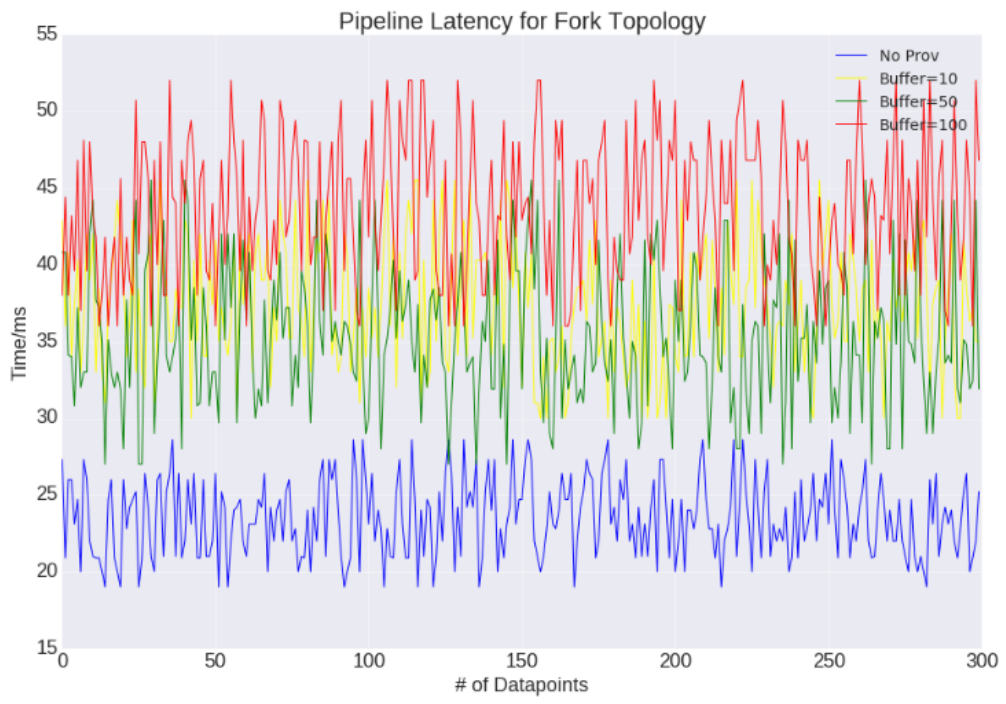


Figure 3.24: Pipeline Latency for Fork Topology with Full Context (Sum of Hop:1 and Hop:3 3.21 using subsample of collected Data points)

Conclusion As seen from figure 3.24 and figure3.24 that the provenance system perform better when the buffer capacity setting is set to 50 messages. The reason behind this could be the implementation of the pipeline itself and also consider another functionality of the provenance system HeartBeat messages of the system which create network overhead. Latency difference could also be due to the bottleneck of the GatewayC. Fifty multiply by message size sent is a perfect size for the network bandwidth used by the service hence the difference in changing the buffer size explained without making a network congestion on the whole system being the sweet spot of the provenance system.

3.4 Failure Detection Benchmarks

The purpose of this benchmarks is to find the following four types of failures,

- Node Failure
- Channel Link Failure
- Provenance Daemon Failure
- Pipeline Daemon Failure

3.4.1 Node Failure

The purpose of this benchmark is to find the time taken to detect a node failure in the topology. Node failure is here meant to be the nodes that run the provenance daemon and pipeline daemon and not the sensors. If any of the nodes that are running the provenance API daemon and pipeline daemon are failed then messages sent via this nodes will be failed to reach the other nodes.

We need to measure the time taken metric for our provenance system to detect such a node failure. In our provenance database, heartbeat messages are sent by each node are stored in the HEARTBEAT table.

So the main tables that are needed to perform this benchmark are,

- Node
- HeartBeat

3.4.1.1 Method to generate a node failure

Method to simulate a node failure is to kill the container(node) itself. Here the workload generator will introduce failures continuously at regular intervals by killing the docker container and restarting the docker container. This killing and restarting process will happen throughout the benchmark.

3.4.1.2 Logic used to detect node failure

We will be querying the Node table to get the list of nodes. Once we got the node id for a particular node, then we query the heartbeat table to get the rows with that id.

Once we get the rows, if heartbeat rows are available for the node then we can be sure that the node is active. We also need to perform an additional check to find out if the timestamp field is less than 30 seconds old. Because each node will be sending the heartbeat messages every 30 seconds and we need to be sure that the node is active currently. We will store the timestamp value each time in the benchmark and compare it next time with the previous value to find out if the heartbeat message is the latest one.

3.4.2 Link Failure

The purpose of this benchmark is to find the time taken to detect a channel link failure in the topology. Channel link failure is here meant to be physical link failure. If any of the channel links between the nodes that are running the provenance API daemon and pipeline daemon are failed then messages sent via this nodes will be failed to reach the other nodes.

We need to measure the time taken metric for our provenance system to detect such a failure. In our provenance database, heartbeat messages are sent by each node are stored in the HEARTBEAT table. Also to check if the neighbouring nodes are active and also to detect link failures, we also send messages to neighbours and each node will transmit those messages to the provenance database.

So the main tables that are needed to perform this benchmark are,

- Node
- HeartBeat

3.4.2.1 Method to generate a link failure

There are two ways to generate a link failure,

- During the deployment of the topology, 'host_next' parameter in the topology yaml file which is used to provide a virtual link between containers(nodes). We will be changing the name of this field to some random value and thus making the two containers unreachable. Thereby simulating a link failure between nodes in the provenance system.
- Another method to simulate a link failure is to kill one container(node) itself. Because once the deployment is done and the topology is running in the provenance system, killing the container(node) is the only way to simulate the link failure in the docker environment.

Here we have used the first method and during the deployment itself we provide a wrong connection between the nodes and we run the benchmark test to detect those failures.

3.4.2.2 Logic used to detect link failure

We will be querying the Node table to get the list of successor for each node. Once we got the successor for a particular node, then we query the heartbeat table to get the rows with the id of the successor node and also with its own id.

Once we get the rows, if heartbeat rows are available for the node and its neighbour then we can be sure that both the nodes are active. The next check we need to perform is to find if the channel field in the heartbeat table row with the successor id contains the node id. If the node id is present then we can conclude that channel link is also active. We also need to perform an additional check to find out if the channel field is less than 30 seconds old. Because each node will be sending the heartbeat messages every 30 seconds and we need to be sure that the channel is active currently.

3.4.3 Pipeline Failure

The purpose of this benchmark is to find the time taken to detect a pipeline daemon failure in the topology. Pipeline daemon failure is here meant to be process failure in a node that is running the pipeline API. If the process in a node that is running the pipeline API daemon are failed then heartbeat messages and provenance messages will not be processed in this node and sent to the database and also not sent to the neighbouring nodes in the topology.

We need to measure the time taken metric for our provenance system to detect such a failure. In our provenance database, heartbeat messages are sent by each node from the pipeline daemon and are stored in the HEARTBEAT table.

So the main tables that are needed to perform this benchmark are,

- Node
- HeartBeat

3.4.3.1 Method to generate a pipeline daemon failure

Method to simulate a pipeline daemon failure is to kill the container(node) itself. This is similar to the Node failure detection benchmark we have done earlier. Here the workload generator will introduce failures continuously at regular intervals by killing the docker container and restarting the docker container. This killing and restarting process will happen throughout the benchmark.

3.4.3.2 Logic used to detect pipeline daemon failure

We will be querying the NODE table to get the list of nodes that are running the pipeline daemon. Once we got the node id for a particular node, then we query the heartbeat table to get the rows with that id. Once we get the rows, if heartbeat rows are available for the node then we can be sure that the pipeline daemon is active. We also need to perform an additional check to find out if the timestamp field is less than 30 seconds old. Because each pipeline daemon will be sending the heartbeat messages every 30 seconds and we need to be sure that the pipeline daemon is active currently. We will store the timestamp value each time in the benchmark and compare it next time with the previous value to find out if the heartbeat message is the latest one.

3.4.4 Provenance Daemon Failure

The purpose of this benchmark is to find the time taken to detect a provenance daemon failure in the topology. Provenance daemon failure is here meant to be process failure in a node that is running the provenance API. If the process in a node that is running the provenance API daemon are failed then provenance data will not be processed in this node and sent to the database.

We need to measure the time taken metric for our provenance system to detect such a failure. In our provenance database, provenance data are sent by each node are stored in the PROVENANCETABLE table.

So the main tables that are needed to perform this benchmark are,

- Node

- Provenancetable

3.4.4.1 Method to generate a provenance daemon failure

Once the deployment is done and docker containers are started, workload generator need to run within each node to kill the process. But that will make the same node to always fail the same provenance daemon. Even if the workload generator script is started in all the nodes then all the provenance daemons will fail at the same time. Coordination between workload generators will generate extra load on the provenance system which will not give the exact results. Hence the workload generator here will directly delete the rows from the PROVENANCETABLE table for half of the nodes in the topology. Thus we generate the provenance daemon failure at regular intervals this way while running the benchmark.

3.4.4.2 Logic used to detect provenance daemon failure

We will be querying the Node table to get the list of nodes. Once we got the node id for a particular node, then we query the PROVENANCETABLE table to get the rows with that id. Initially we will mark the provenance daemon active if we found rows greater than 0 and then we will query the PROVENANCETABLE every 10 seconds delay and again check the count. Previous counts will be stored throughout the benchmark. Provenance Daemon will be marked as active if the count has been increased compared to the previous count value. If the count remains unchanged then Provenance daemon will be marked as failed.

3.4.5 Steps to run the failure detection benchmark

Pull the Benchmark repository from the following link, https://github.com/Krymnos/idp-benchmark/tree/Development

Deploy the complete stack in AWS (deployment steps are explained in the deployment chapter). Once the deployment is done, get the public IP address of the Manager node and this IP address will be used to connect to the cassandra database deployed in our stack. Cassandra IP address will be passed as an external parameter to our failure detection benchmark script.

Run the benchmark script like below(replace the IP address accordingly),

```
python startBenchmark.py -i 18.197.129.37
```

This script will automatically run all the four failure detection benchmarks and save the output in a folder named "results".

.1 Appendix

Listing 1: .travis.yaml for pipeline component

```
sudo: true
language: java

services:
  - redis-server

before_install:
    - wget
      ↪ https://repo1.maven.org/maven2/com/google/protobuf/
      ↪ protoc/3.5.0/protoc-3.5.0-linux-x86_64.exe
    - mvn install:install-file
      ↪ -DgroupId=com.google.protobuf
      ↪ -DartifactId=protoc -Dversion=3.5.0
      ↪ -Dclassifier=linux-x86_64-debian -Dpackaging=exe
      ↪ -Dfile=./protoc-3.5.0-linux-x86_64.exe
    - wget
      ↪ http://maven.aliyun.com/nexus/content/groups/public/
      ↪ io/grpc/protoc-gen-grpc-java/
      ↪ 1.8.0/protoc-gen-grpc-java-1.8.0-linux-x86_64.exe
    - mvn install:install-file -DgroupId=io.grpc
      ↪ -DartifactId=protoc-gen-grpc-java
      ↪ -Dversion=1.8.0 -Dclassifier=linux-x86_64-debian
      ↪ -Dpackaging=exe
      ↪ -Dfile=./protoc-gen-grpc-java-1.8.0-linux-x86_64.exe
before_script: cd pipeline_interfaces/java_project/pipeline

script: mvn package

deploy:
  provider: script
  script: ./deploy_docker.sh
  skip_cleanup: true
  on:
    branch: deployment
```

Listing 2: docker deployment script for pipeline component

```
#!/bin/bash
docker login -u="$DOCKER_USERNAME" -p="$DOCKER_PASSWORD"
docker build -t pipeline_java .
docker images
docker tag pipeline_java cloudproto/pipeline_component
docker push cloudproto/pipeline_component
```

Listing 3: Dockerfile for Pipeline Component

```
# Update the repository and install Redis Server
RUN         apt-get update && apt-get install -y
    redis-server
RUN apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/*
    /var/tmp/*

## pipeline stuff
ADD target/pipeline-0.1-jar-with-dependencies.jar app.jar
RUN sh -c 'touch /app.jar'
ENV JAVA_OPTS=""
ENV ARGUMENTS=""
ENV STARTUP_DELAY=20

# default values for provenance system
ENV CONF_LOC=EVAR
ENV NODE_ID=000000
ENV NODE_NAME=NONAME
ENV SUCCESSOR=000000
ENV NEIGHBOURS=000000:127.0.0.1
ENV SINK=cassandra
ENV CASSANDRA_HOST=127.0.0.1
ENV CASSANDRA_PORT=9042
ENV CASSANDRA_KEYSPACE_NAME=provenancekey
ENV CASSANDRA_TABLE_NAME=provenancetable
ENV CASSANDRA_REPLICATION_STRATEGY=SimpleStrategy
ENV CASSANDRA_REPLICATION_FACTOR=1
ENV BUFFER_CAPACITY=10
ENV
    METRICS=meterid,metricid,loc,line,class,app,ctime,stime,rtime

ENTRYPOINT [ "sh", "-c", "/usr/bin/redis-server
    --daemonize yes && sleep ${STARTUP_DELAY} && java
    $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom -jar
    /app.jar $ARGUMENTS " ]
```

Listing 4: Simple Topology without Provenance System

```
version: '3'
```

```
services:
  source:
    image: cloudproto/sensor:latest
    environment:
      - SENSOR_PARAMETERS=-sourceFolder data/20170210
         -frequency 1000 -targetAddress gateway:50051
         -targetType grpc-pipeline
      - STARTUP_DELAY=60
gateway:
    image: cloudproto/pipeline_component:latest
    environment:
      - ARGUMENTS=--port 50051 --host_next endpoint
         --port_next 50051 --storagetime 100 --no_prov
endpoint:
    image: cloudproto/pipeline_component:latest
    environment:
      - ARGUMENTS=--port 50051 --storagetime 100 --no_prov
```

Listing 5: Example for use of Sensor Image by mounting S3 Bucket

```
image: cloudproto/sensor:s3
    privileged: true
    environment:
        - BUCKET=provenancesensordata:oneday
        - AWS_ACCESS_KEY_ID=ADD_YOUR_ACCESS_KEY
        - AWS_SECRET_ACCESS_KEY=ADD_YOUR_SECRET_KEY
        - REGION=eu-central-1
        - SENSOR_PARAMETERS=-sourceFolder
           /mnt/s3/31400010000000000 -targetAddress
           gateway:50051 -targetType grpc-pipeline
        - STARTUP_DELAY=10
```

Listing 6: Example for scalable sensor groups with unique sensor containers

```
version: '3'

# to use the provenance volume it's mandatory to install
    the rexray/s3 plugin
# docker plugin install rexray/s3fs:latest
    S3FS_ACCESSKEY=XXXXX S3FS_SECRETKEY=XXXXXX

volumes:
    provenancesensordata:
        external: true
```

```
services:
    cassandra:
        image: cassandra:latest

## sensor container can retrieve unique sensor ids from
   this service
    idprovider:
        image: cloudproto/idprovider:latest
        environment:
            - START_ID=3140001000000000

    sensorGroupA:
        image: cloudproto/sensor:latest
        depends_on:
            - gateway
            - idprovider
        restart: always
        volumes:
            - provenancesensordata:/mnt
        environment:
            - SENSOR_PARAMETERS=-sourceFolder /mnt/oneday
               -sensorIdProvider idprovider:8080
               -frequency 1000 -targetAddress
               gateway:50051 -targetType grpc-pipeline
            - STARTUP_DELAY=30
[...]
```

List of Tables

List of Figures

Acronyms

Bibliography

[1] dummyauthor. “Dummytitle.” In: vol. 10. 2018, pp. 28–0.

[2] *Google Protocol Buffers.* URL: `https://developers.google.com/protocol-buffers/` (visited on 03/08/2018).

[3] *gRPC.* URL: `https://grpc.io` (visited on 03/08/2018).

[4] *Redis.* URL: `https://redis.io/` (visited on 03/08/2018).