

UNIVERZA V LJUBLJANI  
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Nejc Saje

**Razširljiv nadzor velikih oblačnih  
sistemov**

MAGISTRSKO DELO

ŠTUDIJSKI PROGRAM DRUGE STOPNJE  
RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: doc. dr. Mojca Ciglarič

Ljubljana, 2016



UNIVERSITY OF LJUBLJANA  
FACULTY OF COMPUTER AND INFORMATION SCIENCE

Nejc Saje

**Scalable monitoring of large cloud  
systems**

MASTER'S THESIS

SECOND CYCLE STUDIES

COMPUTER AND INFORMATION SCIENCE

MENTOR: doc. dr. Mojca Ciglarič

Ljubljana, 2016



This thesis is the intellectual property of the author and the Faculty of Computer and Information Science at the University of Ljubljana. A written permission from the author, the mentor and the Faculty of Computer and Information Science is required for publishing this thesis.

The source code of the prototype is available under the MIT license, which is provided verbatim:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

The software is provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.



# DECLARATION OF AUTHORSHIP OF FINAL WORK

I, the undersigned student Nejc Saje, registration number 63100351, the author of the final work of studies:

*Scalable monitoring of large cloud systems* (slov. *Razširljiv nadzor velikih oblačnih sistemov*)

## DECLARE

1. The written final work of studies is a result of my independent work mentored by doc. dr. Mojca Ciglarič.
2. The printed form of the written final work of studies is identical to the electronic form of the written final work of studies.
3. I have acquired all the necessary permissions for the use of data and copyrighted works in the written final work of studies and have clearly marked them in the written final work of studies.
4. I have acted in accordance with ethical principles during the preparation of the written final work of studies and have, where necessary, obtained agreement of the ethics commission.
5. I give my consent to use of the electronic form of the written final work of studies for the detection of content similarity with other works, using similarity detection software that is connected with the study information system of the university member.
6. I transfer to the UL — free of charge, non-exclusively, geographically and time-wise unlimited — the right of saving the work in the electronic form, the right of reproduction, as well as the right of making the written final work of studies available to the public on the world wide web via the Repository of UL.
7. I give my consent to publication of my personal data that are included in the written final work of studies and in this declaration, together with the publication of the written final work of studies.

In Ljubljana, April 22<sup>nd</sup>, 2016

Student's signature:



*I would like to thank my mentor, doc. dr. Mojca Ciglarič, and as. dr. Matjaž Pančur for their advice and guidance. I would also like to thank Sanja and my parents for their patience.*



To my soon-to-be wife, who stands by me  
through thick and thin.



# Contents

Povzetek

Abstract

Razširjen povzetek

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Structure . . . . .	2
<b>2</b>	<b>Field overview</b>	<b>5</b>
2.1	Prior work . . . . .	6
2.2	Scalability . . . . .	7
2.3	Distributed consensus . . . . .	8
<b>3</b>	<b>Product overview</b>	<b>11</b>
3.1	GMonE . . . . .	11
3.2	DARGOS . . . . .	13
3.3	Monalytics . . . . .	14
3.4	Ceilometer . . . . .	15
3.5	Monasca . . . . .	18
<b>4</b>	<b>Dagger: A new distributed monitoring system</b>	<b>21</b>
4.1	Functional requirements . . . . .	22
4.2	Architecture . . . . .	24
4.3	Design and implementation . . . . .	38

## CONTENTS

<b>5</b>	<b>Pilot deployment and requirements verification</b>	<b>53</b>
5.1	Infrastructure outline . . . . .	53
5.2	Scalability testing . . . . .	54
5.3	Reliability . . . . .	63
<b>6</b>	<b>Conclusion</b>	<b>65</b>
6.1	Future work . . . . .	66
	<b>Appendices</b>	<b>73</b>
<b>A</b>	<b>HTTP API</b>	<b>73</b>
A.1	Register session . . . . .	73
A.2	Renew session . . . . .	73
A.3	Publish a new record . . . . .	73
A.4	Publish a new raw record . . . . .	74
A.5	Subscribe to a record stream . . . . .	74

# Abbreviations

<b>VM</b>	Virtual Machine
<b>API</b>	Application Programming Interface
<b>REST</b>	Representational State Transfer
<b>DBMS</b>	Database Management System
<b>SLA</b>	Service Level Agreement
<b>LWM</b>	Low Watermark
<b>JSON</b>	JavaScript Object Notation
<b>TCP</b>	Transmission Control Protocol
<b>RPC</b>	Remote Procedure Call
<b>JVM</b>	Java Virtual Machine
<b>TTL</b>	Time To Live



# Povzetek

**Naslov:** Razširljiv nadzor velikih oblačnih sistemov

Nadzor velikega oblačnega sistema lahko pomeni nadzirati na tisoče naprav, zaradi česar si operaterji podatkovnih centrov ne morejo privoščiti, da bi vse podatke zbirali in analizirali centralizirano. V sklopu magistrske naloge je bil razvit prototip porazdeljenega komunikacijskega in računskega sistema za nadzor oblačnih sistemov, ki je razširljiv, visoko razpoložljiv in je osnovan na principih realnočasnega procesiranja podatkovnih tokov. V osnovi je prototip sistem tipa objavi/naroči, ko pa se akter naroči na podatke, ki še ne obstajajo, se zažene nov računski proces, ki računa in naročniku pošilja zelene podatke. Z gnezdenjem računskih operacij in transformacij lahko podatke, ki jih računajo in objavljajo računski procesi, izračunamo le enkrat v celotnem sistemu, uporabimo pa v veliko različnih operacijah.

**Ključne besede:** oblak, nadzor, skalabilnost, razširljivost, toleranca pri okvarah.



# Abstract

**Title:** Scalable monitoring of large cloud systems

Monitoring a large cloud can mean monitoring potentially thousands of machines, so operators can not afford to gather and analyze the data in a centralized fashion. A prototype of a distributed communication and computation system for monitoring cloud systems has been developed, which is scalable, highly available, and based on real-time stream processing principles. In essence, it is a publish/-subscribe system, but when a subscription is made to a transformation of some data that is not being calculated yet, a new computation process is started that then provides the desired transformation. With nested transformations, data from an existing provider is reused, achieving that a certain computation on a piece of data is only ever performed once in the whole system.

**Keywords:** cloud, monitoring, scalability, availability, fault tolerance.



# Razširjen povzetek

Olačno računalništvo je pojem, ki opisuje paradigmo računalništva, pri kateri računski viri, ki jih uporabnik uporablja, niso v njegovi lasti, ampak mu jih v obliki storitve daje v uporabo ponudnik oblačnih storitev. Čedalje več podjetij se odloča za uporabo tovrstnih storitev, saj so le-te zelo prožne in sposobne se prilagoditi najrazličnejšim trenutnim potrebam. Te prednosti so v prvi vrsti posledica virtualizacije, ki omogoča, da se lahko na eni sami napravi izvaja na stotine ali celo na tisoče izoliranih opravil. V podatkovnem centru pa se lahko nahaja tudi na tisoče takih naprav. Tako uporabniki kot upravljalci oblačnih storitev potrebujejo informacije o stanju svojih opravil, zato da lahko predvidijo in načrtujejo prihodnje računske potrebe, raziščejo vzroke napak ali se s povečanjem kapacitet odzovejo na nenaden porast prometa, vendar pa zaradi velike količine podatkov tradicionalni pristop centraliziranega zbiranja in analiziranja podatkov ne zadostuje.

V sklopu magistrske naloge smo opravili pregled področja nadzora oblačnih sistemov ter postavili teoretični okvir razširljivosti ter porazdeljenega soglasja. Opisali smo obstoječe rešitve za nadzor oblačnih sistemov ter analizirali njihove arhitekture, prednosti in slabosti. Na podlagi tega smo nato zasnovali porazdeljen in dinamičen komunikacijski in računski sistem, ki temelji na principih realnečasne obdelave podatkovnih tokov, in se lahko uporabi kot osnova za izgradnjo generičnega nadzornega sistema. Eno izmed glavnih vodil pri snovanju sistema je bilo to, da se določena računsko operacija na določenih podatkih izvede le enkrat v celotnem sistemu. To dosežemo tako, da operacije nad podatki ločimo na več korakov in potem podatke iz vmesnih korakov ponovno uporabimo pri nadaljnjih operacijah. Če imamo torej več operacij, ki uporabijo isti vmesni rezultat, se bo ta rezultat izračunal le enkrat.

Vozlišče sistema, ki izvaja določeno računsko operacijo na podatkih, mora zato

te podatke dati na razpolago tudi ostalim vozliščem. Naš sistem je zasnovan tako, da vozlišča oglašujejo katere podatke želijo in katere podatke ponujajo. Če neko vozlišče želi podatke, ki jih ne ponuja še nihče, se bo na enem od vozlišč zagnal nov računski proces, ki bo proizvajal željene podatke, če je to mogoče. Vozlišča v sistemu so tako razdeljena na tri vloge: *proizvajalce* (angl. producers), *delavce* (angl. workers) in *naročnike* (angl. subscribers). *Naročniki* obvestijo ostala vozlišča, da želijo prejemati določene podatke. *Proizvajalci* so vir podatkov, ki jih potem pošiljajo zainteresiranim naročnikom. Če pa se naročnik naroči na transformacijo ali računsko operacijo nad določenimi podatki, ki je ne ponuja še nobeno vozlišče v sistemu, bo eno izmed vozlišč *delavcev* prevzelo opravilo in pričelo proizvajati želene podatke. Vozlišča svoje aktivnosti koordinirajo z uporabo močno konsistentne porazdeljene baze parov ključ-vrednost.

Osnovna enota podatkov v našem sistemu je *zapis* (angl. record). Vsak zapis vsebuje edinstveno identifikacijsko vrednost, čas, ime podatkovnega toka, ki mu pripada, ter podatke v prosti obliki. To pomeni, da lahko zapis vsebuje decimalna števila, JSON dokument ali pa kaj povsem drugega. Vsak zapis je del podatkovnega toka (angl. stream), ki ima določeno ime ter neobvezen seznam značk v obliki parov ključ-vrednost. Značke omogočajo filtriranje zapisov, tako da se lahko naročnik naroči na in prejema le določeno podmnožico zapisov podatkovnega toka.

Operacije in transformacije nad podatkovnimi tokovi izvajajo vozlišča delavci z uporabo *opravil* (angl. tasks). Opravila predstavljajo eno računsko operacijo in imajo svoje ime ter opis, ki jih edinstveno identificirata. Dejanska računsko operacija se izvede v *opravilnem vtičniku* (angl. task plugin). Ime opravila pogojuje kateri vtičnik se bo zagnal za izvedbo operacije, opis opravila pa lahko potem vtičnik poljubno razčleni ter tako pridobi dodatne informacije o opravilu, kot je na primer seznam podatkovnih tokov, ki jih opravilo potrebuje za izvedbo operacije. Vozlišča delavci spremljajo zahteve naročnikov po podatkih in če zaznajo zahtevo po podatkih, ki jih ne proizvaža še noben proizvajalec, zaženejo novo opravilo.

Za zagotovitev visoke razpoložljivosti je mogoče opravila opravljati na več vozliščih delavcih vzporedno. To pomeni, da se opravilo na primer zažene na treh delavcih, nato pa ta tri opravila med sabo izvolijo vodjo. Vsa opravila potem prejemajo podatkovne tokove, ki jih potrebujejo za izračun rezultata, rezultate pa naprej naročnikom pošilja le vodilno opravilo. V primeru, da vodilno opravilo

preneha delovati, bo vodenje prevzelo eno izmed preostalih opravil in naročnik bo zopet prejemal zelene podatke. Za zapolnitev vrzeli, nastale s prenehanjem delovanja prejšnjega vodje, se bo zagnalo novo opravilo, ki pa se mora najprej uskladiti z obstoječimi. Vodilnemu opravilu zato pošlje zahtevo po uskladitvi, le-to pa odgovori s posnetkom svojega stanja. Novo opravilo nato svoje stanje prilagodi posnetku stanja vodje ter se naroči na enake podatkovne tokove kot vodilno opravilo. Nato lahko vsa opravila nadaljujejo z obdelavo podatkov.

Naš prototipni sistem je razširljiv na dva načina. Prvi način je razširljivost pri številu opravil. Če nam število opravil, ki jih moramo izvajati, raste, lahko v sistem enostavno dodamo dodatna vozlišča delavce. Vsako novo vozlišče bo lahko prevzelo določeno število novih opravil. V primeru pa da nam namesto števila opravil narašča količina podatkov v podatkovnem toku, ki ga moramo obdelati z enim opravilom, pa razširljivost ni tako enostavna, saj smo omejeni z ozkim grlom enega opravila. V tem primeru lahko razširljivost dosežemo z razčlenitvijo našega opravila na več nivojev opravil, kjer na prvem nivoju več opravil obdeluje posamezne disjunktne podmnožice podatkovnega toka, na naslednjem nivoju pa se dobljene rezultate združi v končen rezultat. To nam omogoči, da podatkovni tok razdelimo na manjše dele, ki jih potem obdelamo vzporedno.

Pripravili smo pilotno postavitve prototipnega sistema, s pomočjo katere smo preverili funkcionalne zahteve, ki smo jih postavili ob začetku razvoja. Testiranje je potekalo v oblaknem sistemu Amazon Elastic Compute Cloud, kjer je bil na treh navideznih napravah nameščen prototipni sistem, na eni konsistentna baza parov ključ-vrednost uporabljena za koordinacijo, ter na eni programska oprema za testiranje. Opravili smo meritve razširljivosti v različnih scenarijih. V scenariju naraščajočega števila opravil se je razširljivost izkazala za linearno, saj smo z večimi vozlišči delavci obdelali sorazmerno večje število opravil. V scenariju, kjer smo omejeni z ozkim grlom enega opravila sistem po pričakovanjih ni bil razširljiv, ko pa smo scenarij prilagodili z razdelitvijo dela na več opravil, pa je bila razširljivost po pričakovanjih zopet linearna. Izmerili smo tudi kako količina režije, potrebna za koordinacijo med opravili, narašča v odvisnosti od števila opravil in časa. Izkazalo se je, da v obeh primerih količina potrebne režije narašča linearno. Na koncu smo opravili še test visoke razpoložljivosti, kjer smo določeno opravilo poganjali na treh vozliščih vzporedno. Nato smo ob naključnih točkah v času ustavili ter znova

zagnali naključno izbrano vozlišče. Naročnik je prejel vse pričakovane rezultate opravila v pravilnem vrstnem redu.

# Chapter 1

## Introduction

Cloud computing is a term describing a computing paradigm in which the computing resources used to achieve a goal are not managed by the user but are rather provided as a service by a service provider. Enterprises are increasingly using cloud computing to solve their IT needs. Whether public or private, cloud computing provides unprecedented power and flexibility when it comes to dynamically provisioning and scaling workloads. These benefits come as a result of virtualization and containerization, which enable hundreds or even thousands of isolated workloads on a single machine in a data-center of potentially thousands of machines. Cloud operators and users alike need to know about the status of their workloads in order to plan for the future properly, investigate faults and respond to a spike in real-time demand. This results in a very large number of entities that need to be monitored and a huge amount of data to be processed.

The amount of entities that need to be monitored in a cloud system is not static, but changes over time and can even increase by an order of magnitude in the course of a cloud system's lifetime. It is crucial that the monitoring solution is able to adapt to different and increasing volumes of data.

It is not enough though to simply store the gathered data and forget about it. The main purpose of monitoring is getting insight from the collected data, which requires doing different calculations on it. The most common approach in monitoring is to first store the collected data in a database and then perform calculations either on-demand or periodically.

Some cloud operators are starting to use stream processing tools to perform

calculations on the collected data in real-time, because the stream analogy fits the cloud monitoring use-case really well. They are combining existing monitoring and stream processing tools such as Apache Storm, which enables them to perform computations scalably on huge amounts of data in real-time.

The existing stream processing tools, however, were not designed with processing monitoring data in mind. They are designed to operate on big streams of data, not small fine-grained ones such as are produced by measuring a single metric on a single entity.

That is why we designed a stream processing system with monitoring in mind. It is meant to operate on many small, fine-grained streams, which it can then combine into bigger streams or perform computations on them in real-time and on-demand. We can use existing data collection agents, such as Nagios plugins or Ceilometer pollsters, to feed data into the system, where it is made available for the consumers. A consumer can be a simple service that stores all the streams into a database or an operator that has just opened a web dashboard and is requesting live monitoring data. A consumer can also request a computation on the data, such as a calculation of an average over five minutes. The requested computation is then provided to him in real-time. Another use-case for computations on data is real-time alarming and alerting, where a service handling alarms can subscribe to the appropriate computation of a stream and alert the user when the computation exceeds a certain threshold.

## 1.1 Structure

In Chapter 2 we provide an overview of cloud computing and the monitoring of cloud systems. Furthermore, the prior work in the field is reviewed and the concepts of scalability and distributed consensus are described. Some of the existing products that can be used for cloud monitoring are reviewed in Chapter 3. Their architectures are analyzed, the distinguishing features highlighted and assessments made about how complex they are to deploy in a data-center. Chapter 4 first specifies the requirements for a new distributed monitoring system and then describes the architecture which should satisfy those requirements. The design and implementation of the individual components of the prototype are then described in

detail. Verification of the prototype satisfying the set requirements is performed in Chapter 5. Finally, in Chapter 6, we present our conclusions and discuss the possibilities for future work.



# Chapter 2

## Field overview

Cloud Computing, as defined by NIST, is a [15]:

*”Model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”*

Additionally, five essential characteristics are recognized:

1. *On-demand self-service.* The users of a cloud system can provision computing resources and workloads themselves, without requiring any human contact with the cloud provider.
2. *Broad network access.* The resources provisioned are accessible through standard network mechanisms.
3. *Resource pooling.* The resources that the provider has at their disposal are pooled together to be rented to multiple users in a multi-tenant model. The resources are dynamically allocated and relocated according to demand.
4. *Rapid elasticity.* Users are able to elastically provision and release resources, sometimes automatically, according to their needs.
5. *Measured service.* Cloud systems optimize and control resource use by using metering capabilities. The use of resources can be monitored, controlled, and

reported, allowing both the provider and the user to optimize their strategy.

The monitoring systems that existed when cloud computing started to emerge were largely designed for monitoring other large-scale environments such as grid computing and traditional data-centers. These systems did not take into account the rapid elasticity of cloud resources, requiring frequent and dynamic adaptation to freshly created and destroyed resources. They also provided data and insight solely to the owner of the data-center. With cloud computing, resources are used by multiple users so the data and the reports need to take that separation into account.

The research done on the topic of monitoring cloud systems has thus largely focused on addressing the dynamicity and multi-tenancy of cloud systems.

In this chapter we will first review prior scientific work in section 2.1 and then focus on some of the theoretical concepts that are important to the area of distributed cloud monitoring in sections 2.2 and 2.3.

## 2.1 Prior work

De Chaves et al. [7] present an architecture of a system for monitoring private cloud systems. They make the assumption that multi-tenancy is not a primary objective in private clouds so they make use of an existing monitoring system without explicit support for monitoring cloud systems, Nagios. Their monitoring framework is modular and extendable, but they focus primarily on short-term monitoring and do not deal with storing and analyzing the data they gather.

Montes et al. [17] perform a hollistic analysis of cloud monitoring. They analyze the cloud as a set of different layers that need to be monitored and the monitoring itself as a set of different monitoring visions. Visions differ by their point of view of the cloud, for example, monitoring the cloud from the viewpoint of a cloud provider is different than monitoring the cloud from the viewpoint of a user. They present an architecture that enables and exposes the different monitoring visions, however, they do not focus on scalability or eliminating single points of failure of their monitoring system.

Povedano-Molina et al. [21] put a big emphasis on making their system distributed and scalable. The agents of their monitoring system communicate via a

publish-subscribe messaging system, which gives them flexibility and modularity. They do not focus on the storage or analysis of the gathered monitoring data.

Real-time analysis of monitoring data and responding to anomalous events is the focus of an article by Kutare et al. [14] Their monitoring system is based on locally processing the gathered data which is then combined with the calculated results and propagated across the system. They allow for multiple different topologies ranging from centralized to distributed ones. Since the data is processed on the same node where it is gathered, this can lead to unwanted and unpredictable load on the monitored nodes.

Some products in the industry are beginning to view monitoring as akin to processing streams of data, such as for example Monasca [16]. They make use of Apache Kafka [13] for distributed queues and Apache Storm's [22] predefined computation graphs to process streams of monitoring data, but are limited by the static nature of the predefined computations in delivering an extensible solution.

## 2.2 Scalability

Scalability of a monitoring system is the ability of a system to monitor a growing amount of work gracefully [4]. Traditionally, the metrics used to measure the scalability of algorithms [10] are *Speedup*  $S$ , defined as

$$S(k) = \frac{\text{time}(1)}{\text{time}(k)}$$

which compares the time it takes to complete the work on one processor to the time it takes to complete it on  $k$  processors. An ideal speedup is a speedup of value  $S(k) = k$ , which means that running work on  $k$  processors means we complete the work  $k$ -times faster.

A metric that is derived from speedup is *Efficiency*  $E$ ,

$$E(k) = \frac{S(k)}{k}$$

which has the ideal value of  $E(k) = 1$ . This means that the efficiency of performing work is not decreasing as we add more processors.

Finally, a metric called *Scalability*  $\psi$  from scale  $k_1$  to scale  $k_2$  is the ratio of efficiency figures,

$$\psi(k_1, k_2) = \frac{E(k_2)}{E(k_1)}$$

Its ideal value is also  $\psi(k_2, k_1) = 1$ , which means that when we move our workload to a different scale, our efficiency does not decrease.

Jogalekar et. al [11] argue that a more general form of a scalability metric is required for distributed systems because the jobs that are being run and the manner in which they are being run is more complex than an algorithm running a single job to completion. For that reason, they propose a scalability metric based on *productivity*. If the productivity is maintained as the scale changes, the system is regarded as scalable. Given the quantities

- $\lambda(k)$  = throughput in responses per second at scale  $k$ ,
- $f(k)$  = average *value* of each response,
- $C(k)$  = cost at scale  $k$ , expressed as running cost per second,

they define productivity  $F(k)$  as the value delivered per second, divided by the cost per second:

$$F(k) = \frac{\lambda(k)f(k)}{C(k)}$$

The scalability metric is then the ratio of the two scales' productivity figures:

$$\psi(k_1, k_2) = \frac{F(k_2)}{F(k_1)}$$

The system is then considered scalable from one configuration to the next if the productivity keeps pace with the increasing costs. They arbitrarily choose a threshold of 0.8 and say that the system is scalable if  $\psi > 0.8$  and not that the threshold value should reflect the acceptable cost-benefit ratio.

## 2.3 Distributed consensus

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members [18]. This allows us to build reliable systems out of unreliable components. Distributed consensus is a fundamental problem of computer science and lies at the heart of many distributed systems.

In order for us to say that a set of nodes have reached a consensus, the following conditions must be met [20]:

1. Agreement: all correct nodes arrive at the same value;
2. Validity: the value chosen is one that was proposed by a correct node;
3. Termination: all correct nodes eventually decide on a value.

A correct node is one that is currently running, so it has not stopped or it has already recovered after a stop. The consensus problem in an asynchronous system requires that agreement and validity are satisfied for any number of non-Byzantine failures and all three must be satisfied when the number of non-Byzantine failures is less than a certain threshold. Non-Byzantine failures are failures in which a node's failure doesn't result in an arbitrary behaviour of the node. The number of correct nodes required to reach an agreement is called a *quorum* and depends on the number of nodes in the consensus cluster.

A quorum is a majority of the nodes, so for a set of  $n$  nodes, it requires at least  $\frac{n}{2} + 1$  nodes. For example, a cluster of 5 nodes needs 3 nodes to form a quorum. This means it can tolerate 2 node failures and still reach consensus. If 3 nodes were to fail, a quorum would not be formed and consensus could not be achieved.

### 2.3.1 Brief overview of Raft

Raft [18] approaches distributed consensus as a problem of a replicated state machine log. If different machines have the same state machine log, they can execute the commands in order and since they are deterministic, they all eventually reach the same state.

The nodes in a Raft cluster first elect a leader among themselves. All changes to the state must then go through the leader. The election is performed with nodes requesting votes from other nodes. If a node receives votes from a majority of the nodes, it is elected leader. The leader must then send periodic heartbeats to other nodes to let them know that it is still in charge. If the leader dies, a new round of election is performed and a new leader elected.

When a client issues a new request, that request is treated as a command to be executed by the state machine. The leader appends it to its state machine log and replicates it to all of its followers' state machine logs. When the command has been replicated on a majority of nodes, the leader applies the command to its state

machine and notifies other nodes to do so as well. The log entry is then considered to be committed and Raft guarantees that committed entries are durable and will eventually be executed by all available state machines.

The state machine approach is very useful in practice. For example, if we are building a distributed consistent key-value database, we can model the key-value pairs as the state of the state machine and individual client requests to set a certain key to a certain value as log entries that are applied to the state machine.

# Chapter 3

## Product overview

Since cloud monitoring is a very active field in both research and industry, quite a few products have been developed to address the specific needs of monitoring large cloud computing environments. In the following chapter, we will provide an overview of some of these products' architecture, the complexity of deploying them and their distinguishing features.

### 3.1 GMonE

Developed by Montes et al. [17] GMonE is a general-purpose cloud monitoring tool intended to address all needs of modern cloud architectures. The authors have performed an analysis of cloud monitoring needs and platforms existing at the time and defined a general-purpose architecture aimed to be applicable to all areas of cloud monitoring.

#### 3.1.1 Architecture overview

GMonE has a plugin-based monitoring agent, called GMonEMon, that can be used to monitor both physical and virtual cloud layers. The GMonEMon uses plugins to gather data, which it then optionally aggregates and sends to the GMonEDB processes subscribed to it via Java RMI.

GMonEDB receives data from GMonEMon processes and stores it in a database for archiving purposes. Several different databases can be used for storing the data

since it uses a database abstraction. It then exposes the stored data to the user via the GMonEAccess programming library.

GMonEAccess provides a common interface to access the GMonE monitoring system and can be used to obtain monitoring data and configure and manage the GMonE infrastructure at runtime.

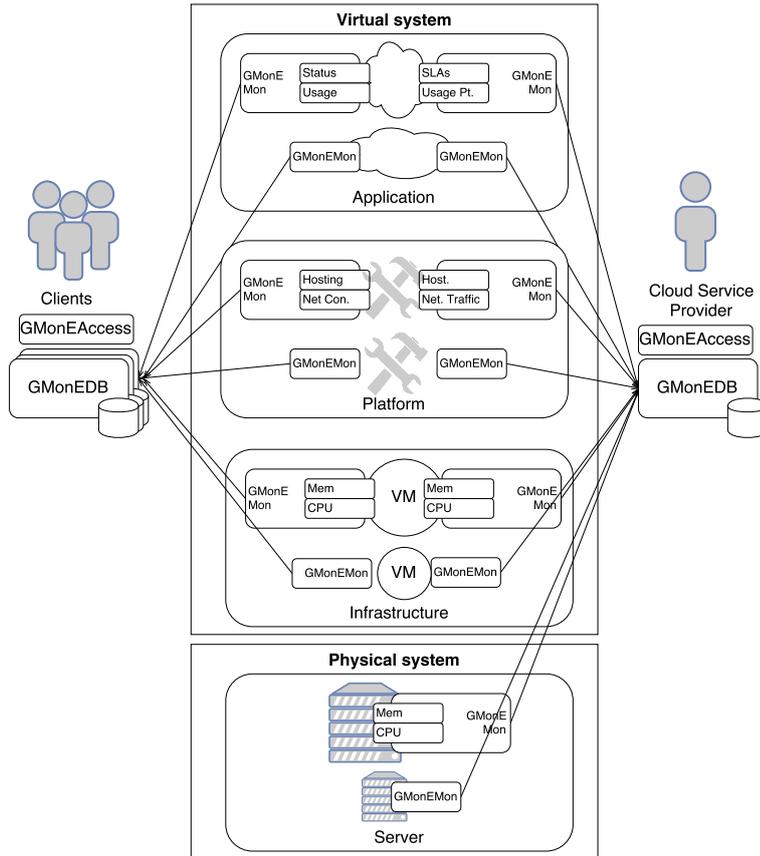


Figure 3.1: GMonE architecture.

### 3.1.2 Deployment complexity

The entire GMonE suite is contained in a single Java jar file to simplify deployment and maximize portability. The GMonEDB processes do not provide horizontal scalability so there is no need for complex coordination, instead, the required amount of GMonEDB instances should be determined up front.

## 3.2 DARGOS

Povedano-Molina et al. [21] developed a decentralized distributed monitoring architecture built atop a publish/subscribe paradigm.

### 3.2.1 Architecture overview

DARGOS is based on two processes called Node Monitoring Agent (NMA) and Node Supervisor Agent (NSA). NMA processes gather data from their local node and make that data available to interested NSAs. NSAs are responsible for collecting monitoring data from nodes and make them available to interested parties via an API.

The communication is based on the DDS standard by the OMG group [19], which describes peer to peer publish/subscribe mechanism. Publishers and subscribers discover each other automatically and match whenever they have a compatible topic.

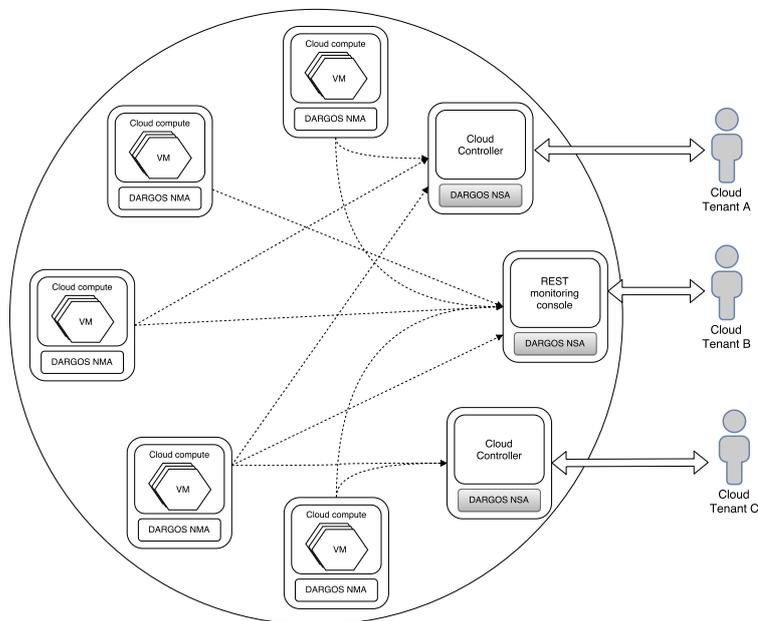


Figure 3.2: DARGOS architecture.

### 3.2.2 Distinguishing features

**Value filtering** Agents can be configured to only publish a new sample if a significant change in the monitored value occurs. For example, while CPU usage remains low, e.g. [0-25]%, the agent doesn't publish any samples. Only when the CPU usage increases outside that predefined range, a new sample is published.

**Time-based filtering** Agents can be configured to only post up to a configured number of updates per time interval in order to reduce network load.

**Host summary** If a certain NSA is interested in all sensor information from a NMA, it can subscribe to its "host summary" topic instead of subscribing to all sensor topics, which reduces network load.

## 3.3 Monalytics

Developed by Kutare et al. [14], Monalytics focuses primarily on online data analysis. It tries to keep the computations done on the data close to the source of that data and it enables dynamic adjustments to the computations being done.

### 3.3.1 Architecture overview

The architecture of Monalytics consists of Agents and Brokers. Agents collect data from virtual and physical infrastructure and do any local processing if required. Brokers gather up the data from multiple agents and perform computations on that data. The computations are modeled as computation graphs, which are executed on the Brokers present in the system in a distributed manner, which means that a Broker must pass its data to all other Brokers that need its data to perform their computations. The way the brokers are connected is very flexible and can be either centralized, a tree hierarchy, a peer-to-peer topology or a combination of the three.

The Brokers in each logical zone of the monitored cloud elect a leader amongst themselves. The Zone Leader is then responsible for deployment and configuration of computation graphs across sets of Brokers and for supervising their execution.

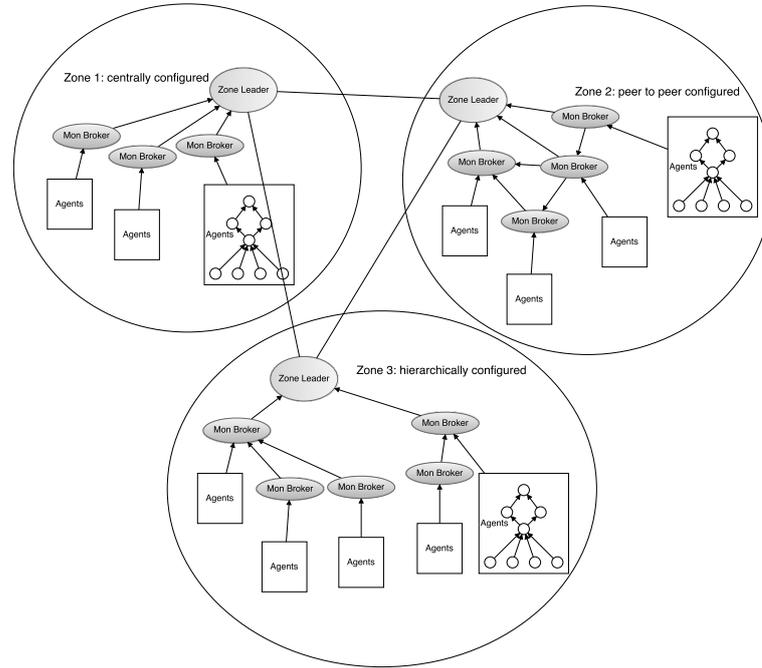


Figure 3.3: Monalytics topology.

### 3.3.2 Distinguishing features

**Computation graphs** Analysis is not centralized but is performed in a distributed manner. For example, each Broker applies data aggregation and analysis functions on the data streams received from Agents, raises alerts when necessary and then propagates its raw and/or analyzed data to other Brokers participating in the same computation graph.

## 3.4 Ceilometer

Ceilometer is the official monitoring and metering project for the OpenStack cloud platform. Development has started in 2012 to create an infrastructure for collecting measurements inside OpenStack clouds. Initially, it was focused primarily on metering but the scope has been expanded to include monitoring virtual and physical resources as well. The initial focus on metering has had a big impact

on the way Ceilometer was designed, which caused the project to struggle with performance when the scope was expanded. There is ongoing work to redesign Ceilometer in order for it to cope with the amount of metering and monitoring data that big clouds produce.

### 3.4.1 Architecture Overview

Different types of agents are used for data collection. Central Agent polls OpenStack services' APIs to collect data about users' usage. For example, OpenStack Block Storage service API is polled for details about how much disk space each user is using. Compute Agent is present on each server where VMs are run and is polling the hypervisor about the status of its VMs. Notification Agent is subscribed to the OpenStack's notification bus and collects notifications from other OpenStack services, such as notifications about VMs being created.

The agents then convert the data they gather into samples and push them through their local sample pipeline. The pipeline can consist of zero or more transformers, which perform different transformations on samples, and one or more publishers, which publish the samples. The transformers and publishers are modular and can be easily added. The default is to use a publisher to publish samples to Ceilometer's message queue, but publishing samples over UDP and HTTP is also currently supported.

If the samples are published to Ceilometer's metering queue, a Ceilometer Collector also needs to be running. The Collector reads samples from the queue and uses the configured dispatchers to either save samples to a database, POST them over HTTP or write them to a log file.

The samples that are saved to the database can be accessed via a rich query API which also supports computing statistics over time ranges.

Ceilometer has support for widely-dimensioned alarms, which means that the alarm definitions are very flexible. For example, one alarm can target the CPU usage of a single VM while another targets the average CPU usage of all VMs in a cluster. Alarms are evaluated by performing a query to the Ceilometer API to compute the statistic specified in the alarm definition, which means they are not computed in real time.

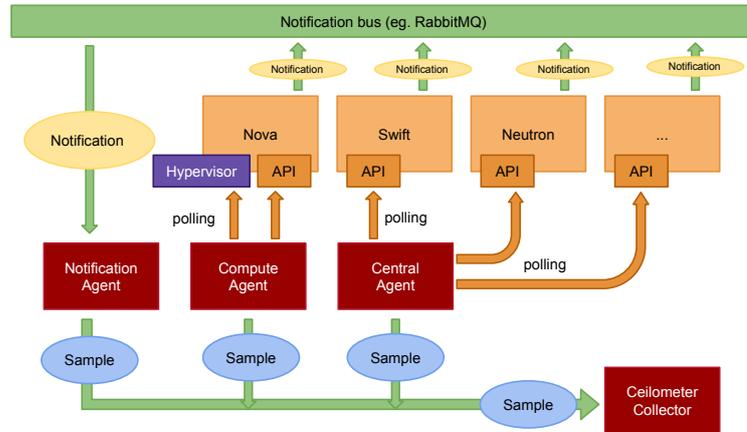


Figure 3.4: Ceilometer architecture.

### 3.4.2 Deployment complexity

A basic Ceilometer deployment does not require any additional or exotic software in an OpenStack cloud environment as it can use the existing messaging queues and databases already in place for running the cloud. When monitoring larger clouds it is advisable to deploy a dedicated queue and database for Ceilometer.

### 3.4.3 Distinguishing features

**Backend-agnostic** Ceilometer is implemented in a very backend-agnostic way.

It uses OpenStack's Oslo Messaging library for its use of messaging, which is an abstraction over different transport methods and currently supports RabbitMQ, ZeroMQ and Qpid. For storing samples it uses a storage abstraction and supports storing samples in MySQL, PostgreSQL, MongoDB and HBase.

**Rich agent-local computations** Since samples pass through a pipeline of customizable transformers before being published, Ceilometer is very flexible in how the data can be aggregated or altered locally before being published over the network.

## 3.5 Monasca

Monasca is a monitoring system for OpenStack cloud software platform, designed to be performant and scalable. The project has been started by HP and Rackspace in order to address some shortcomings of the official OpenStack monitoring and metering platform, Ceilometer. Besides gathering and storing metrics about the cloud, it also offers interactive querying of metrics and historical information via a REST API.

### 3.5.1 Architecture Overview

Monasca uses a microservices architecture. The monitoring system is composed of many small services that communicate using REST APIs. This means that the components are completely decoupled and can be easily updated or replaced.

Each system being monitored by Monasca has a Monasca Agent running on it. Monasca Agent consists of several sub-components that provide support for gathering system metric such as CPU utilization and available memory, Nagios and statsd plugins and health checks for a plethora of services such as MySQL, RabbitMQ and others.

Agents use the Monasca API service to post the metrics into the monitoring system using HTTP POST calls. The API service then publishes the data it receives onto the distributed publish/subscribe messaging system Apache Kafka. Subscribing to the fanout queue are many services that make use of the gathered data. The Persister service simply reads the data from the queue and saves it to the database. Transform Engine applies user-defined data transformation rules to the received data and pushes the transformed data back to the Kafka queue. Anomaly and Prediction Engine uses the Apache Storm real-time computation system to learn from the data, issue predictions and detect anomalies using several anomaly detection schemes like Kolmogorov-Smirnov test and NuPIC engine [9]. When an anomaly is detected, an alert is pushed to the Kafka queue. Notification engine subscribes to the Kafka topic that Anomaly and Prediction Engine posts and executes predefined actions for those alerts, such as sending a text message or an e-mail to an administrator.

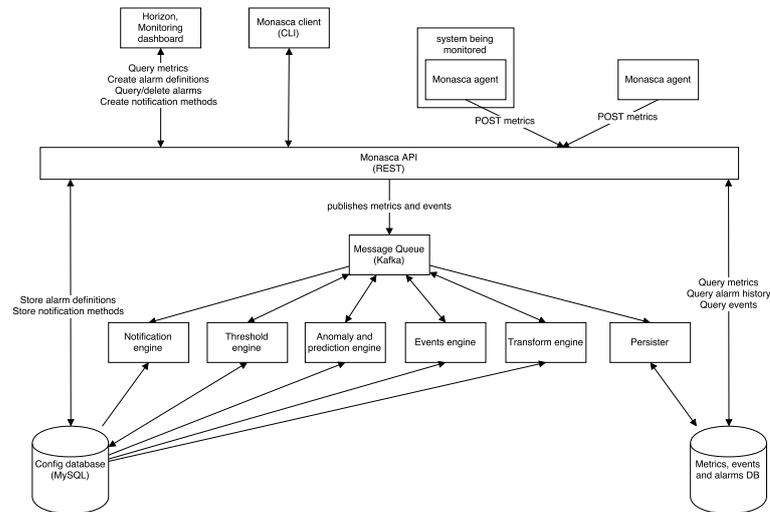


Figure 3.5: Monasca architecture.

### 3.5.2 Deployment complexity

Monasca requires a deployment of Apache Kafka distributed messaging system and Apache Storm real-time computational engine, both of which in turn require a deployment of an Apache ZooKeeper distributed configuration cluster. This makes the deployment of Monasca a rather complex task with a lot of additional services to deploy and manage.

### 3.5.3 Distinguishing features

**Alarm definitions** are predefined or user provided alarm templates. A user then doesn't have to create individual alarms for each resource or service. That is instead handled by Monasca, which creates template-based alarms for resources and services as they appear.



## Chapter 4

# Dagger: A new distributed monitoring system

Most of the existing work in the field focuses on the way data is collected in a monitored cloud system or the way it is made available to different interested parties and presented. The matter of transferring and performing computations on said data feels almost like an afterthought, even though it is of utmost importance for ensuring reliability and scalability, which are sought-after properties of a monitoring solution.

In a similar way, the computations on the data the monitoring systems gather, which is performed for the purposes of analysis, alerting and reporting, is offloaded to the system's chosen DBMS. This is often suboptimal performance-wise because it can overload the DBMS with repetitive queries that require the DBMS to re-read and re-process a big chunk of data every time a new measurement comes in, while the results only change a small amount.

To this end, we set out to design a solution for both transferring monitoring data and performing in-flight real-time computations on it. We wanted our solution to provide the architects and builders of concrete monitoring systems with a communication and computation layer on which they can build higher level monitoring systems that handle specifics such as data collection, multi-tenancy, reporting, configurable alerting etc.

In the following chapter, we first discuss and define the functional requirements

for a scalable distributed monitoring system in Section 4.1. Then we outline the concepts and architecture for the new system in Section 4.2 and finally discuss the implementation and the design of individual components in Section 4.3.

## 4.1 Functional requirements

In order to meet the needs of the operators, a cloud monitoring system is required to satisfy several properties about the way it handles faults, the sizes of the loads and the use cases it can be used for. We discuss said properties in the following section.

### 4.1.1 Reliability

Monitoring is an essential part of cloud infrastructure as it is used by critical components such as billing, SLA compliance verification and resource management [1]. That is why it is very important that the monitoring system remains functional in the face of problems such as failing nodes and network failures.

Therefore, a monitoring system must not contain a single point of failure. If one of the machines in charge of processing monitoring data fails, all of the data must still be processed and none of the data can be lost. This implies that the monitoring system must somehow keep track of what work it has already completed to insure that all the data is properly processed.

When performing a computation on a stream of data, each piece of data must be processed only once. This is important for both billing, where a customer could otherwise be charged twice, and monitoring alike, where it could trigger a false alarm on the usage of some resource.

When processing data from multiple sources, the data is not guaranteed to be received in order. The monitoring system must take careful steps to insure that all the relevant data is included in the computation.

In the case of a failure of an external system that consumes monitoring data, it is very desirable that the stream of monitoring data can be rewinded. That is, giving the data consumers an option to re-receive data from a certain recent point in time on.

### 4.1.2 Scalability

In order to provide horizontal scalability, as defined in Section 2.2, the monitoring system must be distributed. This means that the processes performing monitoring tasks are able to run on multiple networked computers and coordinate their actions by communicating [6]. Coordination in a distributed system is a nontrivial problem that is solved by coordination protocols such as Paxos or Raft.

There are two kinds of scalability when it comes to processing streams of data. The first kind is coping with an increasing number of streams. As we increase the number of streams we need to process, we must also be able to handle all the streams necessary. This can be achieved by distributing the processing of said streams onto different nodes. The second kind is coping with an increasing size of a stream. When we increase the number of records that are produced in a stream in a given timeframe, we must be able to still efficiently process that stream. To achieve this, we must have a way of distributing the records of the stream themselves to different nodes. So we need to have a way of partitioning the stream in logical partitions, depending on the type of processing we are performing.

### 4.1.3 Extensibility

The users of monitoring systems often want to perform custom computations on their data such as custom analytics or error prediction. Providing the ability to specify those computations in a programming language of choice provides a lot of value to the user. It enables them to use the language they are most proficient in or reuse their existing computation codebase.

### 4.1.4 Requirements

Based on the above analysis, we wanted to build a dynamic stream processing framework with the following requirements:

- It should enable simple publishing of and subscribing to data streams and computations on those streams.
- The computation topology should not be predefined, but dynamic and built on-the-fly as needed.

- Computations on data streams should be user-programmable.
- It should support persistent state of the computations.
- It should handle out-of-order data gracefully.
- It should deliver data exactly once.
- Latency should not increase as the system scales to more machines.

## 4.2 Architecture

We designed a dynamic real-time stream processing framework called Dagger. It is based on publishing of and subscribing to streams of data.

The design is centered around the principle that ideally, we would never have to perform the same calculation on the same data twice or in two different places. From this, it follows that whenever a computation is taking place, its results need to be made available to other nodes in the system.

This leads to a system where nodes advertise:

- what data they want,
- what data they offer.

If a node wants not only raw data, but some computation performed on it, a different node might step in to provide that computation and make the results available to the interested node. In order to do that, it needs to, in turn, receive the raw data to perform the computation on.

The nodes in the system are thus divided into three roles: *producers*, *workers* and *subscribers*.

**Subscribers** declare that they are interested in a certain stream or streams of data. **Producers** produce streams of data and deliver them to interested parties. In case a subscriber becomes interested in a transformation of or a computation on a certain stream of data, a **worker** steps in and provides that transformation or computation. The flow of data is shown in Figure 4.1.

The nodes coordinate using a strongly consistent distributed coordination layer.

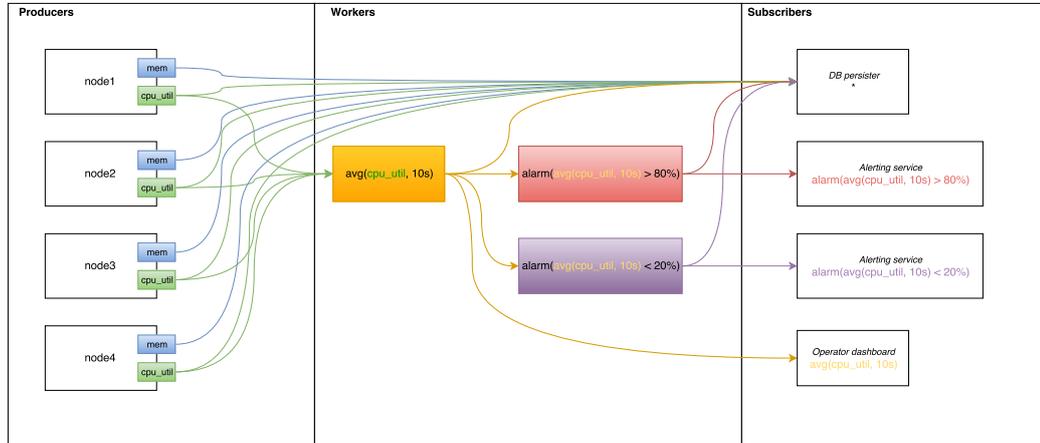


Figure 4.1: An overview of a Dagger system, where publisher nodes publish data, subscriber nodes are interested in that data and worker nodes perform and provide computations on that data.

### 4.2.1 Concepts

The smallest unit of data in Dagger is a *record*. A record is part of a stream identifiable by a name describing its contents. A computation on a stream is performed by a *task*. A task subscribes to a stream or multiple streams it needs to perform computations on and produces a stream containing the results of the computation.

#### Record

A record is the smallest unit of data in Dagger. Each record contains its own unique identifier, a timestamp, the name of the stream it is a part of, and free-form data. This means that a record can contain a floating point number, a JSON document or something else entirely.

#### Stream

Records are grouped into streams which are identified by a name and an optional list of key-value pairs called tags in the form of `streamName{tag1=value1, tag2=value2}`. The tags allow server-side filtering, so if a node wants to subscribe to

just a certain tag value, it will only receive records that have a matching tag value.

## Task

Tasks are run by workers and perform transformations or computations on one or more streams. They are identified by a name and a description in the form of *taskName(taskDescription)*. The name and the description uniquely identify a certain task, for example `avg(cpu_util, 5s)`.

The actual computation is performed by the *task plugin*. The task plugin to run is identified by the *taskName* portion of the task identifier and is then passed the *taskDescription*. From there, the task plugin can parse the free-form description and return the properties of the task to be run. That includes the streams the task needs to subscribe to and the type of the task.

Currently, there are two types of tasks: stateful and stateless. Stateful tasks maintain a state which is replicated across multiple worker nodes according to replication settings. Stateless tasks are not replicated.

### 4.2.2 Coordination

Dagger is a distributed system. In order to avoid a single point of failure, there is no central authority that handles the state of the system and distributes work. Instead, we use a distributed key-value database, which in turn uses a distributed consensus protocol like Paxos or Raft to keep the state consistent across all of the nodes. This enables us to have an approximation of a central authority, which is, in this case, a resilient cluster of nodes running a key-value database. This type of key-value databases allows us the use of several coordination primitives like locking, sessions and key expiry.

Acquiring a lock on a certain key means that only one entity at a time is holding that lock and only the entity holding it can modify the locked key. We can use these facts to implement higher level abstractions like leader election.

When a Dagger service starts up, it first registers itself with the key-value database. It does so by creating a *session*. In order to prove to the key-value database that the service is still alive and operational, the service needs to send the database a heartbeat within every predefined time interval. If a service fails to

send a heartbeat in time, it is considered inoperational and its session is destroyed. We have two options regarding the locked keys the now inoperational service had acquired - releasing the locks or deleting the keys altogether. This enables us to implement transient keys - keys that are only present as long as the entity creating them is present. In turn, we can use keys to advertise for example the resources that entity has available, but automatically delete that advertisement when the entity goes offline.

The third primitive we make use of are watches. Watches enable us to execute a long poll query to the key-value database and specify a key prefix. The query returns when any of the keys matching the prefix change. This enables us to be notified of changes we are interested in inside the key tree.

We call the interface to this key-value database the *coordinator*.

### 4.2.3 Interactions

Subscriber expresses interest in a stream of data via the coordination layer. When a producer produces a new record in a certain stream, it sends it to all subscribers that are listed in the coordination layer as interested in that stream.

When a subscriber is interested in a computation or a transformation of a stream, workers ensure that a task is being run and performing the required computation. When a subscriber subscribes to a certain computation, the task producing it might already be running because some other subscriber has expressed interest in that computation beforehand so the task has already been started.

#### Subscribing and publishing

Figure 4.2 shows the interactions between publishers, subscribers and the coordination layer. The producer that wants to publish stream X must first register itself with the Coordinator as a publisher of stream X. Doing so enables the potential subscribers determine whether there are any publishers at all producing said stream. After registering, it sets up a watch on the subscribers subtree of stream X. That way, it receives an updated list of subscribers every time the list of subscribers to stream X changes.

When a subscriber subscribes to stream X, it creates a transient entry in the

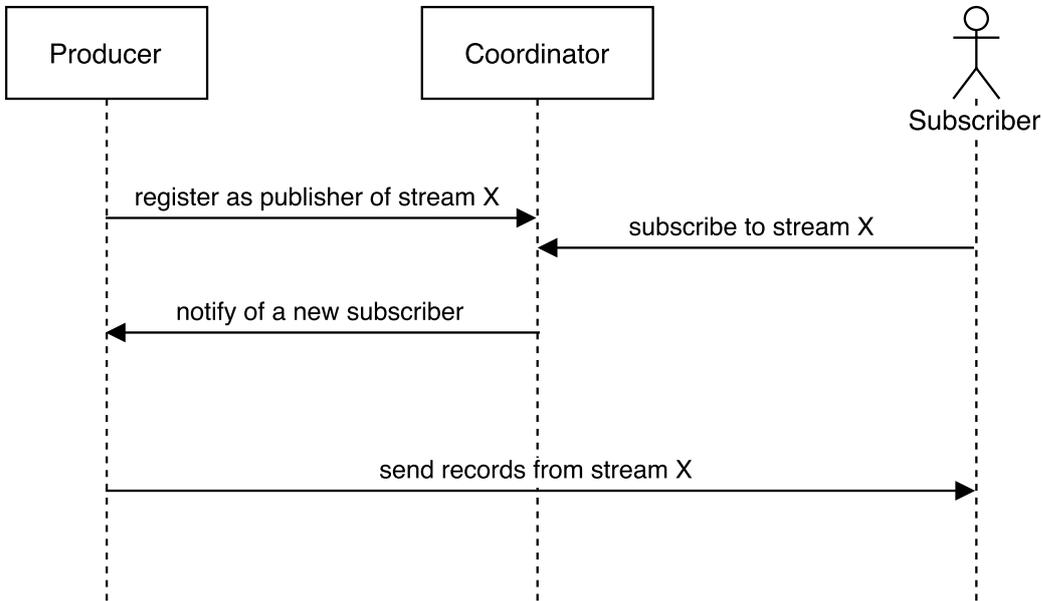


Figure 4.2: A diagram of interactions between publishers, subscribers and the coordination layer in a Dagger system.

stream X subtree, noting that it is interested in receiving records belonging to stream X. Because the entry is transient, it is deleted when the subscriber goes offline.

All the producers of stream X thus receive a notification when a subscriber joins or parts the stream X’s subscribers list and can then send the interested subscribers the appropriate records as they stream in.

### Task management

Task streams are treated exactly the same as regular data streams. When a subscriber subscribes to a stream, it checks whether there are any producers publishing this stream, as shown in Figure 4.3. If there are none (or not enough), it creates a task request in the task list. The task list contains tasks that are in need of a worker.

Worker nodes watch for changes in the task list. When a new available task is added, they try to acquire a lock on it. The node that successfully acquires the lock then starts executing the appropriate task.

#### 4.2.4 Data transfer

The first option we considered for transferring records between nodes was a regular message queue. It quickly became apparent that while the message queue paradigm suffices for the subscribers to discover publishers, it does not provide the capability for the potential publishers to be notified of subscribers wanting a yet non-existing data.

That's why we chose to use our coordination layer for discoverability and a more barebones approach like TCP for data transfer. There are two options to consider when implementing a server-client data transfer. The first is the pull model, where the clients ask the server for new data. If the server has new data available for the client, it returns it as a response. The benefit of this is the natural batching of data to be transferred, as the server is batching the data until the client is ready to receive them. This also allows for easy transfer control as the receiver is never overwhelmed and can process the data at its own pace. On the other hand, in cases where there isn't any new data waiting for the client, the client needs to use long polling. This means that the server does not reply to a request until it has new data for the client. This results in a lot of open connections and unnecessary communication in cases where the data rate is slower than the connection timeout.

With the push model, the client doesn't need to keep checking for new data. When new data becomes available, the server simply sends it to the client. When data is scarce, the server doesn't need to keep a connection open indefinitely. It can simply open a new connection if the previous one has already timed out. We do need to take special care to both maximize the available bandwidth and processing capabilities of the client on one hand and to not overwhelm the client with data on the other. We maximize the bandwidth by sending records to the client as fast as we can without waiting for an acknowledgement. But in order to avoid overwhelming the subscriber, we set a limit for a maximum number of yet unacknowledged records. When we reach this maximum, we wait until it drops below the limit until we start sending records again.

We implemented the communication as remote procedure calls, similar to Aki-dau et. al [2] for MillWheel. Each node that wants to receive records must expose a method that allows the publisher to send it a record. When a receiver receives a record via this method it must then process it, which usually means routing it to

its subcomponents interested in the stream that the received record belongs to.

### 4.2.5 Timestamp ordering of streams

If a node is receiving a stream from multiple upstream publishers, like, for example, receiving the stream `cpu_util` from all the nodes that run virtual machines, the records aren't guaranteed to be received in order. When we receive a record with timestamp  $t$  from publisher A and a record with timestamp  $t+2$  from publisher B, we cannot be sure that we have the complete sequence of records with timestamps  $[t, t+2]$ . There could be a record from A that is delayed on the wire with timestamp  $t+1$ , as is illustrated in Figure 4.4.

That's why we include a low watermark timestamp (LWM) with our records [2]. A low watermark is a timestamp that indicates that all of the data up to it has been received. It is calculated per each node and stream and is included in the records that the particular node produces in that stream. This way a subscriber can be sure it has received all the records from an upstream publisher up to a certain timestamp and can trigger the necessary processing on earlier records.

### 4.2.6 Persistence

When performing a task on a stream, it is very often necessary to keep some state in between processing individual records. For example, when calculating a sum of values, we need to keep a running total. A task also needs to keep track of which records it has already processed so as not to process the same record twice in case of a retry. And finally, the records that a task produces must be persisted so they are not lost in case the node executing the task fails before the produced records are delivered to the subscribers. In order to avoid the per-record additional latency and round trip to an external database, while also reducing the deployment complexity, we store the task state locally.

### 4.2.7 Historical subscriptions

Since we are already persisting produced records so that they are not lost before delivery, we decided to provide the ability for subscribers to subscribe to a stream onward from a certain timestamp. For example, if a producer without support

for historical subscriptions produces some data before any subscriber wishing to receive that data is online, no one receives the produced data and the records are lost. With historical subscriptions, the producer persists the record into an output buffer where they are kept for a configurable time period such as an hour, a day or a week, depending on the amount of data produced. When a subscriber subscribes to the producer's stream, it can either receive records from the time it has subscribed on, or it can receive records from a recent timestamp on. In practice, this enables us to for example re-run an external consumer process after discovering and fixing a bug in its logic.

### 4.2.8 Failover

A node executing a certain task might fail. Since we do have the task state persisted to disk, we could just restart the node and continue our processing. This approach results in a time frame in which no records are being processed by this task and so the subscribers are not getting real time results. It also fails completely when the node is unable to restart because of for example a disk failure.

To alleviate this, we provide the option to run tasks in a master/slave replication mode. This way, the same task is run on multiple nodes in parallel. Each node is being sent the same records, is performing the same processing and is persisting the same state. They use the coordination layer to participate in a *task group* and elect a master between them. Only the master task actually pushes the data to subscribers. The illustration of what happens when the master task goes offline is provided in Figure 4.5.

### 4.2.9 Synchronization

When running a task in a master/slave replication mode, the workers that are part of the task group need a way to synchronize the task's state between them. That's why when a slave task is first started, it must first synchronize its state with the current master. It does so by first looking up the current master's address and then submitting a synchronization RPC to the master. The master replies with the snapshot of the task's state. The slave applies the state snapshot to its task and uses additional information stored in the snapshot to subscribe to the necessary

input streams from the correct timestamps on. The historical subscriptions come in handy with synchronization, because the master and slave tasks do not need to proceed in lockstep. The master (and other slaves) can continue to process records while the new slave is in the process of synchronization. When the slave finishes applying the snapshot, it can start processing records from the point in time to which the master had gotten before it has created the snapshot.

#### 4.2.10 Scalability

In comparison to the systems that use a database for calculations or a single message queue for all their metrics, our use of fine-grained streams provides us with a certain level of scalability in itself. If we are performing a task on individual fine-grained streams, we can simply add worker nodes that will take over those tasks as we add more streams, as is demonstrated in Figure 4.6.



Figure 4.3: A diagram of interactions between a node that is interested in some data that does not yet exist, and a worker that will provide that data for the subscriber.

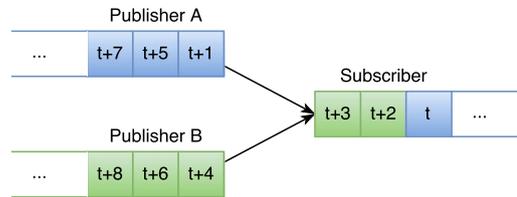


Figure 4.4: When receiving from multiple publishers, we cannot expect to receive the records in the correct order.

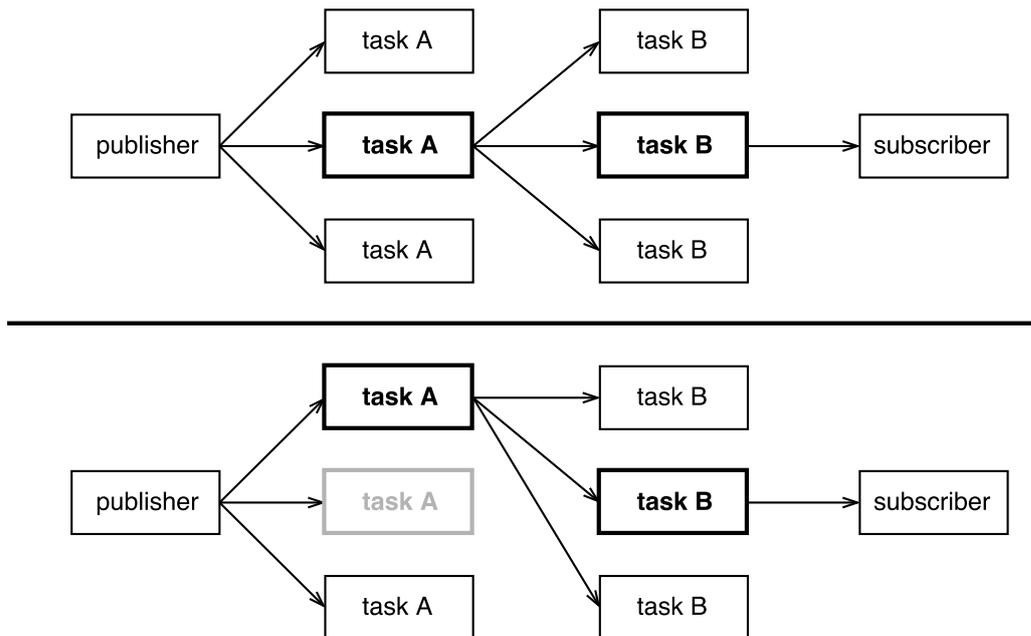


Figure 4.5: The upper half of the figure depicts a situation where tasks A and B are replicated onto three workers. All the workers receive records, but only the master sends its results forward. The lower half of the figure depicts what happens when a master node goes offline; one of the slave nodes takes over as master and starts sending its results forward.

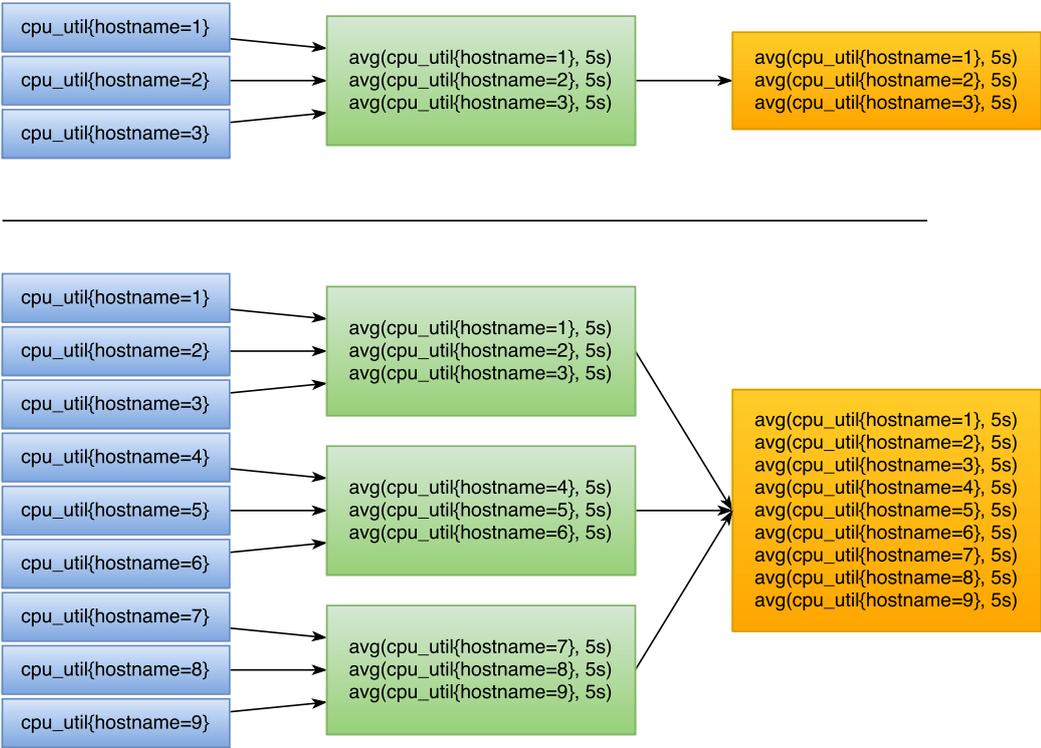


Figure 4.6: Adding worker nodes to process additional streams.

In case we want to perform a task on, for example, all of the streams in the system, we provide the so-called *matching* functionality. Matching is similar to Apache Storm’s stream grouping [22] but in reverse. Instead of partitioning a big stream onto a preexisting number of tasks, with matching, we specify the grouping we want and then new tasks are spawned automatically when new groups appear.

Say we’re interested in average CPU usage of our clusters. We could subscribe to the stream of averages for specific clusters by hand, for example:

```
avg(cpu_util{cluster=app}, 5s)
avg(cpu_util{cluster=db}, 5s)
avg(cpu_util{cluster=memcached}, 5s)
```

Instead of subscribing to different streams providing that computation by hand, we can simply use the matching mechanism to subscribe to all those streams at once by subscribing to `match(cpu_util by cluster in avg($1, 5s))`.

Subscribing to this stream will provide us with records from all the above streams, with the added benefit that if we spawn a new cluster, the matching task will automatically subscribe to the stream calculating its average. So if a new stream `cpu_util{cluster=monitoring}` appears, the stream `avg(cpu_util{cluster=monitoring}, 5s)` will be added to our matching stream.

You can even match by multiple tags. Subscribing to

```
match(
  disk_usage, disk_iops
  by host, device
  in alarm(max($1, 5min) > 90% or
           avg($2, 1min) > 80%)
)
```

will provide us with the specified alarms but with calculations grouped by *host* and *device* tags. So if there are the following streams in our system:

```
disk_usage{host=h1,device=/dev/sda1atompdf}
disk_usage{host=h1,device=tmpfs}
disk_usage{host=h2,device=/dev/sda1}
disk_usage{host=h2,device=tmpfs}
disk_iops{host=h1,device=/dev/sda1}
```

```
disk_iops{host=h1,device=tmpfs}
disk_iops{host=h2,device=/dev/sda1}
disk_iops{host=h2,device=tmpfs}
```

four alarm tasks will be created, one for each *host* and *device* combination:

```
alarm(
  max(disk_usage{host=h1,device=/dev/sda1}) > 90%
  or avg(disk_iops{host=h1,device=/dev/sda1}) > 80%
)
```

```
alarm(
  max(disk_usage{host=h1,device=tmpfs}) > 90%
  or avg(disk_iops{host=h1,device=tmpfs}) > 80%
)
```

```
alarm(
  max(disk_usage{host=h2,device=/dev/sda1}) > 90%
  or avg(disk_iops{host=h2,device=/dev/sda1}) > 80%
)
```

```
alarm(
  max(disk_usage{host=h2,device=tmpfs}) > 90%
  or avg(disk_iops{host=h2,device=tmpfs}) > 80%
)
```

On the other hand, matching just by the *device* tag would give us only two alarms:

```
alarm(
  max(disk_usage{device=/dev/sda1}) > 90%
  or avg(disk_iops{device=/dev/sda1}) > 80%
)
```

```
alarm(
  max(disk_usage{device=tmpfs}) > 90%
```

```
    or avg(disk_iops{device=tmpfs}) > 80%
  )
```

This type of usage is very useful for the concept of alarm templates [16], where we define one alarm that then applies to several different groups of streams.

We can also use the matching functionality to perform computations on streams that are too large to be processed by a single worker. For example, if we want to see the number of all incoming requests in a data-center, we could subscribe to `count(request)`. But this would pull in streams of `request` from all the hosts in the data-center, which could be too much for one worker. Instead, we can subscribe to

```
sum(
  match(
    request by hostname
    in count($1)
  )
)
```

which will result in counts being computed in a separate task for each different hostname. After that, the aggregated counts will be summed together. By combining computations this way, we can perform calculations on streams that are too large for one single worker alone.

### 4.3 Design and implementation

When deciding on which technologies to use for implementation, our guiding principles were

1. performance,
2. prototyping speed,
3. deployment complexity of the final product.

In addition, we also designed Dagger in a modular way so many backing technologies like the distributed key-value store or the persistence layer can be interchanged with alternative libraries or solutions.

In the following section, we discuss the technologies and approaches chosen for implementation and the design of individual components.

### 4.3.1 Programming language

A lot of modern distributed systems are written in Java, such as Apache Kafka, Twitter Storm and Zookeeper. Java provides a rich ecosystem and an optimized performant runtime. But it also requires a fairly large runtime, which can be a burden in cases like containerized deployments, where we want the containers to be as lean and overhead-free as possible.

Recently, new distributed systems that are written in Go, Google's up-and-coming systems programming language with powerful concurrency mechanisms [8], have started emerging. It enables programmers fast prototyping and provides a garbage collector, but it is actually compiled to machine code so speed is not compromised.

Go programs usually compile to a single binary file that can be run without requiring the deployer to install additional software.

Fast prototyping, powerful concurrency mechanisms and lightweight deployment are the reasons we chose Go for our implementation.

### 4.3.2 Distributed consensus

Distributed consensus, as discussed in Section 2.3, is in practice commonly handled by tools that encapsulate distributed consensus into a set of coordination primitives. Most of the established distributed systems use Apache Zookeeper[3] for coordination primitives. Since Zookeeper requires a JVM, requiring its use with Dagger would negate the benefits we gained by choosing a non-JVM programming language.

Thankfully, there are several alternatives to Zookeeper in the Go ecosystem, the foremost of which are Etcd and Consul. Both Etcd and consul are first and foremost distributed key-value stores. They both use the RAFT [18] algorithm to ensure consistency of the data in the face of network and node failures.

With Dagger, we make heavy use of ephemeral keys. Ephemeral keys are keys that are deleted when a node that has created them goes offline. A node going

offline can have different definitions, the easiest of which is the TTL mechanism. If a node fails to submit a heartbeat to the key-value store in time, the key-value store will consider that node offline.

With Etcd, ephemeral keys are implemented with a per-key TTL. This means that in order to keep a key from being deleted after we have created it, we need to submit a heartbeat for it within a certain timeframe. This holds for all the keys we create, so if we want a thousand ephemeral keys, we need to submit a thousand heartbeats within a certain timeframe, one for each key.

Consul, on the other hand, has a concept of sessions. Each node can create a session and then associate its ephemeral keys with that session. The node only needs to submit a heartbeat for the session, not for every key. So a thousand ephemeral keys only require one heartbeat. This approach better suits our use-case, so we chose Consul as the distributed key-value store to provide us with coordination primitives.

### 4.3.3 Components

In addition to Dagger agents forming a directed acyclic graph of publishers, computations and subscribers, the agents themselves are implemented as a tree of objects that know how to process a record, as shown in Figure 4.7. So when a record is received, it starts out at the root of the tree and is then processed by the subtrees.

The record processing objects satisfy the *RecordProcessor* interface, which means it must simply know how to process a record and return an error if it has failed to do so.

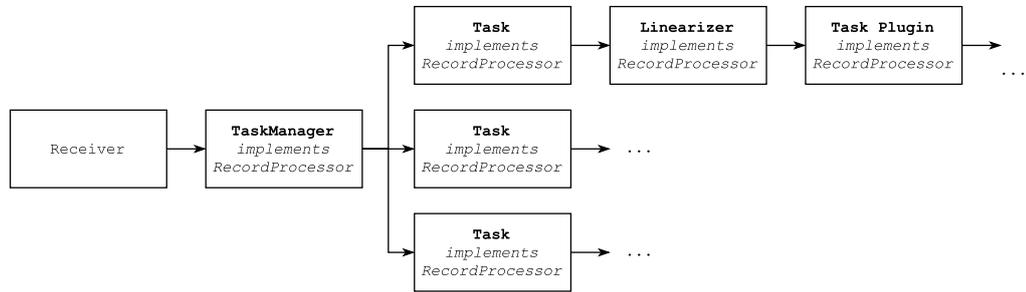


Figure 4.7: A tree of RecordProcessor objects process every record that comes into the node.

Composing different record processors into different trees gives us the three distinct agents. Each type of agent uses the components it needs to achieve its task. Which components are used by which agents are shown in Table 4.1. To illustrate how the components are tied together in an agent, the Figure 4.8 shows the flow of the records through the agent and the coordination that takes place between components. The relationships between the individual components in a particular agent is shown in a UML class diagram in Figure 4.9.

	Receiver	Linearizer	Task manager	Output Buffer	Stream Dispatcher
Subscriber	x	x			
Worker	x	x	x	x	x
Publisher				x	x

Table 4.1: Table showing which components are used by which agent types.

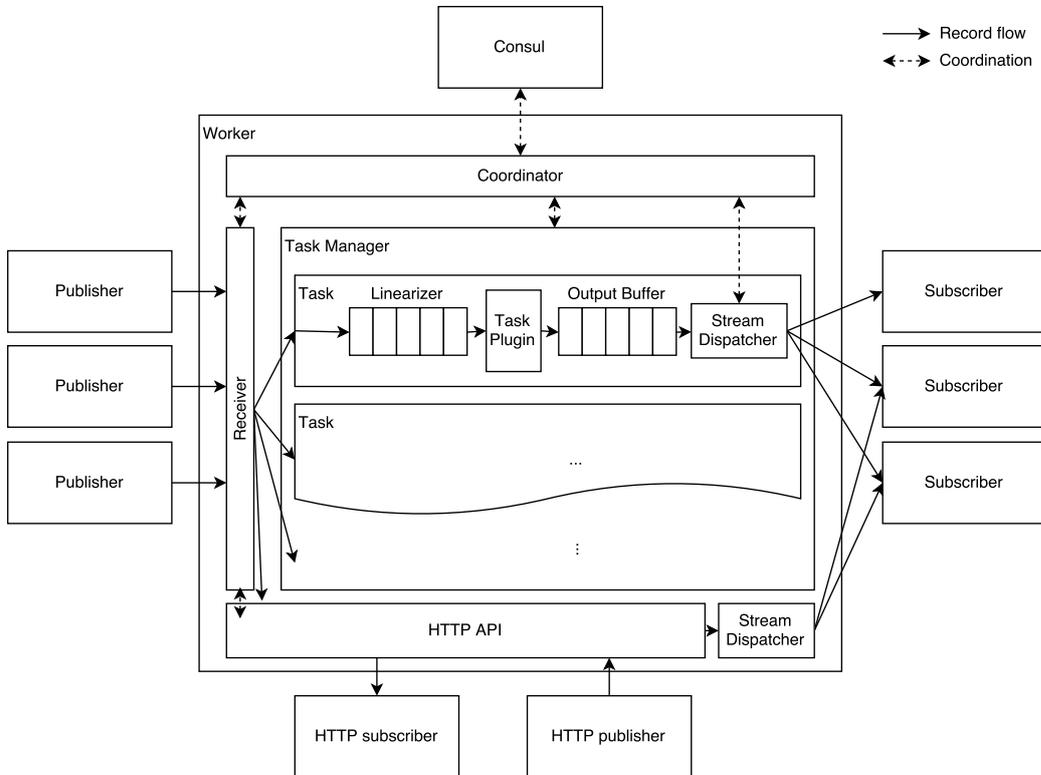


Figure 4.8: A diagram of Dagger components composed into a worker. The diagram shows how the components are tied together in a particular agent and how the records flow from one component to another.

## Coordinator

The *Coordinator* is an interface to the chosen coordination layer. It is designed as a generalized set of instances that can be implemented by different concrete implementations in order to support multiple distributed consensus systems in addition to Consul, like Zookeeper or Etc. The *Coordinator* interface is divided into logical subinterfaces, which enables components to request a subset of the coordination functionality, which eases testing and encourages good design.

The *Coordinator* interface is composed of the following methods and subinterfaces:

**Start()** Starts the coordinator

**Stop()** Stops the coordinator

**RegisterSession()** Registers a new session

**RenewSession(string)** Renews an existing session

**SubscribeCoordinator** Subinterface for subscribers

**PublishCoordinator** Subinterface for publishers

**TaskCoordinator** Subinterface for managing tasks

**ReplicationCoordinator** Subinterface for replication and failover

The chosen distributed consensus backend, Consul, provides the coordination primitives in the form of a key-value store. We make use of this key-value store by maintaining a distributed tree of keys and values that describe which agents are publishing which streams, which agents are subscribing to which streams, who is the leader of the task group and a list of tasks that need to be run.

Keys in Consul are strings that are usually logically separated by the slash ("/") character. This separation combined with the ability to query all the keys with a given prefix gives us the ability to emulate a tree. The outline of the key-value tree we use is as follows:

```
dagger/  
  publishers/  
    <streamID>/  
      <sessionID>  
      <sessionID2>  
      ...  
    <streamID2>/  
      ...  
  subscribers/  
    <streamID>/  
      <tcpAddress>  
      <tcpAddress2>  
      ...
```

```

    <streamID>/
      ...
tasks/
  <task>
  <task2>
  ...
task_groups/
  <streamID>/
    publishers_leader
  <streamID2>/
    ...

```

The components that subscribe to streams use a subset of the *Coordinator*'s functionality, namely the methods that allow subscribing and unsubscribing and ensuring that there are enough publishers publishing a certain stream. The *SubscribeCoordinator* interface consists of:

**SubscribeTo(...)** Subscribes to stream `streamID` by creating a key containing the node's TCP address `tcpAddr` at the path `dagger/subscribers/<streamID>/<tcpAddr>`

**UnsubscribeFrom(...)** Unsubscribes from a stream by deleting the node's TCP address `dagger/subscribers/<streamID>/<tcpAddr>`

**EnsurePublisherNum(...)** Ensures there are enough publishers for stream `streamID` by watching the path `dagger/publishers/<streamID>/` and counting the number of its children. If there are not enough publishers listed, we create a new task by creating a key `dagger/tasks/<streamID>`.

**CheckpointPosition(...)** Checkpoints our position in a stream by updating the value at key `dagger/subscribers/<streamID>/<tcpAddr>` to contain our position in the stream.

**WatchTagMatch(...)** Notifies when a new stream appears that matches the specified topic and tags, used by the matching mechanism. It does so by watching the prefix `dagger/publishers/<topic>`. When a new stream

appears that has the same topic, the method checks if the stream's tags match the desired ones. If they do, it reports a new matching stream.

The components that publish streams use the *PublishCoordinator* interface:

**RegisterAsPublisher(...)** Register as a publisher of stream `streamID` by creating a key that contains the node's session ID at path `dagger/publishers/<streamID>/<sessionID>`.

**RegisterAsPublisherWithSession(...)** Register as a publisher of the specified stream, but with a custom session. Used by the HTTP API component, where the publishes that use the HTTP API have their own session IDs.

**DeregisterAsPublisher(...)** Deregister as a publisher of stream `streamID` by deleting the key `dagger/publishers/<streamID>/<sessionID>`.

**WatchSubscribers(...)** Notifies when a new subscriber subscribes to stream `streamID` by watching the prefix `dagger/subscribers/<streamID>/`.

**GetSubscriberPosition(...)** Check from which timestamp on we should send the subscriber records by reading the value at key `dagger/subscribers/<subscriberTCPAddr>`.

**WatchSubscriberPosition(...)** Notify when a subscriber updates its position in a stream by watching for changes at key `dagger/subscribers/<subscriberTCPAddr>`.

The components that compete for and execute tasks use the *TaskCoordinator* interface:

**WatchTasks(...)** Notify when tasks new tasks appear or are deleted by watching the changes at the prefix `dagger/tasks/`.

**AcquireTask(...)** Try to acquire the task `streamID` by trying to acquire a lock on the key `dagger/tasks/<streamID>`.

**TaskAcquired(...)** If successful in acquiring the lock, mark it as acquired so no-one tries to acquire it by deleting the key `dagger/tasks/<streamID>`.

**ReleaseTask(...)** Release the task to someone else by releasing the lock on the key. Used in case of a problem setting up the task.

The components that are replicated in a master/slave failover scenario use the *ReplicationCoordinator* interface:

**JoinGroup(streamID StreamID) (GroupHandler, error)** Join a group by setting up a task that keeps track of the current leader and tries to acquire leadership when the leader does not exist. The leader of a group is identified by its TCP address written as a value at key `dagger/task_groups/<streamID>/publishers_leader`. At the beginning of the task and on change of the key, the group members try to acquire a lock on said key. If they succeed, they store their own TCP address as the value and thus become the leader of the group.

## Receiver

The receiver component has two responsibilities. The first is exposing an RPC interface that publishers use to send it streams. The second is routing the received records to other interested components.

Components use the following methods to subscribe and unsubscribe from streams:

**SubscribeTo(StreamID, Timestamp, RecordProcessor)**

**UnsubscribeFrom(StreamID, RecordProcessor)**

On the RPC side, the receiver exposes the following methods:

**SubmitRecord(record Record) string** A publisher that is publishing a stream and knows that a receiver at a certain TCP address is interested in that stream then calls the `SubmitRecord` method. This method then routes the record to other interested components in the node.

**Sync(streamID StreamID) []byte** When a task running on this receiver's node is a leader of a group, slaves can use the `Sync()` method to receive a byte array snapshot of that task.

## Task Plugin

In order to provide maximum flexibility to the users, task plugins can be implemented in any programming language of choice. The worker node runs the

provided script or executable and communicates with it over standard input and standard output. The communication is textual and uses JSON-RPC [12] as the protocol. Both sides can utilize abstract away the JSON communication and operate with RPC methods only. As the method calls are encoded in JSON, the parameters passed must also be JSON encodable.

The plugin must expose the following RPC methods:

**GetInfo(definition string) (ComputationPluginInfo, error)** The worker uses

the `GetInfo()` method to pass the plugin the task's definition string. The plugin can parse the definition in any way it wants and returns additional task information to the worker. The definition is the user-defined free-form string that is specified in the stream ID, e.g. `task(definition)`. Usually, the definition includes user-provided information about what the task needs to do. If a task's stream ID is `avg(cpu_util, 5s)`, "*avg*" is the task identifier that defines which task plugin will be run, and "*cpu\_util, 5s*" is the definition string. The plugin parses that definition and derives that it needs to calculate an average of the `cpu_util` stream ID over five seconds. The information about which streams the task needs to subscribe to is returned to the worker along with information about whether the task should be stateful or stateless.

**SubmitRecord(t \*Record) ([]\*Record, error)** When a worker receives a new

record from the stream the task is subscribed to, it passes the plugin that record via the `SubmitRecord()` method. The plugin performs the computation, saves its state and returns the list of records that were produced by processing the received record. The list may be empty or it may contain multiple records.

**GetState() ([]byte, error)** If the task is a master of its task group, slaves will request snapshots of the task. The worker gets a snapshot of the plugin's state by using the `GetState()` method.

**SetState([]byte) error** When a task is not the master but a slave, the worker will get the snapshot from the master and apply the state to the plugin via the `SetState()` method.

## Task

A Task is a component that orchestrates other components into a unit of work. It is a *RecordProcessor* so theoretically, it could simply receive the record and throw it away. But usually, Tasks compose other components such as a *Linearizer* to sort the records from multiple sources by their timestamps, a *Task Plugin* to perform work on the record, and a *Stream Dispatcher* to deliver the potentially produced records to interested subscribers.

Usually, tasks are also replicated among multiple workers. That's why tasks must provide an interface that allows slave nodes to request the Task's snapshot and to set up a new task from an existing snapshot gotten from a master node. So in addition to implementing a *RecordProcessor* interface, a Task component must provide the following methods:

**GetSnapshot()** (`[]byte, error`) Returns a byte array snapshot of the task's state. The task must gather and include state from all of its subcomponents such that an identical task can be recreated on another node using the provided information.

**Sync()** (`Timestamp, error`) Synchronizes a newly created task with the master node. This means getting a snapshot of the state from the master node and applying that snapshot to the newly created task's state. It also returns a timestamp of the last record that was processed and included in the received snapshot, so we know from which point on to subscribe to our input streams.

**Run(`chan error`)** Run starts executing the task, which means that the task is actively processing incoming records.

**Stop()** Stops the task.

## Task Manager

The Task Manager is in charge of managing tasks on a worker node. It uses the coordination layer to watch for available tasks and tries to acquire them when they appear. If it succeeds in acquiring the task, it creates a new *Task* object, synchronizes it with a master node and starts the task's plugin (if any). When

a task is no longer necessary, the Task Manager stops the task and its plugin gracefully.

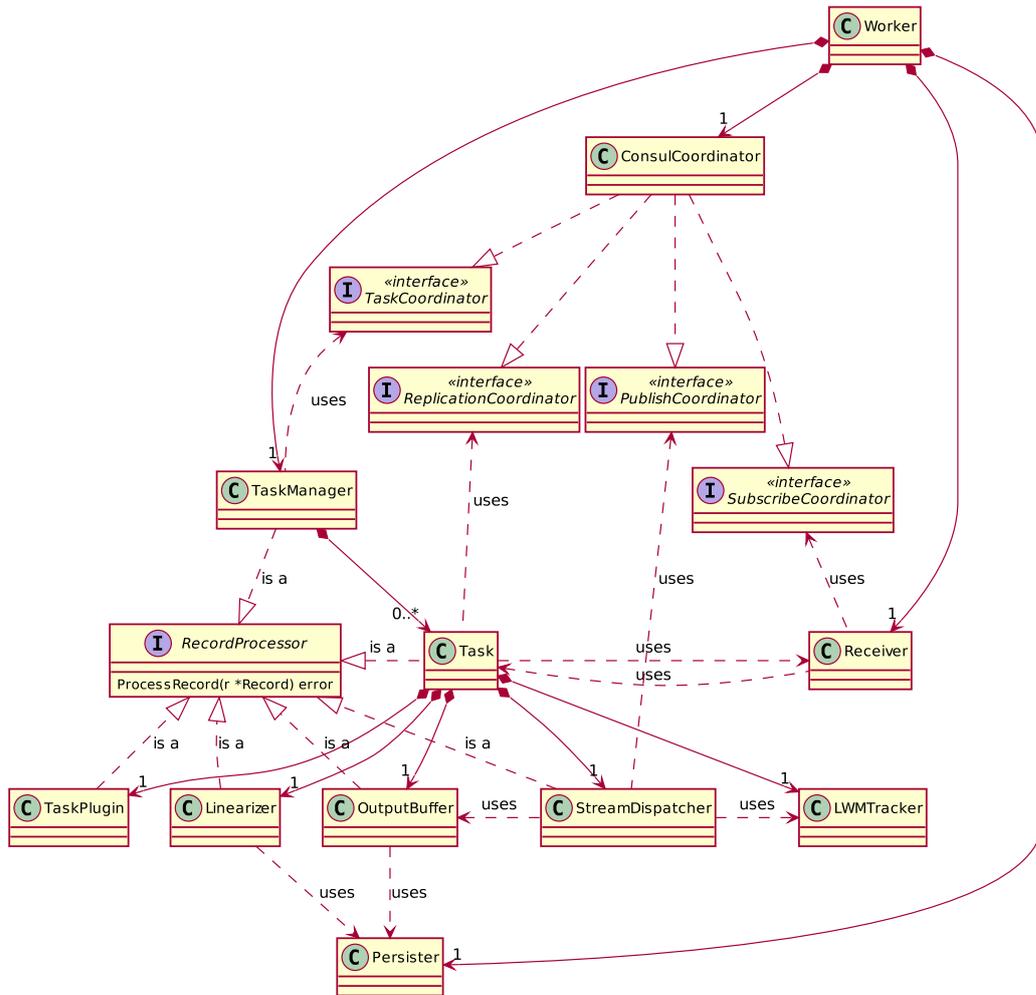


Figure 4.9: An UML class diagram of the Worker agent displaying the relationships between individual components.

## Linearizer

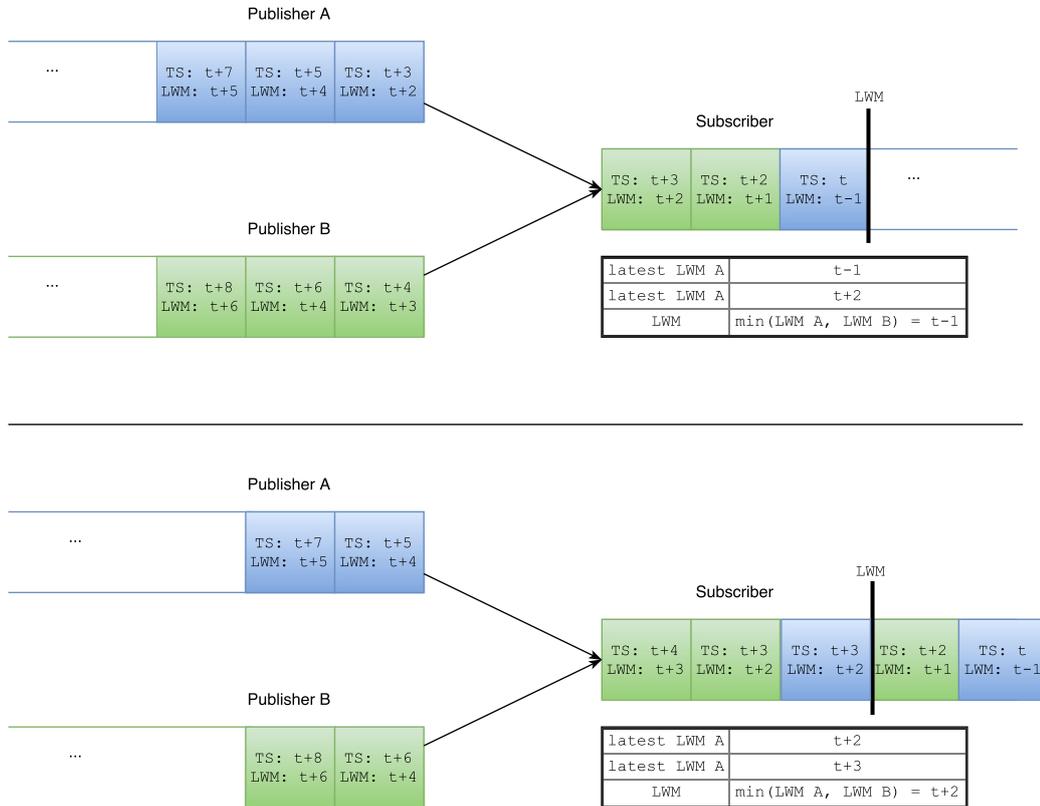


Figure 4.10: LWM value tells us that no more records with timestamp earlier than the LWM will be arriving from this publisher. This means we can process records up to the timestamp that is the minimum of all publishers' LWMs.

When receiving records from multiple sources, the order in which we receive them is not necessarily in increasing timestamp order. The *Linearizer* component buffers and sorts the incoming records and forwards them to other *RecordProcessors* in increasing timestamp order. It does so by maintaining a sorted buffer of received records and only forwarding the records up to a point to which it is sure no more records will arrive and be inserted. It decides when and how much of the buffer to flush by using the LWM values in the received records, as shown in Figure 4.10. The LWM value in a record is a timestamp that tells us that no more

records with a timestamp earlier than LWM will be arriving.

### **Stream Dispatcher**

Using the coordination layer, the Stream Dispatcher watches for the subscriber to the stream it is dispatching. This means that it gets notifications when there is a new subscriber or when an existing subscriber unsubscribes. Each subscriber has its own iterator, which means it is being sent the records at its own pace. The records are sent asynchronously to utilize as much of the available bandwidth as possible, but we allow only a maximum of a predefined number of unacknowledged records at any given time. This prevents us from overwhelming a slow client and to adapt to its processing speed.

The Stream Dispatcher is also performing server-side filtering of streams. If a subscriber is subscribed to for example `cpu_util{hostname=abc}`, the Stream Dispatcher will only send it the records that contain a tag `hostname` with value `abc`.

### **HTTP API**

The primary method of communication for third-party software is via the HTTP interface, which allows publishing records and subscribing to streams. The HTTP API is part of the worker nodes, because it needs to both subscribe to streams and receive records, and publish them at the same time.

Downstream subscribers rely on LWM timestamps to know when no new data is coming from a certain producer. Until the LWMs of all producers pass a certain timestamp, subscribers generally cannot be sure that no more records will be incoming that have a timestamp earlier than that. But if a producer is using the HTTP API to publish records, how can the HTTP API know when the producer will not produce any more records? That's why the producer that is using a HTTP API must register a publishing session before publishing any records and it must maintain that session by executing heartbeat requests. This way, when a heartbeat request isn't received in time, the HTTP API can mark the producer as finished and the subscribers will stop waiting for new records from it.

The endpoints that the HTTP API exposes are described in Appendix A.

# Chapter 5

## Pilot deployment and requirements verification

We set up a pilot deployment of our new system in order to assess the deployment complexity and verify that the requirements we have set for our prototype have been satisfied. In this chapter we will first describe our environment and deployment details, then we will describe which tests we performed and finally we will present and analyze the outcome of the testing.

### 5.1 Infrastructure outline

The pilot deployment of Dagger consists of three worker processes running on three Amazon EC2 `t2.micro` instances. The `t2.micro` instances have one CPU core and one gigabyte of memory each, running the Ubuntu 14.04 LTS operating system. Since Dagger consists of a single binary, no additional software is needed on the worker nodes.

The Dagger worker processes coordinate via a single Consul process running on its own EC2 instance. A highly available Consul cluster would require at least three nodes, but since the mode of Dagger operation does not change with the amount of Consul nodes, we can assume without loss of generality that our Consul node is stable and thus use only one in our testing.

The tests are run off the fifth EC2 instance, which simulates both producers

and subscribers. The outline of the full testing deployment is shown in Figure 5.1.

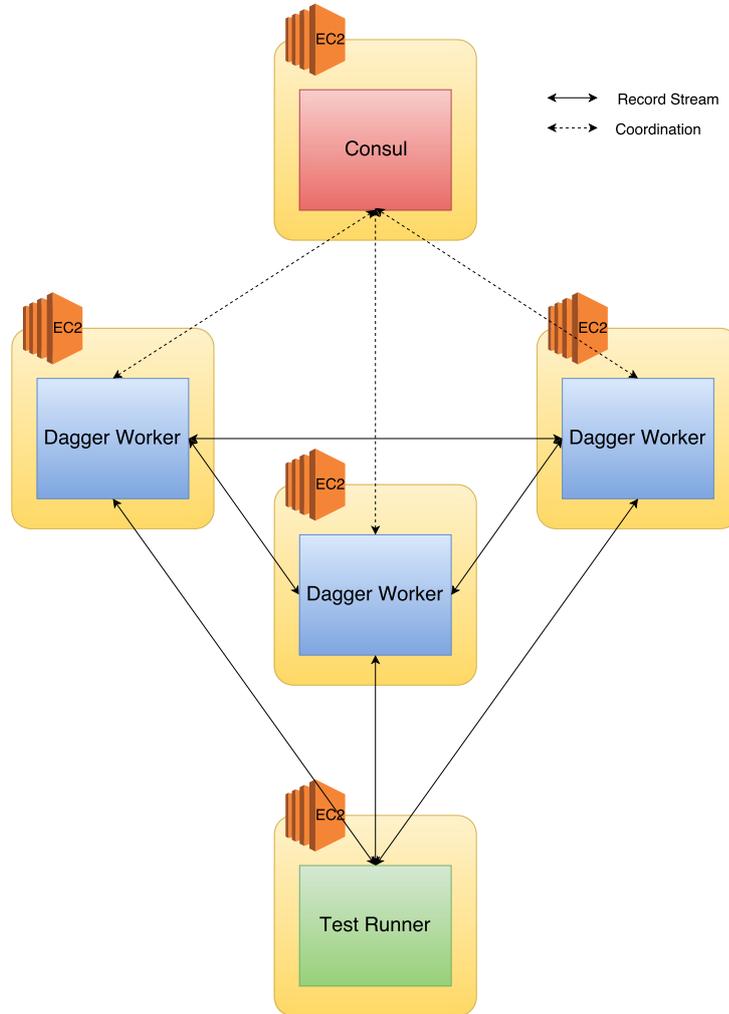


Figure 5.1: Outline of the test deployment of Dagger in the Amazon Elastic Compute Cloud.

## 5.2 Scalability testing

First and foremost we wanted to measure the scalability properties of the Dagger stream processing system. In order to do that, we have devised a set of tests that measure throughput in different use cases and situations.

A Dagger task has been created for the purpose of testing that has an artificial processing time of  $100ms$  per record. This simulates a task that for example performs a computationally heavy update of a machine learning model for detecting anomalies in a stream.

In our tests, we will have one or many producers of a single stream, which simulates monitored machines reporting monitoring data for one metric, for example `cpu_util`.

On the other side, we will have one or many subscribers subscribing to different tasks or combinations of tasks operating on said streams. This will simulate for example an alarm or a set of alarms that are set to fire when certain properties of the metric stream are detected.

All of the test scenarios have different computation graphs, but they will all run for 60 seconds and the number of records that have been successfully processed and the number of records produced will be measured.

### 5.2.1 Scenario 1: One producer, one subscriber

This test setup is intended to provide a sanity check and a baseline for later tests. In a real world analogy, it is equivalent to a single node publishing a single metric of its monitoring data and a single alarm checking the values of the aggregated stream for crossed thresholds.

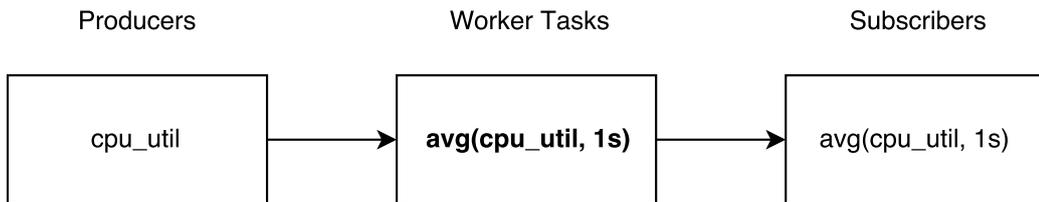


Figure 5.2: Computation graph for the test with one publisher and one subscriber.

The results of this test are displayed numerically in Table 5.1 and graphically in Figure 5.3. Since we only have one task running, the number of worker nodes has no effect on the write and read throughput. Since the task takes  $100ms$  of

processing time per record and then outputs one record each second, the results of about 10 records per second being written and 1 record per second being read matches the expectations.

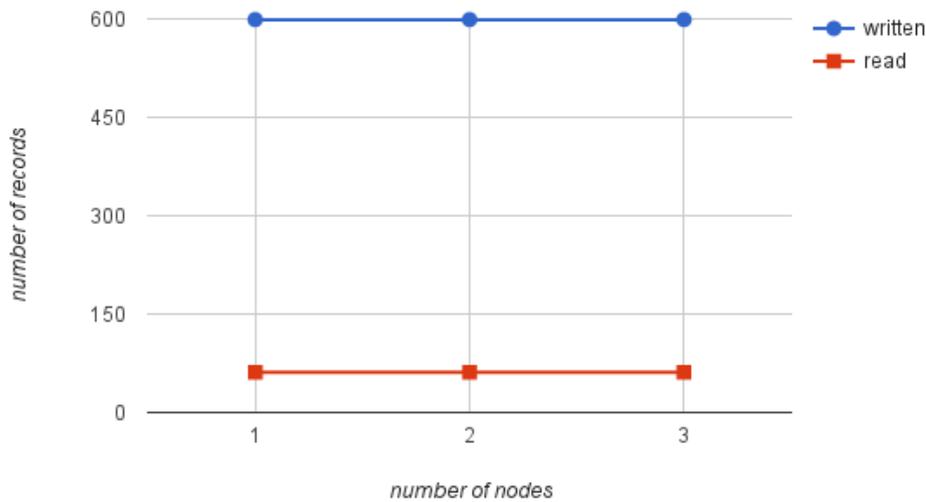


Figure 5.3: Plot of the test results in the one producer, one subscriber test scenario.

### 5.2.2 Scenario 2: Many producers, many subscribers

The next test scenario is intended to simulate a situation where we have many nodes publishing a metric and on the other side we have many alarms checking the values of the aggregated streams. In this scenario, each subscriber is subscribed to an aggregation of one single stream, so if we want all the streams to be checked, we need as many subscribers as we have producers.

The results of this test are displayed in Table 5.1 and plotted in Figure 5.5. In this test scenario, we have multiple tasks to process. Running more worker nodes means those tasks get distributed to multiple workers and the expensive processing is taking place in parallel. The throughput thus scales linearly with the number of worker nodes.

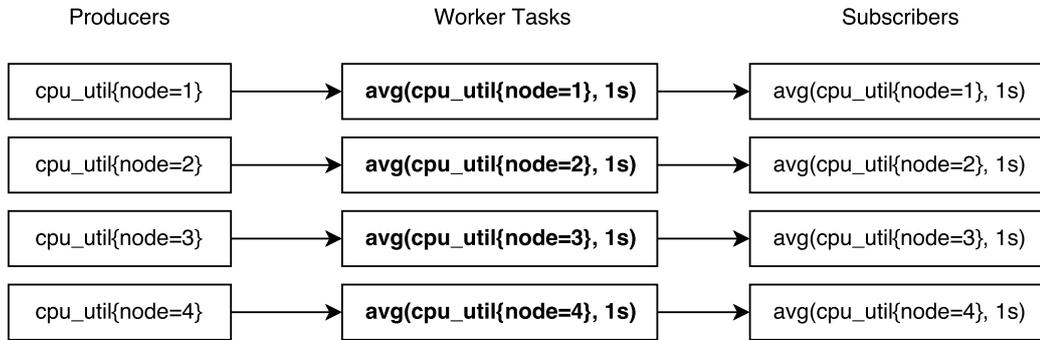


Figure 5.4: Computation graph for the test with many publishers and many subscribers.

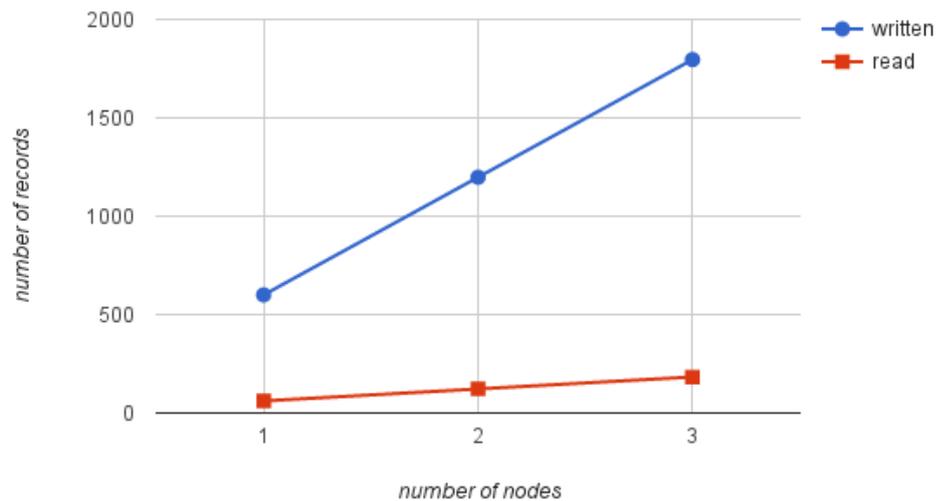


Figure 5.5: Plot of the test results in the many producers, many subscribers test scenario.

### 5.2.3 Scenario 3: Many producers, one subscriber

A situation that commonly occurs in a data-center is that we want to monitor many streams aggregated together. An example is one alarm checking for crossed thresholds in an aggregated stream of many nodes' metric streams. This leads to a task subscribing to many streams and so that task can become a bottleneck in the system.

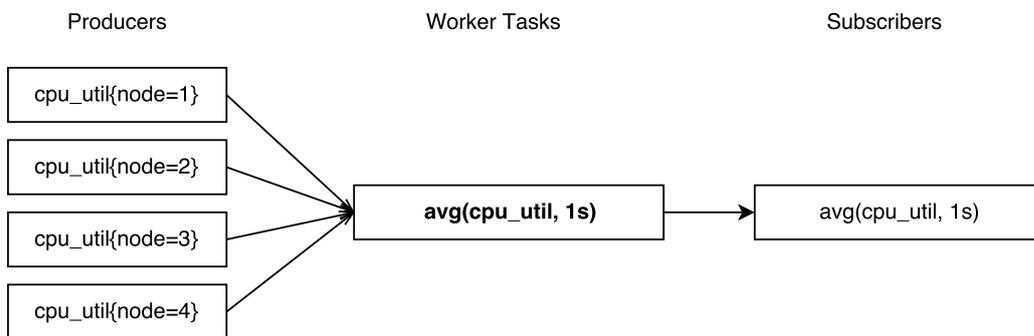


Figure 5.6: Computation graph for the test with many publishers and one subscriber.

The results of this test are displayed in Table 5.1 and plotted in Figure 5.7. In this situation, we have the same number of producers as in Scenario 2, but they are all routed to one single task. Since we only have one task that needs to process all the additional load, that task is still limited by its record processing time. Thus, the number of worker nodes has no effect on throughput.

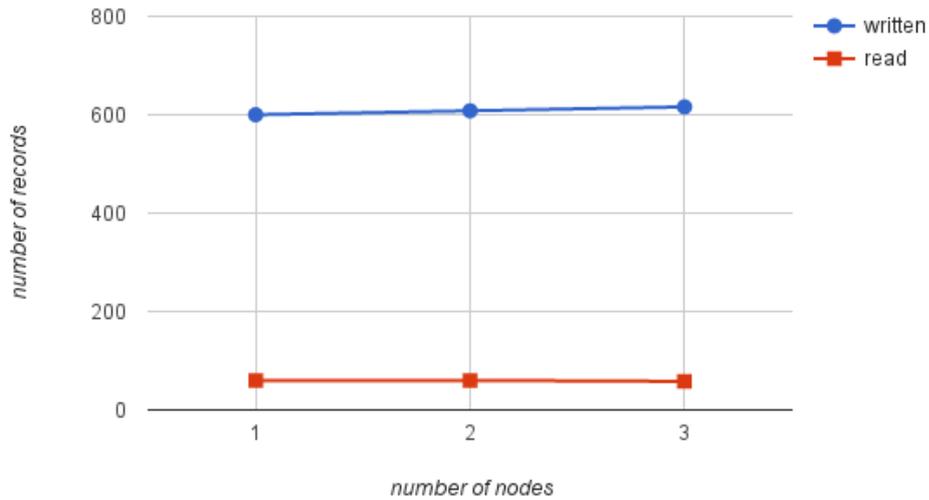


Figure 5.7: Plot of the test results in the many producers, one subscriber test scenario.

#### 5.2.4 Scenario 4: Many producers, one subscriber, with stream matching

In order to deal with a situation of multiple producers and one subscriber more efficiently, Dagger enables the use of stream matching, which allows us to compose the result in multiple stages of computations. For example, when multiple nodes are producing streams of a certain metric, the streams are for example tagged with the name of the rack that the node resides in. With the use of stream matching, we can first aggregate streams of the nodes in the same racks in the first step, and in the second step, we aggregate the streams from individual racks to get the final results.

The results of this test are displayed in Table 5.1 and plotted in Figure 5.9. In this case, we still have only one final subscriber, but in contrast to Scenario 3, that final result is calculated by multiple spawned tasks and then joined together. This results in the fact that even though we are computing one final result, the

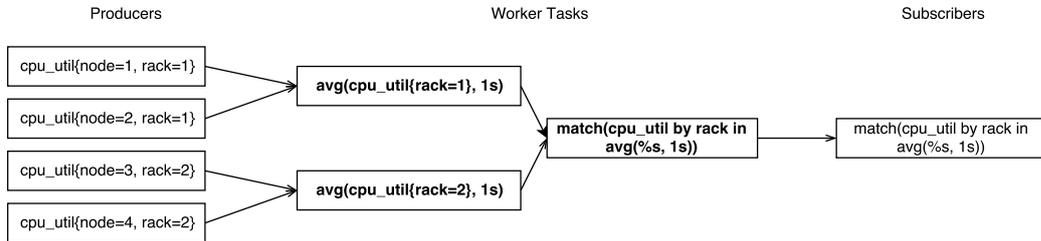


Figure 5.8: Computation graph for the test with many publishers and one subscriber, but with the use of stream matching.

throughput of the calculation scales linearly with the number of worker nodes since it is computed by multiple computations in parallel that can be distributed onto multiple nodes.

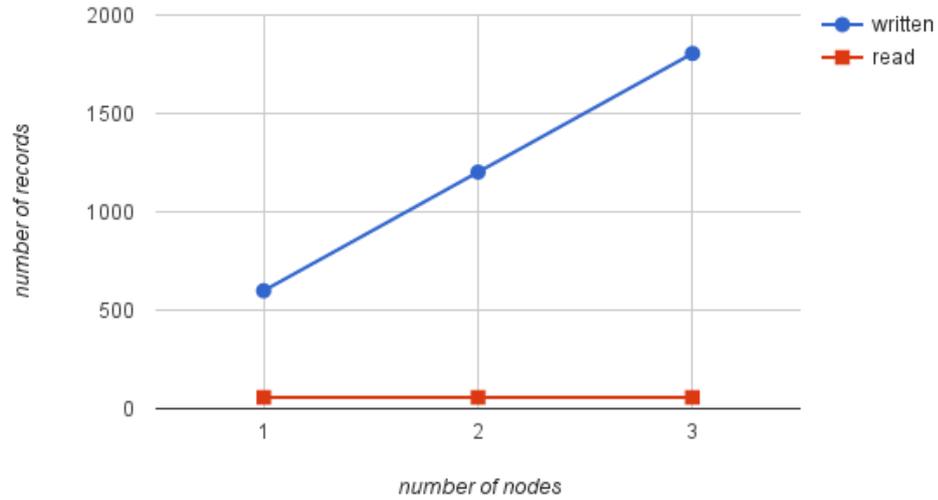


Figure 5.9: Plot of the test results in the many producers, one subscriber, with stream matching test scenario.

Table 5.1: The results of all test scenarios.

nodes	Scenario 1		Scenario 2		Scenario 3		Scenario 4	
	written	read	written	read	written	read	written	read
1	599	61	600	61	600	60	599	58
2	600	61	1198	122	608	60	1202	58
3	599	60	1796	183	616	58	1805	58

### 5.2.5 Coordination overhead

With the presence of coordination, we must make sure that the amount of coordination needed does not grow faster than the amount of records that need to be processed. In this test, we measured the overhead of coordination by comparing the number of requests made to the coordination layer to the amount of work that is being processed.

First, we tested the number of requests made to the coordination in relation to the throughput with which the task was processing records. The results are shown in Table 5.2 and show that the coordination overhead is independent of the number of records processed by a task in a given timeframe.

Secondly, we performed measurements of coordination requests with an increasing number of tasks and with increasing amount of time those tasks were processing records. The results are displayed in Figure 5.10. They show that each node requires some initial coordination requests for setting up, but the amount scales linearly with the number of tasks. The number of requests also grows linearly with time, since the coordination layer relies on long polling and even with no changes, the polling requests expire and must be performed again.

Table 5.2: The number of coordination requests with respect to processing throughput.

records per minute	coordination requests per minute
59	32
602	29
5850	30

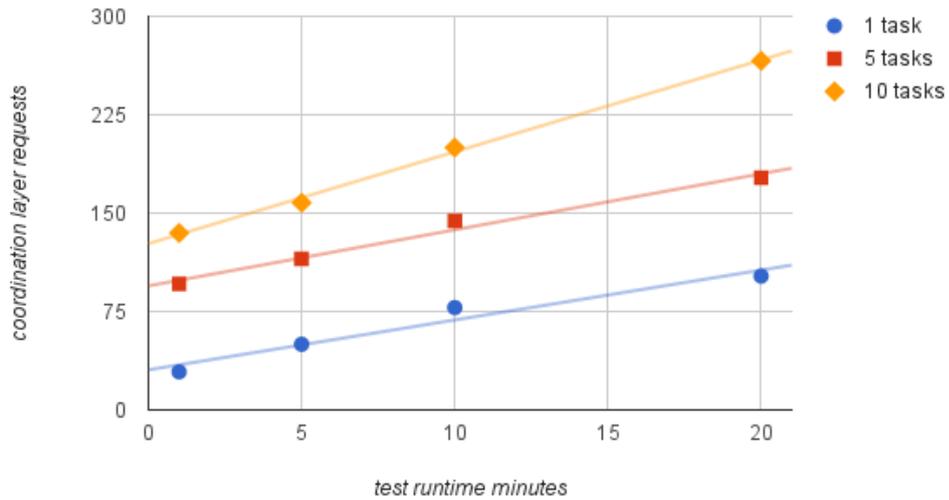


Figure 5.10: Plot of the amount of coordination requests with respect to the number of tasks and the amount of time passed.

## 5.3 Reliability

For testing reliability and failover, we used the same deployment as for the scalability testing. In this test scenario, we had one producer, one subscriber and a replication factor of 3. This means that the task that was processing the producer's records and sending the results to the subscriber was running on three nodes in parallel.

We set up the test script so the subscriber published records that contained an integer value that was incremented by one in every new record produced by the producer. We started the producer and the publisher and then randomly killed and restarted the Dagger Worker processes on the worker nodes. At the end, we examined the resulting log files and determined that all the records were received by the subscriber in order and no records were lost. In addition, the failover procedure took the amount of time that the TTL timeout was configured in the coordination layer, which is consequently the amount of time that needs to pass

before other nodes consider the failed node non-functional.

# Chapter 6

## Conclusion

Modern data-center operators are faced with the task of monitoring an increasing number of servers. Not only that - they need to be able to react to changes in their behaviour in real time. Most of the current monitoring systems still rely on centralized data processing for analytics and anomaly detection.

As a basis for a generic distributed monitoring and analytics solution, we have developed a distributed and dynamic communication and computation system. It follows the principle of only ever calculating something once by separating transformations on data into multiple steps and then reusing the results of the intermediate steps in the subsequent steps of the transformation. This means that if multiple computations make use of a partial result, for example, the average CPU usage over the last five minutes, that result will only be calculated once.

This behaviour is achieved by a two-way advertisement from both publishers and subscribers. Publishers advertise what data they offer and subscribers advertise what data they want. This fact allows the system to recognize when some data is desired by some subscribers, but no publishers exist that have that data available. In such a case, the system can dynamically run a new process that then provides the desired data to the interested subscribers, such as for example calculating an average of a data stream over time.

Aside from the focus on usability and efficiency, we also designed the system to be scalable and highly available. Scalability of the system stems from the fact that our system is inherently distributed and computation tasks can be performed on any node. Performing increasing number of computations is therefore as simple as

adding new computation nodes. Processing an increasing amount of data with a single computation, on the other hand, requires breaking down that computation into multiple steps, where parts of the data are processed in multiple parallel computations and then joined in the final computation.

In order for our system to be highly available, we introduced redundancy and failover. A single computation task can be run on multiple nodes in parallel, which allows the system to tolerate node failures. As long as one of the nodes that the computation task is running on is healthy, the subscriber will receive the computation data. This is achieved by the computation tasks forming groups and electing a leader amongst themselves. When a leader crashes, a different computation task takes over and provides the data to subscribers.

We set up a pilot deployment of our system and tested it in various use cases. The results of the testing confirmed that the design of our prototype is scalable and highly available. The prototype demonstrates the feasibility of a new monitoring and analytics paradigm that is based on dynamic on-demand real-time stream processing.

## 6.1 Future work

Currently, all of the worker nodes contend for the available tasks, regardless of how many tasks they already have. The next step would be to implement a scheduling algorithm in the workers, which would allow a more even distribution of work. It could take the worker's system resources and even the estimated complexity of the task into account.

The system's extensibility is achieved by the user providing their own computation binary. Currently, a new process is started for every computation task on a worker. If we have many small tasks running on a worker node, this means that many processes are running which can be suboptimal. The plugins could be implemented in a way that would only require one process per task type.

There are also a lot of areas in which our prototype can be improved and optimized implementation-wise. Currently, data records are serialized into JSON objects. The serialization and deserialization of JSON in Go is non-optimal when using the standard library encoder. The performance could be improved by using

an optimized library or by using a serializing strategy like Protocol Buffers [23] or Cap'n Proto [5].



# Bibliography

- [1] Giuseppe Aceto et al. “Cloud monitoring: A survey”. In: *Computer Networks* 57.9 (2013), pp. 2093–2115.
- [2] Tyler Akidau et al. “MillWheel: fault-tolerant stream processing at internet scale”. In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1033–1044.
- [3] *Apache ZooKeeper - Home*. URL: <https://zookeeper.apache.org/> (visited on 01/10/2016).
- [4] André B. Bondi. “Characteristics of scalability and their impact on performance”. In: *Proceedings of the 2nd international workshop on Software and performance*. ACM, 2000, pp. 195–203.
- [5] *Cap’n Proto: Introduction*. URL: <https://capnproto.org/> (visited on 04/03/2016).
- [6] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. Pearson Education, 2005.
- [7] Shirlei Aparecida De Chaves, Rafael Brundo Uriarte, and Carlos Becker Westphall. “Toward an architecture for monitoring private clouds”. In: *Communications Magazine, IEEE* 49.12 (2011), pp. 130–137.
- [8] *Documentation - The Go Programming Language*. URL: <https://golang.org/doc/> (visited on 01/10/2016).
- [9] Jeff Hawkins, Subutai Ahmad, and Donna Dubinsky. “Hierarchical temporal memory including HTM cortical learning algorithms”. In: *Technical report, Numenta, Inc, Palto Alto* (2010).

- 
- [10] Mark D Hill. “What is scalability?” In: *ACM SIGARCH Computer Architecture News* 18.4 (1990), pp. 18–21.
- [11] Prasad Jogalekar and Murray Woodside. “Evaluating the scalability of distributed systems”. In: *Parallel and Distributed Systems, IEEE Transactions on* 11.6 (2000), pp. 589–603.
- [12] *JSON-RPC*. URL: <http://json-rpc.org> (visited on 01/25/2016).
- [13] Jay Kreps, Neha Narkhede, Jun Rao, et al. “Kafka: A distributed messaging system for log processing”. In: NetDB. 2011.
- [14] Mahendra Kutare et al. “Monalytics: online monitoring and analytics for managing large scale data centers”. In: *Proceedings of the 7th international conference on Autonomic computing*. ACM. 2010, pp. 141–150.
- [15] Peter Mell and Tim Grance. “The NIST definition of cloud computing”. In: *National Institute of Standards and Technology* 53.6 (2009), p. 50.
- [16] *Monasca*. URL: <https://wiki.openstack.org/wiki/Monasca> (visited on 04/03/2015).
- [17] Jesús Montes et al. “GMonE: A complete approach to cloud monitoring”. In: *Future Generation Computer Systems* 29.8 (2013), pp. 2026–2040.
- [18] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *Proc. USENIX Annual Technical Conference*. 2014, pp. 305–320.
- [19] Gerardo Pardo-Castellote. “Omg data-distribution service: Architectural overview”. In: *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*. IEEE. 2003, pp. 200–206.
- [20] Marshall Pease, Robert Shostak, and Leslie Lamport. “Reaching agreement in the presence of faults”. In: *Journal of the ACM (JACM)* 27.2 (1980), pp. 228–234.

- 
- [21] Javier Povedano-Molina et al. “DARGOS: A highly adaptable and scalable monitoring architecture for multi-tenant Clouds”. In: *Future Generation Computer Systems* 29.8 (2013), pp. 2041–2056.
  - [22] Ankit Toshniwal et al. “Storm@ twitter”. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.
  - [23] Kenton Varda. “Protocol buffers: Google’s data interchange format”. In: *Google Open Source Blog*, Available at least as early as Jul (2008).



# Appendix A

## HTTP API

### A.1 Register session

Register a new publishing session. Returns Session ID.

**Endpoint** POST /register

**Response** 200 OK, Session ID

### A.2 Renew session

Renew a session. Needs to be called before the configured timeout runs out.

**Endpoint** PUT /renew

**Parameters** session=<Session ID>

**Response** 200 OK

### A.3 Publish a new record

Publishes a new record containing the specified data on the specified stream.

**Endpoint** POST /publish

**Parameters** session=<Session ID>, data=<data>

**Response** 201 Created

## A.4 Publish a new raw record

Publishes a new prebuilt record on the specified stream.

**Endpoint** POST /publish\_raw

**Parameters**

session=<Session ID>, s=<Stream ID>, data=<data>, timestamp=<timestamp>

**Response** 201 Created

## A.5 Subscribe to a record stream

Subscribes to a stream. The consumer then receives records in a streaming fashion via the opened HTTP connection.

**Endpoint** GET /listen

**Parameters**

s=<Stream ID>

**Response** 200 OK