Runlet: A cross-platform IoT tool for interactive job execution over heterogeneous devices with reliable message delivery

Vandré Leal Cândido

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2020

Vandré Leal Cândido

Runlet: A cross-platform IoT tool for interactive job execution over heterogeneous devices with reliable message delivery

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Flávio de Oliveira Silva

Uberlândia
2020

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

C386 2020

Cândido, Vandré Leal, 1992-
Runlet: A cross-platform IoT tool for interactive job execution over heterogeneous devices with reliable message delivery [recurso eletrônico] / Vandré Leal Cândido. - 2020.

Orientador: Flávio De Oliveira Silva.
Dissertação (Mestrado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.di.2020.716
Inclui bibliografia.
Inclui ilustrações.

1. Computação. I. Silva, Flávio De Oliveira,1970-, (Orient.). II. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:

Gizele Cristine Nunes do Couto - CRB6/2091

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Mestrado Acadêmico, 25/2020, PPGCO				
Data:	13 de agosto de 2020	Hora de início:	09:00	Hora de encerramento:	12:00
Matrícula do Discente:	11722CCP011				
Nome do Discente:	Vandré Leal Cândido				
Título do Trabalho:	Runlet: A cross-platform IoT tool for interactive job execution over heterogeneous devices with reliable message delivery				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Sistemas de Computação				
Projeto de Pesquisa de vinculação:	-				

Reuniu-se, por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Rodrigo Sanches Miani - FACOM/UFU; Augusto José Venâncio Neto - DIMAP/UFRN e Flávio de Oliveira Silva - FACOM/UFU, orientador do candidato.

Os examinadores participaram desde as seguintes localidades: Augusto José Venâncio Neto - Natal-RN; Rodrigo Sanches Miani e Flávio de Oliveira Silva - Uberlândia-MG. O discente participou da cidade de Uberlândia-MG.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Flávio de Oliveira Silva, apresentou a Comissão Examinadora e o candidato agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado.

Esta defesa faz parte dos requisitos necessários à obtenção do título de Mestre.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Referência: Processo nº 23117.045576/2020-46 SEI nº 2184663

This dissertation is dedicated to my parents and friends who encouraged me during critical phases of this study.

Acknowledgments

Firstly, I would like to express my gratitude to my advisor Prof. Dr. Flávio de Oliveira, for his guidance, knowledge, and motivation. Besides my advisor, I would like to thank my friend and teammate, Gustavo Brito Sampaio, for his work and efforts to make Runlet a reality. Last but not least, I would like to thank my family and friends for their continuous support.

"The energy of the mind is the essence of life."
(Aristotle)

Abstract

The heterogeneous and dynamic nature of the Internet of Things (IoT) creates challenges that go beyond the traditional computer-based network model. These challenges are commonly related to the fragmented and unpredictable mixture of devices with individual capabilities that may pose a barrier to achieving interoperability and managing devices in the context of IoT.

This study addresses interoperability and management challenges by introducing a tool for achieving interactive job execution over a network of heterogeneous devices. The proposed tool, named Runlet, is a cross-platform application that runs across many architectures and operating systems, such as ARM, Linux, macOS, and Windows. It uses both the protocol Advanced Message Queuing Protocol (AMQP) and the broker RabbitMQ for reliable message delivery.

The protocol AMQP is an open standard Machine-to-Machine (M2M) publish/subscribe messaging protocol optimized for high-latency and unreliable networks that enables client applications to communicate with conforming messaging middleware brokers. RabbitMQ is an open-source lightweight message broker that supports various messaging protocols and can be deployed on-premises and in the cloud.

The architecture of Runlet is discussed in detail both conceptually and computationally, including the reasoning behind architectural decisions and selected technologies. The evaluation is conducted through an experimental approach that assesses interactivity and reliability on a testbed of devices composed of single-board ARM computers and laptop devices. The experimental results show that the application offers interactivity under different scenarios and provides reliable message delivery even after node and server failures.

Keywords: Runlet. IoT. AMQP. RabbitMQ.

List of Figures

List of Tables

Acronyms list

ACL Access Control Lists

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

ARM Advanced RISC Machine

AWS Amazon Web Services

CoAP Constrained Application Protocol

CLI Command-Line Interface

DOM Document Object Model

DRBD Distributed Replicated Block Device

DTLS Datagram Transport Layer Security

E2EE End-to-End Encryption

ER Entity-Relationship

FIFO 'First In, First Out'

FP Functional Programming

FRP Functional Reactive Programming

GUI Graphical User Interface

HA High Availability

HTTP Hypertext Transfer Protocol

IoT Internet of Things

JAAS Java Authentication and Authorization Service

LIFO 'Last In, First Out'

M2M Machine-to-Machine

MQTT MQ Telemetry Transport

OOP Object Oriented Programming

ORM Object Relational Mapper

QoS Quality of Service

REST Representational State Transfer

SASL Simple Authentication and Security Layer

SQL Structured Query Language

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

UI User Interface

VDOM Virtual Document Object Model

VMs Virtual Machines

XEP XMPP Extension Protocols

XMPP Extensible Messaging and Presence Protocol

Contents

Chapter 1

Introduction

The heterogeneous and dynamic nature of the IoT creates challenges that go beyond the traditional computer-based network model. These challenges are commonly related to the unpredictable mixture of devices with individual capabilities that may pose a barrier to achieving interoperability and managing devices in the context of IoT (ELKHODR; SHAHRESTANI; CHEUNG, 2016).

The IEEE describes interoperability as "the ability of two or more systems or components to exchange information and to use the information that has been exchanged" (GERACI et al., 1991). Achieving interoperability is indispensable for devices across different networks and a difficult task given the IoT's competitive nature and rapidly evolving wireless technologies. This commonly results in integration issues where heterogeneous devices cannot communicate with one another (ELKHODR; SHAHRESTANI; CHEUNG, 2016).

In addition to that, IoT solutions need to handle unique management scenarios that go beyond the traditional capabilities of remote control, monitoring, and maintenance of devices. For example, a system that supports remote monitoring of tasks across a fleet of heterogeneous devices may highly benefit from a tool that offers the ability to monitor and troubleshooting errors in real-time, reducing maintenance costs and accelerating the response time of maintenance tasks.

This study fills the gap in the state of the art by presenting a tool that achieves interactive job execution over a network of heterogeneous devices with reliable message delivery. The tool, named Runlet, grants the user the capability of interacting with the shell from any connected device during job execution, which is particularly useful in a scenario that requires user input for successful script execution. A few potential use areas include Continuous Integration (CI), Continuous Delivery (CD), and remote monitoring and management of devices.

Runlet is a cross-platform application that runs across many architectures and operating systems. It uses both the protocol AMQP and the broker RabbitMQ for reliable message delivery. The AMQP protocol is an open standard M2M publish/subscribe messaging protocol optimized for high-latency and unreliable systems that enable client applications to communicate with conforming messaging middleware brokers (AMQP, 2020). AMQP uses queues for delayed delivery of messages, allowing messages to be queued for offline nodes and delivered once they are online. This feature is crucial in a dynamic scenario such as an IoT network in which nodes join and leave the network frequently due to network instability or power outages.

The application is evaluated using an experimental approach that assesses interactivity and reliability on a testbed of devices composed of single-board ARM computers and laptop devices. The results show that interactivity is accomplished under a variety of scenarios and devices with different architectures. It also shows that message delivery is reliable after both node and server failures.

1.1 Motivation

The primary motivation of this study is to introduce a new cross-platform solution for interactive job execution over a network of heterogeneous devices that is optimized for unreliable networks. The idea came up after the author noticed a lack of solutions providing means of interacting with jobs on remotely connected ARM devices like the Raspberry Pi. The proposed application addresses not only that but also gives the ability to performing such actions from different platforms and operating systems.

1.2 Objectives and Research Challenges

This study has the primary objective of introducing a cross-platform tool for interactive job execution across a network of distributed heterogeneous devices with reliable message delivery. A few secondary objectives are also set:

- ❏ Present the reasoning behind the set of open-source components selected for dealing with interactive execution on a network of distributed devices.
- ❏ Describe Runlet's architecture and the motivation behind architectural choices.
- ❏ Show the application in action using an experimental approach that evaluates interactivity and reliability.

1.3 Hipothesys

Interactive job execution can be accomplished on a network of distributed heterogeneous devices with reliable message delivery through a cross-platform application built on top of open-source components and libraries.

1.4 Contributions

This study contributes with a tool for interactive job execution that leverages the power of the messaging protocol AMQP and the broker RabbitMQ for reliable message delivery on heterogeneous devices.

1.5 Research Method

The literature review provides a comparative analysis of communication protocols and message brokers that are commonly used in the context of IoT The review compares communication protocols AMQP, CoAP, MQTT, and XMPP, and message brokers Apache Qpid, Apache ActiveMQ, and RabbitMQ. The selection is arbitrary based on web search results and open-source community engagement. The search engines used are ACM Digital Library [1], CiteSeerX [2], IEEE Xplore Digital Library [3], Google Scholar [4] and ResearchGate [5].

The architectural choices are presented in detail, considering the characteristics of the selected protocol and broker. It also considers the set of open-source components and libraries that were arbitrarily selected for developing the tool based on cutting-edge features and personal preference. The application is composed of a daemon service written in Golang and a cross-platform desktop manager written in JavaScript using Electron. The desktop manager provides the ability to create, manage, and trigger jobs, view timestamped logs created by each job execution, and a summary of recently triggered jobs. The daemon is connected to the application and is responsible for creating and managing service workers that fulfill requests.

An experimental approach is used to evaluate Runlet in regards to interactivity and reliability. The experiments involve the assessment of interactivity under different scenarios of jobs executions, the addition of nodes to the RabbitMQ cluster, and the observation of the broker's behavior after node and server failures. The testbed of devices includes single-board ARM devices and laptop devices.

1.6 Expected Results

A cross-platform tool for interactive job execution over a network of heterogeneous devices that delivers messages in a reliable manner and recovers from failures without losing data.

[1] https://dl.acm.org
[2] https://citeseerx.ist.psu.edu
[3] https://eeexplore.ieee.org
[4] https://scholar.google.com
[5] https://researchgate.net

1.7 General Scheme of Dissertation

This study has five (5) chapters. Chapter two (2) presents a literature review that compares communication protocols and message brokers to contextualize the contribution of this study. This chapter also discusses related work from an academic and commercial standpoint. Chapter three (3) outlines Runlet's architecture and the reasoning behind architectural decisions and technology choices. Chapter four (4) describes the experimental evaluation that assesses interactivity and reliability. Chapter five (5) elucidates the study conclusions, including main contributions, technical contributions, and future work.

Chapter 2

Background and Literature Review

This chapter presents a background review of message-oriented protocols and message brokers that are commonly used in the context of IoT. The review is essential to understand and contextualize the reasoning behind the architectural decisions taken during the development phase. It also describes the related work academically and commercially.

2.1 Communication Protocols

This section introduces popular message-oriented protocols, including AMQP, CoAP, MQTT, and XMPP. These protocols define rules, syntax, and semantics for message exchanging. A message is the unit of data sent between clients and servers containing a set of key-value pairs called properties. These protocols were selected arbitrarily based on web search results and open-source community engagement.

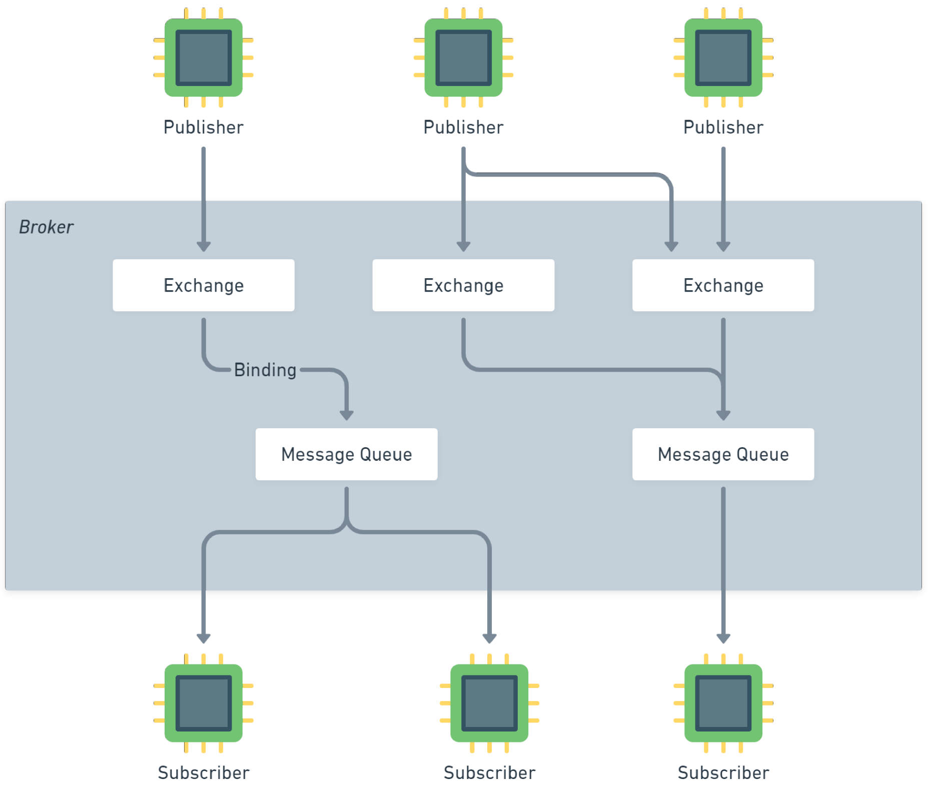
2.1.1 AMQP

The AMQP provides a platform-agnostic method for data transport between applications and is used in various areas such as financial trading, transportation, smart grid, and online gaming. It uses TCP for transport, and Transport Layer Security (TLS) / Secure Sockets Layer (SSL) for security. The following scenarios are mentioned as potential use cases: real-time feed of constantly updating data, delivery when the destination comes online, and interoperability with all popular operating systems (AMQP, 2020).

Brokers receive messages from publishers, also known as producers, and route them to consumers. The messages are self-contained, have no limit in size, does not enforce a particular data format, and support a variety of delivery options such as *point-to-point*, *store-and-forward* and *publish-and-subscribe*. It also has acknowledgments to ensure that all messages are transmitted even after network disruptions (LUZURIAGA et al., 2015).

- ❏ **Exchange**: accepts messages from producers and routes them to message queues according to pre-defined criteria stored on binding tables. A binding is a relationship

Figure 1 – AMQP 0-9-1 Architecture



Source: Elaborated by the author

between an exchange and a message queue. There are many exchange types, however *direct* and *topic* are arguably the most important. The *direct* type routes on a routing key. The *topic* types routes on a specific routing pattern.

- **Message Queue**: stores messages in memory or on disk and forwards them to consumers in sequence. Each queue is independent and has properties such as private or shared, durable or temporary, client-named or server-named.

- **Routing Key**: virtual address that may be used by an exchange to decide how to route a message.

Exchanges are declared with the following properties: name, type, durability, auto-delete, and arguments. The type defines the routing algorithm used to route messages to queues. Durability defines whether exchanges are durable and survive to broker restarts or have to be redeclared. The auto-delete determines whether the exchange is deleted when the last queue is released from it. Arguments are optional and are used by plugins to offer broker-specific features.

Message queues are 'First In, First Out' (FIFO) buffers that hold their messages in memory, on disk, or a combination of these and distribute them between one or more consumers. They may be durable, temporary, or auto-deleted. Durable queues need to be explicitly deleted. Temporary queues are deleted when the server shuts down. Auto-deleted queues are deleted when they are no longer used. It is important to note that the only way to ensure FIFO behavior is to have only a single consumer connected to a queue. Otherwise, they may not behave like FIFO and are called "weak-FIFO".

The version 1.0 is an approved ISO/IEC international standard protocol developed by the OASIS open standards consortium (ISO, 2020), and differs radically from versions 0-9-1 / 0-9 / 0-8. Version 1.0 does not attempt to define a broker and, therefore, does not describe exchanges, binding, and routing keys.

2.1.2 CoAP

The Constrained Application Protocol (CoAP) is a M2M protocol designed for constrained nodes and networks in the IoT that features reliable delivery, simple congestion control, and flow control. The protocol is based on the Representational State Transfer (REST) model and is an Internet Standards Document (RFC 7252). It uses minimal device resources and works on micro-controllers with as low as 10 KiB of RAM and 100 KiB of code space (COAP, 2020).

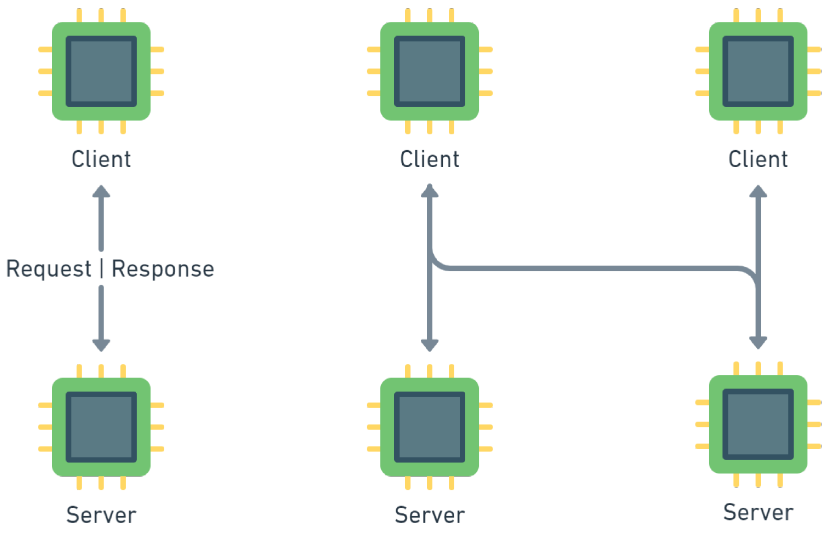
It was designed for a smaller footprint using User Datagram Protocol (UDP) rather than Transmission Control Protocol (TCP), and provides security by using Datagram Transport Layer Security (DTLS) over UDP. One drawback of using DTLS to secure UDP is that DTLS does not support multicast, which is a significant disadvantage of CoAP when compared to other messaging protocols. In addition to that, DTLS handshakes require additional packets that increase network overhead and may shorten the lifespan of devices that run on batteries. (KARAGIANNIS et al., 2015)

CoAP uses the request/response pattern, and Hypertext Transfer Protocol (HTTP) methods GET, POST, PUT, and DELETE to provide resource-oriented interactions in a client-server architecture. There's a proposal for enabling publish/subscribe communication through a broker, which is currently in draft (KOSTER; KERäNEN; JIMENEZ, 2019). This broker would facilitate many-to-many communication, including *store-and-forward* messaging between two or mode nodes, and would also avoid the need for direct reachability between clients, making it suitable for event-oriented scenarios in IoT.

Messages are exchanged asynchronously and may arrive out of order, appear duplicated, or go missing as the protocol is bound to UDP. For this reason, CoAP has its reliability mechanism featuring stop-and-wait retransmission with exponential back-off for confirmable messages and duplicate detection for both confirmable and non-confirmable messages. (SHELBY et al., 2014). There are four (4) types of messages that are used to ensure reliability:

- **Reset**: indicates that a confirmable or non-confirmable message was received, but

Figure 2 – COAP Architecture



Source: Elaborated by the author

context is missing to process it.

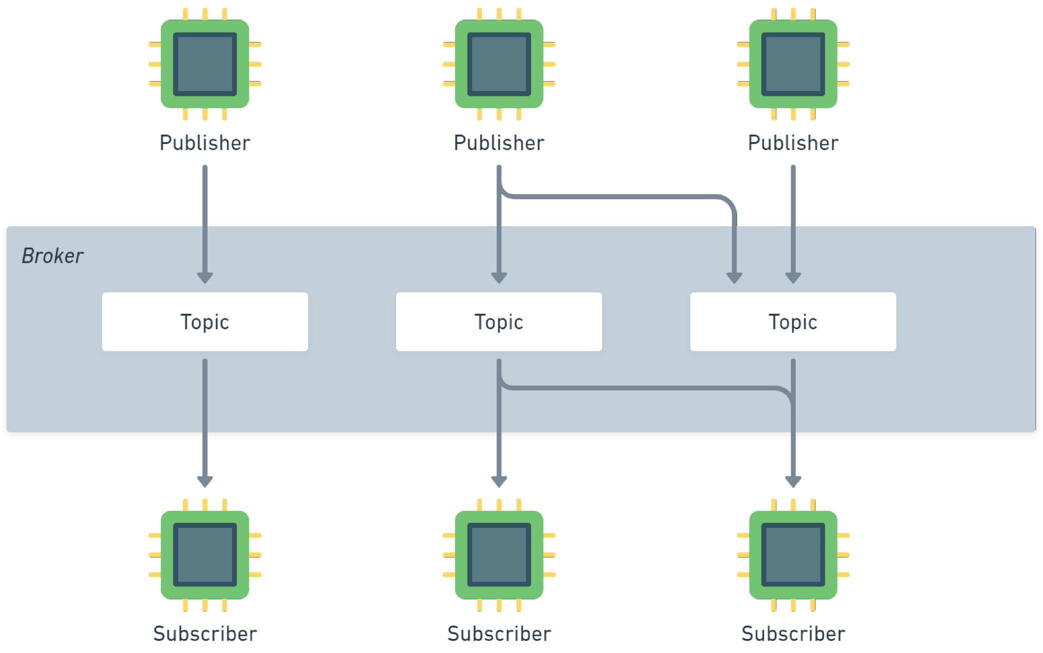
- ❏ **Acknowledgment**: acknowledges that a specific confirmable message was received, but does not indicate if any request encapsulated in the message was successful.
- ❏ **Confirmable**: messages that require an acknowledgment.
- ❏ **Non-confirmable**: messages that do not require an acknowledgment.

A proposal for using CoAP over reliable transports such as TCP/TLS was introduced on RFC 8323 due to the existence of networks that do not forward UDP packets because of geographic connectivity challenges and rate-limited traffic (BORMANN et al., 2018). However, the use over UDP is still the most recommended for IoT environments since more reliable transports lead to larger packet sizes, more round trips, and increased RAM requirements.

2.1.3 MQTT

The MQ Telemetry Transport (MQTT) is also a M2M lightweight protocol for the Internet of Things designed for constrained devices and unreliable networks. The protocol uses the publish/subscribe method pattern based on the principle of publishing messages and subscribing to topics. Each client is a publisher that sends information to topics or/and a subscriber that receives messages from a subscribed topic. Client subscriptions may be either to an explicit topic or multiple topics using wildcards (MQTT, 2020).

Figure 3 – MQTT Architecture



Source: Elaborated by the author

The broker is the middleware agent that holds topics and receives messages from publishers to deliver to subscribers. Topics are treated as a hierarchy, allowing arrangements similar to file systems using a slash (/) separator. Topics are always dynamic and created silently by the broker if they do not exist previously, following one of MQTT's design principles of requiring as little administrative setup as possible. The broker discards a topic with no subscribers unless the publisher indicates that the topic is retained. This feature allows that new subscribers receive an instant update with the most current value of a topic rather than waiting for a new update to occur.

MQTT uses TCP for transport and TLS/SSL for security. There is also a simplified version that runs over UDP; however, it is not on the scope of this study as reliable delivery is not yet a feature of this version (ZAVALISHIN, 2020). The protocol defines three levels of Quality of Service (QoS) for message delivery (OASIS, 2020). Higher levels are more reliable but involve higher latency and higher network bandwidth due to the increased overhead associated with delivery confirmation.

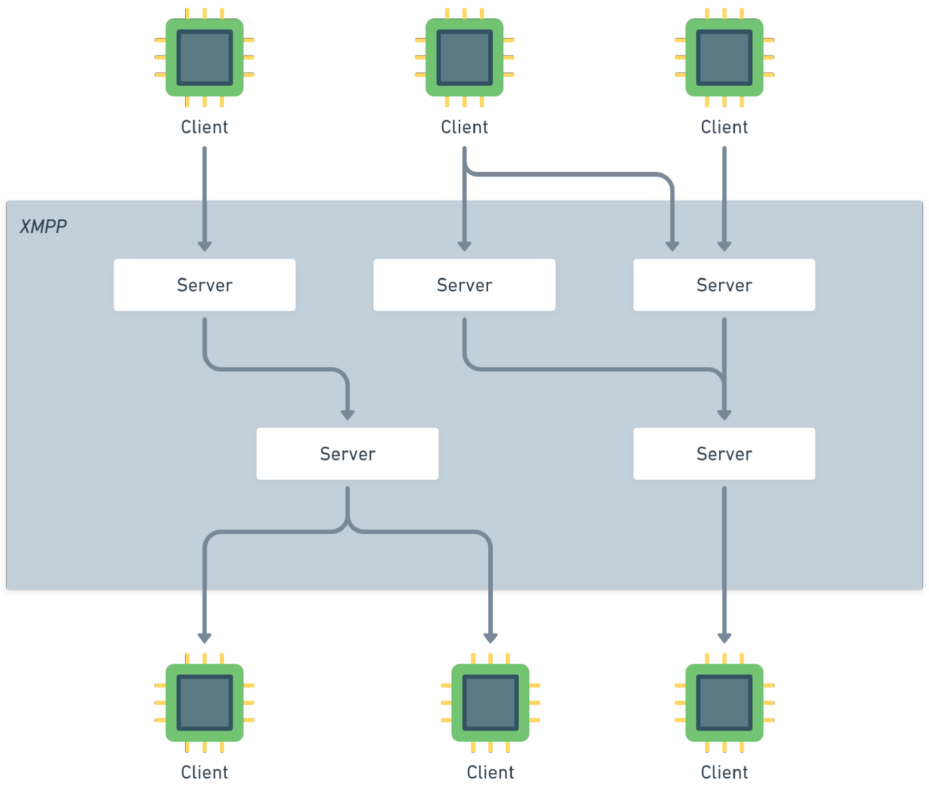
- **at most once**: messages are delivered to the best efforts, but losses may occur. It is suitable for cases in which the loss of an individual message is not so important.
- **at least once**: messages are assured to arrive, but duplicates may occur.
- **exactly once**: messages are assured to arrive only once. It is useful in applications where message loss or duplication may lead to unwanted effects.

2.1.4 XMPP

The Extensible Messaging and Presence Protocol (XMPP), previously known as Jabber, is a protocol that supports low latency message exchange by design since it was conceived as a decentralized alternative to popular consumer-oriented instant messaging (IM) services at the time of its conception, such as ICQ, AIM, and MSN (XMPP, 2020). The protocol is extensible, as the name suggests, and allows the specification of XMPP Extension Protocols (XEP) that extend its functionality.

Similarly to AMQP and MQTT, XMPP uses TCP for transport, TLS/SSL for security, and supports the publish/subscribe pattern. However, it does not provide any QoS options and depends exclusively on TCP mechanisms for reliability, which makes the protocol impractical for M2M applications. The XML tags also demand additional parsing overhead that increases power consumption (KARAGIANNIS et al., 2015).

Figure 4 – XMPP Architecture



Source: Elaborated by the author

XMPP applications are usually deployed in decentralized client-server architectures and communicate either via *client-to-server* stream or *server-to-server* stream. Clients are not able to communicate directly since a server is required to open the communication channel between them. Figure 4 shows an example that follows the *server-to-server* stream to share resources between clients.

A server manages connections with clients and establishes the XML stream that exchanges resources with other entities in the network. A client connects to a server to have access to the network and authenticates via Simple Authentication and Security Layer (SASL). Server to server connection is possible after allowing inter-domain communication (NĂSTASE; SANDU; POPESCU, 2017).

Comparison

Table 1 compares protocols according to the following criteria: (1) transport, (2) security, (3) brokered architecture, (4) support to the publish/subscribe communication pattern, (5) QoS levels, (6) flexible routing, and (7) header size in bytes. The analysis provided by this comparison table is important to validate the reason why the protocol AMQP is selected.

CoAP is UDP-based and has lower overhead when compared to other TCP-based protocols for this reason. However, the lack of TCPs retransmission mechanisms may cause more packet loss. In addition to that, support to publish/subscribe is not yet supported by CoAP, which makes the protocol not suitable for event-oriented scenarios. MQTT has lower overhead than AMQP but does not offer flexible routing. XMPP has the disadvantage of not offering QoS support when compared to others.

Table 1 – Comparison of Communication Protocols

	AMQP	CoAP	MQTT	XMPP
Transport	TCP/UDP	TCP/UDP	TCP/UDP	TCP
Security	TLS/SSL	DTLS	TLS/SSL	TLS/SSL
Broker	X	-	X	-
Publish/Subscribe	X	-	X	X
Service Levels (QoS)	X	X	X	-
Flexible Routing	X	-	-	-
Header Size (bytes)	8	2	4	0

Source: Elaborated by the author

2.2 Message Brokers

This section presents an overview of popular message brokers that support the protocol AMQP, including Apache Qpid, Apache ActiveMQ, and RabbitMQ. A message broker is an intermediate entity that offers messaging capabilities via standard or custom protocols (MAGNONI, 2015). We also define other important concepts before we delve into the features offered by each message broker:

- Clusters: a group of messaging servers that spread the load of sending and consuming messages across many nodes, allowing a system to be scaled horizontally by adding new nodes to the cluster.
- High Availability (HA): stands for the system's ability to remain operational after the failure of one or more nodes. The degree of high availability support varies between messaging systems.
- Backup and Recovery: procedures to create and store copies of data to ensure data integrity and consistency, avoid data loss during service and maintenance and easily set up new nodes from previous backups.
- Federation: allows transmission of messages between brokers without requiring clustering, which is useful for loose coupling of nodes and clusters under different administrative domains.
- Persistence: commonly used by message brokers to store messages and queues on memory, disk, or custom database services.
- Logging: the process of collecting and storing data from events that are commonly used to audit behavior over a specific period.
- Access Control: regulates user access to view and use resources from messaging systems. Users may have their access granted to a particular set of features or system actions.
- Web Management Console: a management interface accessed over the web to view and control aspects of the broker, such as clusters, nodes, connections, queues, messages, and users.
- Metric Tools: tools that offer metrics to monitor server performance and resource consumption, usually as multi-dimensional time-series data that is used to generate ad-hoc graphs, tables, and alerts.

2.2.1 Apache Qpid

Apache Qpid is an open-source project by the Apache Software Foundation that builds messaging tools based on the protocol AMQP (QPID, 2020). The project has two types of components, *messaging APIs* for the development of AMQP applications and *messaging servers* for the deployment of AMQP networks.

Messaging APIs:

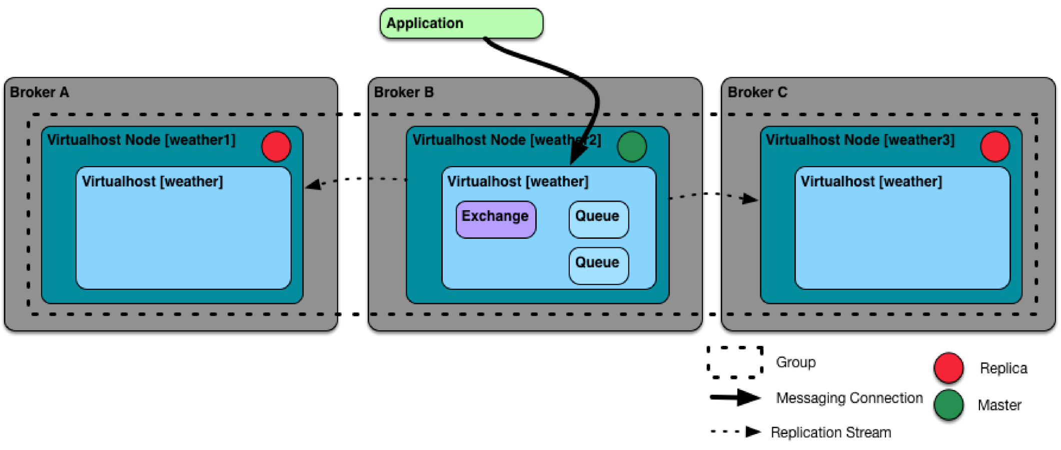
- *Qpid Proton*: toolkit library to allow any application to use AMQP.
- *Qpid JMS*: JMS client built using Qpid Proton.
- *Qpid Messaging API*: connection-oriented messaging API.

Messaging servers:

- *Broker-J*: JAVA message broker that stores, routes, and forwards messages.
- *C++ Broker*: C++ message broker that stores, routes, and forwards messages.
- *Dispatch router*: message router built on Qpid Proton.

Only the brokers, Broker-J and C++ Broker, are further analyzed in this study for comparison with Apache ActiveMQ and RabbitMQ. Qpid brokers are full-featured middlewares that offer clustering, high availability, backup and recovery, persistence, logging, access control, among other features. A fundamental difference among them is that Broker-J uses an entity called *virtualhost*. A *virtualhost* is an independent container that performs messaging inside a container called *virtualhost node*. Each *virtualhost node* has a single *virtualhost*.

Figure 5 – Broker-J group of three nodes deployed across three brokers



Source: (QPID, 2020)

Clustering is offered on both brokers with an active/passive mode for high availability. In this mode, many brokers work together to form a highly available group of two or mode nodes capable of handling failures. A single broker stays active serving the clients, while others are backups in the event of a failure. All operations in the *active* node are automatically replicated to *backup* nodes, so a *backup* node may take over and become the *active* node, if the *active* node fails. Broker-J supports backup and recovery by manually copying files after stopping the broker and nodes.

The C++ Broker supports broker federation by creating message routes that deliver messages to exchanges on the destination broker. Federation are configured using a *pull route* in which the destination broker subscribes to the source queue on the source broker, or a *push route* in which the source broker contacts the destination broker to send messages. Broker-J documentation does not mention support to broker federation.

Both brokers claim to provide pluggable persistence in addition to the regular message persistence required for their functioning. However, no additional details were found by the author at the time of writing. In regards to logging, both brokers support a variety of loggers, including a FileLogger capable of writing a log file to the file system, a ConsoleLogger capable of writing to `stdout` or `stderr`, and a SyslogLogger capable of writing to a remote syslog daemon. The following severity logs are found on both: ERROR, WARN, INFO, DEBUG, TRACE. However, only Broker-J has a web management console for managing the broker. The web interface also allows to (1) add, remove, and monitor queues and virtual hosts; (2) inspect, move, copy or delete messages; (3) configure high availability, and (4) change logging severity.

Access control is achieved via authentication and authorization. Authentication to verify the identity of a user using SASL and a rule-based authorization mechanism using Access Control Providers with Access Control Lists (ACL) rules. An ACL rule has matching criteria that determine the actions each user is allowed to perform and the objects that can be accessed.

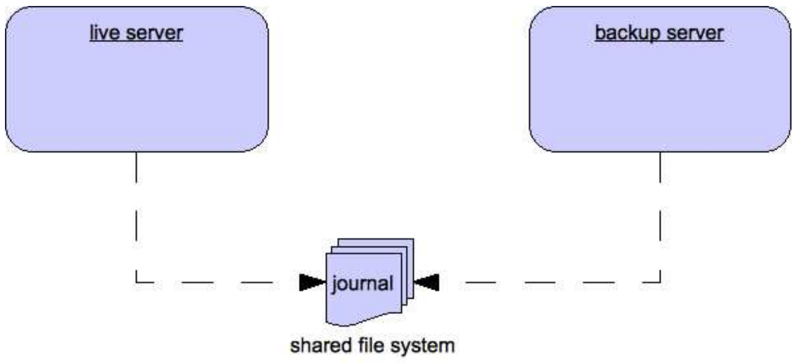
2.2.2 Apache ActiveMQ

Apache ActiveMQ is an open-source messaging server developed by the Apache Software Foundation that supports multiple message-oriented protocols, including AMQP, MQTT, STOMP, and OpenWire. It currently has two versions, the classic 5.x broker and the next generation Artemis broker that will become version 6 once it reaches a sufficient level of feature parity with 5.x (ACTIVEMQ, 2020). It supports clustering, high availability, backup and recovery, federation, persistence, logging, access control, and comes with a web management console.

The next generation (Artemis) is further analyzed for being more recent. It provides a configurable clustering model where messages may be load balanced between the nodes in the cluster, and automatically redistributed between nodes to prevent resource starvation on any particular node. High availability is accomplished by linking servers together as *live - backup* groups (active/passive) where each live server has one or more backup servers that only become operational when a failure occurs. In this scenario, the backup node stays in passive mode, ready to take over any time. The previous live server has the priority to become the next live server when the current live server experiences a failure. Automatically stopping the current live server once the previous live server comes back up is also possible.

There are two supported policies for backup, *shared store* and *replication*. The *shared store* policy shares an entire data directory between live and backup servers using a shared file system. The data is loaded from the persistent storage in the shared file system when a failure occurs, and a backup server takes over, which means that no replication happens between live and backup nodes. The *replication* policy duplicates all the data received by the live server on backup nodes. A backup server needs to synchronize all the data from the live server before becoming operational, which is a downside depending on the amount of data to be synchronized and connection speed.

Figure 6 – ActiveMQ Shared Store for High Availability



Source: (ACTIVEMQ, 2020)

There are two types of federation methods available, *address federation* and *queue federation*. *Address federation* behaves like a multicast between connected brokers, meaning that every message sent to an address on `Broker-A` will be delivered to every queue on that broker, but also to all attached queues on `Broker-B`. A *queue federation* provides a method of balancing the load of a queue across remote brokers by linking queues together and making them act as a single logical queue. It is suitable for cases such as increasing queue capacity and migrating between two clusters.

The persistence may be done using *file journal* or *JDBC Store*. *File Journal* saves data on disk and is highly optimized for the messaging use case. *JDBC Store* uses JDBC to connect to a database vendor such as PostgreSQL, MySQL or Apache Derby. There are many loggers available that log either to the console or file or both of them. The severity level in which messages are logged are FATAL, ERROR, WARN, INFO, DEBUG, or TRACE.

Access control is achieved via authentication and authorization. Authentication is performed by the `ActiveMQJAASSecurityManager` manager that supports any standard Java Authentication and Authorization Service (JAAS) login module. However, a legacy and deprecated `ActiveMQSecurityManager` that reads user credentials from properties file is also supported. While ActiveMQ 5.x only has three (3) permission types (read, write and admin), Artemis has nine (9) permission types for authorizing access to queue addresses. Each type grants permissions to a specific list of roles.

Artemis comes with a management console powered by (HAT, 2020). The console supports the visualization of dashboards, threads, connections, sessions, consumers, producers, addresses, and queues; and also allows the creation of new addresses and queues. Metrics are exported to a variety of monitoring systems via the Micrometer facade.

2.2.3 RabbitMQ

RabbitMQ is a popular open-source message broker developed by Pivotal Software. It is based on the Erlang language, runs on all major operating systems, and was originally conceived to support AMQP 0-9-1. However, the broker also supports AMQP 1.0, MQTT, and STOMP via plugins (RABBITMQ, 2020). Similarly to ActiveMQ Artemis, it supports clustering, high availability, backup and recovery, federation, persistence, logging, and access control.

A cluster replicates all the data and state required for the operation of a broker across all nodes, except message queues. Queues are visible and reachable from all cluster nodes but reside on a single node by default. Replicating queues is possible with *mirrored queues* for high availability. Each mirrored queue has its own *master* node and one or more *mirrors*. All queue operations happen on the *master* node first and then are propagated to *mirrors*. If the node that hosts the *master* fails, the oldest *mirror* is promoted to master as long as it synchronizes. Although this feature enhances availability, it does not balance load across nodes since all participating nodes do all the work.

High Availability is improved using Pacemaker and DRBD for an active/passive setup. The Pacemaker is an open-source resource manager for clusters that provides corrects response to any failure or cluster state (CLUSTERLABS, 2020). Distributed Replicated Block Device (DRBD) is a distributed replicated storage system that provides the shared storage in which the active node will write messages. Durable queues and persistent messages from a node may be recovered once a failure occurs in this setup. This technique combined with clustering is used to scale beyond a single node and preserve persistent messages in the event of node failure.

Every node has a data directory that stores all the data from that node, including definitions and message store data. Definitions are schema, metadata, or topology data that can be exported and imported via the HTTP API and CLI tools. Messages can also be backed up. However, a node must be first stopped before accessing its message store.

The entire cluster must be stopped to avoid losing messages or having duplicates in a cluster with *mirrored queues*.

Federation allows both exchanges and queues to be federated via a plugin. A federated exchange or queue is capable of receiving messages from one or more remote exchanges and queues on other brokers, called *upstreams*. A federated exchange routes messages published upstream to a local queue, while a federated queue lets a local consumer receive messages from an upstream queue.

Persistent messages are persisted to disk as soon as they reach the queue, while transient messages are persisted to disk only when the memory is under pressure. However, it is important to note that persistent messages may also be kept in memory as the persistence layer attempts to persist as much data to disk as possible. RabbitMQ uses the Lager logging library to support different sources, such as file, console, and syslog. Severity levels of log messages include CRITICAL, ERROR, WARNING, INFO, DEBUG, and NONE.

Access control is achieved using authentication and authorization. Authentication to identify who the user is, and authorization to determine what the user is allowed to do. Authentication happens via credentials (username/password pairs) or X.509 certificates. It is performed after an application connects to RabbitMQ and before it is able to perform any operations. The server checks user permissions after authentication and authorizes access to resources such as virtual hosts, queues, and exchanges. Permissions are distinguished between *configure*, *write* and *read* operations.

RabbitMQ comes with a management plugin that contains an HTTP API for management and monitoring of nodes and clusters, a web management console, and a command-line tool `rabbitmqadmin`. The management tool is very convenient for development and includes the following features: (RABBITMQ, 2020)

- ❑ Create, list, and delete exchanges, queues, bindings, users, and permissions.
- ❑ Monitor message rates per queue, exchange, channel, or globally.
- ❑ Monitor node resources, including memory usage breakdown, available disk page, and bandwidth usage on inter-node communication links.
- ❑ Import and export broker definitions, including users, permissions, queues, exchanges, binding, parameters, and policies. This feature is used for setup automation of new nodes and clusters, as well as backup and restore.

A certain amount of overhead is introduced by the plugin since it is bound to the system being monitored. For that reason, it is recommended to use long-term metric storage and visualization services for production systems, such as Prometheus and Grafana. These tools have built-in support and offer more powerful and customizable UI, decoupling of the monitoring system, lower overhead, and collection of node-specific metrics that are more resilient to node failures.

Starting with version 3.8, RabbitMQ offers a unique feature flag subsystem for upgradability. Feature flags are a mechanism that controls the features that are enabled or available on all cluster nodes, allowing RabbitMQ nodes to determine if two versions are compatible. If so, then two nodes with different versions can live in the same cluster, and upgrades happen without shutting down the entire cluster. All supported features flags are enabled by default when a node is started for the first time.

Comparison

Table 2 compares Qpid Broker-J, Qpid C++ broker, ActiveMQ Artemis, and RabbitMQ, according to the following criteria: (1) AMQP version support, (2) programming language, (3) clustering, (4) high availability, (5) backup and recovery, (6) persistence, (7) logging, (8) access control, (9) federation, (10) web management console, and (11) built-in support for metric tools.

They are all similar in terms of features. The most notable differences are the lack of federation support on Qpid Broker-J, the lack of a web management console on Qpid C++ Broker, and the lack of metric tools on both Qpid brokers. It's also important to note that there are slight differences between how a feature is supported across brokers. For instance, even though all brokers support logging, the severity levels differ.

Table 2 – Comparison of Message Brokers

	Qpid Broker-J	Qpid C++ Broker	ActiveMQ Artemis	RabbitMQ
AMQP Version Support	1.0 0-10 0-9-1 0-9 0-8	1.0 0-10	1.0	1.0 (plugin) 0-9-1
Language	Java	C++	Java	Erlang
Clustering	X	X	X	X
High Availability	X	X	X	X
Backup and Recovery	X	X	X	X
Persistence	X	X	X	X
Logging	X	X	X	X
Access Control	X	X	X	X
Federation	-	X	X	X
Web Management Console	X	-	X	X
Metric Tools Support	-	-	X	X

Source: Elaborated by the author

2.3 Related Work

There are several studies that evaluate messaging protocols and distributed message brokers. However, to the best of the author's knowledge, there is no other academic study pursuing the goal of performing interactive job execution across a fleet of network devices with reliable message delivery.

From a commercial standpoint, many popular cloud solutions offer similar capabilities such as Amazon Web Services (AWS) IoT from Amazon, Azure IoT Hub from Microsoft, Google IoT Core from Google, and ThingsBoard. Table 3 compares these solutions in terms of features. The comparison is from a quantitative standpoint and was made considering the features based on official documentation and some briefs tests.

These solutions offer a similar list of features such as authentication, device management, publish/subscribe model for communication, device assignment, job scheduling, activity logs, and management UI. AWS IoT Device Management supports HTTP, MQTT, and WebSockets (AMAZON, 2020). Azure IoT Hub supports HTTP, AMQP, AMQP over WebSockets, MQTT, and MQTT over Websockets (AZURE, 2020a). Google IoT Core supports HTTP and MQTT (GOOGLE, 2020). ThingsBoard supports HTTP, CoAP, and MQTT (THINGSBOARD, 2020).

Table 3 – IoT Cloud Solutions

	AWS IoT Core	Azure IoT Hub	Google IoT Core	ThingsBoard
Protocols	HTTP MQTT WebSockets	HTTP AMQP MQTT AMQP over WebSockets MQTT over WebSockets	HTTP MQTT	HTTP CoAP MQTT
CLI Tool	X	X	X	X
Authentication	X	X	X	X
Device Management	X	X	X	X
Publish/Subscribe	X	X	X	X
Device Assignment	X	X	X	X
Job Scheduling	X	X	X	X
Management UI	X	X	X	X
Activity Logs	X	X	X	X
Interactive Job Execution	-	-	-	-

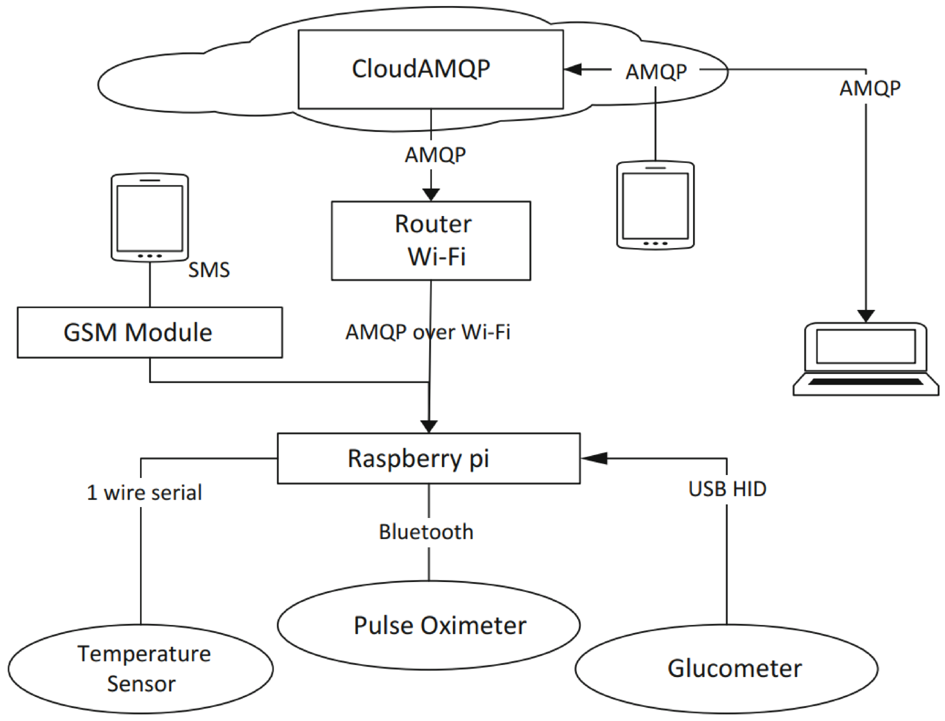
Source: Elaborated by the author

A lack of an interactive terminal to view and interact with running jobs is observed on all compared solutions despite the common offering of an interactive shell for remote monitoring and management. However, it's important to point out that Runlet was conceived as a proof of concept and does not intend to compete commercially with the aforementioned cloud solutions.

A selection of academic studies are presented next, ranging from monitoring systems to surveys that evaluate messaging protocols and message brokers. However, no mention is made of interactive job execution in none of the studies.

(KRISHNA; SASIKALA, 2019) introduces a healthcare monitoring system that monitors patients' vitals in real-time and sends the information to a web server for persistence and decision-making. Sensors are connected to a Raspberry Pi that acts as a gateway and sends the data to a server using AMQP through the board's Wi-Fi, as demonstrated in figure 7.

Figure 7 – Healthcare Monitoring System Design



Source: (KRISHNA; SASIKALA, 2019)

(LIANG; CHEN, 2018) introduces the design of a real-time data acquisition and monitoring system for medical data on smart campuses. The design includes both AMQP and RabbitMQ for data exchanging and the HL7 protocol to facilitate information management via standard-compliant messages. The HL7 standard applies to the exchange of medical records before different medical systems, which is also the main difference between this study and (KRISHNA; SASIKALA, 2019) given that both introduce a similar concept for healthcare monitoring.

(KOSTROMINA; SIEMENS; YURII, 2018) presents a concept for a resilient meteorological monitoring system using AMQP. The system design considers the need for stable and resilient delivery of data without losses, even in case of network outages and a predictive monitoring system that monitors memory usage on single-board computers to prevent system crashes caused by lack of available memory. RabbitMQ is the selected message broker since it implements the protocol AMQP and has features like federation, clustering, and persistence.

Table 4 compares previously mentioned systems in terms of protocols, reliable message delivery, prototype, and interactive execution. Interactive job execution is not applicable in none of the studies, which is further motivation for conducting this research.

Table 4 – Academic Systems

Study	AMQP	RabbitMQ	Reliable Message Delivery	Prototype	Interactive Job Execution
(LIANG; CHEN, 2018)	X	X	X	-	-
(KOSTROMINA; SIEMENS; YURII, 2018)	X	X	X	-	-
(KRISHNA; SASIKALA, 2019)	X	X	X	X	-

(HAPP et al., 2017) discusses features that a message-oriented middleware has to meet the requirements of the IoT domain. It evaluates which features are supported by some well-known open/pub solutions, including AMQP, MQTT, XMPP, and ZeroMQ. A quantitative evaluation is performed in terms of throughput and latency measurements. XMPP performs considerably worse than others in both aspects. AMQP performs better than XMPP, but message throughput and average delays lack behind ZeroMQ and MQTT for large numbers of small messages. However, it is important to note that MQTT doesn't allow messaging clients directly, and ZeroMQ is no full-featured broker like AMQP.

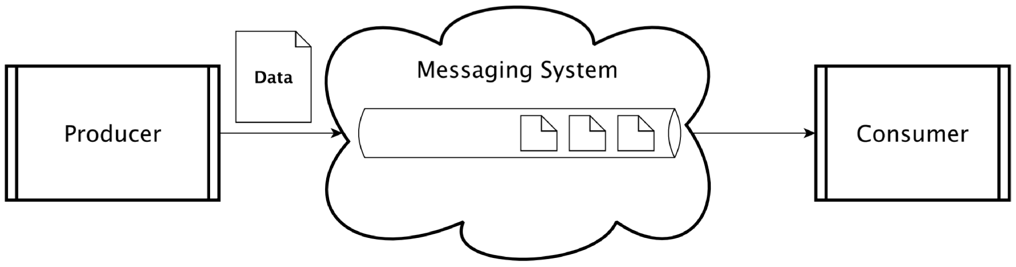
(KARAGIANNIS et al., 2015) and (DHAS; JEYANTHI, 2019) survey the pros and cons of a diverse list of application layer protocols, including AMQP, CoAP, MQTT, and XMPP. The comparison underlies aspects such as transport, architecture, security, reliability, and quality of service. (NAIK, 2017) also evaluates the aforementioned protocols based on empirical evidence from the literature but does not consider dynamic network conditions and re-transmission of packets incurred by overheads, which may affect comparison results. The computational capabilities of devices also influence the selection of a protocol.

(MAGNONI, 2015) does a similar work overviewing main concepts, technologies, and services related to messaging solutions. The overview includes a walk-through of communication protocols AMQP, MQTT and STOMP; message brokers ActiveMQ, Apache Kafka, RabbitMQ, and ZeroMQ; and use cases where messaging-based communication has been successfully adopted. The walk-through highlights features and differences between selected protocols and message brokers.

(LUZURIAGA et al., 2015) evaluates AMQP and MQTT over unstable and mobile networks. The authors define unstable networks as networks in which links are frequently modified or broken without control, like mobile and wireless networks in urban environments. The experimental results demonstrated that both protocols are robust and have similar jitter behavior. AMQP delivery followed a 'Last In, First Out' (LIFO) order during message bursts, which caused message consumption in reverse order.

(DOBBELAERE; ESMAILI, 2017) presents a comparison framework to position Apache Kafka and RabbitMQ quantitatively and qualitatively. The quantitative comparison includes aspects such as time decoupling, routing logic, delivery guarantees, ordering guarantees, availability, transactions, multicast, and dynamic scaling. The quantitative com-

Figure 8 – Messaging for loosely coupled communication



Source: (MAGNONI, 2015)

parison contrasts latency and throughput based on results obtained from an experimental setup. The study also highlights features that are unique to each system and best-suited scenarios. It's concluded that the choice does not have to be an exclusive one. A combination of both might be the best option since both have their best-suited cases and are capable of achieving low latency results over multiple nodes.

(ROSTANSKI; GROCHLA; SEMAN, 2014) evaluates the performance of RabbitMQ under different clustering scenarios, considering requirements of scalability, availability, and fault tolerance. The results indicate that while single queues on clustered nodes are crucial to performance, performance may still be improved by spreading queues between nodes if fault tolerance is a requirement.

Chapter 3

Runlet

This chapter presents the architecture and reasoning behind architectural decisions and technology choices. It also details each one of the components, including the cloud storage, daemon, database, Graphical User Interface (GUI), and server.

3.1 Introduction

Runlet is a cross-platform application that features a daemon and a desktop manager. The desktop manager includes both the daemon and a full-featured GUI that provides an easy to use interface for managing jobs across a fleet of connected devices. The Advanced RISC Machine (ARM) distribution supports ARMv6, ARMv7, and ARM64.

3.1.1 The Queuing Protocol

CoAP and XMPP protocols are discarded due to some of their characteristics described in the previous chapter. XMPP does not provide any QoS options (DHAS; JEYANTHI, 2019), which makes the protocol not suitable for Runlet, ensuring reliable message delivery is a crucial aspect of the application. CoAP seems like a great option from a resource usage and network bandwidth standpoint. However, it is discarded due to its lack of support for the publish/subscribe model (KARAGIANNIS et al., 2015). The publish/subscribe model is a must-have for reaching optimal performance as job logs are updated continuously.

MQTT is an excellent option and very similar to AMQP in many aspects. They both have brokered architectures, support the publish/subscribe model, use TCP for transport and TLS/SSL for security, offer QoS levels for reliability, with MQTT having the advantage of smaller header size (DHAS; JEYANTHI, 2019). However, extra features provided by AMQP made the trade-off advantageous despite the increased overhead and latency. AMQP has the advantage of having flexible message routing, reliable message queues with fine-grained control over bindings, and multiple types of exchanges (AMQP, 2020).

3.1.2 The Message Broker

RabbitMQ is the selected broker for the following reasons

- ❏ Web management UI for short-term metric collection and monitoring offers many administrative features convenient for development.
- ❏ Built-in support for using long-term metric visualization tools such as Prometheus and Grafana in production.
- ❏ Concise documentation and tutorials.
- ❏ Feature flag subsystem for upgradability.
- ❏ Very engaged community.

ActiveMQ Artemis also comes with a web management console, supports metric tools, and has well-organized documentation and active community (ACTIVEMQ, 2020). Nonetheless, the decision ultimately came down to a personal preference for RabbitMQ's management tool and built-in support to Prometheus and Grafana in production. Qpid Broker-J and Qpid C++ Broker, on the other hand, fall short on development tools and lack support for metrics (QPID, 2020). The Qpid community is also not as active and engaged as both RabbitMQ and ActiveMQ communities.

3.2 Architecture

This section details the conceptual architecture composed of the following components: GUI, daemon, server, database, and cloud storage. Fig. 9 shows the conceptual architecture and communication flow between components.

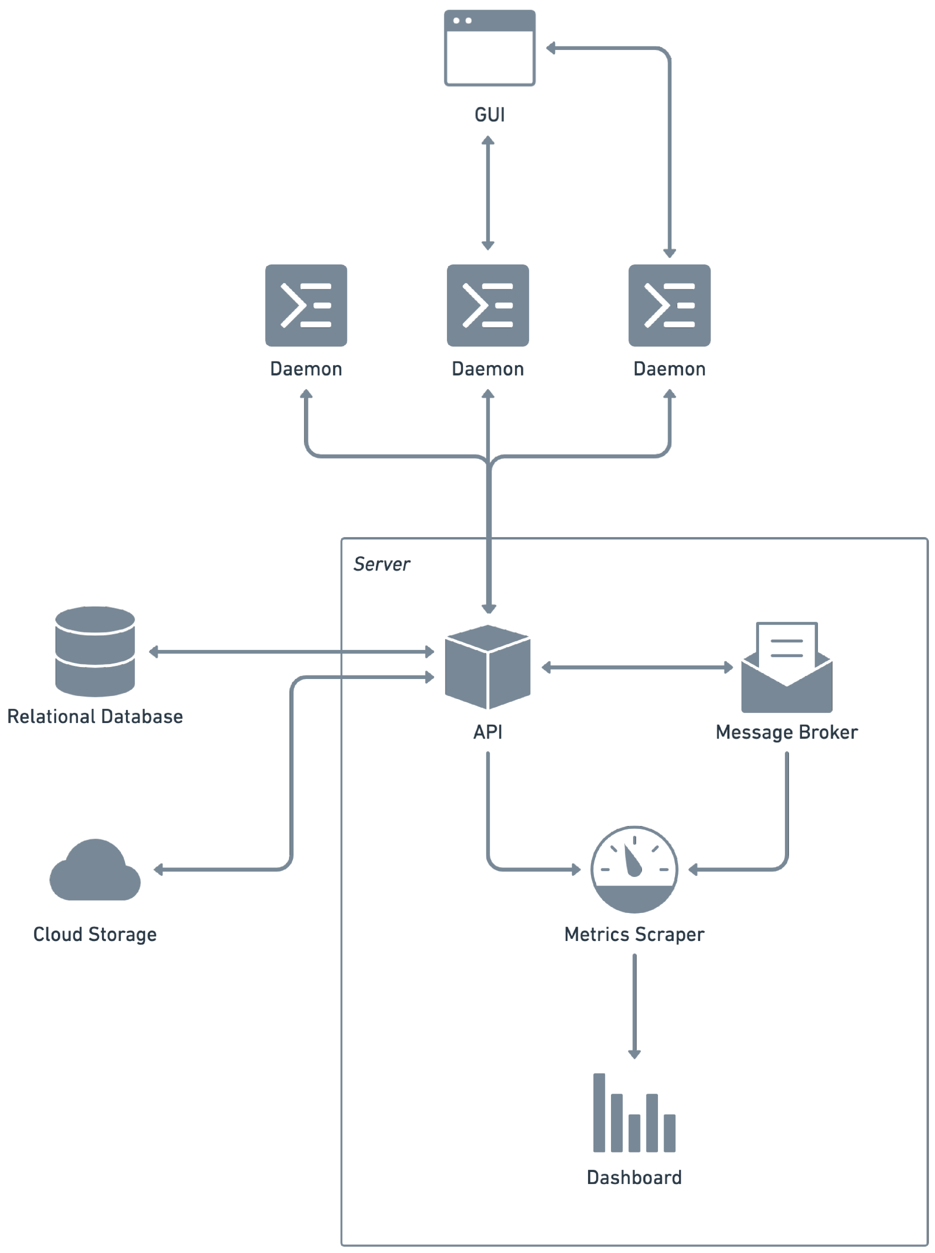
GUI

The GUI is a cross-platform application that runs on all major operating systems such as Mac, Linux, and Windows. It must provide a friendly interface for management and monitoring of devices and apply concepts of reactive programming to observe, compute, and react only to job changes.

Daemon

The daemon does job orchestration and triggers the server by job requests. It fulfills user requests via Command-Line Interface (CLI) or GUI. In addition to that, it provides an extra layer of security by encrypting logs on the local disk.

Figure 9 – Runlet Conceptual Architecture



Source: Elaborated by the author

Database

The database is capable of storing data from workspaces, users, jobs, and relationships. Not only that, but it also provides relational operators to manipulate data, be highly scalable in terms of data it can store, and support a great number of concurrent users.

Cloud Storage

A cloud-based solution is used exclusively for log storing. This eliminates ownership costs associated with managing a data storage infrastructure and ensures data durability, availability, and security. The object storage server offers the following features:

- ❏ **Erasure Coding**: data protection mechanism that breaks data into sectors and store encoded redundant data pieces across different storage media.
- ❏ **Bitrot Protection**: the server never reads corrupted data and is capable of healing corrupted objects on the fly, ensuring data integrity.
- ❏ **Encryption**: support to multiple encryption schemes to protect data at rest.
- ❏ **Identity Management**: ability to identify and control user access.

Server

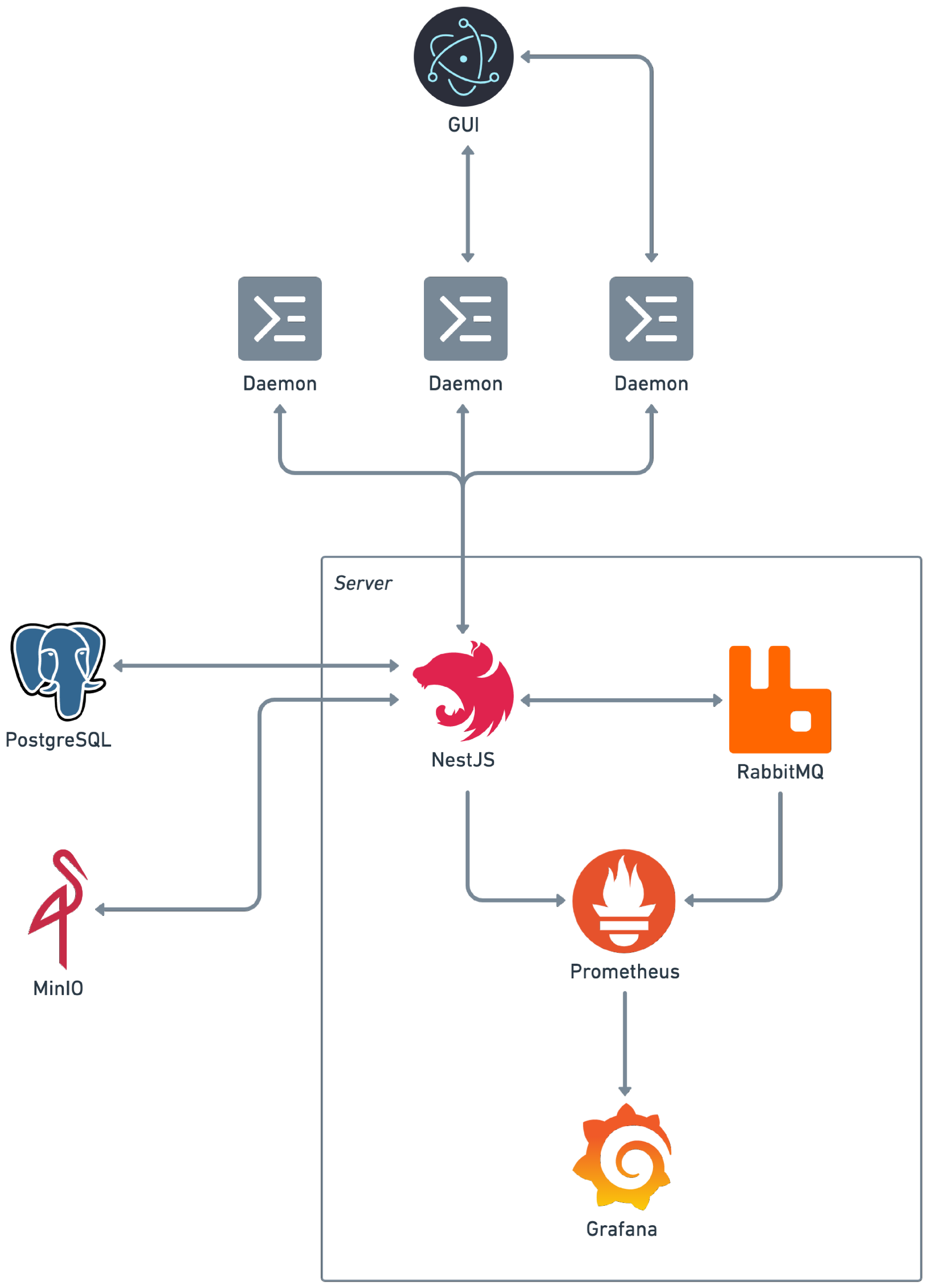
The server is the central point of the architecture. It orchestrates the communication between daemons and message broker through an API, persists data in the relational database, and store files in the cloud. A step-by-step example of how the desktop manager handles a request is described next.

1. The desktop manager transmits a job execution request to the daemon.
2. The daemon sends the request to the corresponding API endpoint.
3. The controller method tied to the endpoint receives the request and enqueues the message on the message broker.
4. The device is then notified that a new execution is requested and starts executing the job at the earliest convenience. All the other connected devices start receiving the output data.

3.3 Implementation

This section details how components are implemented, including programming languages, frameworks, libraries, and hosting services. It also describes the structure of a Runlet job and the reactive model followed by the desktop manager to reduce the number of computations and CPU workload. Figure 10 presents the conceptual architecture with entities replaced by selected technologies.

Figure 10 – Runlet Architecture Overview



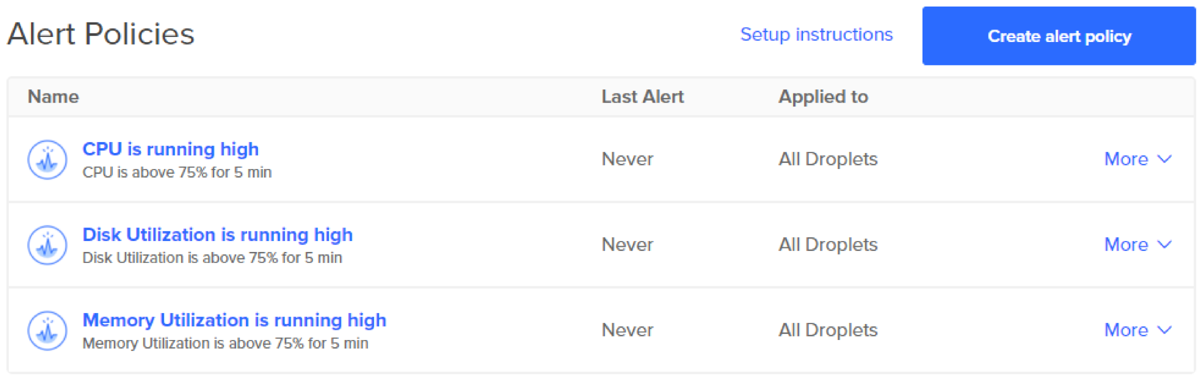
Source: Elaborated by the author

3.3.1 Server

The server consists of a NestJS API, a RabbitMQ message broker, and metric tools Prometheus and Grafana for monitoring. It is currently hosted as a droplet on DigitalOcean. A droplet is a Linux-based Virtual Machines (VMs) flexible in type, size, and resource allocation (DIGITALOCEAN, 2020b). The machine is resized whenever deemed necessary and offer the following features:

- **Monitoring**: the performance metrics of a Droplet is monitored in real-time, including bandwidth traffic, CPU usage, and disk I/O. Not only that, but alert policies may also be defined to receive alerts whenever the utilization of a resource reaches a certain threshold. Figure 11 shows three examples of alert policies that are available for a droplet.
- **Backups**: automatically created Droplet images may be enabled at weekly intervals. This feature provides an easy way to revert a Droplet to a previous state or create new Droplets from an image.
- **Snapshots**: on-demand disk images that are taken before server changes.
- **API**: interface that enables management of Droplets using conventional HTTP requests, including resizing, rebooting, and enabling backups.

Figure 11 – Example of Alert Policies for a Droplet



Name	Last Alert	Applied to	
CPU is running high CPU is above 75% for 5 min	Never	All Droplets	More
Disk Utilization is running high Disk Utilization is above 75% for 5 min	Never	All Droplets	More
Memory Utilization is running high Memory Utilization is above 75% for 5 min	Never	All Droplets	More

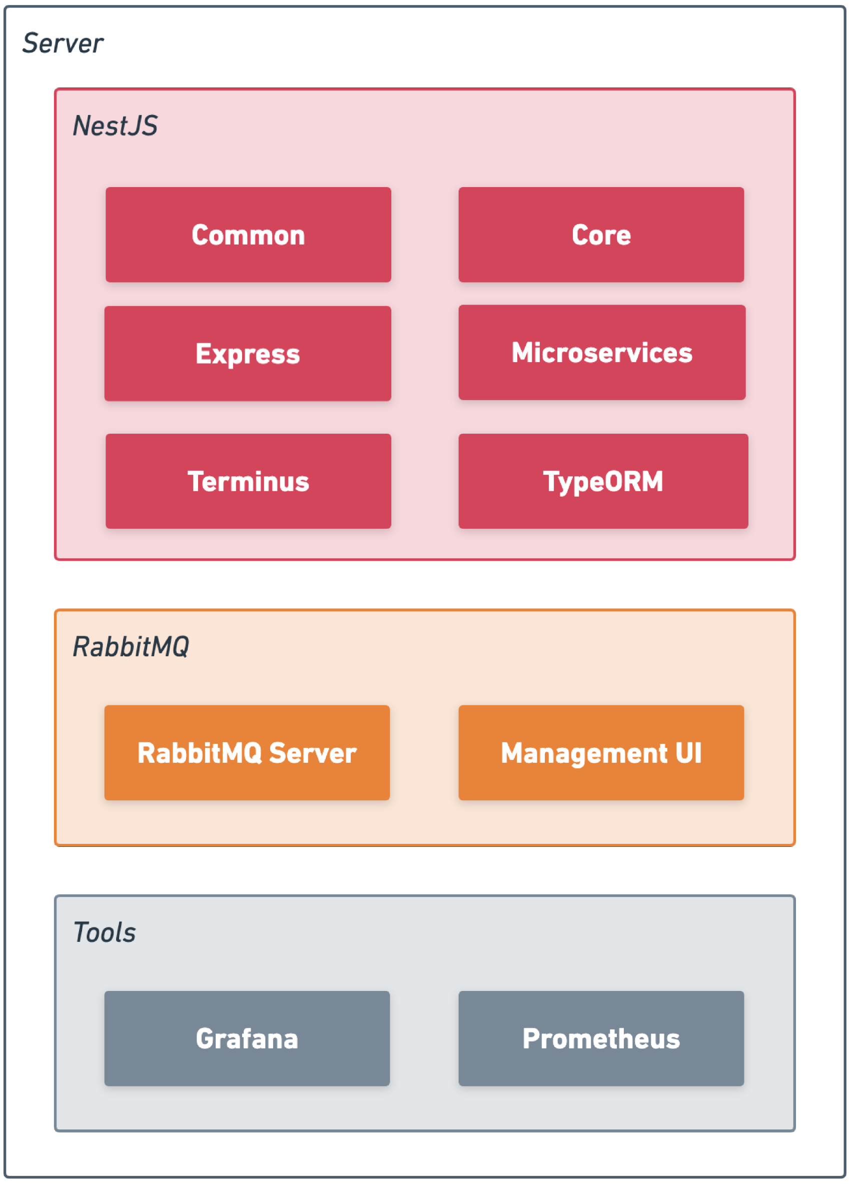
Source: (DIGITALOCEAN, 2020a)

NestJS

There are a great amount of Node.js web frameworks that fit the purpose of creating a server-side middleware capable of exposing an Application Programming Interface (API), such as NestJS, AdonisJS, FeathersJS, hapi, Koa, and LoopBack. NestJS has been selected due to its simplicity, testing utilities, Object Relational Mapper (ORM) recipes, built-in support for microservices, and community engagement.

It has a modularized architecture, uses JavaScript, supports TypeScript, and combines elements of Object Oriented Programming (OOP), Functional Programming (FP), and Functional Reactive Programming (FRP). Under the hood, it abstracts common HTTP Server frameworks like Express and Fastify but also exposes their API to developers (NESTJS, 2020). Figure 12 presents an overview of server components, including NestJS modules, RabbitMQ subcomponents, and metric tools.

Figure 12 – Server Components



Source: Elaborated by the author

- Common: package with HTTP constants, interface, module, and service.
- Core: package with core functions and classes such as hooks, injectors, interceptors, router, services, and so on.
- Express: underlying framework used for building the HTTP API.
- Microservices: built-in microservices, including a RabbitMQ transporter.
- Terminus: offers health checks to check whether services are healthy or not.
- TypeORM: ORM used to interact with the database.

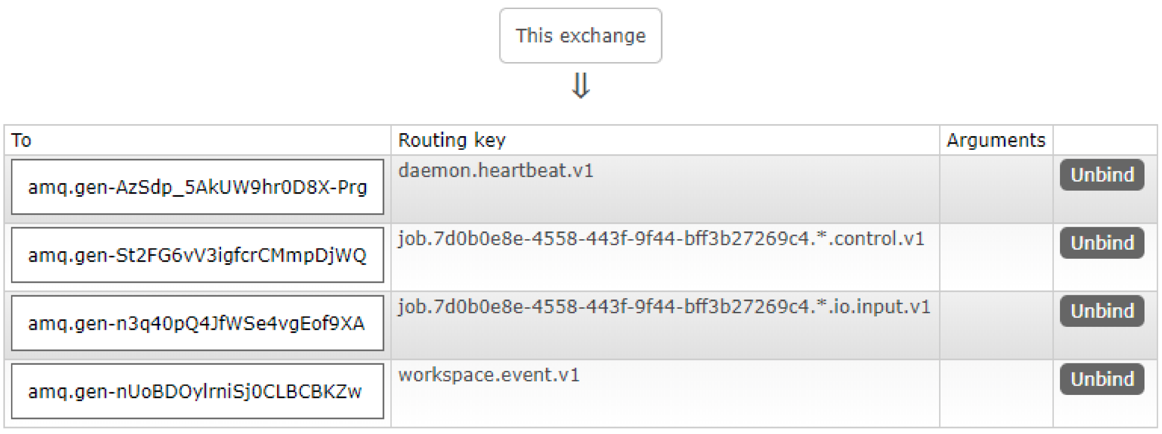
RabbitMQ

RabbitMQ is the message broker that handles message exchanging between network devices. It has two subcomponents: (1) a server operator where all broker features are configurable, such as clustering, high availability, persistence, and access control; and (2) a web-based tool for external management, monitoring, and short-term visualization of broker events.

A *topic* exchange is created for each user. This type of exchange is used for multicast routing of messages as it routes messages to one or many queues bound with a matching binding key. All exchanges are durable and survive broker restart. On the other hand, queues are temporary and short-lived to reduce workload and avoid leaving unused queues behind failed connections. They are both *exclusive*, and *auto-deleted*, meaning that they are only used by their declaring connections and are automatically deleted when the last consumer is gone or cancels its subscription.

Queues are also created following the single responsibility principle that dictates that each queue has a single concern. That means that jobs end up having multiple queues to control execution rather than having a single queue for everything. Figure 13 shows a few examples of routing keys to exclusive queues.

Figure 13 – Examples of RabbitMQ exchange bindings



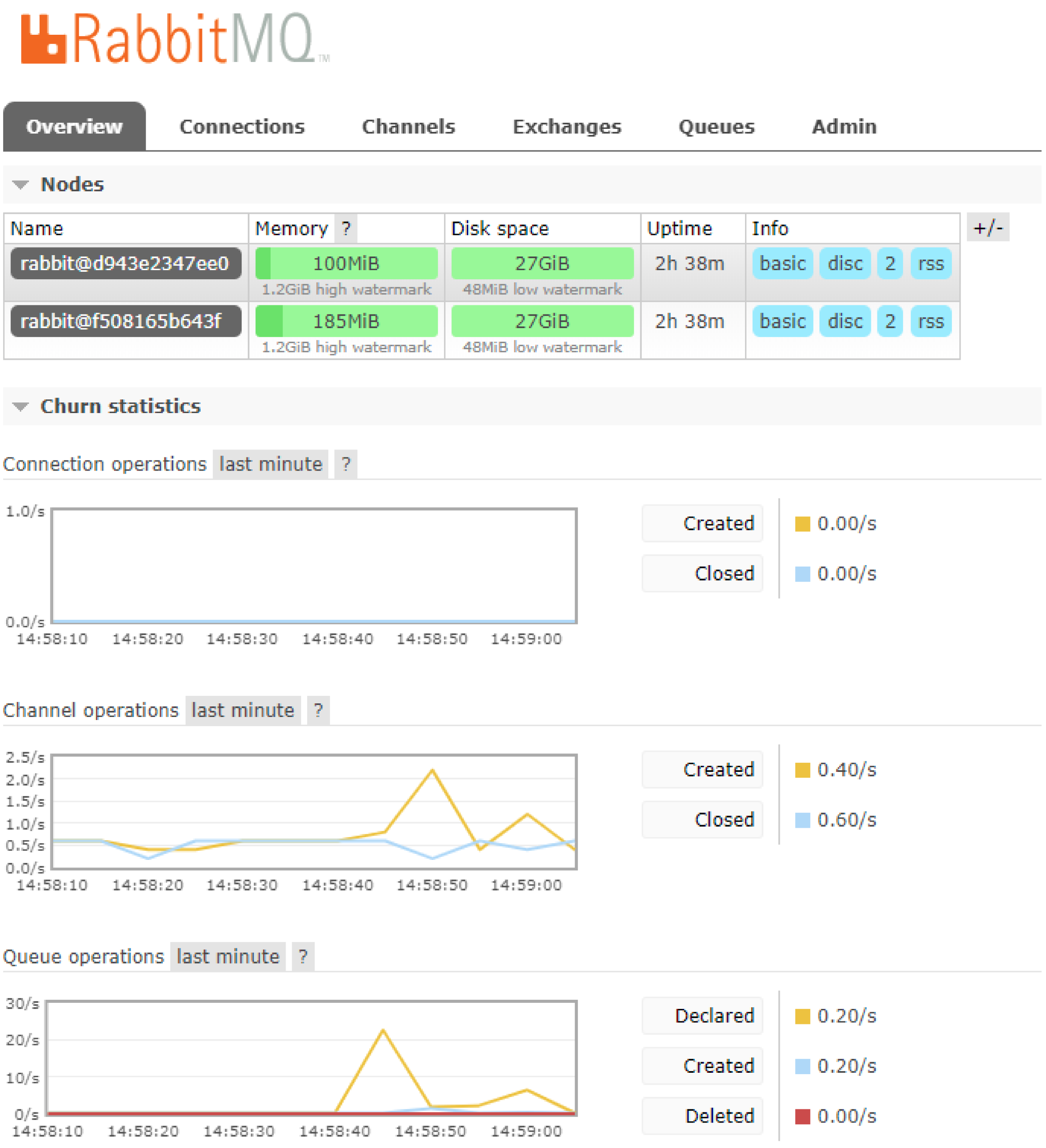
To	Routing key	Arguments	
amq.gen-AzSdp_5AkUW9hr0D8X-Prg	daemon.heartbeat.v1		Unbind
amq.gen-St2FG6vV3igfcrCMmpDjWQ	job.7d0b0e8e-4558-443f-9f44-bff3b27269c4.*.control.v1		Unbind
amq.gen-n3q40pQ4JfWSe4vgEof9XA	job.7d0b0e8e-4558-443f-9f44-bff3b27269c4.*.io.input.v1		Unbind
amq.gen-nUoBDOylrniSj0CLBCBKZw	workspace.event.v1		Unbind

Source: Elaborated by the author

- `job.{id}.*.control.v1`: the pattern that binds to a queue with messages that control whether the execution of a job is stopped or killed.
- `job.{id}.*.io.input.v1`: the pattern that binds to a queue with streaming input messages from a specific job identifier.
- `workspace.event.v1`: the pattern that binds to a queue with workspace events.

Figure 14 shows the web management UI. The overview page contains information about nodes and churn statistics such as connection operations, channel operations, and queue operations at a given period. The tabbed navigation also has links to connections, channels, exchanges, queues, and admin settings.

Figure 14 – RabbitMQ web management UI



Name	Memory ?	Disk space	Uptime	Info
rabbit@d943e2347ee0	100MiB 1.2GiB high watermark	27GiB 48MiB low watermark	2h 38m	basic disc 2 rss
rabbit@f508165b643f	185MiB 1.2GiB high watermark	27GiB 48MiB low watermark	2h 38m	basic disc 2 rss

Connection operations last minute ?

Created	🟨 0.00/s
Closed	🟦 0.00/s

Channel operations last minute ?

Created	🟨 0.40/s
Closed	🟦 0.60/s

Queue operations last minute ?

Declared	🟨 0.20/s
Created	🟦 0.20/s
Deleted	🟥 0.00/s

Source: Elaborated by the author

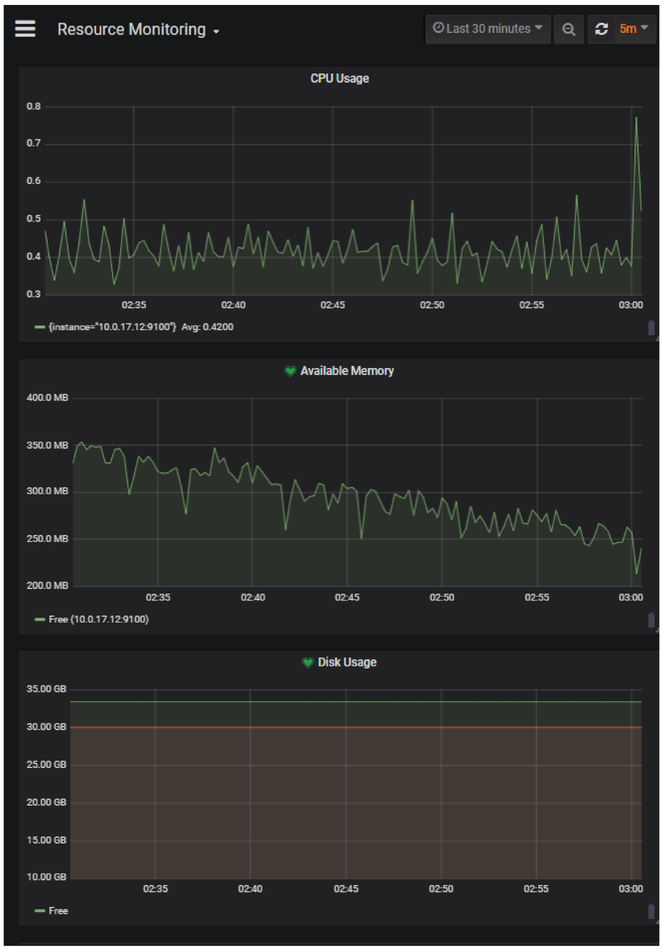
The current infrastructure entails one (1) cluster with two (2) nodes, and uses *queue mirroring* to enhance broker availability in the case of node failure. *Lazy queues* are also used for some queue types to move their contents to disk as early as possible, and only load them in RAM when requested by consumers. This technique reduces memory consumption and gives support to long queues. Both *queue mirroring* and *lazy queues* features are supported via user policies. These policies may be updated at any time using the CLI tool or management UI.

Prometheus and Grafana

Prometheus is a monitoring and alerting toolkit that scrapes metrics from jobs, either directly or via an intermediary gateway (PROMETHEUS, 2020). Grafana is a visualization platform that gives the ability to query and visualize metrics through dashboards, including data collected by Prometheus (GRAFANA, 2020). The server collects long-term metrics via Prometheus and uses Grafana dashboards for monitoring.

Figure 15 shows the dashboard for resource monitoring over 30 minutes, including CPU Usage, Available Memory, and Disk Usage. Alerts are also set for high usage levels with notifications sent via Slack.

Figure 15 – Grafana - Resource Monitoring



Source: Elaborated by the author

3.3.2 Database

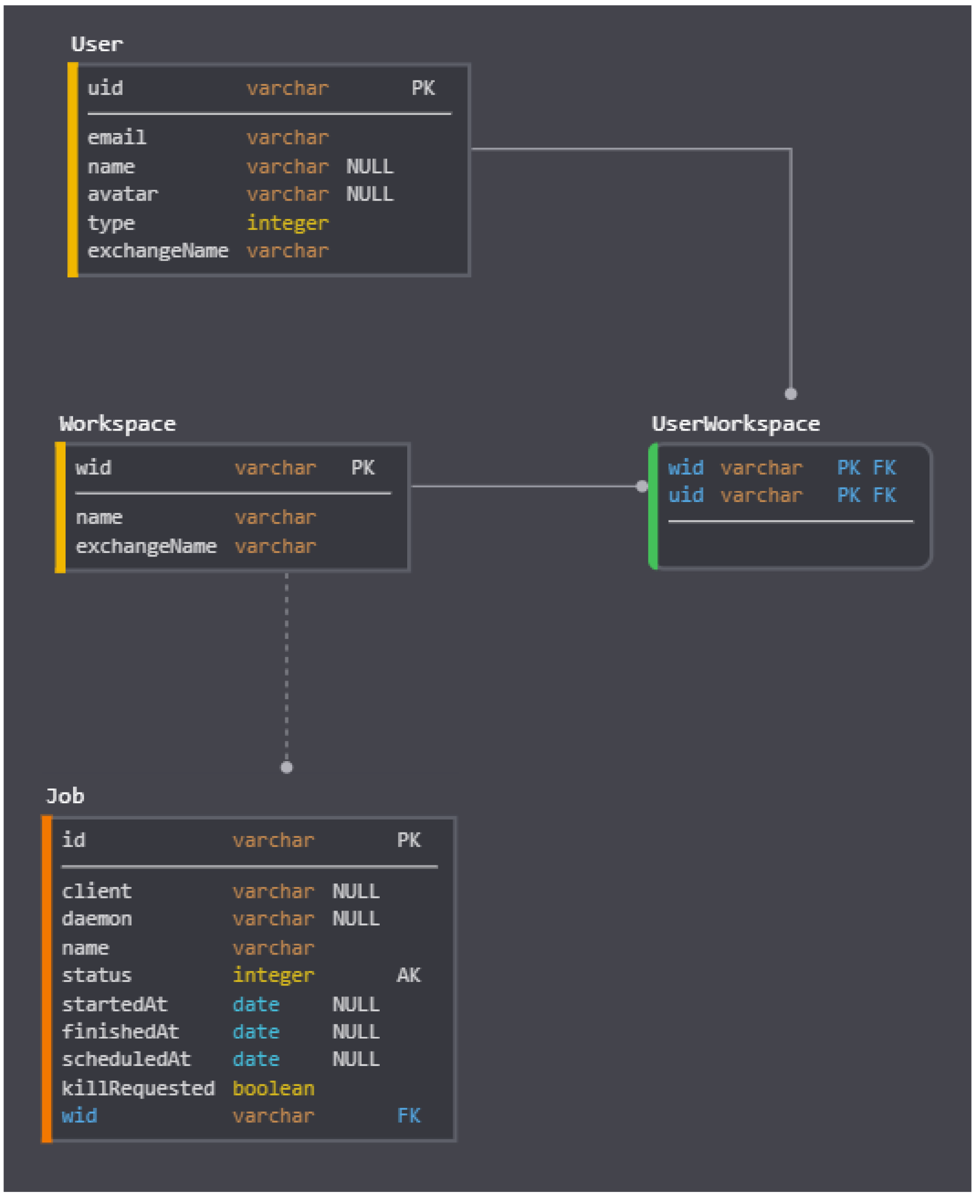
Runlet uses PostgreSQL, an open-source object-relational database that uses the Structured Query Language (SQL) language and is highly scalable both in the amount of data it can manage and in the number of concurrent users it accommodates (POSTGRESQL, 2020). The database is hosted on DigitalOcean as a managed database isolated from the droplet, which reduces the complexity of setting up and maintaining it for production. The service also offers automatic maintenance and updates, daily backups, end-to-end security, and automated recovery in the event of a failure.

The database stores data from workspaces, users, and jobs, as shown in figure 16. Table *Workspace* includes workspace id, name, and RabbitMQ's exchange name. Table *user* includes user id, email, name, avatar, type, and RabbitMQ's exchange name. Table *Job* includes job id, client, daemon, job name, status, when the execution started and finished, scheduled date if scheduled, and a boolean that indicates if the job must be killed. Each workspace must have one or more users, a relationship that is accomplished through a *UserWorkspace* mapping table.

The database does not store job logs. They are shared between devices using RabbitMQ's messaging system. This decision reduces security concerns related to leakage of sensitive information that records may have and improves performance by ensuring that updates are queued and delivered reliably without the need for querying and storing large amounts of data from a database. Most columns' names are self-explanatory. However, there are a few integer columns that match enumerated types according to the following mapping:

- ❑ Job Status:
  0. Pending: the job is waiting for execution.
  1. Running: the job is being executed.
  2. Error: an error has occurred while executing the job.
  3. Finished: the job finished its execution.
- ❑ User Type
  0. Basic: default account type.
  1. Pro: professional account with more features.

Figure 16 – Database ER model



```
User
uid           varchar        PK
email         varchar
name          varchar  NULL
avatar        varchar  NULL
type          integer
exchangeName  varchar
```

```
Workspace
wid           varchar   PK
name          varchar
exchangeName  varchar
```

```
UserWorkspace
wid  varchar   PK FK
uid  varchar   PK FK
```

```
Job
id            varchar        PK
client        varchar  NULL
daemon        varchar  NULL
name          varchar
status        integer        AK
startedAt     date     NULL
finishedAt    date     NULL
scheduledAt   date     NULL
killRequested boolean
wid           varchar        FK
```

Source: Elaborated by the author

3.3.3 Cloud Storage

Log storing is accomplished with MinIO, an open-source distributed object storage server designed to be cloud-native, performant, scalable, and lightweight. The server runs as lightweight containers by external orchestration services and is highly efficient in its use of CPU and memory resources, even under increased loads (MINIO, 2020). It also offers erasure coding, bitrot protection, and encryption schemes for data at rest.

MinIO stores objects using the same storage capacity of the droplet used for the server. However, DigitalOcean's block storage service may be used to attach a volume to the server and extend the storage capacity beyond the server's disk size.

3.3.4 Daemon

The daemon, written in Golang, is responsible for job orchestration and has an internal queue that executes jobs in the received order, FIFO, as well as a parallelism controller that helps to prevent CPU throttling by limiting the number of jobs that run in parallel. It runs on machine startup as a background process capable of recovering from crashes through a process restart.

Each daemon communicates with the server directly. It acts both as a producer and a consumer simultaneously, meaning that jobs are triggered and executed from any active node in the network. It supports the execution of selected methods via CLI by appending the term `runlet` to method names, which is useful for script automation and remote access on devices without a graphical interface. A brief description of currently supported methods is presented next.

- **runlet @config**: to view and edit jobs.
- **runlet @run**: to run jobs by name.
- **runlet @signin**: to sign in to account using either a URL or a QR code.
- **runlet @signout**: to sign out from account.

The daemon is also responsible for End-to-End Encryption (E2EE) of logs, local disk persistence, and data dispatch for cloud storage. Logs are first encrypted and persisted on the local disk for quick access and then submitted to the cloud storage server. Logs are only retrieved from the cloud when the local copy is outdated, according to the rules:

1. **If the log does not exist locally**: a local copy is created from the remote copy and used for reading operations.
2. **If log exists locally**: the hash of the local copy is sent to the server and compared with the hash present in the metadata of the remote copy. The remote copy replaces the hash's local copy if different, and the remote copy has a more recent modification date. Otherwise, the local copy is used for reading operations, avoiding unnecessary updates.

Jobs

Jobs are the basic blocks of Runlet and are defined either by manual inclusion via CLI or using the GUI. Each job has the following properties:

- **name**: name of the job.
- **description**: brief description of the job.
- **showDuration**: boolean value that indicates whether the duration is displayed after execution or not.
- **script**: single or multi-line string containing the commands that are executed by the job.
- **cwd**: optional property to change the current directory.
- **where**: defines where the job is executed. It gives the ability to set the execution for local devices, all connected devices, any online device, or a custom list of devices. The local device is selected by default.
- **whereCustom**: optional property to provide a custom list of devices, if the selected value for the property 'where' is 'custom'.
- **every**: optional property for scheduled execution. Cron-like and human-readable syntaxes are valid.
- **entrypoint**: optional property to define a command or script executed as the entry point. Default values are `/bin/bash` for Linux and macOS, and `cmd.exe` on Windows.

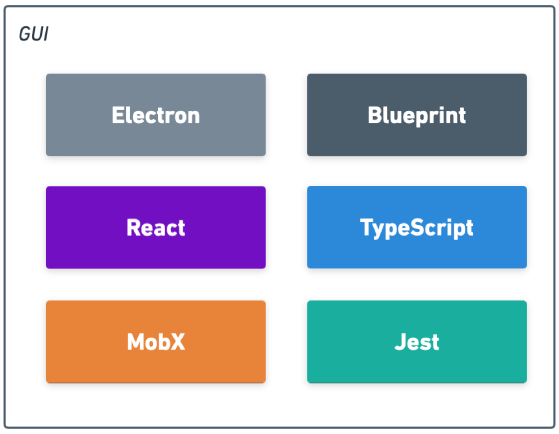
The following example shows the syntax of a job named `runlet-website`. This job has a multi-lined script string that opens Runlet's website and displays the job duration after execution.

```
runlet-website:
  script: |-
    echo "Opening website..."
    open https://runlet.app
  description: website
  showDuration: true
```

3.3.5 GUI

The GUI is built with JavaScript, HTML, and CSS on top of well-adopted open-source libraries such as Electron, React, TypeScript, MobX, Blueprint, Xterm.js, and Jest. Electron is a library developed by GitHub for building cross-platform desktop applications by combining Chromium and Node.js into a single runtime environment system (ELECTRON, 2020). Chromium is used for rendering pages, while the Node.js API is used for lower-level system actions such as managing process events and interacting with the file system.

Figure 17 – GUI Components



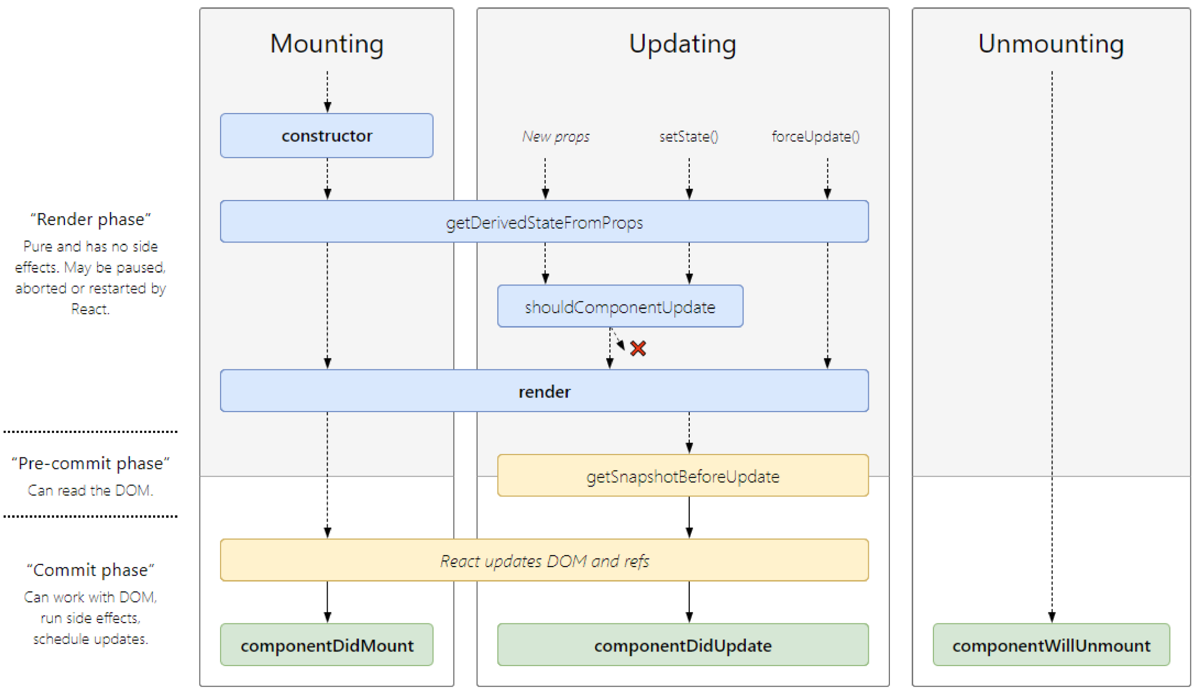
Source: Elaborated by the author

React is a component-based library for building user interfaces. Components are encapsulated, highly composable, and capable of managing their state (REACT, 2020). One of the main advantages of React is the usage of a Virtual Document Object Model (VDOM) to reduce the number of costly node tree mutations on updates. Also, each component has a list of lifecycle methods to run code at particular times in the process, giving developers control over mounting, updating, and unmounting components. Figure 18 presents the lifecycle methods diagram.

TypeScript is a typed superset of JavaScript that offers support for static checking, statement completion, code refactoring, type annotations, and the latest JavaScript features from ECMAScript 2015 and future proposals (TYPESCRIPT, 2020). TypeScript compiles to plain JavaScript and is used alongside React to get all the tooling benefits it offers.

Blueprint is a React-based toolkit that offers a set of components optimized for complex, data-dense desktop interfaces (BLUEPRINT, 2020). The set features more than 30 highly accessible components that cover basic element behaviors and include default CSS styles. Some of the frequently imported components in Runlet are buttons, cards, lists, menus, form controls, and text inputs.

Figure 18 – React Lifecycle Methods Diagram



Source: (MAJ, 2020)

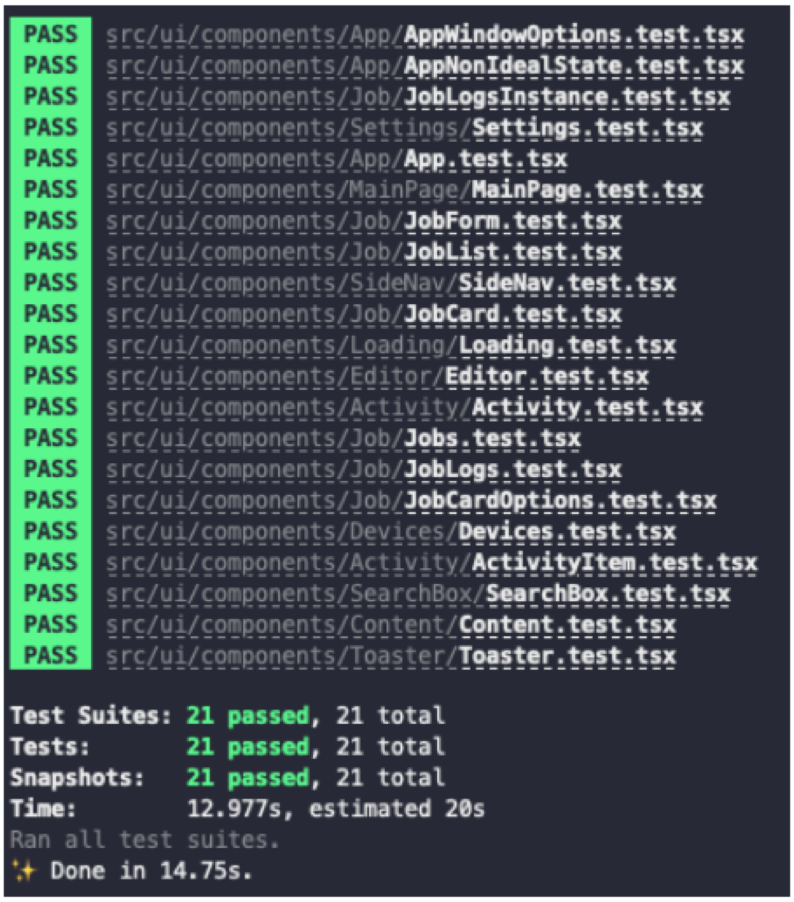
Xterm.js is a component that gives the ability to attach fully-featured terminals, such as bash, vim, and tmux, to applications (XTERM.JS, 2020). Runlet uses it to attach job outputs to terminal instances, giving users the ability to observe and interact with running jobs. This component supports rich Unicode, theming, addons and has been adopted by other popular projects such as VS Code, Hyper, and Theia.

MobX is a library for state management that applies concepts from functional reactive programming to observe, compute, and react to state changes (MOBX, 2020). It's a powerful library since it provides the mechanism to store and update an observable application state. This observable state is used by React to perform User Interface (UI) updates or VDOM mutations when strictly needed.

Runlet uses MobX to keep an observable state that reacts only to state changes rather than continuously check for updates on a timeframe basis. This pattern reduces the number of computations, the CPU workload and makes the application much more performant since Document Object Model (DOM) mutations only happen when relevant data is updated.

Jest is a JavaScript Testing Framework that offers a suite of tools for code testing, including snapshot testing to keep track of changes on large objects, code coverage information, and mock functions to test links between code (JEST, 2020). Runlet uses snapshot testing to keep track of component changes and spot bugs as soon as they are introduced. Figure 19 shows a Jest report of Runlet components tests.

Figure 19 – Jest Tests



```
PASS  src/ui/components/App/AppWindowOptions.test.tsx
PASS  src/ui/components/App/AppNonIdealState.test.tsx
PASS  src/ui/components/Job/JobLogsInstance.test.tsx
PASS  src/ui/components/Settings/Settings.test.tsx
PASS  src/ui/components/App/App.test.tsx
PASS  src/ui/components/MainPage/MainPage.test.tsx
PASS  src/ui/components/Job/JobForm.test.tsx
PASS  src/ui/components/Job/JobList.test.tsx
PASS  src/ui/components/SideNav/SideNav.test.tsx
PASS  src/ui/components/Job/JobCard.test.tsx
PASS  src/ui/components/Loading/Loading.test.tsx
PASS  src/ui/components/Editor/Editor.test.tsx
PASS  src/ui/components/Activity/Activity.test.tsx
PASS  src/ui/components/Job/Jobs.test.tsx
PASS  src/ui/components/Job/JobLogs.test.tsx
PASS  src/ui/components/Job/JobCardOptions.test.tsx
PASS  src/ui/components/Devices/Devices.test.tsx
PASS  src/ui/components/Activity/ActivityItem.test.tsx
PASS  src/ui/components/SearchBox/SearchBox.test.tsx
PASS  src/ui/components/Content/Content.test.tsx
PASS  src/ui/components/Toaster/Toaster.test.tsx

Test Suites: 21 passed, 21 total
Tests:       21 passed, 21 total
Snapshots:   21 passed, 21 total
Time:        12.977s, estimated 20s
Ran all test suites.
✨ Done in 14.75s.
```

Source: Elaborated by the author

3.3.6 Builds and Deployments

Builds and deployments are fully automated with Azure Pipelines, a cloud-hosted platform to continuously build, test, and deploy applications to any platform and cloud (AZURE, 2020b). The platform offers integration to GitHub and has agents with full pipeline support for Linux, macOS, and Windows.

Runlet has a single pipeline that pulls the source code from a private repository and builds the application for all platforms. The artifacts generated are drafted for release on GitHub every time a version change is detected, and are released upon manual review. Version numbers and increments follow the set of rules defined by the Semantic Versioning system authored by (PRESTON-WERNER, 2020).

Chapter 4

Experimental Evaluation

This chapter describes the experimental evaluation conducted to prove the hypothesis that Runlet offers interactive job execution with reliable message delivery across a network of heterogeneous devices. Section 4.1 describes the computational power of the server and the testbed of devices. Section 4.2 describes the nature of the experiments. Section 4.3 discusses the experimental results.

4.1 Testbed Description

The server, currently hosted on DigitalOcean, is a standard droplet with two (2) vCPUs, four (4) GB RAM, fifty (50) GB SSD storage, and four (4) TB of data transfer. A vCPU is a processing power unit that corresponds to a single hyperthread on a processor core (DIGITALOCEAN, 2020b). The testbed of devices used for the experiments is presented in table 5, including device model, CPU, RAM, operating system, and an alias used for study reference.

Table 5 – Testbed Devices

Alias	Device Model	CPU	RAM	Operating System
win	Dell G5	Intel Core i7-9750H @ 2.60GHz	16GB	Windows 10
mac	Macbook Pro Late 2013	Intel Core i5-4288U @ 2.60GHz	8GB	macOS Catalina 10.15.3
pi-1	Raspberry Pi Model B Rev 2	Broadcom BCM2835 ARMv6 @ 700MHz	512MB	Raspbian GNU/Linux 10
pi-2	Raspberry Pi 2 Model B Rev 1.1	Broadcom BCM2836 ARMv7 @ 900MHz	1GB	Raspbian GNU/Linux 10
pi-3	Raspberry Pi 3 Model B Plus Rev 1.3	Broadcom BCM2837B0 ARMv7 @ 1.4GHz	1GB	Raspbian GNU/Linux 9.11

Source: Elaborated by the author

4.2 Experiments

This section describes the experiments that evaluate Runlet in regards to interactivity and reliability. Interactivity is tested under three scenarios of job executions, and reliability is investigated by observing the broker after node and server failures.

4.2.1 Interactivity

Three (3) experiments are conducted to evaluate the user's capability of interacting with job executions and making decisions when requested. A different job is executed on each experiment to demonstrate not only unique types of interactivity but also how Runlet handles the rendering of different outputs.

The first experiment verifies the ability to trigger a macOS job from a Windows manager. The job is called `brew-cask-upgrade` and executes two commands: (1) `neofetch` and (2) `brew cu -a`. `neofetch` is a command-line system tool written in bash that displays information about the operating system and hardware (DYLANARAPS, 2020). `brew cu -a` is a command from a tool called `brew-cask-upgrade`, which is a command-line tool for upgrading outdated apps installed by Homebrew Cask (BUO, 2020). Homebrew Cask is a tool that extends Homebrew and is used for installation and management of macOS applications distributed as binaries (HOMEBREW, 2020).

The second experiment verifies the ability to trigger a job `rpi-neofetch` from a macOS manager on two Raspberry Pi devices, `pi-1` and `pi-2`. The job initially installs the package neofetch using apt-get and then runs neofetch to obtain system information. This is accomplished executing two commands (1) `apt-get install neofetch` and (2) `neofetch`, and differs from the first experiment in a few ways:

1. A desktop manager is used to trigger the execution of a job on ARM devices that only have the daemon installed.
2. The devices have different hardware configuration and set of installed packages, which may require user interaction at some point to confirm operations.
3. It shows that individual decisions may be made for each device when a job is executed across a network of devices under different conditions.

The third experiment shows how a Raspberry PI may be remotely monitored using htop, which is an interactive text-mode process viewer for Unix systems (MUHAMMAD, 2020). This viewer is highly configurable and gives the option to view information such as CPU load, memory consumption, hostname, tasks, load averages, and uptime.

4.2.2 Reliability

Reliability stands for the system's ability of delivering queued messages after failure. Two types of failures are investigated in this study: (1) node failure and (2) server failure. The method used to investigate each one is presented next.

1. **Node failure**: the number of nodes in the cluster is downscaled from four (4) nodes to a single node. This change does not cause any issues as RabbitMQ tolerates individual nodes' failure as long as there are other known nodes in the cluster at the time. It may also not affect queues as the current infrastructure uses *queue mirroring* to replicate queues across nodes.

2. **Server failure**: the single node is removed for a brief period to observe how the broker reacts in the absence of nodes. It does not cause any significant issues to the server apart from becoming temporarily unavailable as the broker is configured to use *lazy queues*, a policy that moves messages from a large number of queue types to disk as early as possible and loads them in memory only when requested. These messages are expected to be retrieved from the disk after new nodes become operational. All queues are also likely to return.

4.3 Discussion

This section describes the results from each experiment through screenshots of executed jobs and monitoring metrics from Prometheus and Grafana.

4.3.1 Interactivity

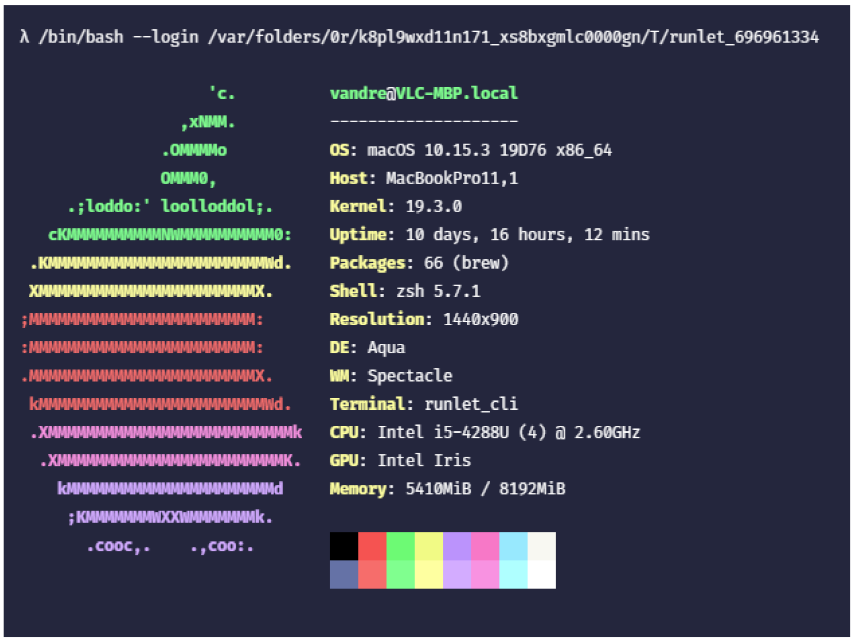
Three (3) experiments evaluate interactivity. Each one of them contains a brief description of the experiment, the job definition in YAML syntax, and the output explanation supported by screenshots.

Experiment 1

A job is triggered by a Windows desktop manager to update system applications on a macOS device using homebrew.

```
brew-cask-upgrade:
  script: |-
    neofetch;
    brew cu -a;
  description: brew cask upgrade
  showDuration: true
  where: custom
  whereCustom:
  - mac
```

Figure 20 – Output of (1) neofetch



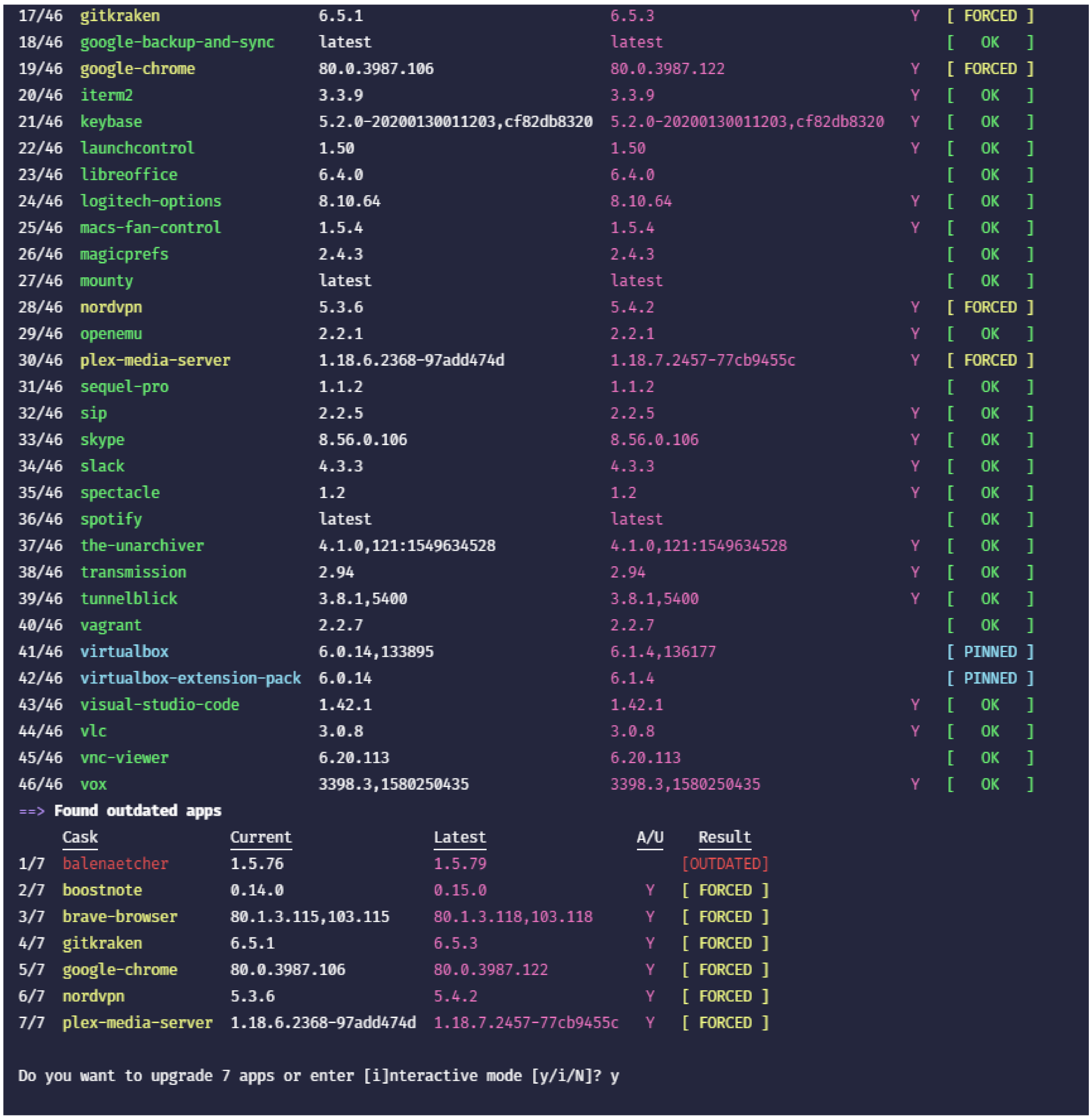
```
λ /bin/bash --login /var/folders/0r/k8pl9wxd11n171_xs8bxgmlc0000gn/T/runlet_696961334

                    'c.          vandre@VLC-MBP.local
                 ,xNMM.          --------------------
               .OMMMMo           OS: macOS 10.15.3 19D76 x86_64
               OMMM0,            Host: MacBookPro11,1
     .;loddo:' loolloddol;.      Kernel: 19.3.0
   cKMMMMMMMMMMNWMMMMMMMMMM0:    Uptime: 10 days, 16 hours, 12 mins
 .KMMMMMMMMMMMMMMMMMMMMMMMWd.    Packages: 66 (brew)
 XMMMMMMMMMMMMMMMMMMMMMMMX.      Shell: zsh 5.7.1
;MMMMMMMMMMMMMMMMMMMMMMMM:       Resolution: 1440x900
:MMMMMMMMMMMMMMMMMMMMMMMM:       DE: Aqua
.MMMMMMMMMMMMMMMMMMMMMMMMX.      WM: Spectacle
 kMMMMMMMMMMMMMMMMMMMMMMMMWd.    Terminal: runlet_cli
 .XMMMMMMMMMMMMMMMMMMMMMMMMMMk   CPU: Intel i5-4288U (4) @ 2.60GHz
  .XMMMMMMMMMMMMMMMMMMMMMMMMK.   GPU: Intel Iris
    kMMMMMMMMMMMMMMMMMMMMMMd     Memory: 5410MiB / 8192MiB
     ;KMMMMMMMWXXWMMMMMMMk.
       .cooc,.    .,coo:.
```

Source: Elaborated by the author

Figure 20 shows the output of command (1) neofetch, while figure 21 shows the output generated by (2) `brew cu -a`. The initial output lists all installed applications, including current and latest versions, and a list of outdated apps. The option `-a` passed with `brew cu` indicates that applications that have the auto-update functionality are also listed. The question "Do you want to upgrade 7 apps or enter [i]nteractive mode [y/i/N]?" is shown after the list of outdated apps and requires user interaction. The meaning of which option is presented below.

- y: confirms that all outdated apps will be updated.
- i: enters the interactive mode that allows casks to be manually selected.
- N: skips update of outdated apps.

Figure 21 – Output of (2) brew cu -a before confirmation [y]

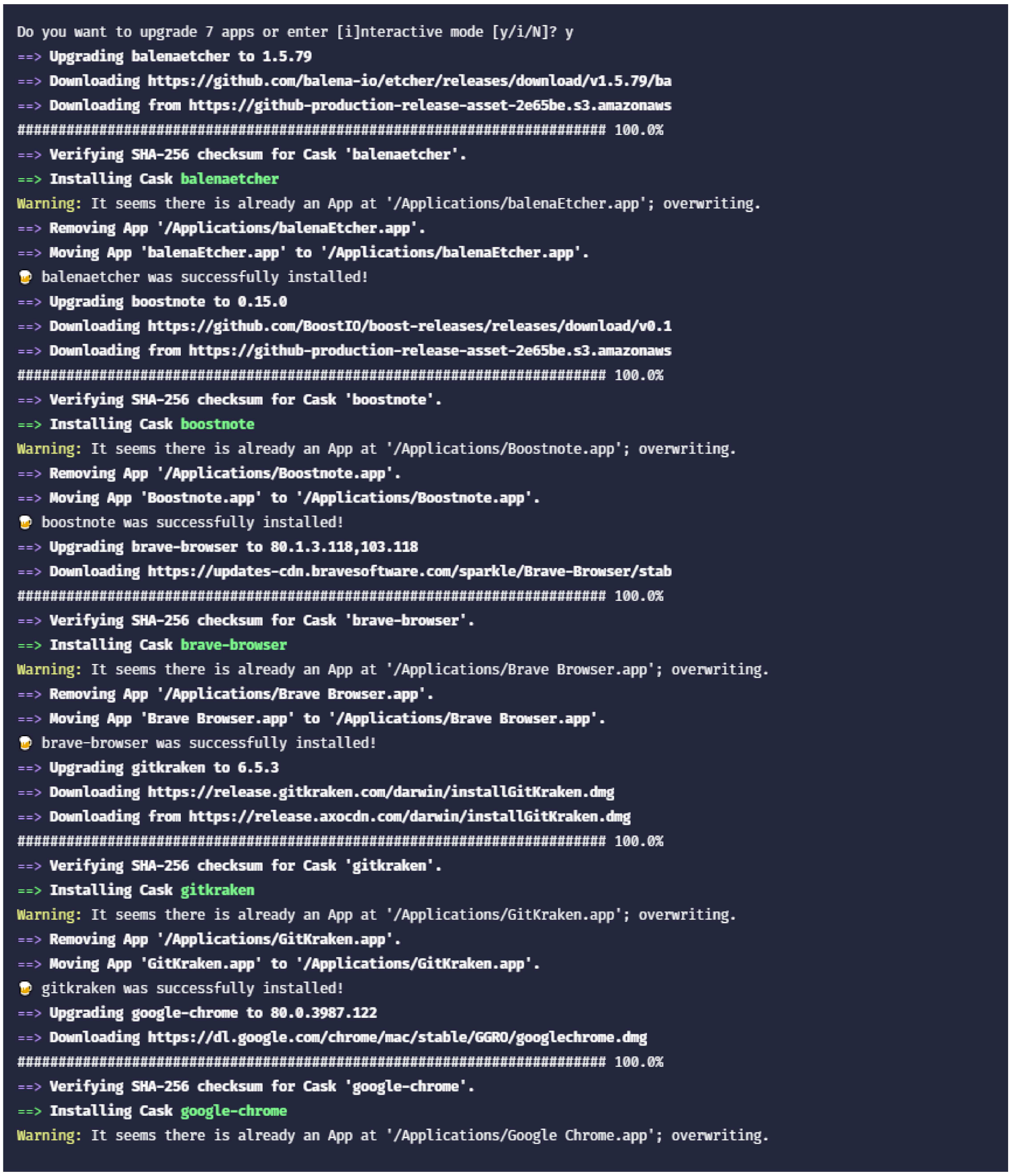


```
17/46  gitkraken                  6.5.1                            6.5.3                            Y  [ FORCED ]
18/46  google-backup-and-sync     latest                           latest                              [   OK   ]
19/46  google-chrome              80.0.3987.106                    80.0.3987.122                    Y  [ FORCED ]
20/46  iterm2                     3.3.9                            3.3.9                            Y  [   OK   ]
21/46  keybase                    5.2.0-20200130011203,cf82db8320  5.2.0-20200130011203,cf82db8320  Y  [   OK   ]
22/46  launchcontrol              1.50                             1.50                             Y  [   OK   ]
23/46  libreoffice                6.4.0                            6.4.0                               [   OK   ]
24/46  logitech-options           8.10.64                          8.10.64                          Y  [   OK   ]
25/46  macs-fan-control           1.5.4                            1.5.4                            Y  [   OK   ]
26/46  magicprefs                 2.4.3                            2.4.3                               [   OK   ]
27/46  mounty                     latest                           latest                              [   OK   ]
28/46  nordvpn                    5.3.6                            5.4.2                            Y  [ FORCED ]
29/46  openemu                    2.2.1                            2.2.1                            Y  [   OK   ]
30/46  plex-media-server          1.18.6.2368-97add474d            1.18.7.2457-77cb9455c            Y  [ FORCED ]
31/46  sequel-pro                 1.1.2                            1.1.2                               [   OK   ]
32/46  sip                        2.2.5                            2.2.5                            Y  [   OK   ]
33/46  skype                      8.56.0.106                       8.56.0.106                       Y  [   OK   ]
34/46  slack                      4.3.3                            4.3.3                            Y  [   OK   ]
35/46  spectacle                  1.2                              1.2                              Y  [   OK   ]
36/46  spotify                    latest                           latest                              [   OK   ]
37/46  the-unarchiver             4.1.0,121:1549634528             4.1.0,121:1549634528             Y  [   OK   ]
38/46  transmission               2.94                             2.94                             Y  [   OK   ]
39/46  tunnelblick                3.8.1,5400                       3.8.1,5400                       Y  [   OK   ]
40/46  vagrant                    2.2.7                            2.2.7                               [   OK   ]
41/46  virtualbox                 6.0.14,133895                    6.1.4,136177                        [ PINNED ]
42/46  virtualbox-extension-pack  6.0.14                           6.1.4                               [ PINNED ]
43/46  visual-studio-code         1.42.1                           1.42.1                           Y  [   OK   ]
44/46  vlc                        3.0.8                            3.0.8                            Y  [   OK   ]
45/46  vnc-viewer                 6.20.113                         6.20.113                            [   OK   ]
46/46  vox                        3398.3,1580250435                3398.3,1580250435                Y  [   OK   ]
==> Found outdated apps
     Cask               Current                Latest                  A/U    Result
1/7  balenaetcher       1.5.76                 1.5.79                        [OUTDATED]
2/7  boostnote          0.14.0                 0.15.0                   Y    [ FORCED ]
3/7  brave-browser      80.1.3.115,103.115     80.1.3.118,103.118       Y    [ FORCED ]
4/7  gitkraken          6.5.1                  6.5.3                    Y    [ FORCED ]
5/7  google-chrome      80.0.3987.106          80.0.3987.122            Y    [ FORCED ]
6/7  nordvpn            5.3.6                  5.4.2                    Y    [ FORCED ]
7/7  plex-media-server  1.18.6.2368-97add474d  1.18.7.2457-77cb9455c    Y    [ FORCED ]

Do you want to upgrade 7 apps or enter [i]nteractive mode [y/i/N]? y
```

Source: Elaborated by the author

Figure 22 shows an excerpt of the upgrading process that takes place after user confirmation [y]. The progress of each upgrade is displayed individually, which helps to visualize the overall progress.

Figure 22 – Output of (2) brew cu -a after confirmation [y]



```
Do you want to upgrade 7 apps or enter [i]nteractive mode [y/i/N]? y
==> Upgrading balenaetcher to 1.5.79
==> Downloading https://github.com/balena-io/etcher/releases/download/v1.5.79/ba
==> Downloading from https://github-production-release-asset-2e65be.s3.amazonaws
######################################################################## 100.0%
==> Verifying SHA-256 checksum for Cask 'balenaetcher'.
==> Installing Cask balenaetcher
Warning: It seems there is already an App at '/Applications/balenaEtcher.app'; overwriting.
==> Removing App '/Applications/balenaEtcher.app'.
==> Moving App 'balenaEtcher.app' to '/Applications/balenaEtcher.app'.
🍺 balenaetcher was successfully installed!
==> Upgrading boostnote to 0.15.0
==> Downloading https://github.com/BoostIO/boost-releases/releases/download/v0.1
==> Downloading from https://github-production-release-asset-2e65be.s3.amazonaws
######################################################################## 100.0%
==> Verifying SHA-256 checksum for Cask 'boostnote'.
==> Installing Cask boostnote
Warning: It seems there is already an App at '/Applications/Boostnote.app'; overwriting.
==> Removing App '/Applications/Boostnote.app'.
==> Moving App 'Boostnote.app' to '/Applications/Boostnote.app'.
🍺 boostnote was successfully installed!
==> Upgrading brave-browser to 80.1.3.118,103.118
==> Downloading https://updates-cdn.bravesoftware.com/sparkle/Brave-Browser/stab
######################################################################## 100.0%
==> Verifying SHA-256 checksum for Cask 'brave-browser'.
==> Installing Cask brave-browser
Warning: It seems there is already an App at '/Applications/Brave Browser.app'; overwriting.
==> Removing App '/Applications/Brave Browser.app'.
==> Moving App 'Brave Browser.app' to '/Applications/Brave Browser.app'.
🍺 brave-browser was successfully installed!
==> Upgrading gitkraken to 6.5.3
==> Downloading https://release.gitkraken.com/darwin/installGitKraken.dmg
==> Downloading from https://release.axocdn.com/darwin/installGitKraken.dmg
######################################################################## 100.0%
==> Verifying SHA-256 checksum for Cask 'gitkraken'.
==> Installing Cask gitkraken
Warning: It seems there is already an App at '/Applications/GitKraken.app'; overwriting.
==> Removing App '/Applications/GitKraken.app'.
==> Moving App 'GitKraken.app' to '/Applications/GitKraken.app'.
🍺 gitkraken was successfully installed!
==> Upgrading google-chrome to 80.0.3987.122
==> Downloading https://dl.google.com/chrome/mac/stable/GGRO/googlechrome.dmg
######################################################################## 100.0%
==> Verifying SHA-256 checksum for Cask 'google-chrome'.
==> Installing Cask google-chrome
Warning: It seems there is already an App at '/Applications/Google Chrome.app'; overwriting.
```

Source: Elaborated by the author

Experiment 2

A job is triggered by a macOS desktop manager to display hardware and system information from two Raspberry PI devices using neofetch.

```
rpi-neofetch:
  script: |-
    apt-get install neofetch;
    neofetch;
  description: neofetch
  showDuration: true
  where: custom
  whereCustom:
  - pi-1,pi-2
```

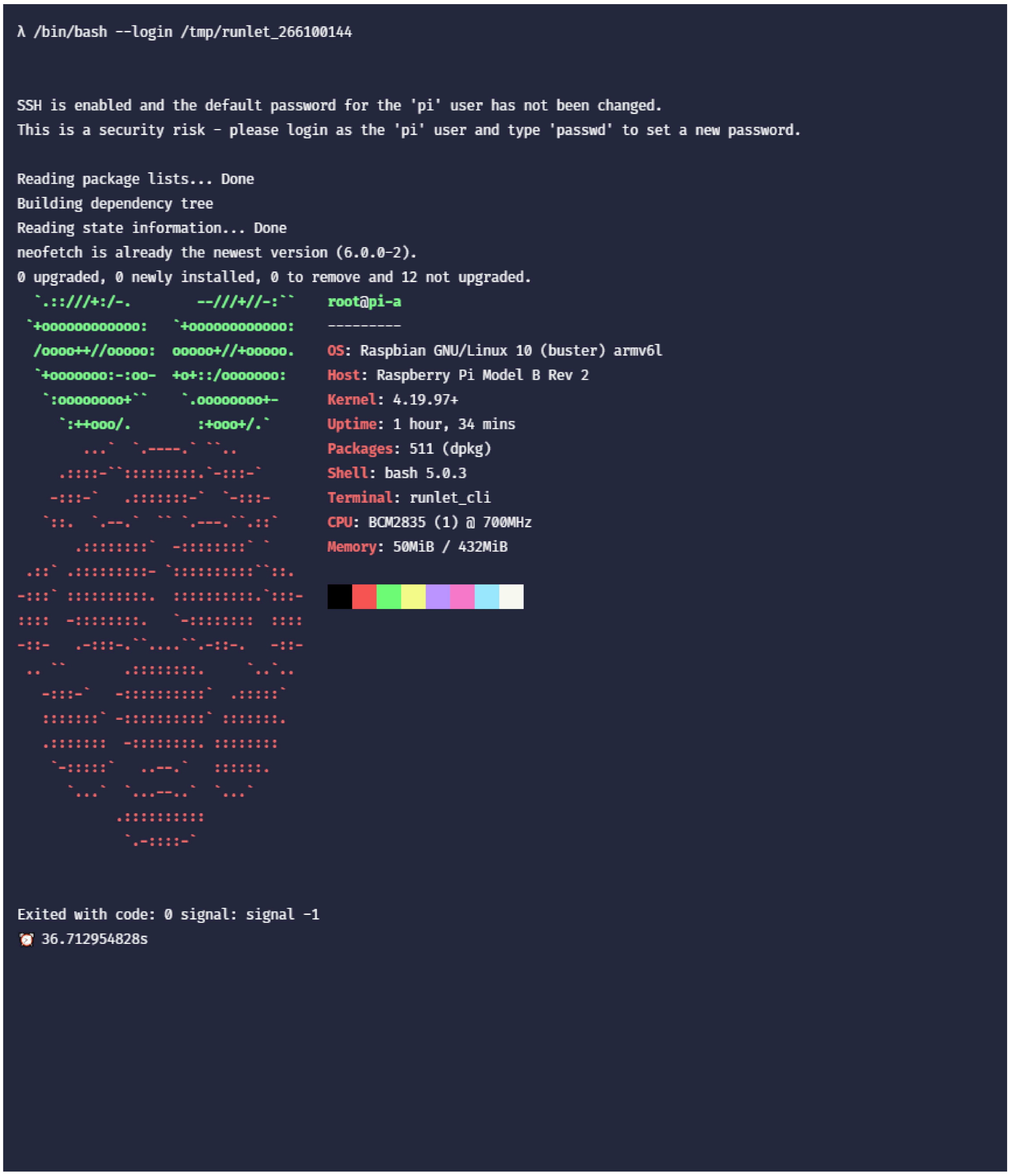
pi-1

The first device is a Raspberry Pi Model B Rev 2 that has neofetch installed and skips the package installation process for this reason. Figure 23 shows the complete output of `pi-a` with both commands executed without any user interaction.

pi-2

The second device is a Raspberry Pi 2 Model B Rev 1.1 that does not have neofetch installed and requires user interaction during its execution. Figure 24 shows the initial output of `pi-b`, including the question "Do you want to continue? [Y/n]" asked to confirm the installation of new packages required by neofetch. Packages are downloaded/installed after confirmation [y], and neofetch is automatically executed right after, as shown in figure 25.

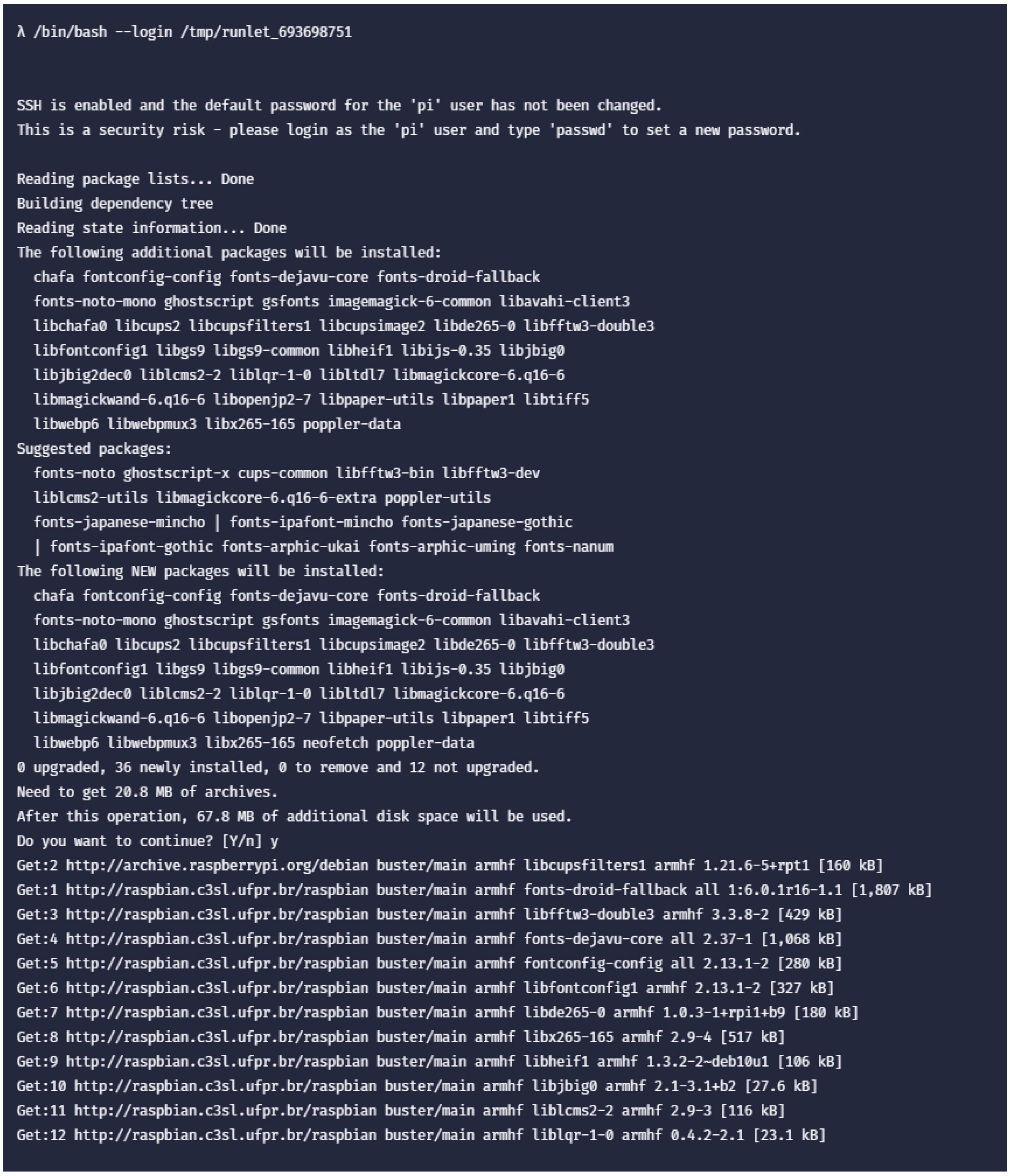
Figure 23 – Complete output of pi-1



```
λ /bin/bash --login /tmp/runlet_266100144


SSH is enabled and the default password for the 'pi' user has not been changed.
This is a security risk - please login as the 'pi' user and type 'passwd' to set a new password.

Reading package lists... Done
Building dependency tree
Reading state information... Done
neofetch is already the newest version (6.0.0-2).
0 upgraded, 0 newly installed, 0 to remove and 12 not upgraded.
  `.::///+:/-.        --///+//-:``    root@pi-a
 `+oooooooooooo:   `+oooooooooooo:    ---------
  /oooo++//ooooo:  ooooo+//+ooooo.    OS: Raspbian GNU/Linux 10 (buster) armv6l
  `+ooooooo:-:oo-  +o+::/ooooooo:     Host: Raspberry Pi Model B Rev 2
   `:oooooooo+``    `.oooooooo+-      Kernel: 4.19.97+
     `:++ooo/.        :+ooo+/.`       Uptime: 1 hour, 34 mins
        ...`  `.----.` ``..           Packages: 511 (dpkg)
     .::::-``:::::::::.`-:::-`        Shell: bash 5.0.3
    -:::-`   .:::::::-`  `-:::-       Terminal: runlet_cli
   `::.  `.--.`  `` `.---.``.::`      CPU: BCM2835 (1) @ 700MHz
       .::::::::`  -::::::::` `       Memory: 50MiB / 432MiB
 .::` .:::::::::- `::::::::::``::.
-:::` ::::::::::.  ::::::::::.`:::-   ⬛🟥🟩🟨🟪🟪🟦⬜
::::  -::::::::.   `-::::::::  ::::
-::-   .-:::-.``....``.-::-.   -::-
 .. ``       .::::::::.     `..`..
   -:::-`   -::::::::::`  .:::::`
   :::::::` -::::::::::` :::::::.
   .:::::::  -::::::::. ::::::::
    `-:::::`   ..--.`   ::::::.
      `...`  `...--..`  `...`
            .::::::::::
             `.-::::-`



Exited with code: 0 signal: signal -1
⏰ 36.712954828s
```

Source: Elaborated by the author

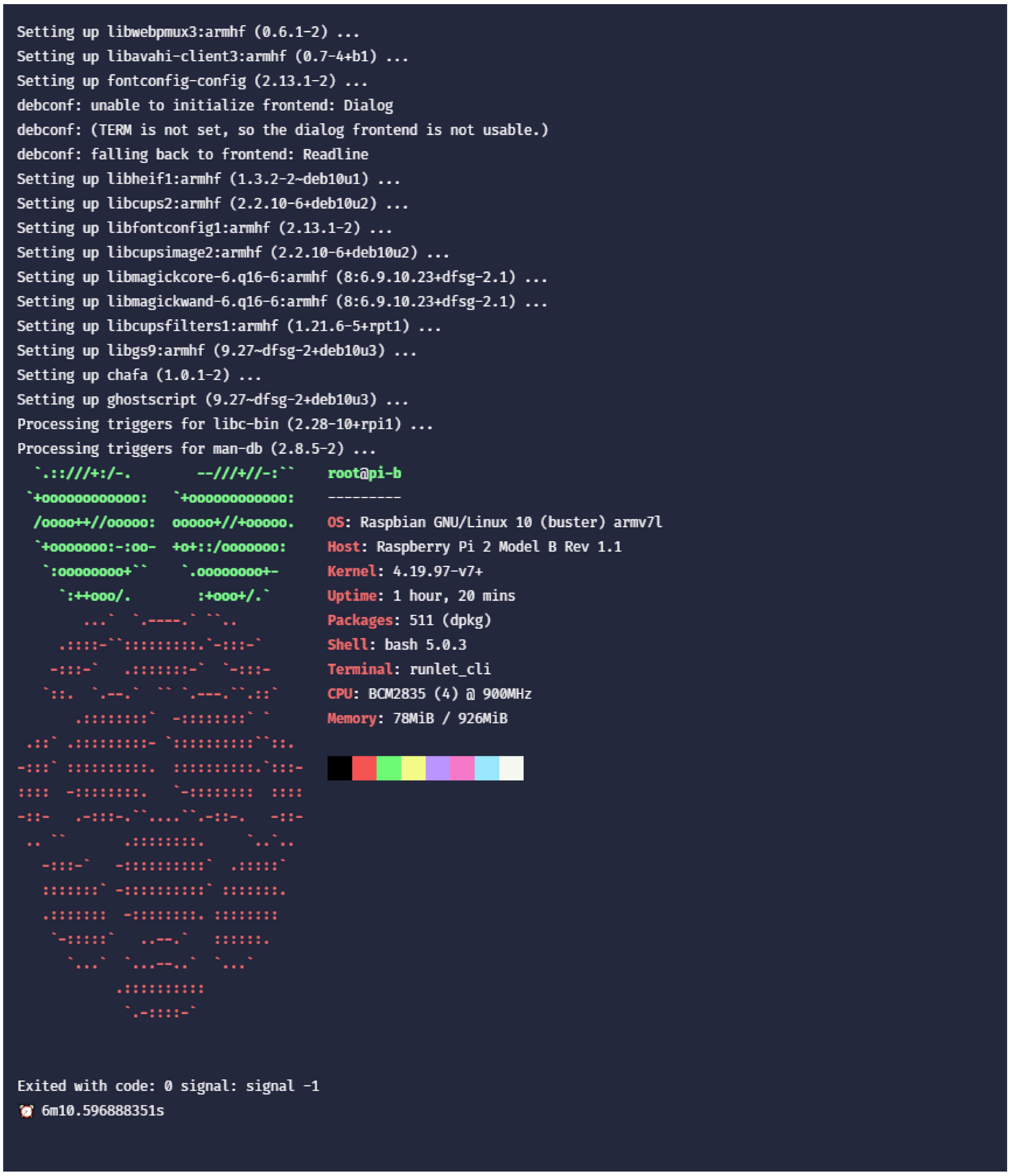
Figure 24 – Initial output excerpt of pi-2



```
λ /bin/bash --login /tmp/runlet_693698751



SSH is enabled and the default password for the 'pi' user has not been changed.
This is a security risk - please login as the 'pi' user and type 'passwd' to set a new password.

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  chafa fontconfig-config fonts-dejavu-core fonts-droid-fallback
  fonts-noto-mono ghostscript gsfonts imagemagick-6-common libavahi-client3
  libchafa0 libcups2 libcupsfilters1 libcupsimage2 libde265-0 libfftw3-double3
  libfontconfig1 libgs9 libgs9-common libheif1 libijs-0.35 libjbig0
  libjbig2dec0 liblcms2-2 liblqr-1-0 libltdl7 libmagickcore-6.q16-6
  libmagickwand-6.q16-6 libopenjp2-7 libpaper-utils libpaper1 libtiff5
  libwebp6 libwebpmux3 libx265-165 poppler-data
Suggested packages:
  fonts-noto ghostscript-x cups-common libfftw3-bin libfftw3-dev
  liblcms2-utils libmagickcore-6.q16-6-extra poppler-utils
  fonts-japanese-mincho | fonts-ipafont-mincho fonts-japanese-gothic
  | fonts-ipafont-gothic fonts-arphic-ukai fonts-arphic-uming fonts-nanum
The following NEW packages will be installed:
  chafa fontconfig-config fonts-dejavu-core fonts-droid-fallback
  fonts-noto-mono ghostscript gsfonts imagemagick-6-common libavahi-client3
  libchafa0 libcups2 libcupsfilters1 libcupsimage2 libde265-0 libfftw3-double3
  libfontconfig1 libgs9 libgs9-common libheif1 libijs-0.35 libjbig0
  libjbig2dec0 liblcms2-2 liblqr-1-0 libltdl7 libmagickcore-6.q16-6
  libmagickwand-6.q16-6 libopenjp2-7 libpaper-utils libpaper1 libtiff5
  libwebp6 libwebpmux3 libx265-165 neofetch poppler-data
0 upgraded, 36 newly installed, 0 to remove and 12 not upgraded.
Need to get 20.8 MB of archives.
After this operation, 67.8 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:2 http://archive.raspberrypi.org/debian buster/main armhf libcupsfilters1 armhf 1.21.6-5+rpt1 [160 kB]
Get:1 http://raspbian.c3sl.ufpr.br/raspbian buster/main armhf fonts-droid-fallback all 1:6.0.1r16-1.1 [1,807 kB]
Get:3 http://raspbian.c3sl.ufpr.br/raspbian buster/main armhf libfftw3-double3 armhf 3.3.8-2 [429 kB]
Get:4 http://raspbian.c3sl.ufpr.br/raspbian buster/main armhf fonts-dejavu-core all 2.37-1 [1,068 kB]
Get:5 http://raspbian.c3sl.ufpr.br/raspbian buster/main armhf fontconfig-config all 2.13.1-2 [280 kB]
Get:6 http://raspbian.c3sl.ufpr.br/raspbian buster/main armhf libfontconfig1 armhf 2.13.1-2 [327 kB]
Get:7 http://raspbian.c3sl.ufpr.br/raspbian buster/main armhf libde265-0 armhf 1.0.3-1+rpi1+b9 [180 kB]
Get:8 http://raspbian.c3sl.ufpr.br/raspbian buster/main armhf libx265-165 armhf 2.9-4 [517 kB]
Get:9 http://raspbian.c3sl.ufpr.br/raspbian buster/main armhf libheif1 armhf 1.3.2-2~deb10u1 [106 kB]
Get:10 http://raspbian.c3sl.ufpr.br/raspbian buster/main armhf libjbig0 armhf 2.1-3.1+b2 [27.6 kB]
Get:11 http://raspbian.c3sl.ufpr.br/raspbian buster/main armhf liblcms2-2 armhf 2.9-3 [116 kB]
Get:12 http://raspbian.c3sl.ufpr.br/raspbian buster/main armhf liblqr-1-0 armhf 0.4.2-2.1 [23.1 kB]
```

Source: Elaborated by the author

Figure 25 – Final output excerpt of pi-2



```
Setting up libwebpmux3:armhf (0.6.1-2) ...
Setting up libavahi-client3:armhf (0.7-4+b1) ...
Setting up fontconfig-config (2.13.1-2) ...
debconf: unable to initialize frontend: Dialog
debconf: (TERM is not set, so the dialog frontend is not usable.)
debconf: falling back to frontend: Readline
Setting up libheif1:armhf (1.3.2-2~deb10u1) ...
Setting up libcups2:armhf (2.2.10-6+deb10u2) ...
Setting up libfontconfig1:armhf (2.13.1-2) ...
Setting up libcupsimage2:armhf (2.2.10-6+deb10u2) ...
Setting up libmagickcore-6.q16-6:armhf (8:6.9.10.23+dfsg-2.1) ...
Setting up libmagickwand-6.q16-6:armhf (8:6.9.10.23+dfsg-2.1) ...
Setting up libcupsfilters1:armhf (1.21.6-5+rpt1) ...
Setting up libgs9:armhf (9.27~dfsg-2+deb10u3) ...
Setting up chafa (1.0.1-2) ...
Setting up ghostscript (9.27~dfsg-2+deb10u3) ...
Processing triggers for libc-bin (2.28-10+rpi1) ...
Processing triggers for man-db (2.8.5-2) ...
  `.::///+:/-.        --///+//-:``    root@pi-b
 `+oooooooooooo:   `+oooooooooooo:    ---------
  /oooo++//ooooo:  ooooo+//+ooooo.    OS: Raspbian GNU/Linux 10 (buster) armv7l
  `+ooooooo:-:oo-  +o+::/ooooooo:     Host: Raspberry Pi 2 Model B Rev 1.1
   `:oooooooo+``    `.oooooooo+-      Kernel: 4.19.97-v7+
     `:++ooo/.        :+ooo+/.`       Uptime: 1 hour, 20 mins
        ...`  `.----.` ``..           Packages: 511 (dpkg)
     .::::-``:::::::::.`-:::-`        Shell: bash 5.0.3
    -:::-`   .:::::::-`  `-:::-       Terminal: runlet_cli
   `::.  `.--.`  `` `.---.``.::`      CPU: BCM2835 (4) @ 900MHz
       .::::::::`  -::::::::` `       Memory: 78MiB / 926MiB
 .::` .:::::::::- `::::::::::``::.
-:::` ::::::::::.  ::::::::::.`:::-   ⬛🟥🟩🟨🟪🟪🟦⬜
::::  -::::::::.   `-::::::::  ::::
-::-   .-:::-.``....``.-::-.   -::-
 .. ``       .::::::::.     `..`..
   -:::-`   -::::::::::`  .:::::`
   :::::::` -::::::::::` :::::::.
   .:::::::  -::::::::. ::::::::
    `-:::::`   ..--.`   ::::::.
      `...`  `...--..`  `...`
            .::::::::::
             `.-::::-`



Exited with code: 0 signal: signal -1
⏰ 6m10.596888351s
```

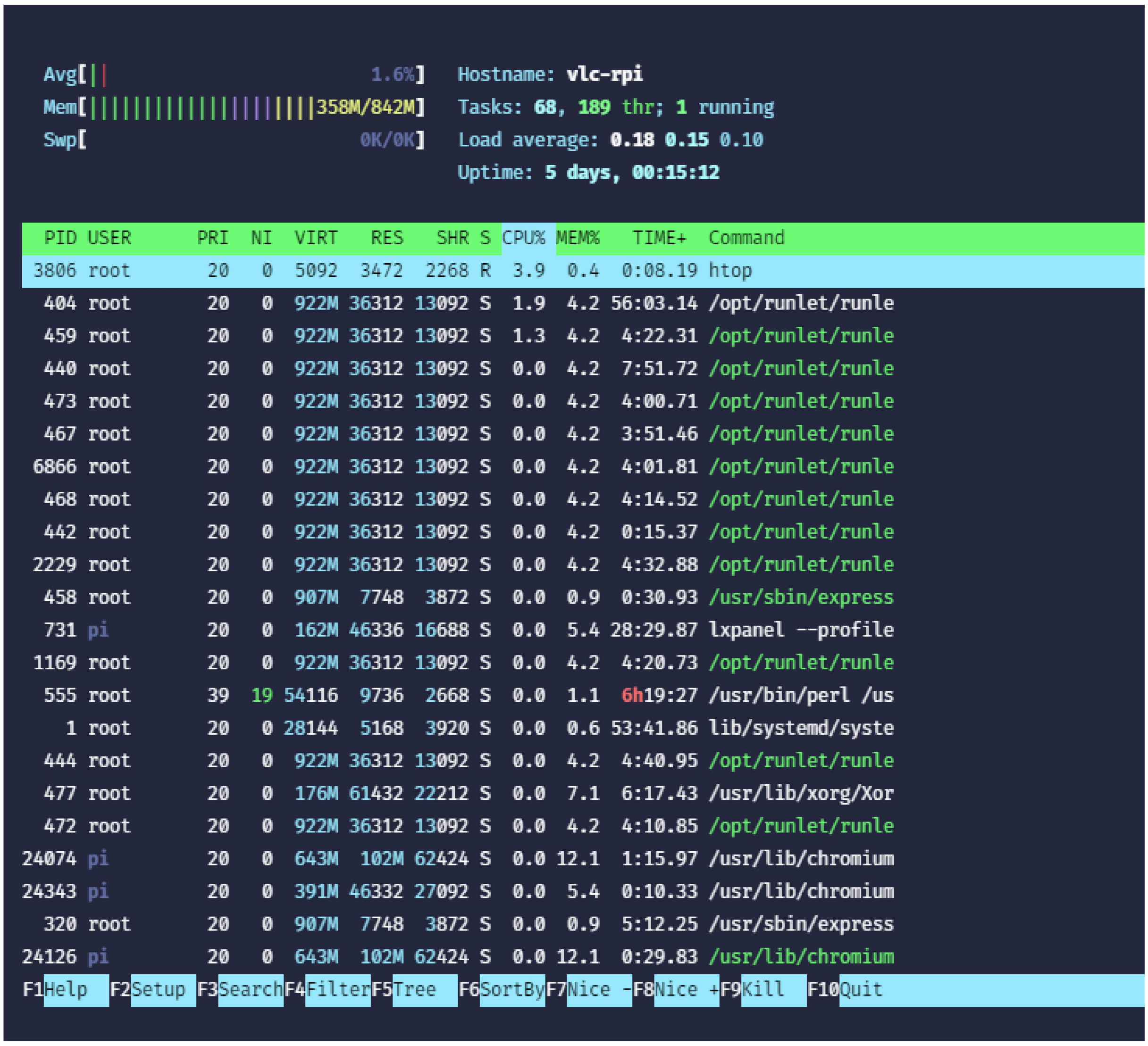
Source: Elaborated by the author

Experiment 3

A job is triggered by a Windows desktop manager to display running processes of a Raspberry PI using htop.

```
rpi-htop:
  script: htop
  description: htop
  showDuration: true
  where: custom
  whereCustom:
  - pi-3
```

Figure 26 – Initial output of htop

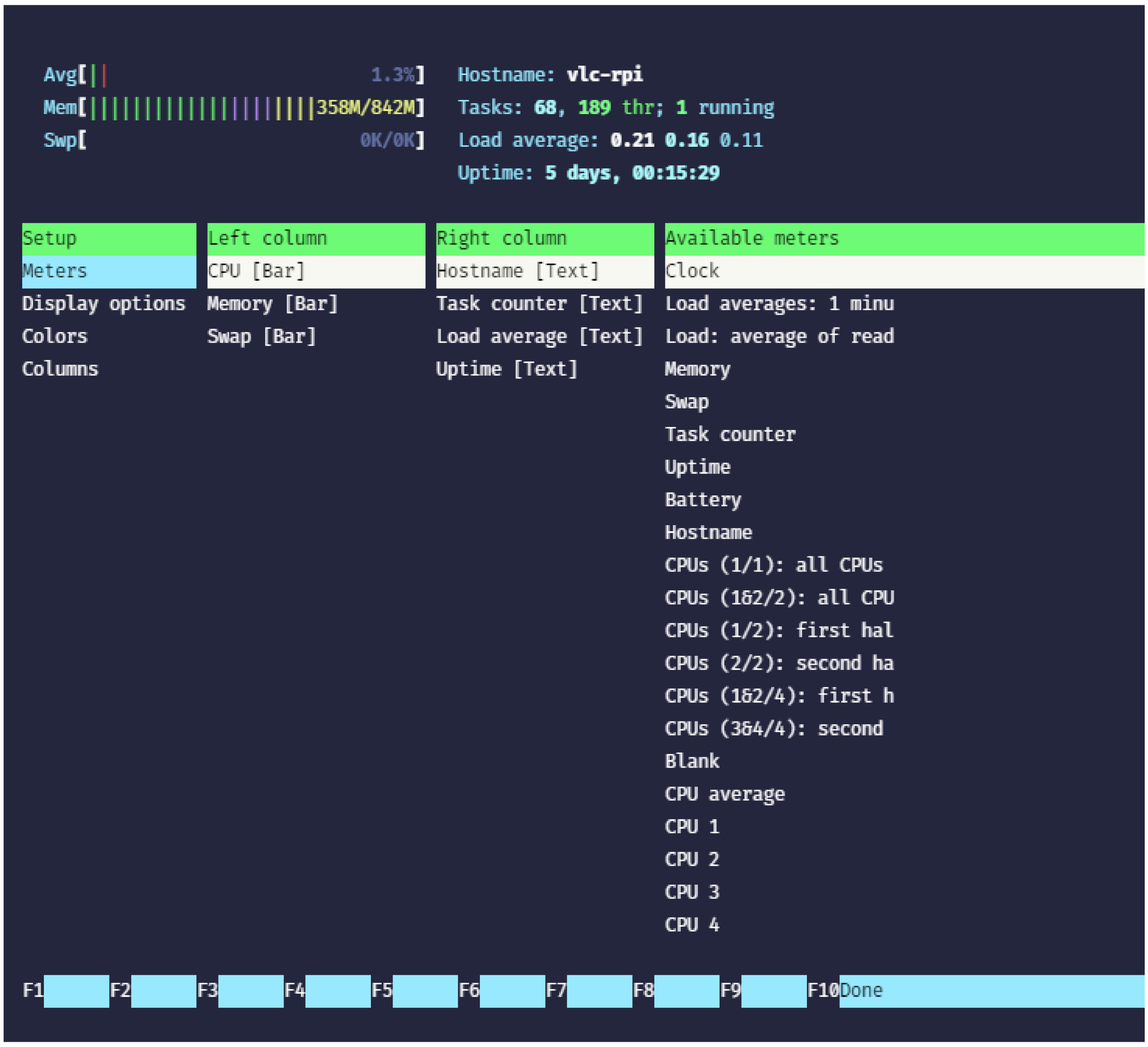


```
  Avg[||                           1.6%]   Hostname: vlc-rpi
  Mem[||||||||||||||||||||||||358M/842M]   Tasks: 68, 189 thr; 1 running
  Swp[                            0K/0K]   Load average: 0.18 0.15 0.10
                                           Uptime: 5 days, 00:15:12

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 3806 root       20   0  5092  3472  2268 R  3.9  0.4  0:08.19 htop
  404 root       20   0  922M 36312 13092 S  1.9  4.2 56:03.14 /opt/runlet/runle
  459 root       20   0  922M 36312 13092 S  1.3  4.2  4:22.31 /opt/runlet/runle
  440 root       20   0  922M 36312 13092 S  0.0  4.2  7:51.72 /opt/runlet/runle
  473 root       20   0  922M 36312 13092 S  0.0  4.2  4:00.71 /opt/runlet/runle
  467 root       20   0  922M 36312 13092 S  0.0  4.2  3:51.46 /opt/runlet/runle
 6866 root       20   0  922M 36312 13092 S  0.0  4.2  4:01.81 /opt/runlet/runle
  468 root       20   0  922M 36312 13092 S  0.0  4.2  4:14.52 /opt/runlet/runle
  442 root       20   0  922M 36312 13092 S  0.0  4.2  0:15.37 /opt/runlet/runle
 2229 root       20   0  922M 36312 13092 S  0.0  4.2  4:32.88 /opt/runlet/runle
  458 root       20   0  907M  7748  3872 S  0.0  0.9  0:30.93 /usr/sbin/express
  731 pi         20   0  162M 46336 16688 S  0.0  5.4 28:29.87 lxpanel --profile
 1169 root       20   0  922M 36312 13092 S  0.0  4.2  4:20.73 /opt/runlet/runle
  555 root       39  19 54116  9736  2668 S  0.0  1.1  6h19:27 /usr/bin/perl /us
    1 root       20   0 28144  5168  3920 S  0.0  0.6 53:41.86 lib/systemd/syste
  444 root       20   0  922M 36312 13092 S  0.0  4.2  4:40.95 /opt/runlet/runle
  477 root       20   0  176M 61432 22212 S  0.0  7.1  6:17.43 /usr/lib/xorg/Xor
  472 root       20   0  922M 36312 13092 S  0.0  4.2  4:10.85 /opt/runlet/runle
24074 pi         20   0  643M  102M 62424 S  0.0 12.1  1:15.97 /usr/lib/chromium
24343 pi         20   0  391M 46332 27092 S  0.0  5.4  0:10.33 /usr/lib/chromium
  320 root       20   0  907M  7748  3872 S  0.0  0.9  5:12.25 /usr/sbin/express
24126 pi         20   0  643M  102M 62424 S  0.0 12.1  0:29.83 /usr/lib/chromium
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Source: Elaborated by the author

Figure 26 shows the initial output of htop, which is a list of system processes including PID, user, percentage of CPU and memory used, virtual memory, and time in execution. The bottom bar has all the options that are available through function keys.

Figure 27 – htop setup results for keyword 'runlet'



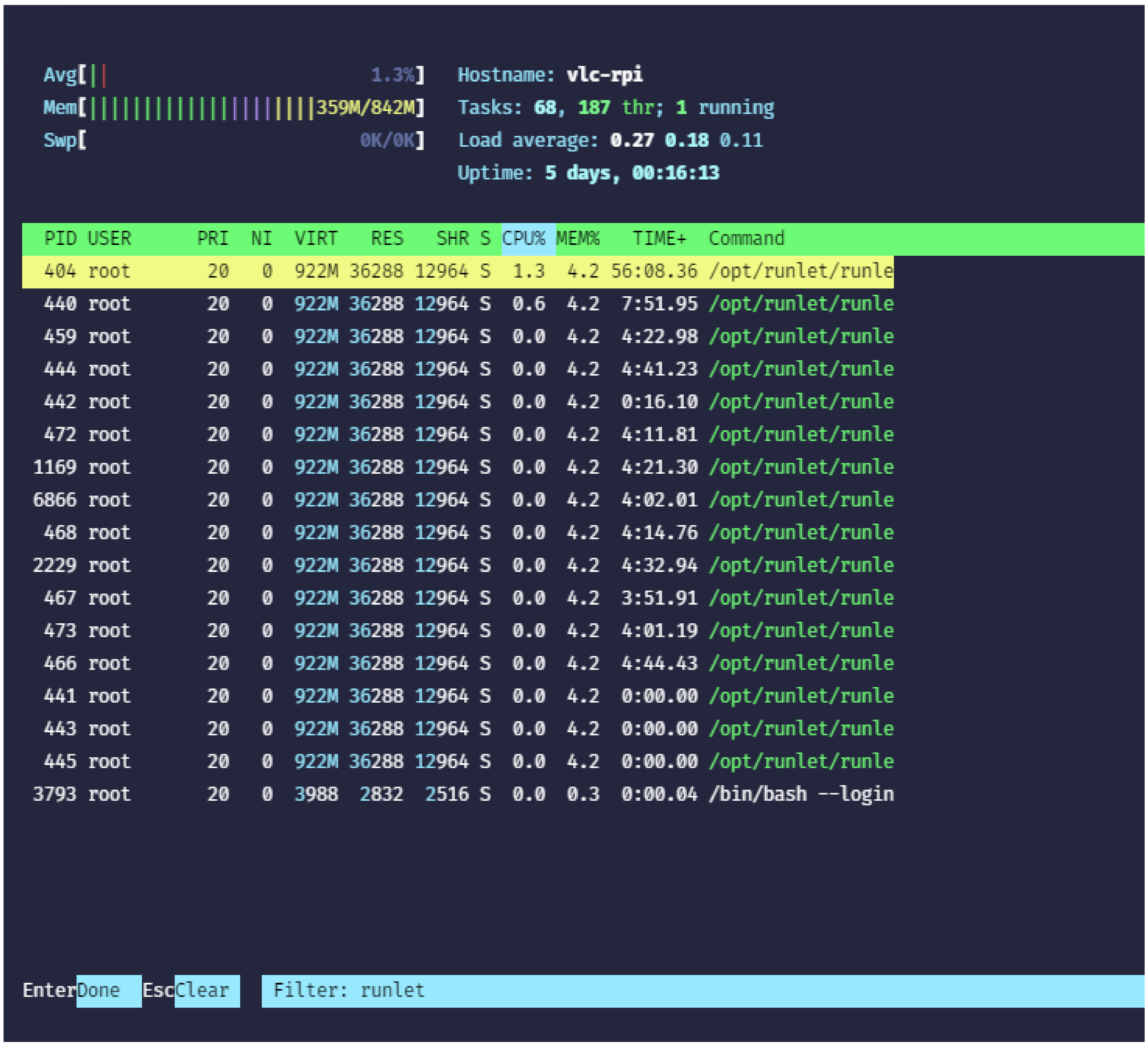
```
  Avg[||                          1.3%]   Hostname: vlc-rpi
  Mem[||||||||||||||||||||||358M/842M]   Tasks: 68, 189 thr; 1 running
  Swp[                          0K/0K]   Load average: 0.21 0.16 0.11
                                          Uptime: 5 days, 00:15:29

Setup            Left column        Right column         Available meters
Meters           CPU [Bar]          Hostname [Text]      Clock
Display options  Memory [Bar]       Task counter [Text]  Load averages: 1 minu
Colors           Swap [Bar]         Load average [Text]  Load: average of read
Columns                             Uptime [Text]        Memory
                                                         Swap
                                                         Task counter
                                                         Uptime
                                                         Battery
                                                         Hostname
                                                         CPUs (1/1): all CPUs
                                                         CPUs (1&2/2): all CPU
                                                         CPUs (1/2): first hal
                                                         CPUs (2/2): second ha
                                                         CPUs (1&2/4): first h
                                                         CPUs (3&4/4): second
                                                         Blank
                                                         CPU average
                                                         CPU 1
                                                         CPU 2
                                                         CPU 3
                                                         CPU 4

F1      F2      F3      F4      F5      F6      F7      F8      F9      F10Done
```

Source: Elaborated by the author

Figure 27 shows all the options available under setup when the key `F2` is pressed. These options customize meters shown at the top, as well as change display options, colors, and active columns. Navigation using arrow keys is done with no hassle as Runlet is able to capture all keyboard strokes. The key `F10` may be pressed at any time to indicate that the setup is done and return to the previous screen.

Figure 28 shows the output of key `F4` in the main screen. This option allows the user to filter processes, which is particularly useful in remote monitoring of running processes as the tool is capable of killing processes.

Figure 28 – htop search option



```
  Avg[||                      1.3%]   Hostname: vlc-rpi
  Mem[||||||||||||||||||||||359M/842M]   Tasks: 68, 187 thr; 1 running
  Swp[                        0K/0K]   Load average: 0.27 0.18 0.11
                                       Uptime: 5 days, 00:16:13

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
  404 root       20   0  922M 36288 12964 S  1.3  4.2 56:08.36 /opt/runlet/runle
  440 root       20   0  922M 36288 12964 S  0.6  4.2  7:51.95 /opt/runlet/runle
  459 root       20   0  922M 36288 12964 S  0.0  4.2  4:22.98 /opt/runlet/runle
  444 root       20   0  922M 36288 12964 S  0.0  4.2  4:41.23 /opt/runlet/runle
  442 root       20   0  922M 36288 12964 S  0.0  4.2  0:16.10 /opt/runlet/runle
  472 root       20   0  922M 36288 12964 S  0.0  4.2  4:11.81 /opt/runlet/runle
 1169 root       20   0  922M 36288 12964 S  0.0  4.2  4:21.30 /opt/runlet/runle
 6866 root       20   0  922M 36288 12964 S  0.0  4.2  4:02.01 /opt/runlet/runle
  468 root       20   0  922M 36288 12964 S  0.0  4.2  4:14.76 /opt/runlet/runle
 2229 root       20   0  922M 36288 12964 S  0.0  4.2  4:32.94 /opt/runlet/runle
  467 root       20   0  922M 36288 12964 S  0.0  4.2  3:51.91 /opt/runlet/runle
  473 root       20   0  922M 36288 12964 S  0.0  4.2  4:01.19 /opt/runlet/runle
  466 root       20   0  922M 36288 12964 S  0.0  4.2  4:44.43 /opt/runlet/runle
  441 root       20   0  922M 36288 12964 S  0.0  4.2  0:00.00 /opt/runlet/runle
  443 root       20   0  922M 36288 12964 S  0.0  4.2  0:00.00 /opt/runlet/runle
  445 root       20   0  922M 36288 12964 S  0.0  4.2  0:00.00 /opt/runlet/runle
 3793 root       20   0  3988  2832  2516 S  0.0  0.3  0:00.04 /bin/bash --login




EnterDone  EscClear   Filter: runlet
```

Source: Elaborated by the author

4.3.2 Reliability

Reliability is investigated by observing the broker's behavior after two failure events. The impact is observed through a Grafana dashboard that monitors the following metrics using Prometheus:

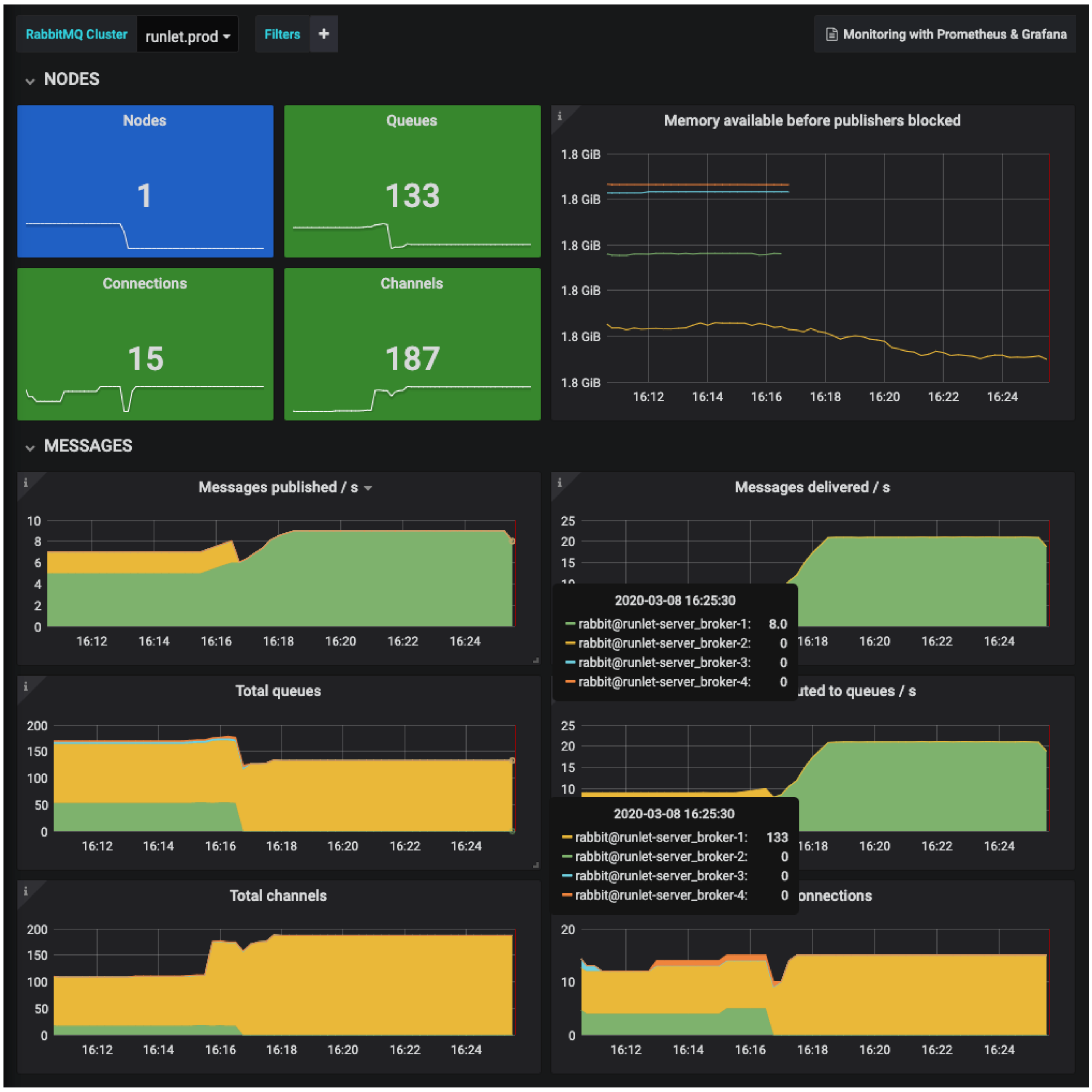
- ❑ Total number of nodes, queues, connections, and channels.
- ❑ Per-node memory available before publishers blocked.
- ❑ Messages published, delivered, and routed to queues in a per-node and per-second g ranularity.
- ❑ Total number of queues, channels, and connections per node per second.

Node Failure

The first experiment investigates how the broker reacts to node failure by changing the cluster composition from four (4) nodes to a single node at about 16:15. Figure 29 shows that all messages, queues, channels, and connections are kept in the remaining node after the removal of three (3) nodes, which means that no messages are lost.

Different colors may be used by Grafana to indicate the same node across graph panels. For example, green is designated for node one (1) on messages published, delivered, and routed to queues, while yellow is designated to the same node for total queues, channels, and connections.

Figure 29 – Cluster after node failure



RabbitMQ Cluster
runlet.prod
Filters
Monitoring with Prometheus & Grafana
NODES
Nodes
1
Queues
133
Connections
15
Channels
187
Memory available before publishers blocked
1.8 GiB
MESSAGES
Messages published / s
Messages delivered / s
Total queues
uted to queues / s
Total channels
onnections
16:12
16:14
16:16
16:18
16:20
16:22
16:24
2020-03-08 16:25:30
rabbit@runlet-server_broker-1: 8.0
rabbit@runlet-server_broker-2: 0
rabbit@runlet-server_broker-3: 0
rabbit@runlet-server_broker-4: 0
2020-03-08 16:25:30
rabbit@runlet-server_broker-1: 133
rabbit@runlet-server_broker-2: 0
rabbit@runlet-server_broker-3: 0
rabbit@runlet-server_broker-4: 0

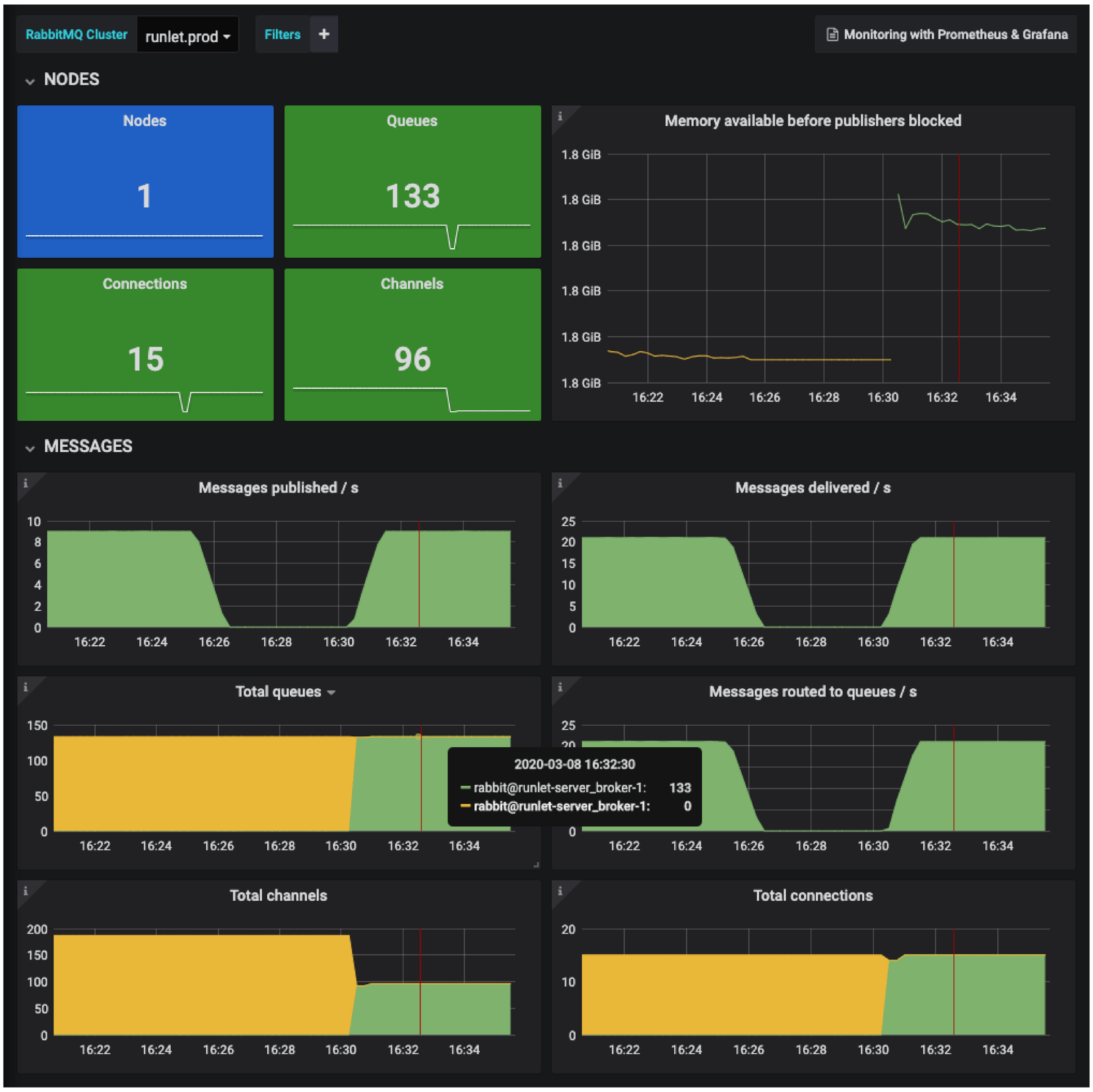
Source: Elaborated by the author

A slight increase in memory consumption happens on the remaining node as it starts to handle all the load. The difference between the number of messages published and routed to queues for delivery may be attributed to messages routed to multiple queues. The number of queues, channels, and connections varies according to the workload of messages published and delivered.

Server Failure

The second experiment investigates how the broker reacts to a server failure by removing the only active node at about 16:25.

Figure 30 – Cluster after server failure



```
2020-03-08 16:32:30
rabbit@runlet-server_broker-1:   133
rabbit@runlet-server_broker-1:     0
```

Source: Elaborated by the author

Figure 30 shows that messages stop being transmitted for about 5 minutes during the failure, but the process resumes as soon as the node joins the cluster again to take over. The node establishes all the old connections and recovers persisted messages and queues, which also results in no lost messages. The number of channels drops significantly as previous connections and channels are closed. Grafana changes the color of the node from yellow to green to differentiate instances of the same node.

4.3.3 Analysis with Related Work

This subsection compares Runlet to systems presented in Section 2.3 regarding protocols, reliable message delivery, prototype, and interactive execution. Table 6 presents the comparison.

Table 6 – Academic Systems

Study	AMQP	RabbitMQ	Reliable Message Delivery	Prototype	Interactive Job Execution
(LIANG; CHEN, 2018)	X	X	X	-	-
(KOSTROMINA; SIEMENS; YURII, 2018)	X	X	X	-	-
(KRISHNA; SASIKALA, 2019)	X	X	X	X	-
Runlet	X	X	X	X	X

Interactive job execution is not applicable in none of the studies apart from Runlet. Besides that, this is the only study that presents an experimental evaluation that assesses interactivity and reliability.

Chapter 5

Conclusions

This study introduces a tool that achieves interactive job execution across heterogeneous devices using the protocol AMQP and the message broker RabbitMQ. A few potential use areas include include Continuous Integration (CI), Continuous Delivery (CD), and remote monitoring and management of devices.

The literature review performed in chapter 2 provides a clear understanding of commonalities and differences among communication protocols and message brokers commonly used in the context of IoT. The comparison reassures that the protocol AMQP and the message broker RabbitMQ are the right choices for Runlet due to their unique messaging capabilities.

AMQP was selected due to its flexible message routing, reliable message queues, and multiple exchange types. RabbitMQ was selected based on a personal preference for the built-in web management tool and support for Prometheus and Grafana, which are considered advantages over Apache Artemis. The complete reasoning behind both choices is presented in section 3.1.

The architecture and the motivation behind architectural decisions are discussed in detail in section 3.2, including a conceptual view that introduces all the components and an implementation view with entities replaced by selected technologies. The reasoning behind the set of open-source components selected for dealing with interactive execution is also presented throughout chapter 3.

The experimental evaluation is conducted in chapter 4 and proves the hypothesis that interactive job execution on heterogeneous devices is achieved using AMQP under a variety of scenarios and that message delivery is reliable even after node and server failures. Thus, it can be concluded that both the primary and secondary study objectives defined in chapter 1 were achieved.

5.1 Main Contributions

The study contributes with an approach to interactive job execution across heterogeneous devices using the protocol AMQP and the message broker RabbitMQ. This combination showed to be very useful in the experimental evaluation as the broker could recover messages and queues after node and server failure with no messages lost.

5.2 Technical Contributions

The application has been released on GitHub and is easily found over the internet. The website with more information about the app, documentation, and download options is available at https://runlet.app. Runlet has been downloaded over 3.000 times at the time of writing, considering all package options and release versions.

The application was registered on INPI (*Instituto Nacional da Propriedade Intelectual*) under process BR 51 2019 001725-0, and published in the RPI (*Revista da Propriedade Intelectual*) 2537, page 7 on August 20, 2019.

5.3 Future Work

The roadmap for future work includes an experimental evaluation to benchmark scalability by measuring message throughput and latency as the network grows. Also, the analysis of the security measures put in place to ensure data protection from cyberattacks.

Bibliography

ACTIVEMQ, A. **Apache ActiveMQ**. 2020. <https://activemq.apache.org>. [Online; accessed 15-March-2020].

AMAZON. **AWS IoT Device Management**. 2020. <https://aws.amazon.com/iot-device-management>. [Online; accessed 15-March-2020].

AMQP. **Advanced Message Queuing Protocol**. 2020. <https://www.amqp.org>. [Online; accessed 15-March-2020].

AZURE, M. **Azure IoT Hub**. 2020. <https://azure.microsoft.com/en-us/services/iot-hub>. [Online; accessed 15-March-2020].

_____. **Azure Pipelines**. 2020. <https://azure.microsoft.com/en-us/services/devops/pipelines>. [Online; accessed 15-March-2020].

BLUEPRINT. **Blueprint**. 2020. <https://blueprintjs.com>. [Online; accessed 15-March-2020].

BORMANN, C. et al. **CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets**. [S.l.], 2018. Disponível em: <https://tools.ietf.org/html/rfc8323>.

BUO. **homebrew-cask-upgrade**. 2020. <https://github.com/buo/homebrew-cask-upgrade>. [Online; accessed 15-March-2020].

CLUSTERLABS. **Pacemaker**. 2020. <https://clusterlabs.org/pacemaker>. [Online; accessed 15-March-2020].

COAP. **CoAP**. 2020. <https://coap.technology>. [Online; accessed 15-March-2020].

DHAS, Y. J.; JEYANTHI, P. A review on internet of things protocol and service oriented middleware. In: IEEE. **2019 International Conference on Communication and Signal Processing (ICCSP)**. 2019. p. 0104–0108. Disponível em: <https://doi.org/10.1109/ICCSP.2019.8698088>.

DIGITALOCEAN. **DigitalOcean**. 2020. <https://www.digitalocean.com>. [Online; accessed 15-March-2020].

_____. **Droplets**. 2020. <https://www.digitalocean.com/products/droplets>. [Online; accessed 15-March-2020].

DOBBELAERE, P.; ESMAILI, K. S. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In: **Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems**. New York, NY, USA: Association for Computing Machinery, 2017. (DEBS '17), p. 227–238. ISBN 9781450350655. Disponível em: <https://doi.org/10.1145/3093742.3093908>.

DYLANARAPS. **neofetch**. 2020. <https://github.com/dylanaraps/neofetch>. [Online; accessed 15-March-2020].

ELECTRON. **Electron**. 2020. <https://electronjs.org>. [Online; accessed 15-March-2020].

ELKHODR, M.; SHAHRESTANI, S.; CHEUNG, H. The internet of things: new interoperability, management and security challenges. **arXiv preprint arXiv:1604.04824**, 2016. Disponível em: <https://doi.org/10.5121/ijnsa.2016.8206>.

GERACI, A. et al. **IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries**. IEEE Press, 1991. Disponível em: <https://dl.acm.org/doi/book/10.5555/574566>.

GOOGLE. **Cloud IoT Core**. 2020. <https://cloud.google.com/iot-core>. [Online; accessed 15-March-2020].

GRAFANA. **Grafana**. 2020. <https://grafana.com/grafana>. [Online; accessed 15-March-2020].

HAPP, D. et al. Meeting iot platform requirements with open pub/sub solutions. **Annals of Telecommunications**, Springer, v. 72, n. 1-2, p. 41–52, 2017. Disponível em: <https://doi.org/10.1007/s12243-016-0537-4>.

HAT, R. **Hawtio**. 2020. <https://hawt.io/>. [Online; accessed 15-March-2020].

HOMEBREW. **homebrew-cask**. 2020. <https://github.com/Homebrew/homebrew-cask>. [Online; accessed 15-March-2020].

ISO. **ISO/IEC 19464:2014**. 2020. <https://www.iso.org/standard/64955.html>. [Online; accessed 15-March-2020].

JEST. **Jest**. 2020. <https://jestjs.io>. [Online; accessed 15-March-2020].

KARAGIANNIS, V. et al. A survey on application layer protocols for the internet of things. **Transaction on IoT and Cloud computing**, v. 3, n. 1, p. 11–17, 2015. Disponível em: <https://www.researchgate.net/publication/303192188_A_survey_on_application_layer_protocols_for_the_Internet_of_Things>.

KOSTER, M.; KERäNEN, A.; JIMENEZ, J. **Publish-Subscribe Broker for the Constrained Application Protocol (CoAP)**. [S.l.], 2019. Work in Progress. Disponível em: <https://datatracker.ietf.org/doc/html/draft-ietf-core-coap-pubsub-09>.

KOSTROMINA, A.; SIEMENS, E.; YURII, B. A concept for a high-reliability meteorological monitoring system using amqp. In: BIBLIOTHEK, HOCHSCHULE ANHALT. **Titel: Proceedings of the 6th International Conference on Applied Innovations in IT**. 2018. Disponível em: <https://doi.org/10.25673/5590>.

KRISHNA, C. S.; SASIKALA, T. Healthcare monitoring system based on iot using amqp protocol. In: SMYS, S. et al. (Ed.). **International Conference on Computer Networks and Communication Technologies**. Singapore: Springer Singapore, 2019. p. 305–319. ISBN 978-981-10-8681-6. Disponível em: <https://doi.org/10.1007/978-981-10-8681-6_29>.

LIANG, Y.; CHEN, Z. Intelligent and real-time data acquisition for medical monitoring in smart campus. **IEEE Access**, IEEE, v. 6, p. 74836–74846, 2018. Disponível em: <https://doi.org/10.1109/ACCESS.2018.2883106>.

LUZURIAGA, J. E. et al. A comparative evaluation of amqp and mqtt protocols over unstable and mobile networks. In: IEEE. **2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)**. 2015. p. 931–936. Disponível em: <https://doi.org/10.1109/CCNC.2015.7158101>.

MAGNONI, L. Modern messaging for distributed systems. **Journal of Physics: Conference Series**, IOP Publishing, v. 608, p. 012038, may 2015. Disponível em: <https://doi.org/10.1088/1742-6596/608/1/012038>.

MAJ, W. **React Lifecycle Methods Diagram**. 2020. <http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram>. [Online; accessed 15-March-2020].

MINIO. **MinIO**. 2020. <https://min.io>. [Online; accessed 15-March-2020].

MOBX. **MobX**. 2020. <https://mobx.js.org>. [Online; accessed 15-March-2020].

MQTT. **MQTT**. 2020. <http://mqtt.org>. [Online; accessed 15-March-2020].

MUHAMMAD, H. **htop**. 2020. <https://hisham.hm/htop>. [Online; accessed 15-March-2020].

NAIK, N. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. In: IEEE. **2017 IEEE international systems engineering symposium (ISSE)**. 2017. p. 1–7. Disponível em: <https://doi.org/10.1109/SysEng.2017.8088251>.

NĂSTASE, L.; SANDU, I. E.; POPESCU, N. An experimental evaluation of application layer protocols for the internet of things. **Studies in Informatics and Control**, v. 26, n. 4, p. 403–412, 2017. Disponível em: <https://doi.org/10.24846/v26i4y201704>.

NESTJS. **NestJS**. 2020. <https://nestjs.com>. [Online; accessed 15-March-2020].

OASIS. **MQTT Version 5.0**. 2020. <https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html>. [Online; accessed 15-March-2020].

POSTGRESQL. **PostgreSQL**. 2020. <https://www.postgresql.org>. [Online; accessed 15-March-2020].

PRESTON-WERNER, T. **Semantic Versioning**. 2020. <https://semver.org>. [Online; accessed 15-March-2020].

PROMETHEUS. **Prometheus**. 2020. <https://prometheus.io>. [Online; accessed 15-March-2020].

QPID, A. **Apache Qpid**. 2020. <https://qpid.apache.org>. [Online; accessed 15-March-2020].

RABBITMQ. **RabbitMQ**. 2020. <https://www.rabbitmq.com>. [Online; accessed 15-March-2020].

REACT. **React**. 2020. <https://reactjs.org>. [Online; accessed 15-March-2020].

ROSTANSKI, M.; GROCHLA, K.; SEMAN, A. Evaluation of highly available and fault-tolerant middleware clustered architectures using rabbitmq. In: IEEE. **2014 federated conference on computer science and information systems**. 2014. p. 879–884. Disponível em: <https://doi.org/10.15439/2014F48>.

SHELBY, Z. et al. **The constrained application protocol (CoAP)**. [S.l.], 2014. Disponível em: <https://tools.ietf.org/html/rfc7252>.

THINGSBOARD. **ThingsBoard**. 2020. <https://thingsboard.io>. [Online; accessed 15-March-2020].

TYPESCRIPT. **TypeScript**. 2020. <https://www.typescriptlang.org>. [Online; accessed 15-March-2020].

XMPP. **XMPP**. 2020. <https://xmpp.org>. [Online; accessed 15-March-2020].

XTERM.JS. **Xterm.js**. 2020. <https://xtermjs.org>. [Online; accessed 15-March-2020].

ZAVALISHIN, D. **MQTT/UDP**. 2020. <https://github.com/dzavalishin/mqtt_udp>. [Online; accessed 15-March-2020].