SEVENTH STATE

Observability in RabbitMQ

YOUR GUIDE TO EFFECTIVE MONITORING FOR ULTIMATE PERFORMANCE

WHITEPAPER

Contents

Abstract

This white paper addresses the importance of monitoring and visibility in managing RabbitMQ, a critical component in inter-service communications. Disruptions in RabbitMQ can severely impact system performance, availability, and resilience. We discuss how RabbitMQ's metrics and logs provide insights into message volumes and resource usage, essential for detecting issues and preventing them from escalating. The goal is to equip operational teams with the tools and methodologies necessary for proactive system management and enhanced operational visibility.

A WHITETPAPER BY SEVENTH STATE
THE RABBITMQ EXPERTS

GABOR OLAH
RABBITMQ CONSULTANT

LAJOS GERECS
RABBITMQ CONSULTANT

Introduction

Visibility in the operation of any computer system is important because it enables us to assess if it works well or if there are problems where those problems are. RabbitMQ is no different in this regard, especially that it is a crucial component enabling services to communicate with one another. If anything goes wrong with RabbitMQ then multiple services will be affected and this can have a disproportionate impact on the overall system availability, performance and resiliency.

To avoid problems from escalating we have to understand how the system is currently functioning. RabbitMQ exposes numerous metrics to show the volume of messages as well as the resource usage. It is important to note that the metrics themselves are hardly useful in isolation and understanding the limitations of RabbitMQ is also very important. Detecting problems before they turn into serious incidents takes a lot of pressure from the operations team, and enables capacity planning to expand (or reduce) capacity in an organised manner.

Although the metrics can indicate if the service is not performing well, it is difficult to pinpoint the root cause. RabbitMQ also provides detailed operational logs to reveal what is happening inside the system. This is most useful when we try to identify the cause, and reveal what happened before, during and after the incident.
Based on the metrics and the logs, it is possible to monitor the real time performance of RabbitMQ, but unless somebody is actively watching the numbers, problems can still silently happen. By setting up appropriate alerts and identifying the appropriate severity for the events, the operation team can rely on an external system to notify them in due time to intervene and fix the issue.

The structure of this paper

RabbitMQ is a mature product with a long history of features. Before we delve into the recommended monitoring solution, we must discuss the ways RabbitMQ provided for monitoring purposes. These old techniques are not efficient enough for today's standards, but are still widely used. RabbitMQ 4.0 will deprecate some of the built-in monitoring in favour of a much more efficient and capable open-source solution. We will discuss this in this chapter.

The new monitoring solutions available today are much more capable than the built-in ones. RabbitMQ chose Prometheus and Grafana as the supported monitoring technologies. In this chapter we will discuss how this architecture works, what is available in the open-source community to streamline an efficient monitoring system for RabbitMQ. The Grafana ecosystem also provides ways to integrate the monitoring solution with alert management, which we will also cover.

Operational logging can help identify the cause of issues, but unlike numeric values that can be easily visualised in graphs, log messages are textual and require different "visualisation" solutions. Most monitoring stacks have a logging component, so we will not discuss the technical solutions, but instead this chapter will focus on the most important log messages RabbitMQ produces and what actions can be taken if they appear.

We will cover how to detect problems with RabbitMQ clients, what are the symptoms of a system overload and network partitions in the form of case studies in this chapter.

SEVENTH STATE

CHAPTER ONE

The "Old" Ways

With RabbitMQ's long history comes a plethora of solutions to problems. Observability and RabbitMQ monitoring evolved a lot over the years, starting from dedicated monitoring nodes to built-in monitoring features to the management UI to various plugins to expose metrics to third-party systems.

In this chapter we will discuss what are the most important features RabbitMQ exposes over the management UI, how those features are utilised by third-party solutions and explore the reasons why RabbitMQ 4.0 will remove the monitoring part of the management user interface.

The three types of management activities

Before we explore the nitty gritty details of how the management interface works, let's take a look at what kind of functionality it provides. The management interface serves three major roles:

1. It enables administrators to manage users, the built-in access rights, queue and exchange policies and other day-to-day operational features.
2. RabbitMQ users can manage their AMQP topology including declaring or deleting exchanges, queues.
3. It provides performance monitoring to enable administrators (and automated tools) to check the actual performance of the RabbitMQ cluster.

The management UI is a web application over an HTTP(S) API. To be able to use it, users must be configured in RabbitMQ with various access rights. Although our focus is on the monitoring aspects of RabbitMQ, it is important to understand the other priorities that the same service needs to serve too.

Operational Features

The administrative features enable the administrator to manage some aspects of the RabbitMQ cluster. In no particular order these features are the following:

User and access right management: RabbitMQ provides built-in user support as well as integrating with Active Directory (LDAP) or OAuth-based providers.

Limits: some cluster-wide or virtual host related limits can be configured via the management UI. These are the settings that do not require restarting nodes or the whole cluster.

Policies: the AMQP protocol provides very limited capability to changing queue and exchange properties. RabbitMQ enables changing most of the additional (sometimes RabbitMQ exclusive) features to be configured by setting policies.

Feature flags: feature flags enable RabbitMQ to coordinate how newly introduced features are rolled out during and after upgrading. The management UI provides a convenient way to make this easier to manage and monitor which flags are enabled.[1]

Plugin features: some RabbitMQ plugins (e.g. shovel and federation) expose their operational features on the management UI too.

Topology export/import: RabbitMQ enables downloading all runtime and topology configuration (not the messages) via the management UI. This accomplishes most of what is needed for a good back-up solution.

[1] It is recommended and sometimes required to enable all non-experimental feature flags before and after upgrading.

Topology Management

Topology is the phrase we use to describe the collection of queues, exchanges, bindings and a new addition to the RabbitMQ family, streams. The Advanced Message Queuing Protocol (AMQP) is the primary means to interact with RabbitMQ. Client software implementing the AMQP version 0.9.1 protocol[2] can declare (i.e. create) or delete topology entities.

The management interface provides ways for users with adequate AMQP rights to declare, delete or interact with the topology entities. What it doesn't allow is cheating, because it uses an internal AMQP channel and the AMQP commands. Any operation is executed according to the semantics of the AMQP protocol.

Although this seems auxiliary to the observability functionality of the management interface, it enables the operator to make changes to the topology while being able to observe the effect. E.g. if a queue accumulates too many messages due to a client error, then they can decide to empty the queue (i.e. purge it as AMQP calls it).

Monitoring

For a long time the management interface was the way to see the traffic flow or to check the health of the cluster without running commands directly on the RabbitMQ servers. It provided both an API and a friendly UI to interact with RabbitMQ. Newer RabbitMQ versions provide a Prometheus based monitoring solution which will be discussed in the next chapter.

It shows both traffic related information as well as detailed resource usage metrics that can be used to determine if the RabbitMQ system is performing well.

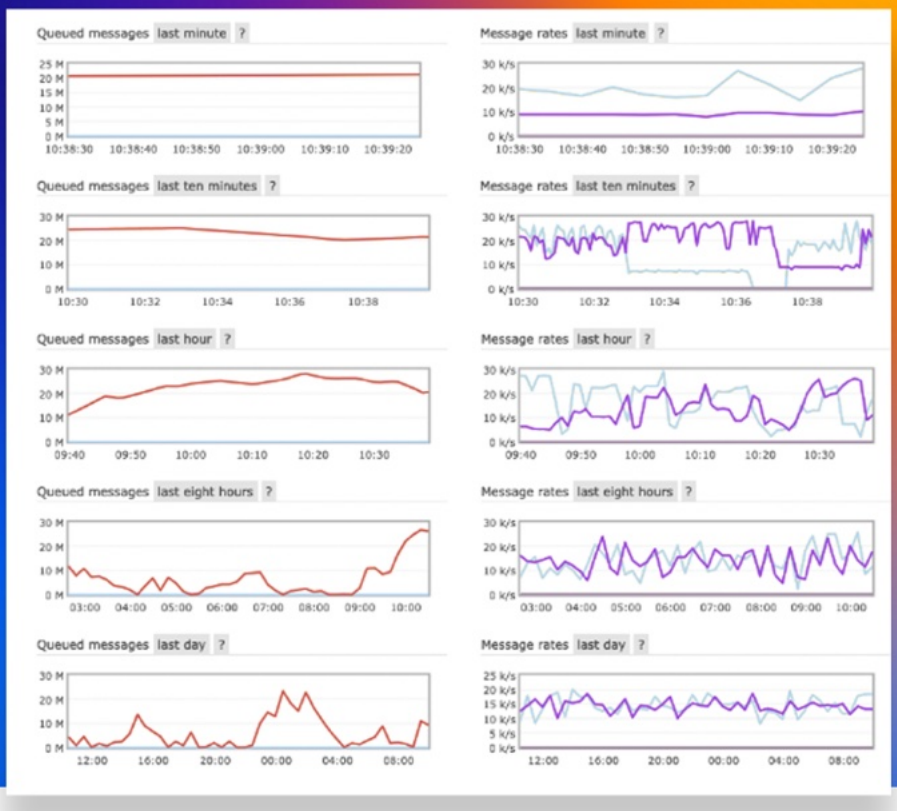
[2] N.B. Even though "AMQP" includes the term "protocol" in its acronym, it's typical to still mention "protocol" when referring specifically to the AMQP protocol.

Traffic Related Metrics

The UI shows the current throughput rates and message statistics as well as historical values for predetermined periods. Instead of going into the details of the individual metrics in this chapter, we will explain them when discussing the Prometheus based solution. The metrics will be removed from the management interface in RabbitMQ 4.0 according to the announced depreciation strategy.

RabbitMQ calculates the rates and counter values in intervals and stores them in temporary tables internally as we'll see in the next section. This interval by default is 5 seconds, so the rates show the average per second value over the last 5 second period. This can hide very short bursts of traffic or show impossible fractional rates. But as long as the statistics collection interval is short enough it gives a very accurate view of the current traffic numbers.

The historical values are given in a graph form for the last minute, 10 minutes, 1 hour or 8 hours, and a day respectively. These predetermined historical charts are useful to see the current values in context while also enabling RabbitMQ to calculate and store the values efficiently. Keep in mind that **RabbitMQ's main purpose is not to provide a world-class monitoring solution, but to efficiently move messages between publishers and consumers.**



System metrics and information

The management UI enables operators to check some of the system metrics and some useful system information too. Although RabbitMQ is a self-contained system (i.e. it doesn't depend on any other services to function) it does run on the operating system and writes data to the network and disk.

RabbitMQ runs in the BEAM Erlang Virtual machine. This system provides soft real time semantics which enables RabbitMQ to serve lots of client connections and queues concurrently while maintaining fair progress even during overload. For each RabbitMQ node the management UI provides some insights into the most important metrics of the VM, e.g. process counts, memory usage and scheduler run queue. These metrics are gathered at regular intervals, but they don't average out for the statistics collection interval.

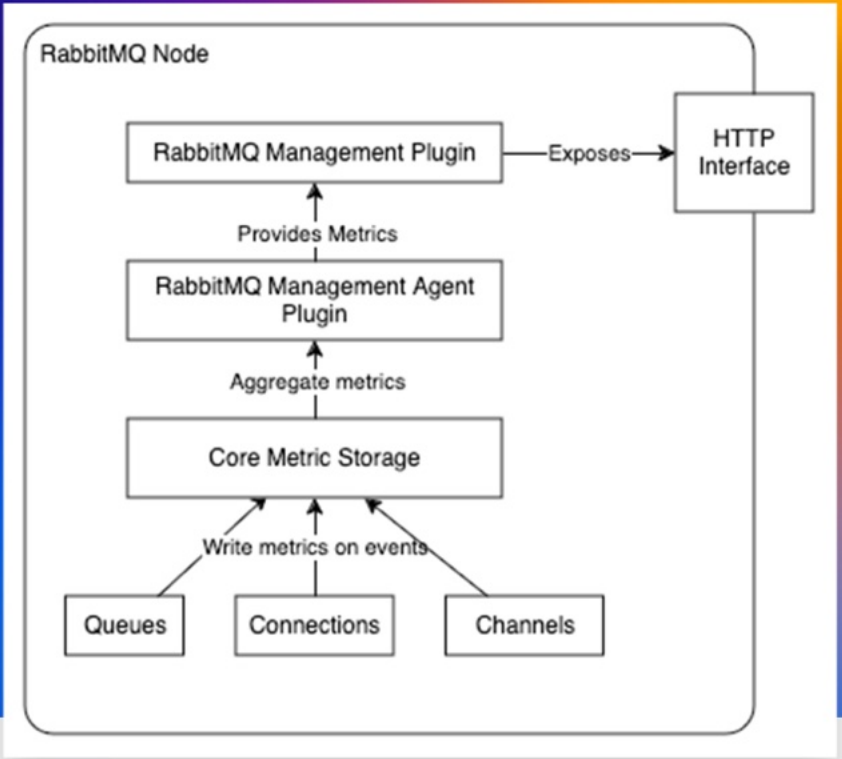
The disk related metrics enable some insights into the health of the disk subsystem. As part of the resiliency guarantees RabbitMQ provides, it writes the messages on disk. This means that both the disk throughput and the disk latency affects the traffic flow. These node level metrics can help determine if one or more nodes in a cluster are contributing to a slowdown or an outage.

For RabbitMQ version 3.x, RabbitMQ also reports the number of persisted messages to disk. In RabbitMQ version 3.8, a replacement for the deprecated classic queue mirroring was introduced. Quorum queues persist messages differently to classic queues and they no longer provide the persistence related metrics. But there are separate metrics for the Raft-based subsystem to see progress exposed as part over the Prometheus end-point.

Architecture Overview

RabbitMQ can be managed by a built-in plugin called the Management User Interface. This is backed by another plugin that enables an HTTP endpoint on all installed RabbitMQ nodes in the cluster. The HTTP endpoint can be independently accessed from the User Interface, and the data is presented in a JSON format.

The HTTP Interface is exposed by the Management Plugin, which collects the information from the Management Agent. Management Agent uses the Core Metrics functionality of RabbitMQ to store historical metrics. Core metrics are collected at intervals and on certain events, ie. connection closed.



Rabbit Core Metrics

RabbitMQ Core hosts the database tables related to metrics of connections, channels, queues, exchanges, and other internal entities. Metrics are updated directly in these tables, for example, when a connection is created, an entry is inserted into the "connection_created" table, or when a message is published a counter is increased in the "exchange_stats" table.

Periodically, by default 2 minutes, there are some procedures in RabbitMQ which clean these tables up by removing entries for entities which no longer exist, i.e. when a connection is closed its entry is removed from the tables.

Management Agent Plugin

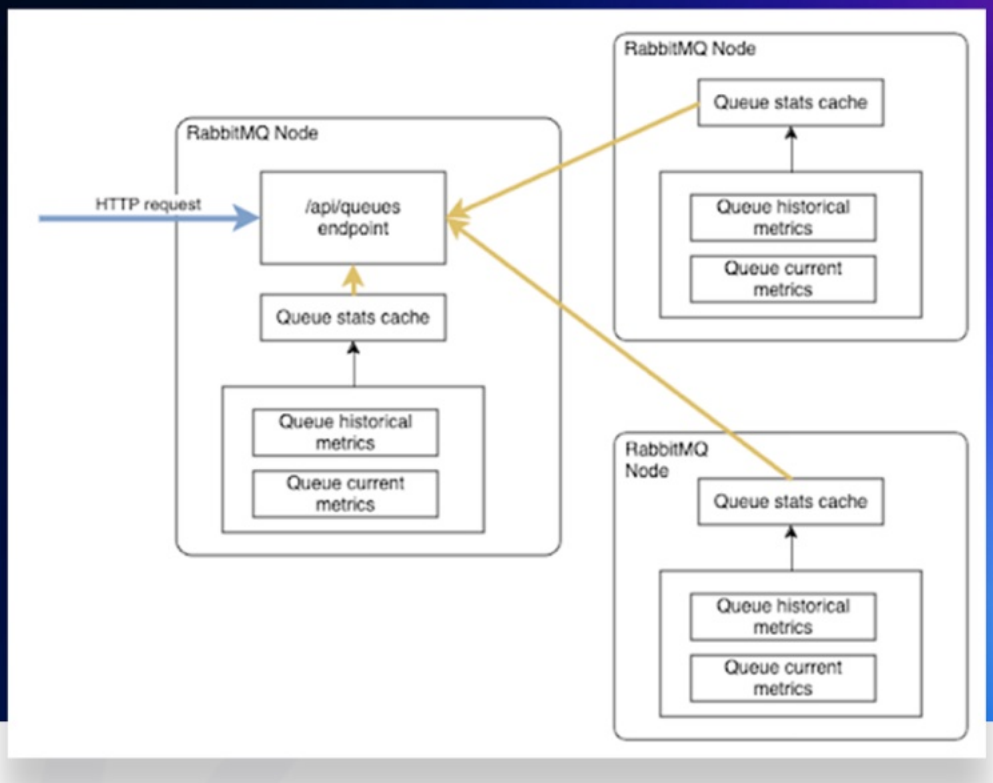
Core Metric Storage does not store historical data, it is only concerned with counters and events. To be able to display the charts on the GUI when the interface is opened, historical metric storage is needed.

The Management Agent plays this role of aggregating the counters into historical data points and storing them in in-memory tables. These aggregated metrics will be stored in yet another set of tables and will be garbage collected periodically.

The Management Agent also subscribes to internal events, such as `connection_closed` or `queue_deleted` to be able to clean up stale metrics from the tables.

The Management Agent Plugin implements the functionality required to query all node local metric data.

Management Plugin



Most of the API endpoints are implemented similarly. When a request is made, information is collected from node local tables and then served as JSON.

The Management Plugin is responsible for setting up the web server to serve the API endpoints and the UI.

The plugin contains metric caches as well, so if repeated requests are made to the same API endpoints in a short amount of time, the same results will be returned.

When a request is made to the web API of RabbitMQ it will collect node local data from all available nodes. This procedure can be slow and time consuming, and can result in high memory usage.

In the past some API endpoints were not only read from the statistics table of RabbitMQ, but they needed to walk through all the related Erlang processes and request information from these. This is no longer the case, making the Management Interface much more stable.

Performance Implications

As the number of connections, channels, and queues in the system increases, the performance of the Management Interface may degrade. This slowdown is primarily due to the demands of aggregating, filtering, paginating, and generating output for the results.

Aggregation, as outlined in the previous chapter, is a continuous process which happens in the background. If there are a lot of these entities, it can take a high amount of CPU and memory to collate the results. For each type, connections, channels, queues, etc. it can take up to a CPU core to have the metrics aggregated. In some of these cases the memory usage of the internal ETS tables grows very large due to the number of entities.

Once metrics are collected from each node, they undergo a process of merging, sorting, and pagination.
The final step involves converting the sorted and paginated results into JSON format. A substantial number of entities in the output can lead to considerable memory usage, ranging from tens to hundreds of megabytes.

These processes generate a significant amount of temporary data, increasing the workload for the garbage collector. Consequently, this can increase the time required to produce the output.

HTTP API Use Cases

As mentioned above the management UI uses an HTTP API to get the data from the RabbitMQ servers. But this API is useful for programmatically getting data from RabbitMQ and in some cases implementing third-party services.

Before the Prometheus interface was introduced to RabbitMQ in version 3.8, the best available way for external monitoring platforms to integrate with RabbitMQ was to set up a monitoring user and use this to access the management API. This provides all the data that is required for the successful monitoring solution, but it is very inefficient. For each request the user must be authenticated and the proper access rights needed to be checked, which is made "worse" if the authentication was outsourced to a third-party solution provider, e.g. Active Directory.

The traffic related metrics are aggregated from all RabbitMQ nodes in the cluster which means that it is enough to query a single node, but it also adds additional load to the monitoring workload. This workload is infrequent and hardly causes issues, but aggregating the data from all nodes has an unfortunate side effect: during a network partition, the aggregation times out during the initial phase of the partition. This means that crucial data can be missing to determine the traffic impact of the network partition.

Operating RabbitMQ by hand depends on thorough operational procedures and is prone to consistency issues. Automating backup procedures (e.g. importing and exporting definitions) or monitoring for the correctness of the topology can be very useful. The management API also enables changing some of the configuration of RabbitMQ (e.g. topology, policies, etc) which can be used to automate some of the operations tasks in a separate tool.

Depricated in RabbitMQ 4.0

RabbitMQ 4.0 will be a milestone in the design of the system. As such it presents some major changes. The relevant one for this paper is the deprecation of the metrics data provided by the management API. Although the data presented is useful and comes with an out of the box installation of RabbitMQ, it comes with performance issues and limited functionality compared to a dedicated monitoring solution. For further information, visit the deprecation announcement.

In the next section we'll describe the recommended monitoring solution for RabbitMQ.

SEVENTH STATE

CHAPTER TWO

The Prometheus Way

Starting with version 3.8, RabbitMQ adopted the Prometheus monitoring model[3] as the recommended way to observe the health and progress of the cluster.

Prometheus is designed around a pull-based model, where it regularly scrapes metrics from configured targets at specified intervals, storing this data in its time-series database. It supports service discovery to dynamically discover targets in various environments. Prometheus's architecture is modular, consisting of several components such as the main Prometheus server for collecting and storing data, Alertmanager for handling alerts, and client libraries for instrumenting application code. Its data model and query language (PromQL) allow for efficient storage and querying of time-series data, enabling users to analyse and aggregate metrics across multiple dimensions. Prometheus is designed to be reliable, allowing for deployments in various setups, including standalone or highly available configurations, without external dependencies.

The open-source Prometheus system is usually deployed with Grafana, which is a powerful and widely used open-source analytics and interactive visualisation web application. It provides a rich set of features for visualising time-series data, making it a popular choice for operations teams to understand and analyse metrics, logs, and traces from multiple sources like Prometheus, Elasticsearch, Graphite, and InfluxDB. Grafana allows users to create, explore, and share dashboards that display visually appealing graphs, charts, and alerts for operational monitoring. It supports a wide range of data sources and offers extensive customization options for dashboards, including a variety of panels and widgets.

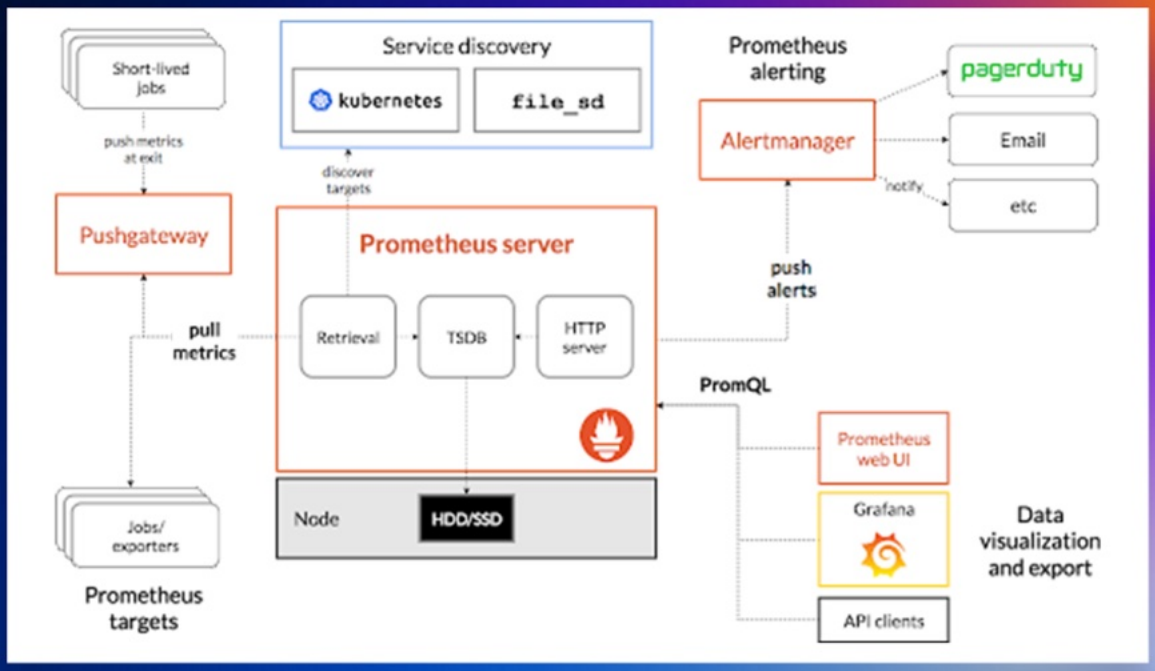
Unlike the management interface of RabbitMQ, these solutions are designed for all the different aspects of monitoring.

[3] It is worth noting that there was a RabbitMQ plugin to export Prometheus developed by the community. (https://github.com/deadtrickster/prometheus_rabbitmq_exporter)

SEVENTH STATE

Architecture Overview

The generic architecture of a Prometheus based system is shown in the following diagram:



Source: https://prometheus.io/assets/architecture.png

Prometheus Server

The Prometheus Server is the heart of the monitoring solution, responsible for gathering metrics from configured targets at specified intervals. It achieves this through a pull model, directly scraping metrics exposed by applications or intermediate devices. The server stores these metrics in a highly efficient time-series database optimised for fast data retrieval and compact storage. Besides data collection and storage, the Prometheus server provides a powerful query language, PromQL, enabling complex data analysis and visualisation directly through its web interface. This component is designed for reliability and can operate in various configurations to ensure high availability and scalability.

Alertmanager

The Alertmanager handles the operational aspect of monitoring by managing alerts generated by the Prometheus server. It is capable of processing hundreds of alerts from multiple Prometheus servers, deduplicating them to prevent noise, and then grouping them intelligently based on configurable rules to reduce information overload. The Alertmanager can route alerts to different endpoints and communication channels such as email, Slack, or SMS, based on severity or team. Moreover, it supports silencing, temporarily muting alerts, and inhibition, which can prevent alerts from being sent based on the status of other alerts, enabling more manageable on-call duty rotations and incident response.

Client Libraries

Prometheus provides a set of libraries that offer various metrics types, such as counters, gauges, histograms, and summaries, allowing for detailed monitoring of application performance and behaviour. Instrumentation with client libraries is a critical step in utilising Prometheus to its full potential, enabling the collection of in-depth, application-specific metrics beyond basic system metrics.

Pushgateway

The Pushgateway is designed for scenarios where the pull model is not feasible, such as with short-lived batch jobs or other ephemeral tasks that might not be running when Prometheus performs its scrape. Jobs can push their final state or metrics to the Pushgateway, making the information available for Prometheus to scrape. This component acts as an intermediary, ensuring that even transient services contribute to the overall monitoring landscape without requiring Prometheus to adapt its scraping strategy.

Service Directory

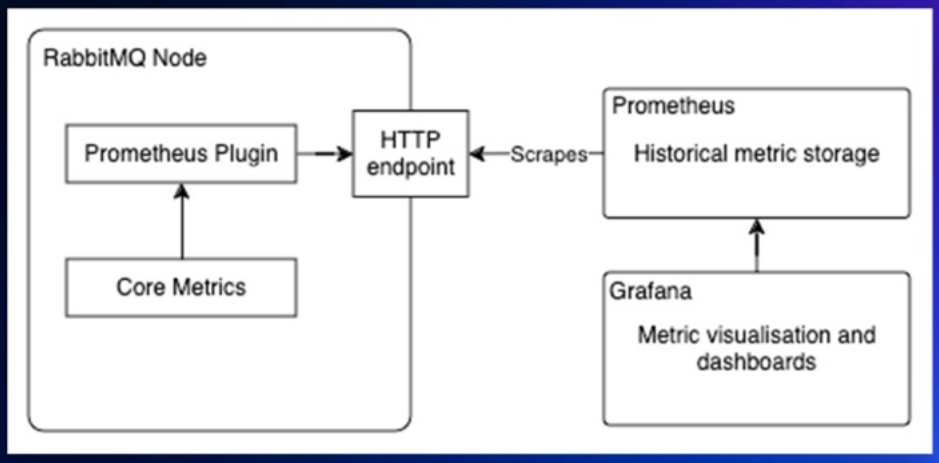
Service discovery mechanisms in Prometheus automate the process of finding targets to monitor within dynamic environments, such as cloud infrastructures or orchestrated container systems. Prometheus supports various service discovery configurations, including Kubernetes, AWS, and more, allowing it to automatically adjust its target list as services are added, removed, or moved across the infrastructure. This capability ensures that Prometheus's monitoring coverage remains comprehensive and up-to-date with minimal manual intervention, crucial for modern, rapidly changing deployments.

Exporters

Exporters play a crucial role in extending Prometheus's monitoring capabilities to third-party systems and services not natively instrumented to expose metrics in the Prometheus format. These external programs collect metrics from various sources like databases, hardware, web servers, and more, transforming and exposing them in a format that Prometheus can scrape and store. Exporters thus significantly broaden the range of applications and infrastructure components Prometheus can monitor, making it a versatile tool in heterogeneous environments.

Together, these components form a cohesive and robust monitoring ecosystem, capable of handling the complex requirements of modern infrastructure monitoring, from data collection and storage to alert management and automatic service discovery.

RabbitMQ in the Prometheus ecosystem



RabbitMQ implements an exporter and provides metrics over an HTTP port (by default 15692). Although we focus on RabbitMQ in this paper it is worth noting that Prometheus (or an equivalent monitoring solution) enables connecting not only RabbitMQ but all other systems in the overall solution. This opens up the possibility to connect metrics or logs originating from different services and provide a global view of events or metrics.

With the Prometheus exporter feature in modern RabbitMQs we advocate to integrate RabbitMQ in the solution-wide monitoring solution (regardless of Prometheus or other technology) and use RabbitMQ Prometheus exporter (and a possible bridge solution) as the standard way to get metrics from RabbitMQ.

This offloads workload from the RabbitMQ nodes to aggregate data from the cluster nodes and enables access to different historical data that RabbitMQ is not capable of retaining.

SEVENTH STATE

Endpoints

Prometheus reads the metrics data from an HTTP interface in RabbitMQ. Depending on the granularity of the data RabbitMQ exposes **three endpoints.**

1. The main and most important endpoint is the global metrics endpoint under `/metrics`. This exposes system level metrics from the RabbitMQ node as well as the main global metrics calculated on the specific node. These metrics can later be aggregated to cluster-wide metrics in Grafana. This endpoint contains all the necessary metrics for the "official" RabbitMQ Overview dashboard to function and provide a good baseline for monitoring RabbitMQ clusters.

   Assuming one does not use the management interface for monitoring purposes, or wants to futureproof the monitoring procedures, the `/metrics` endpoint is not enough to see which individual topology entities are contributing to the metrics. This breakdown or more detailed metrics can be accessed by two different endpoints.

2. Historically, the first endpoint exposing the metrics for all entities on the specific node was `/metrics/per-objects`. This endpoint includes everything that RabbitMQ can report and as such the metric reports can grow fairly large and can cause issues with storage and performance. Regardless, for small installations, this endpoint provides a convenient way to scrape everything.

3. Since the "object metrics" contain everything unconditionally, newer RabbitMQ versions offer another endpoint that accepts query parameters to filter the output to metrics. This is `/metrics/detailed`. This offers the opportunity for Prometheus to send different requests for the different types of metrics, e.g. one query for connection related metrics, one for queue related metrics, etc. In principle, the operator should collect all metrics from RabbitMQ because it is not possible to know what type of data can be useful to investigate any issues.

Using the overview and the detailed endpoints provide an opportunity to gather data at different intervals. E.g. gather the overview metrics in every 20 seconds, along with all the queue related metrics, but only gather connection and channel metrics in 40 second intervals for a system with thousands of connections and channels. In general we don't recommend collecting metrics at a low time resolution, i.e. metrics gathered rarer than 60 seconds provide diminishing value for investigation issues or spotting short bursts of overload.

Grafana for monitoring

Prometheus serves as the main metric database and it does provide a friendly UI to query the data by using the PromQM language, but it is not very convenient to do this every time one wants to check metrics. As a widespread solution Prometheus integrates well with Grafana as a visualisation tool.

Grafana is a powerful and widely used open-source analytics and interactive visualisation web application. It provides a rich set of features for visualising time-series data, making it a popular choice for operations teams to understand and analyse metrics, logs, and traces from Prometheus.

Grafana allows users to create, explore, and share dashboards that display visually appealing graphs, charts, and alerts for operational monitoring. It supports a wide range of data sources and offers extensive customization options for dashboards, including a variety of panels and widgets. With its user-friendly interface, advanced analytics features, and strong community support, Grafana has become an essential tool in the DevOps toolkit for real-time monitoring of IT infrastructure and application performance.

There are pre-made dashboards for RabbitMQ developed by both the RabbitMQ core team and the community. There is a short list of dashboards later in this chapter.

Metrics

In this section we will give an overview of metrics and what they can be used for. In the case studies chapter we will explain how some of the concrete metrics can be utilised for issue investigations.

Cluster global metrics

These metrics reflect counts valid to the whole cluster. The individual RabbitMQ nodes report the local numbers, while Prometheus does the aggregation based on PromQL queries.

Total message count: Total number of messages in the queues.

Unacknowledged message count: Messages consumed but not yet acknowledged by consumers.

Queue count: Total number of queues in the cluster.

Channel count: Total number of channels in the cluster.

Connection count: Total number of connections in the cluster.

Memory, disk and file handle internal RabbitMQ alarms: The column displays a warning if there are any active internal alarms in the cluster.

Connections related metrics

For AMQP connections the most important metrics to collect are the data rates. They help identify if a particular connection is not progressing.

Node: The cluster member which the connection is terminated at.

Connection Pid: The identifier of the connection. This identifier is present in the logs as well.

Incoming and outgoing data rates: The number of bytes sent or received by the connection in each second.

Channel related metrics

The following metrics are associated with the AMQP channels that transact them. It is possible to aggregate them to turn them into global counts.

Consumers: The number of queue subscriptions created on the channel.

Unacknowledged message count: Messages in flight, not yet acknowledged by the consumer application. Unconfirmed message count: Messaging in flight from the publishing application, not yet acknowledged by RabbitMQ.

Consumer prefetch: Maximum number of messages in flight for each consumer subscription.

Global prefetch: Maximum number of messages in flight for the channel.

Uncommitted messages: Messages that have been published within a transaction but not yet committed.

Uncommitted acks: Messages that have been acknowledged within a transaction but not yet committed by the consumer.

Note that the notion of consumers is an AMQP term and as such RabbitMQ treats it differently from producers. From the AMQP's point of view any client can publish on a channel without any specific need to declare this. It is possible to use heuristics to report the number of publishers but it is not guaranteed that it is correct in general.

Queues related metrics

In AMQP the channel (including the exchange) is a separate entity and RabbitMQ implements them in separate Erlang processes. Queues form another abstraction layer in both AMQP and in the implementation, hence some of the metrics share functions with channel metrics. This may seem a duplication of data, but serves an important purpose to determine the performance of RabbitMQ and be able to pinpoint bottlenecks in the message path.

Number of messages in the queue: Total number of messages in the queue, including unacknowledged messages. For best performance, queues should be empty.

Incoming message rate: Rate of new messages inserted into the queue.

Acknowledgment rate: The frequency at which acknowledgments from consumers are received, allowing messages to be removed from the queue.

Unacknowledged messages: The count of messages delivered to consumers that are pending acknowledgment before they can be removed from the queue.

Publisher count: Number of channels publishing to this queue.

Consumer count: The total number of consumers subscribed to the queue.

Queue length: Total number of messages in the queue, including unacknowledged messages. For best performance, queues should be empty.

Message polling rates: Rate of client applications polling the queue for messages. The metrics shown are the following:

- **Queue Polling with Acknowledgment:** In this mode, the queue requires an acknowledgment from the client to confirm message processing before removing the message from the queue.
- **Queue Polling without Acknowledgment:** Messages are immediately removed from the queue once served, with no need for processing acknowledgment from the client.
- **Polling on Empty Queue:** This metric tracks the count of poll events when the queue is empty, indicating unnecessary polling activity.

Stream related metrics

Streams were introduced in RabbitMQ version 3.9 and provide a distributed log data structure. Streams are highly efficient and capable of serving millions of messages per second. The efficiency is made possible by utilising data transferring functionality close to the operation system. This is reflected by the fact that some metrics are not exposed to individual streams or by fewer available metrics compared to queues. Nevertheless, the existing metrics provide enough data to see progress and health of the system. Furthermore, streams support both AMQP and their proprietary protocol.

Number of messages in a stream: Total number of messages in the stream, including unacknowledged messages. Messages in the stream are stored until the length limits are reached.

Number of consumers for a stream: Total number of consumers for the stream. The same stream can be read multiple times by different consumers.

Per publisher message statistics: On the management UI the number of messages published to a stream by a connection is available.

Per consumer message statistics: On the management UI the number of messages consumed from a stream by a connection is available.

Total number of messages received on the stream protocol: The total number of RabbitMQ Stream messages received. Currently RabbitMQ does not expose individual stream receive metrics.

Total number of messages confirmed for publishers on the stream protocol: RabbitMQ confirms a message if it was replicated.

Total number of stream publishers: The total number of stream writers in the cluster.

Alarms

In communication systems alarms are special messages that the system can emit to notify the operator that something is not right. A common protocol for such events is the Simple Network Management Protocol (SNMP). However, RabbitMQ does not follow this and exposes very limited cases of alarms via boolean flags as part of a metric.

Operationally RabbitMQ treats the following events as problems:

- **Used memory exceeding the memory watermark limit**, in which case RabbitMQ automatically blocks incoming messages to avoid a possible system crash.
- **Used disk space exceeding the disk watermark limit,** in which case RabbitMQ automatically blocks incoming messages to avoid filling the available space. This can lead to problems restarting the operating system, which is a serious problem.

These events are obviously serious cases as they affect the primary function of RabbitMQ to transact messages. In comparison RabbitMQ does not make a judgement call if other “important to the business” events happen, e.g. a queue is accumulating messages because it has no consumers. The main difference with the latter is that it is a valid use case for RabbitMQ and the overall system may consider it a usual planned event. The same applies to cases when there are slow consumers, or a short burst of incoming traffic causing temporary high latency.

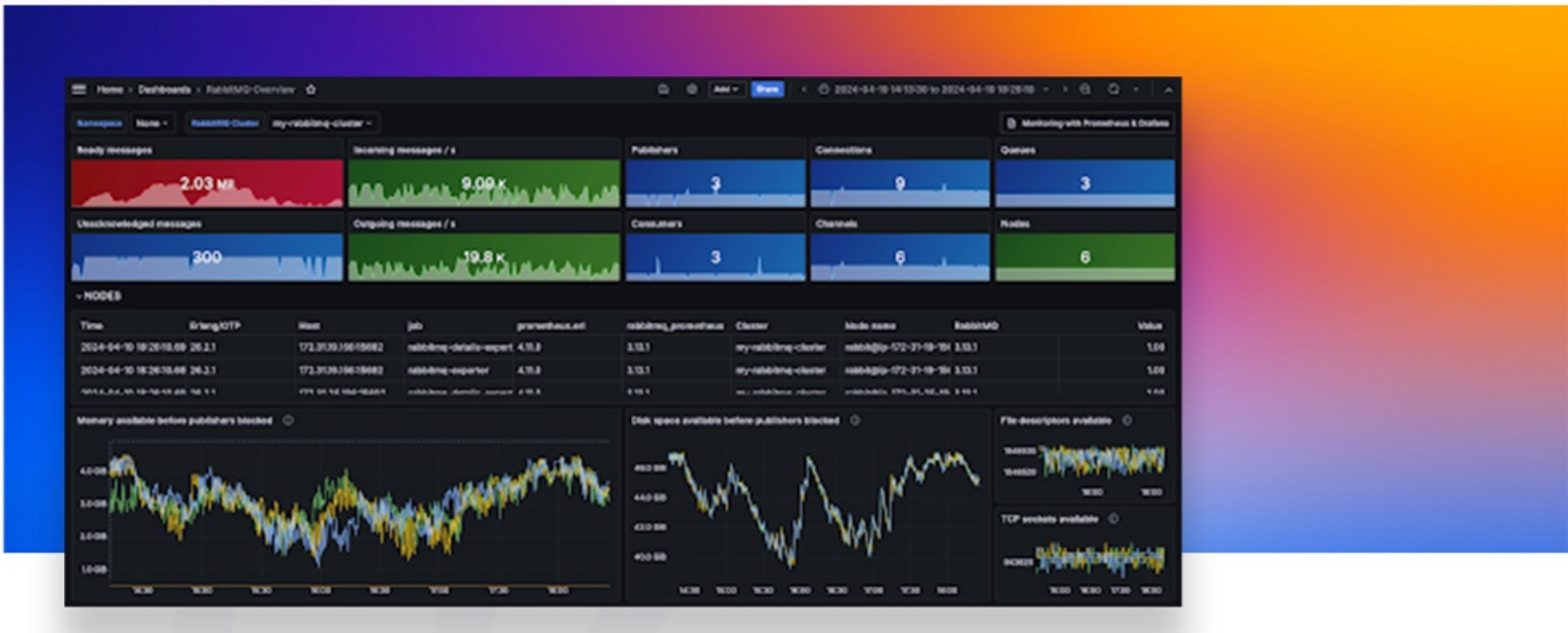
These events are of great importance to determine the overall health of the system, but the criteria can only be codified on a case by case basis. Each RabbitMQ deployment serves a different purpose and what counts as a problem worth intervention or monitoring by the operator varies greatly. This is not a problem with Prometheus and Grafana that enables declaring custom warning and alarm criteria for different conditions or to add complex conditions to these alarms.

Dashboards
“Official” dashboards

The RabbitMQ core team released a set of Grafana dashboards to visualise metrics for various parts of RabbitMQ. In this chapter we give a brief overview of the available dashboards. The RabbitMQ documentation[4] provides a detailed description of how to use these metrics and how to set up the system.

RabbitMQ Overview

The main dashboard is the “RabbitMQ Overview” including details of the main cluster runtime metrics:

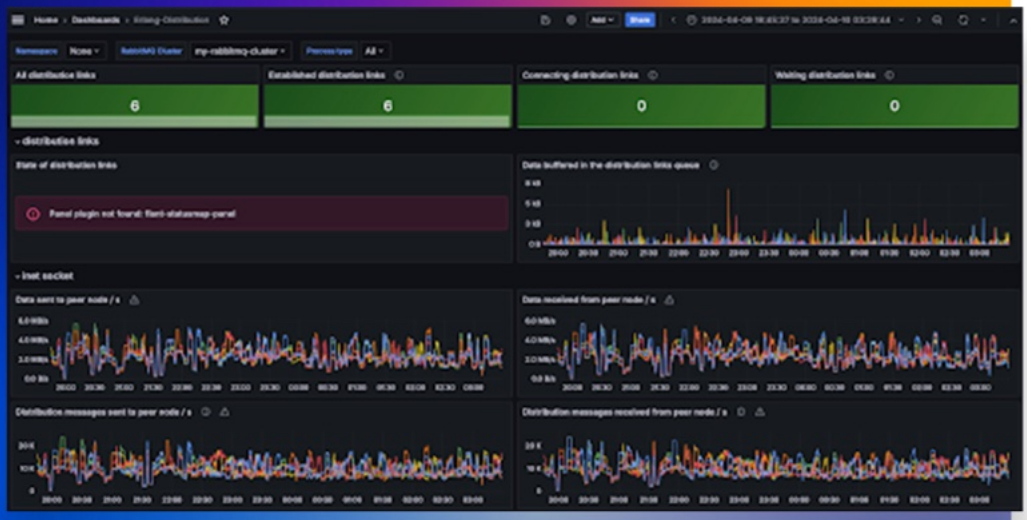


- *Cluster health including memory and disk space metrics*
- *Publisher and consumer numbers and throughput*
- *Aggregated message throughput metrics*
- *Aggregated message counts*
- *Connection, channel and queue churn statistics*

4. https://www.rabbitmq.com/docs/prometheus

Erlang Distribution

This dashboard visualises the different metrics that the BEAM Erlang VM exposes about the TCP connections it uses to enable the cross communication of the RabbitMQ nodes. These metrics are quite low level but they can be used to detect and investigate network partitions.



Erlang Memory Allocators

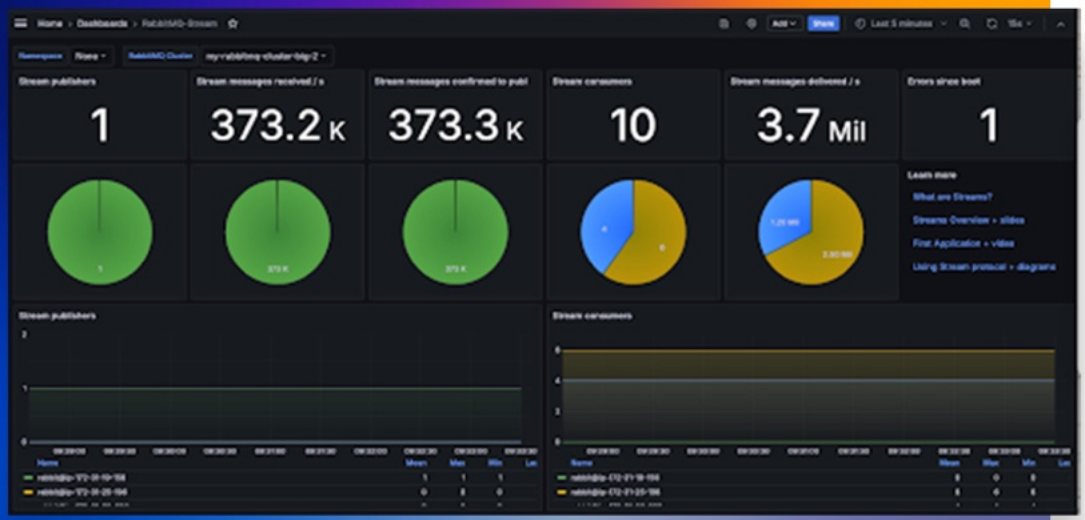
This dashboard displays the internal metrics of the Erlang Virtual Machine's memory usage.

Quorum Queues

This dashboard may have a misleading name but it is very useful to look under the hood of the Raft subsystem that underpins the quorum queue implementation. The individual values on the panels are less important than noticing any divergence between the nodes.

Streams

The streams dashboard is the go to place to see the performance and health of RabbitMQ streams.

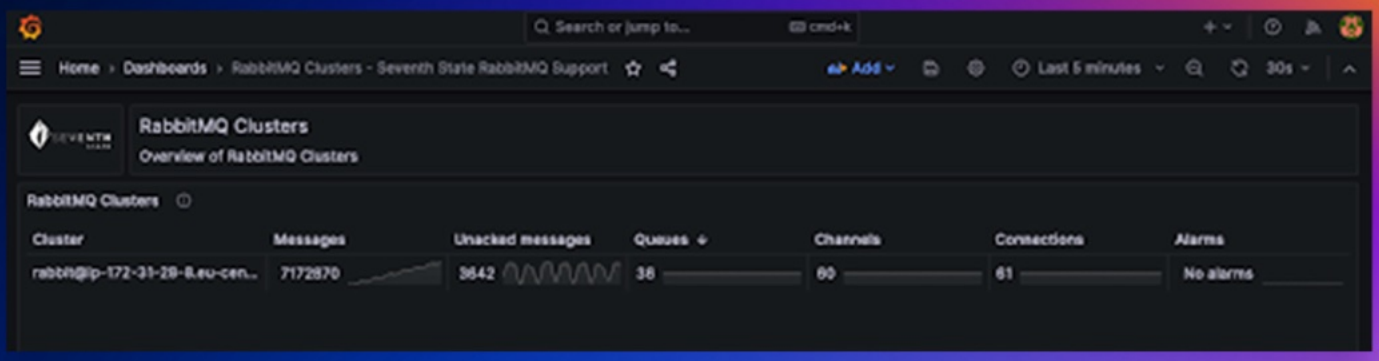


Community Dashboards

Seventh State also published[5] a set of dashboards (available at the Grafana Marketplace[6]) to visualise the detailed metrics exposed by the `/metrics/detailed` endpoint.

Clusters

The Clusters Dashboard provides a view of all RabbitMQ clusters within your environment. It showcases metrics offering a high-level overview critical for managing multiple clusters effectively.



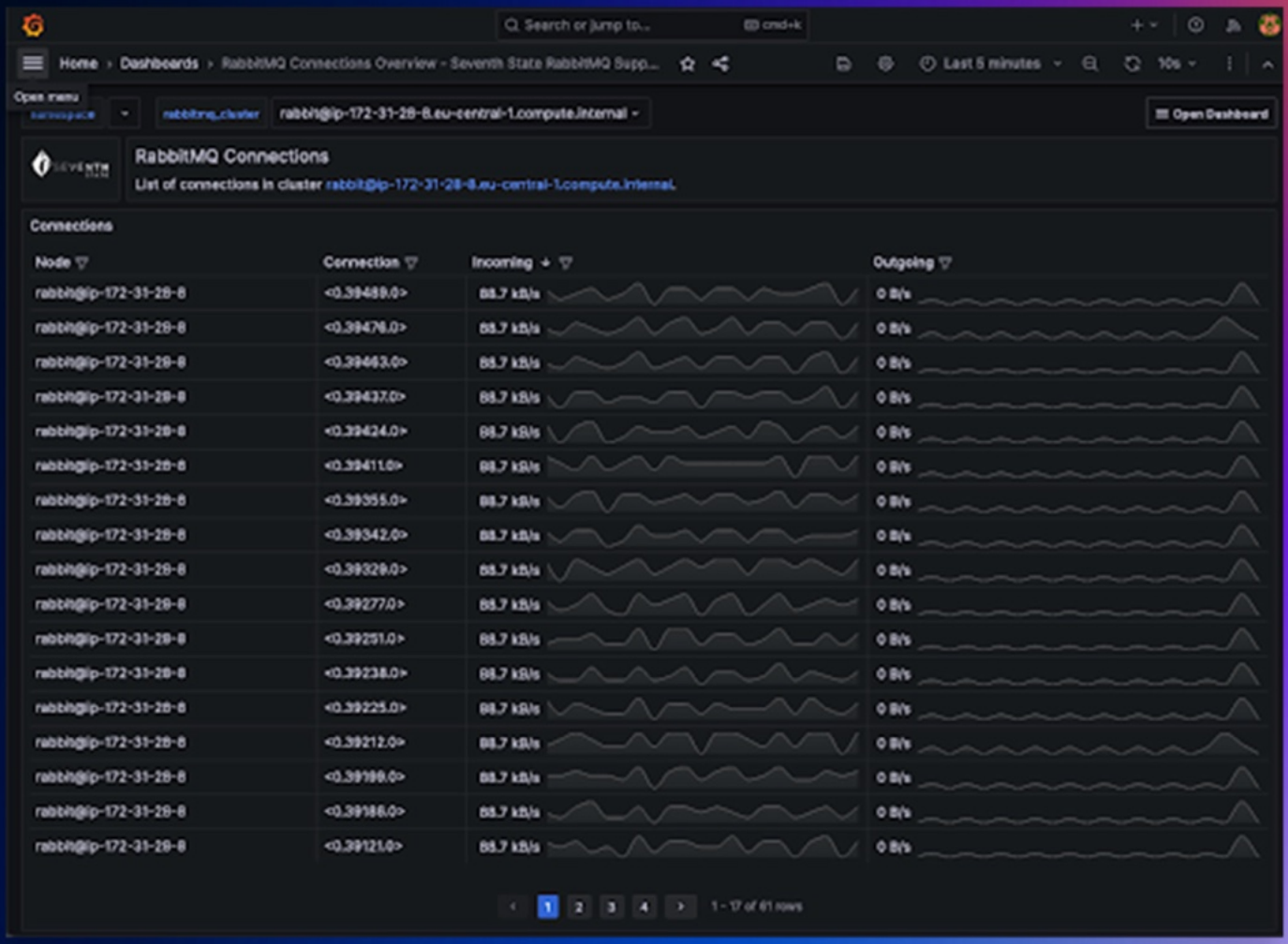
7S

You can download this dashboard from https://grafana.com/grafana/dashboards/20697-rabbitmq-clusters-seventh-state-rabbitmq-support/ .

5. https://seventhstate.io/rabbitmq-monitoring-dashboards/
6. https://grafana.com/orgs/seventhstate

Connections Overview

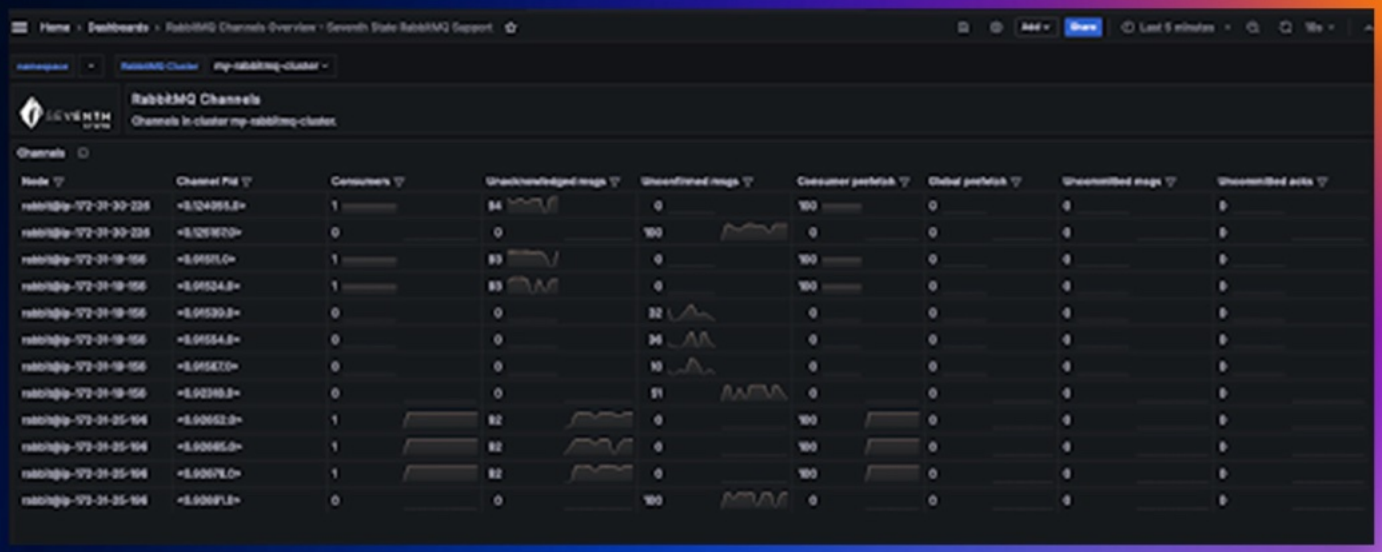
This dashboard focuses on RabbitMQ connections, presenting key metrics like connection throughput.



You can download this dashboard from https://grafana.com/grafana/dashboards/20698-rabbitmq-connections-overview-seventh-state-rabbitmq-support/ .

RabbitMQ Channel Overview

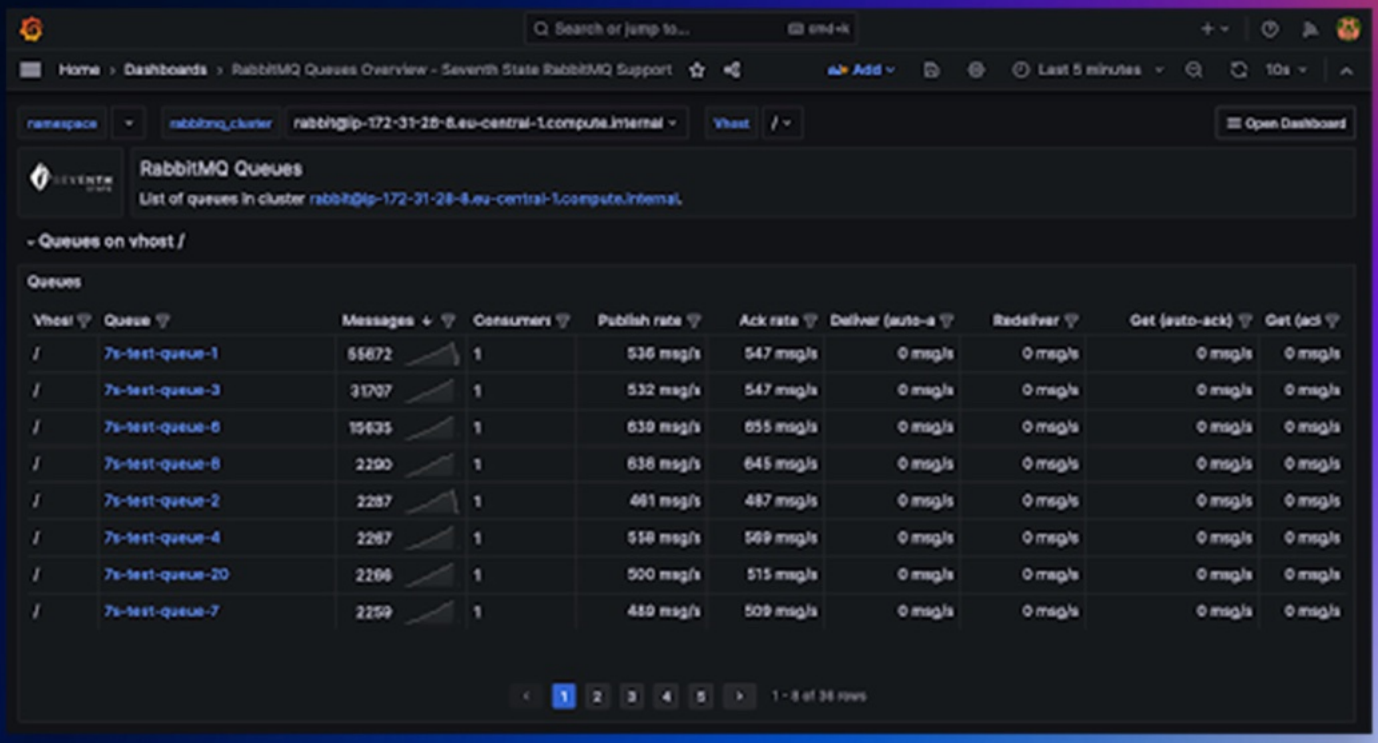
The Channels Overview Dashboard offers insights into RabbitMQ channels, including global counts, message rates per channel, and channel utilisation. Channels are crucial for message throughput, making this dashboard invaluable for tuning and capacity planning.



You can download this dashboard from https://grafana.com/grafana/dashboards/20831-rabbitmq-channels-overview-seventh-state-rabbitmq-support/.

Queues Overview

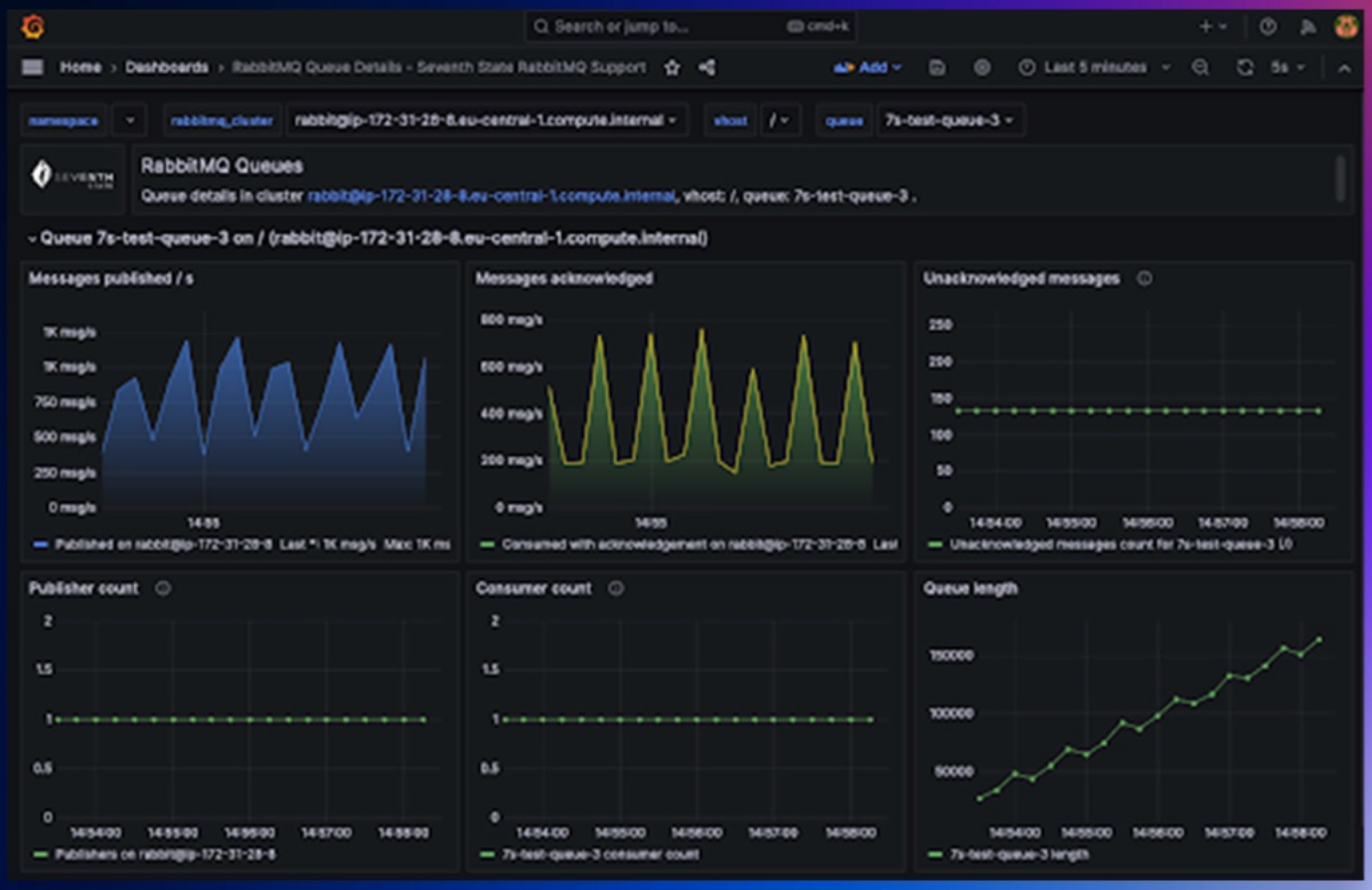
This dashboard gives a detailed view of RabbitMQ queues, displaying real-time metrics like message backlog, publish and delivery rates. It's designed to provide an overview of queue health and ensure messages are being processed.



You can download this dashboard from https://grafana.com/grafana/dashboards/20700-rabbitmq-queues-overview-seventh-state-rabbitmq-support/ .

Queues Details

This dashboard gives a detailed view of a single RabbitMQ queue, displaying historical and real-time metrics like message backlog, publish and delivery rates. It's designed to provide an overview of queue health and ensure messages are being processed.



7S

You can download this dashboard from https://grafana.com/grafana/dashboards/20701-rabbitmq-queue-details-seventh-state-rabbitmq-support/ .

SEVENTH STATE

CHAPTER THREE

Logs and log aggregation

Logs in RabbitMQ

Logs are a form of textual information generated by the broker. The log messages provide a lot of information on events taking place in the RabbitMQ installation. For example, a log message is generated whenever a node goes down or comes up, when a quorum queue is created, connection is opened, or if any error is encountered.

The log entries include timestamps, severity levels, and descriptive messages about the events that occurred. They can help administrators and developers diagnose problems, identify bottlenecks, analyse system performance, and detect any anomalies or misconfiguration within the RabbitMQ infrastructure.

It is important to treat the log messages on the same level as metrics. Not everything can be expressed as numbers that metrics are best suited for, and not every event needs to be expressed as a textual log. When explaining the cause of problems, both logs and metrics must be checked to make an informed decision.

Before we dive into some of the most important log entries in RabbitMQ, we will briefly overview the types of logs RabbitMQ produces. For an Erlang based system like RabbitMQ, we can identify three main types of logs: normal logs, crash logs, and crash dumps.

Normal Logs

These messages are sorted into the usual levels: notice, debug, info, warning and error in increasing order of severity. Most RabbitMQ systems can produce a large amount of log messages when the lower levels are enabled. Over time, the balance was struck to include all log messages from info level and higher for all non-development instances. This is because the severity level also contains semantic information on the nature of the entry. E.g.: a node starting up and joining the cluster is useful information (as opposed to an error) and it is important to keep. While some error log entries do indicate problems in the system, RabbitMQ's self-healing capabilities may render them less severe.

In production systems, we do not recommend enabling debug level logs, as they can lead to high load on the system.

By default, RabbitMQ puts all log messages in a single place where it is easy for the operator to check. But if one or more categories of log messages become overwhelming to the logging system, then they can be separated and their severity filtering can be limited.

RabbitMQ provides the following categories for logging:

Connection: Connection lifecycle event logs for AMQP connections

Channel: Errors and warning logs for AMQP channels

Queue: Mostly debug logs from message queues

Mirroring: Logs related to the now deprecated classic queue mirroring

Federation: Federation plugin logs

Upgrade: Verbose upgrade logs

Default: Generic log category

Crash Logs

These logs are created by errors in the system, typically in unexpected scenarios or when bugs are encountered. In many cases these errors can be ignored because RabbitMQ can recover from non-critical errors. If they recur then they can indicate serious problems.

A crash log includes information about the reason for the crash and stacktrace, for example:

```
[error] <0.31900.5591> ** Generic server rabbit_core_metrics_gc terminating
[error] <0.31900.5591> ** Last message in was start_gc
[error] <0.31900.5591> ** When Server state ==
{state,#Ref<0.2044953014.131072060.18984>,120000}
[error] <0.31900.5591> ** Reason for termination ==
[error] <0.31900.5591> ** {{aborted,{no_exists,[rabbit_queue,
[error] <0.31900.5591> {amqqueue,'_','_','_','_','_','_','_','_','_','_',
[error] <0.31900.5591> '_','_','_','_','_','_','_','_','_','_'}]}},
[error] <0.31900.5591> [{mnesia,abort,1,[{file,"mnesia.erl"},{line,362}]},
[error] <0.31900.5591>
{rabbit_amqqueue,do_list,0,[{file,"rabbit_amqqueue.erl"},{line,1105}]},
[error] <0.31900.5591> {rabbit_amqqueue,list_with_possible_retry,1,
[error] <0.31900.5591> [{file,"rabbit_amqqueue.erl"},{line,1330}]},
```

The error message indicated that `rabbit_core_metrics_gc`, the metric garbage collector crashed, because `no_exists,rabbit_queue`, the rabbit_queue table was not existing.

Because these errors are raised when the unexpected happens it is almost impossible to create operational instructions to diagnose. A RabbitMQ expert is usually needed to diagnose the problem and find the root cause.

Crash Dumps

Erlang crash dumps are files that are automatically generated when an Erlang runtime system crashes. These files contain detailed information about the state of the Erlang virtual machine at the time of the crash, including the status of all processes, system memory usage, the call stack of each process, and other diagnostic information. The files can contain the full memory of the Erlang VM at the time of the crash, therefore they can be very large.

These crash dump files are unique to the BEAM Erlang VM and are not the same as the system generated process memory dumps. These are text files that operate on the Erlang level instead of the opaque VM level memory representations, therefore it is easier to analyse and it is not architecture dependent.

It is very important to save these files if RabbitMQ crashes for later investigation. These files are typically located in the RabbitMQ log directory and are called `erl_crash.dump`.

Log Aggregation

RabbitMQ stores the logs either in files or prints it out on the standard out for a logging subsystem to store it. Either way the logs passively accumulate on the host machine until an operator opens them to check for anything important. This manual approach is adequate for investigating issues but not useful for proactively alerting if anything goes wrong. A solution to this problem is to use a log collecting or aggregation system that collects all the logs from all systems and based on specific patterns it raises alarms.

Among the most favoured log aggregation solutions in the community are Splunk, the ELK stack, Graylog and various Software-as-a-Service platforms. These services enable very efficient troubleshooting techniques by enabling:

- Filtering log messages based on certain criteria
- Sorting log entries based on time from multiple sources
- Displaying a histogram for events
- Showing log entries from connected services (e.g. clients, database)

Compared to manual log reading techniques, these tools especially if they are connected to an alarming service can speed up the issue diagnostics. This approach is also useful for identifying issues that extend beyond a single node and involve cluster-wide interactions. Sometimes problems show up in RabbitMQ (e.g. messages piling up in queues, or error logs indicating client misbehaviour) that are not RabbitMQ issues. These log aggregation tools can speed up pinpointing the issue because all the required information is under one hood.

Important log messages

There are lots of log messages. We highlight some of the most important ones to monitor for and what is the recommended action to take. This list is definitely non-exhaustive and the specific RabbitMQ installation may require monitoring and alerting on further log messages that regularly occur during known incidents on the particular cluster.

Structure of Log Messages

One of the important parts of a log message is the Pid, or process identifier, an identifier which will be unique for the lifetime of the Erlang VM. These are the numbers between angle brackets (< and >). For example, `<0.1476939.0>` is the pid in the following log message:

```
[info] <0.1476939.0> closing AMQP connection <0.1476939.0>
(172.31.36.224:50420 -> 172.31.25.196:5672 - perf-test-configuration-1,
vhost: '/', user: 'testuser').
```

By searching through the logs for `<0.1476939.0>` we can correlate all log messages for this particular connection or other entities in the system. In the following examples all log messages will have Pids associated. If the Pid is the same between two log messages, it means it was produced by the same entity, ie. queue, connection, channel, etc. which makes it easier to track down issues.

Node Lifecycle

Node Startup

```
2024-04-11 04:20:57.989005+00:00 [info] <0.249.0>  Starting RabbitMQ
3.13.1 on Erlang 26.2.1 [jit]
```

When a node is starting up it displays the version of RabbitMQ and Erlang it is using. During the startup procedure the internal databases are set up and synchronised, plugins are started, then at the last step the protocol listeners are enabled.

```
2024-04-11 04:21:28.332467+00:00 [info] <0.656.0> Server startup
complete; 15 plugins started.
```

When the startup is completed RabbitMQ displays the message above. The more queues, exchanges, bindings, plugins a system has, the slower the startup procedure will be.

It is worth monitoring for this message and the time RabbitMQ takes to start. Large deviation from the average startup time is worth investigating.

Node Shutdown

```
2024-04-11 04:20:37.561347+00:00 [info] <0.989.0> RabbitMQ is asked to
stop…
```

When RabbitMQ is stopped by an operator or it is restarting due to a network partition, it will display the message above. This kicks off the stopping procedure, which is the reverse of the startup sequence. First, all connections and listeners are closed, then plugins are stopped, then the node fully stops.

```
2024-04-11 04:20:37.561347+00:00 [info] <0.989.0> RabbitMQ is asked to
stop…
```

At the point of this message all internal RabbitMQ infrastructure has stopped on the node and only the Erlang VM is running. The next step will be either the stopping of the Erlang VM or waiting for the network to recover in case of a partition.

It is important to monitor these events to know when RabbitMQ is down and when any of these events are raised, an operator should check the logs to determine the reason for the node stopping.

Connection Lifecycle

A connection can be tracked by the Process Identifier (Pid) in the log message. In the following example the Pid is <0.1479760.0>. All messages related to the AMQP connection will include this identifier. If the connection is successfully authenticated, the following three lines will be displayed:

```
2024-04-11 03:26:41.417899+00:00 [info] <0.1479760.0> accepting AMQP
connection <0.1479760.0> (172.31.36.224:50628 -> 172.31.25.196:5672)
2024-04-11 03:26:41.418981+00:00 [info] <0.1479760.0> connection
<0.1479760.0> (172.31.36.224:50628 -> 172.31.25.196:5672) has a
client-provided name: perf-test-producer-0
2024-04-11 03:26:41.419898+00:00 [info] <0.1479760.0> connection
<0.1479760.0> (172.31.36.224:50628 -> 172.31.25.196:5672 -
perf-test-producer-0): user 'testuser' authenticated and granted access
to vhost '/'
```

An AMQP connection can have a client-provided name, which will be included in the log messages as well. We recommend that application developers name the connections appropriately for easier debugging. The log messages also contain the source and destination hosts and ports of the connection.

```
2024-04-11 03:31:02.475403+00:00 [info] <0.1479760.0> closing AMQP
connection <0.1479760.0> (172.31.36.224:50628 -> 172.31.25.196:5672 -
perf-test-producer-0, vhost: '/', user: 'testuser')
```

When a connection is closed by performing the appropriate AMQP handshake the message above is displayed.

Memory Alarm

There can be many situations where a node's memory usage in a RabbitMQ cluster grows very high. RabbitMQ has a built-in protection mechanism to avoid being killed by the OOM handler. When the process's memory usage grows to the memory watermark a memory alarm is raised. This is visible in the logs by the following log messages:

```
2024-04-11 03:14:17.949806+00:00 [warning] <0.466.0> memory resource
limit alarm set on node 'rabbit@ip-172-31-19-156'.
2024-04-11 03:14:17.949806+00:00 [warning] <0.466.0> *** Publishers
will be blocked until this alarm clears ***
```

When the alarm is in effect no message flow can happen in the cluster. Usually the alarm is cleared quickly (seconds) due to garbage collection or because RabbitMQ catches up with incoming messages during the publisher block. If RabbitMQ is under very high load or it is under provisioned the RabbitMQ nodes can be stuck in this state for a longer time.

```
2024-04-11 03:14:18.956975+00:00 [warning] <0.466.0> memory resource
limit alarm cleared on node 'rabbit@ip-172-31-19-156'
2024-04-11 03:14:18.957074+00:00 [warning] <0.466.0> memory resource
limit alarm cleared across the cluster
```

When the memory usage is reduced below the memory watermark the alarm is cleared on all nodes in the cluster, traffic can start to flow again.

Disk Alarm

The disk alarm is very similar to the memory alarm explained previously but it applies to the disk usage of RabbitMQ. RabbitMQ triggers this alarm when available disk space is low, reaching the free disk watermark, halting all incoming messages from publishers. Typically, this situation is not a result of excessive publisher activity but rather due to insufficient disk allocation for RabbitMQ and the accumulation of unconsumed messages.

```
2024-04-11 04:14:09.596175+00:00 [info] <0.470.0> Free disk space is
insufficient. Free bytes: 2567765777. Limit: 3000000000
2024-04-11 04:14:09.596305+00:00 [warning] <0.466.0> disk resource
limit alarm set on node 'rabbit@ip-172-31-25-196'.
2024-04-11 04:14:09.596305+00:00 [warning] <0.466.0> *** Publishers
will be blocked until this alarm clears ***
```

In most cases, this error will not go away by itself but by operator intervention of either clearing some queues or allocating more disk to RabbitMQ.

```
2024-04-11 04:14:35.081256+00:00 [info] <0.470.0> Free disk space is
sufficient. Free bytes: 3432234223. Limit: 3000000000
2024-04-11 04:14:35.081347+00:00 [warning] <0.466.0> disk resource
limit alarm cleared on node 'rabbit@ip-172-31-25-196'
2024-04-11 04:14:35.081418+00:00 [warning] <0.466.0> disk resource
limit alarm cleared across the cluster
```

When the alarm is cleared, traffic can flow again.

Node connection loss

In clustered installations the logs display information about clustering connections between the nodes. These connections are highly important for message and metadata replication.

When a connection goes down the following messages are displayed:

```
2024-04-11 05:10:56.690415+00:00 [info] <0.545.0> rabbit on node
'rabbit@ip-172-31-30-226' down
```

The message above indicates that the local node lost knowledge of the RabbitMQ application of the other instance. Either this is caused by the restart of RabbitMQ or by network issues.

```
2024-04-11 05:10:56.697071+00:00 [info] <0.545.0> node
'rabbit@ip-172-31-30-226' down: net_tick_timeout
```

While this message indicates, that the reason for this was that the connection went down due to network issues. Net tick timeout indicates that the heartbeat between the cluster nodes failed.

Pausing

In current clustered RabbitMQ installations when a network connection is lost between the nodes at least one of the nodes need to be restarted. If a node loses connectivity to the majority of the other nodes it will need to pause and wait for the network to recover.

The log message below indicates exactly that condition:

```
2024-04-11 04:57:30.769479+00:00 [warning] <0.548.0> Cluster
minority/secondary status detected - awaiting recovery
```

This starts the normal stopping procedure, except the Erlang VM will keep on running trying to recover the network connections.

```
2024-04-11 04:57:40.995642+00:00 [info] <0.1290.0> Successfully stopped
RabbitMQ and its dependencies
```

When the connections are recovered to the other nodes, RabbitMQ will automatically restart:

```
2024-04-11 04:57:42.008111+00:00 [info] <0.1290.0> RabbitMQ is asked to
start…
```

If these messages are occurring frequently it is recommended to review what is causing networking issues in the infrastructure.

Queue not found

One of the most common issues with client applications is that the queue they are trying to use is not present on the cluster. This state is indicated by the following message:

```
2024-04-11 06:24:04.069312+00:00 [error] <0.11160.0> Channel error on
connection <0.11135.0> (46.139.39.107:50695 -> 172.31.30.226:5672,
vhost: '/', user: 'testuser'), channel 1:
2024-04-11 06:24:04.069312+00:00 [error] <0.11160.0> operation
basic.consume caused a channel exception not_found: no queue
'non-existing-queue' in vhost '/'
```

This means that the application is unable to progress, because it is trying to consume from a queue which does not exist. This kind of scenario will show up in channel churn metrics as well, as any channel level error will close the AMQP channel.

Unexpectedly closed connection

In case there are issues with the client application, such as the one above with a missing queue, or maybe a bad message, it can cause the client application to crash. This usually shows up in the logs as "clients unexpectedly closing" connections.

```
2024-04-11 06:20:37.860346+00:00 [warning] <0.7268.0> closing AMQP
connection <0.7268.0> (172.31.36.224:26646 -> 172.31.19.156:5672 -
perf-test-producer-2, vhost: '/', user: 'testuser'):
2024-04-11 06:20:37.860346+00:00 [warning] <0.7268.0> client
unexpectedly closed TCP connection
```

This message indicates that the AMQP connection was not closed properly with the AMQP closing handshake, but abruptly with a TCP reset. In some cases when the application logic is successfully executed but the AMQP connection closure is not implemented properly the effect is minimal to none. In other cases this log entry indicates that the connected application crashed and it should be investigated why.

SEVENTH STATE

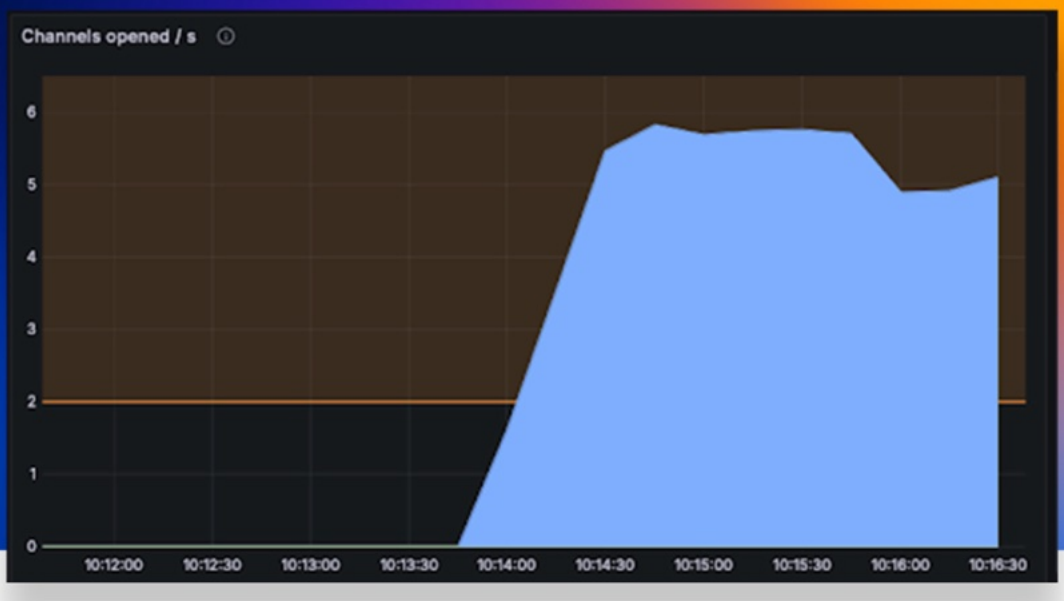
CHAPTER FOUR

Metrics and how to use them?

Case Study 1. Client Troubles

The developers just deployed the new version of the application but it is not working. Seemingly the application is running, but when users try to log in on the website, they are presented with a 500 Internal Server Error, and nothing happens.

While looking at the Grafana dashboard RabbitMQ Overview, we notice that after the redeployment of the application, the channel churn is not zero, it's a few channels opened per second:



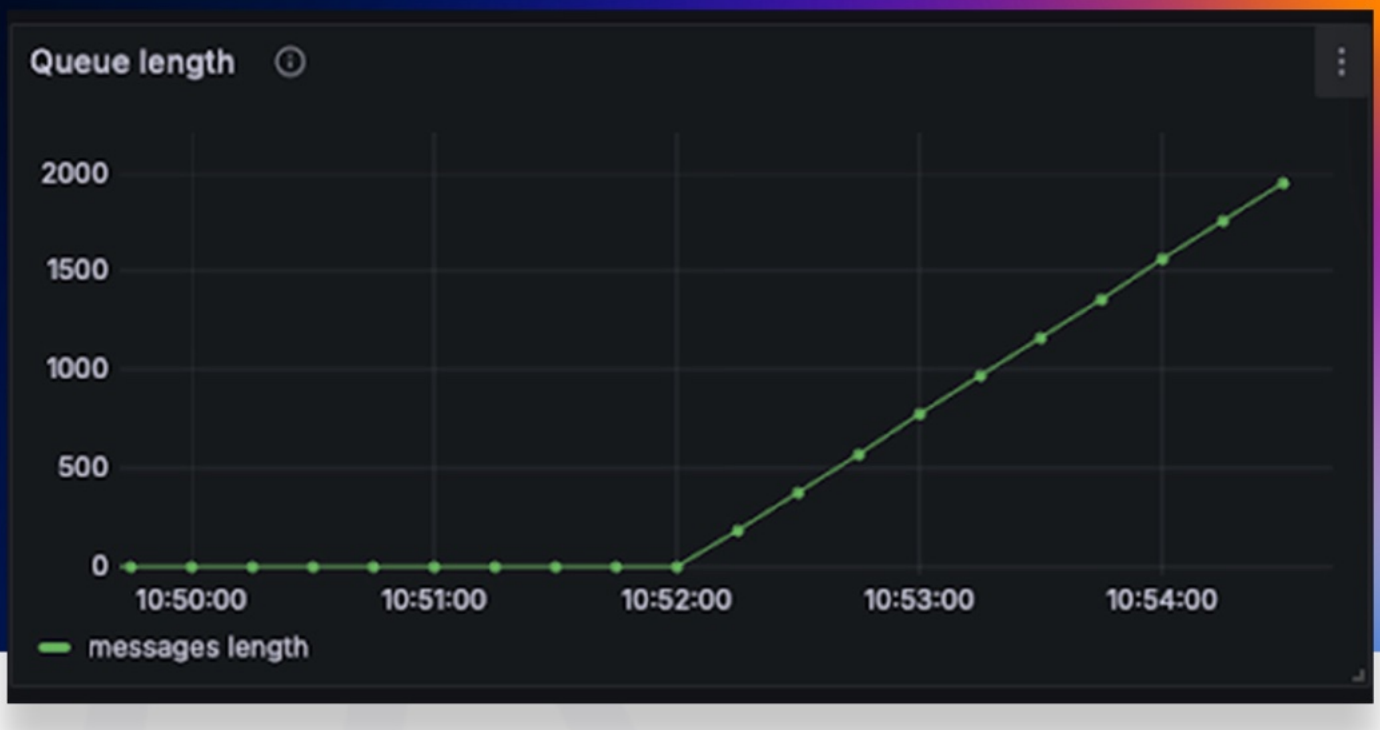
We also see that before the deployment the channel churn was zero. If we take a look at the RabbitMQ logs, we can immediately see what is the cause of the issue:

```
08:20:13.653913+00:00 [info] <0.244750.0> accepting AMQP connection
<0.244750.0> (94.44.249.142:2339 -> 172.31.30.226:5672)
08:20:13.760417+00:00 [info] <0.244750.0> connection <0.244750.0>
(94.44.249.142:2339 -> 172.31.30.226:5672): user 'testuser'
authenticated and granted access to vhost '/'
08:20:13.850177+00:00 [error] <0.244766.0> Channel error on connection
<0.244750.0> (94.44.249.142:2339 -> 172.31.30.226:5672, vhost: '/',
user: 'testuser'), channel 1:
08:20:13.850177+00:00 [error] <0.244766.0> operation basic.consume
caused a channel exception not_found: no queue 'messages' in vhost '/'
08:20:13.927795+00:00 [info] <0.244750.0> closing AMQP connection
<0.244750.0> (94.44.249.142:2339 -> 172.31.30.226:5672, vhost: '/',
user: 'testuser')
```

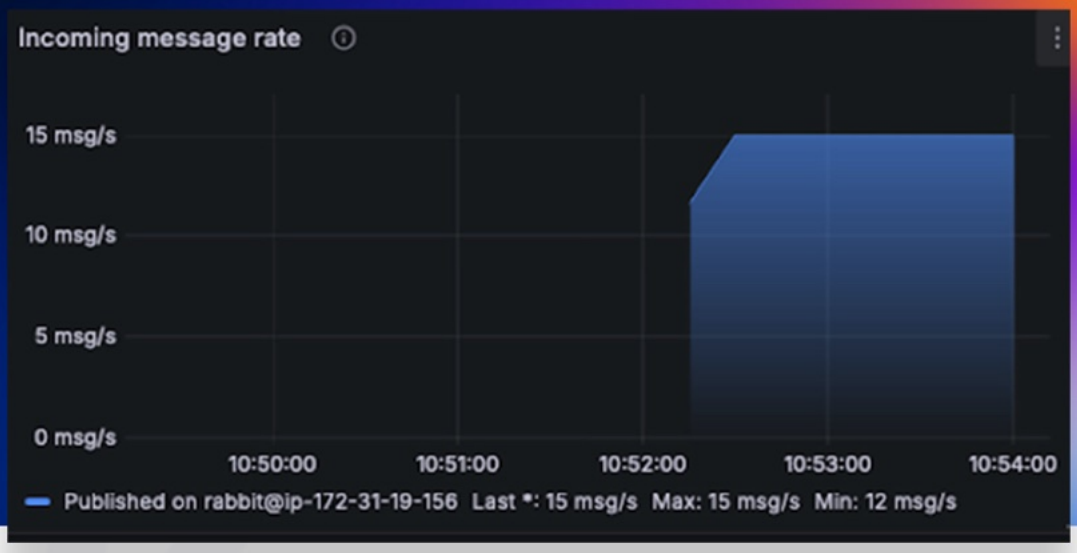
From the logs, we can see that the application is stuck in a reconnect - trying to consume - disconnect loop. The fix for this is to create the queue in some way. In many cases, it's the responsibility of the application to create it, however in some other cases it may be a configurator tool or the RabbitMQ Administrator.

Now that the queue is created, messages start to flow, however the application is slow and users are unable to log in in time.

Now that the queue exists the users started to log in but for many users it times out. The error rates are still elevated but now instead of 500 the users are receiving 408 Request Timeout. By taking a look at the queue in question, we can see that the messages are piling up in it.

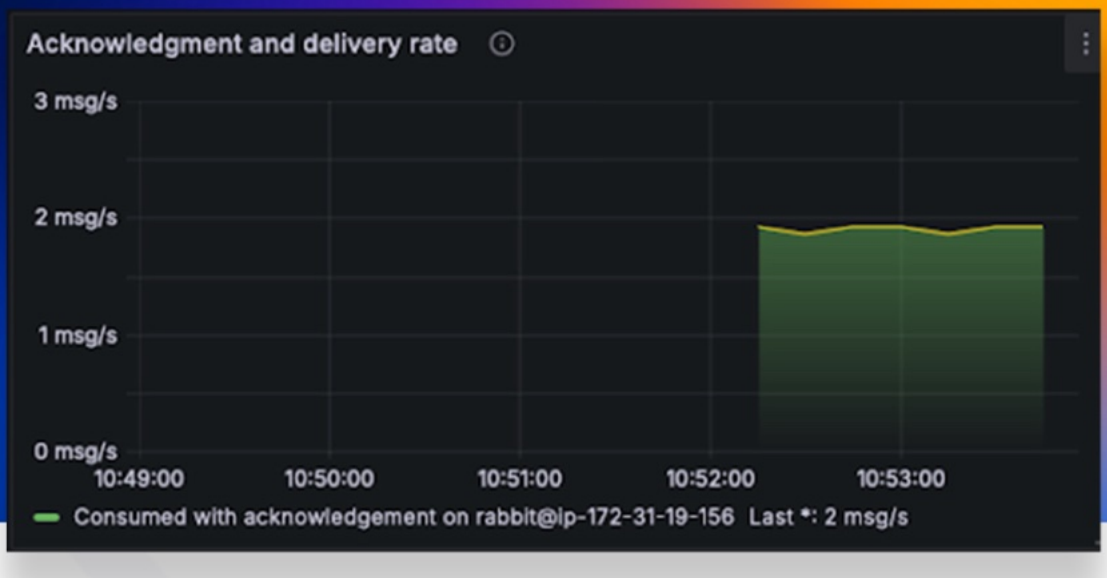


This means that either there is no consumer attached to the queue or that the consumer can not keep up with the load. By comparing the incoming rate of messages and the outgoing rate of messages, we can immediately see that the consumer throughput is well below the required.



We have an incoming message rate of about 15 messages / second.

The consumer is only able to process about 2 messages / second.



It's common to be in this situation in cases where the backend service or database is under high load, or for example if our authentication service is provided by a third party.

We can verify that the consumer is slow by looking at the number of unacknowledged messages. We know that for this consumer the prefetch count is 10, and we see from the chart that it is maxed out.

This definitely pinpoints the issue at the performance of the consumer.

There can be multiple solutions for this kind of scenario, deploying more resources for the consumer application, purging the messages in the queue, or setting up TTLs for the messages so we are not processing stale messages.

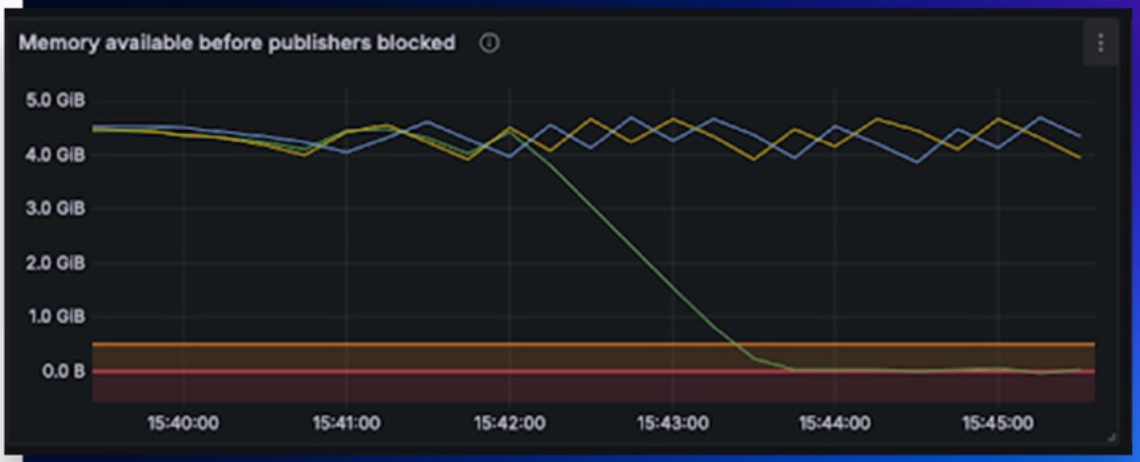
Case Study 2. Overload

For performance optimization, many applications choose not to use publish confirms. This approach, while potentially increasing throughput by reducing the minimal communication overhead, may also increase the risk of causing instability of the RabbitMQ cluster. If no publish confirms are used then RabbitMQ will need to buffer messages in memory which will lead to excessive load on the queues and on the garbage collector, eventually leading to situations where the memory alarm will be raised and traffic will be stopped, in bad cases crash of the node.

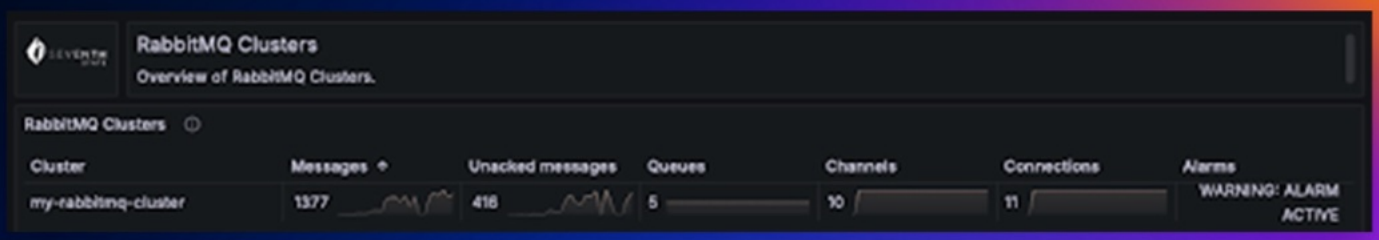
For this exact reason we always recommend using publish confirm in all publisher implementation.

As traffic increases, there will be a point in any installation where RabbitMQ can not handle the incoming message flow fast enough, leading to increased memory usage.

The load from the publishers increases gradually up to a limit.



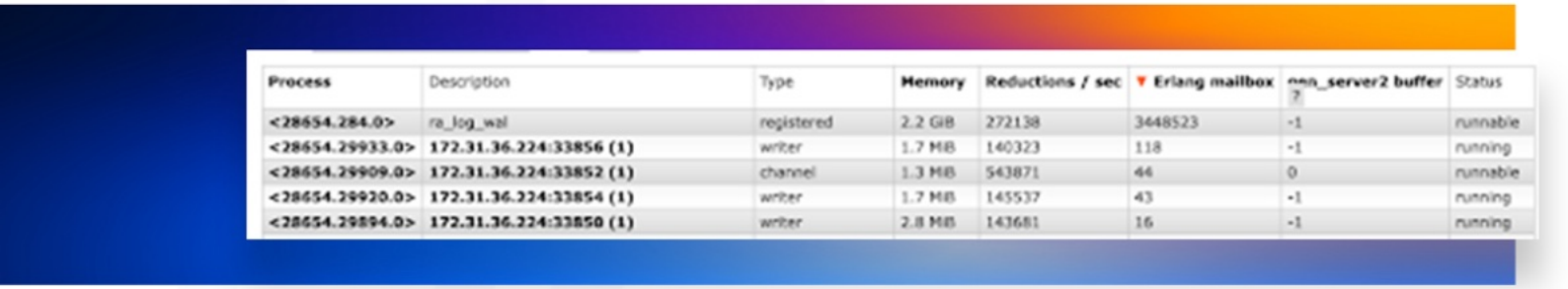
We see that correlating with traffic, as traffic increases the memory usage of one of the nodes is increasing, then reaches the memory alarm threshold (zero on the graph).



7S

The Seventh State Clusters dashboard shows that the cluster is having issues, while we need to be careful, as the memory alarm is only raised for a split second.

By investigating a bit deeper on the "Erlang Memory" dashboard we see that the heap allocator and the ETS allocator has the highest amount of memory allocated correlating with traffic. From this, we can suspect that something is storing a large amount of data in ETS and a process or multiple processes consume a lot of memory.

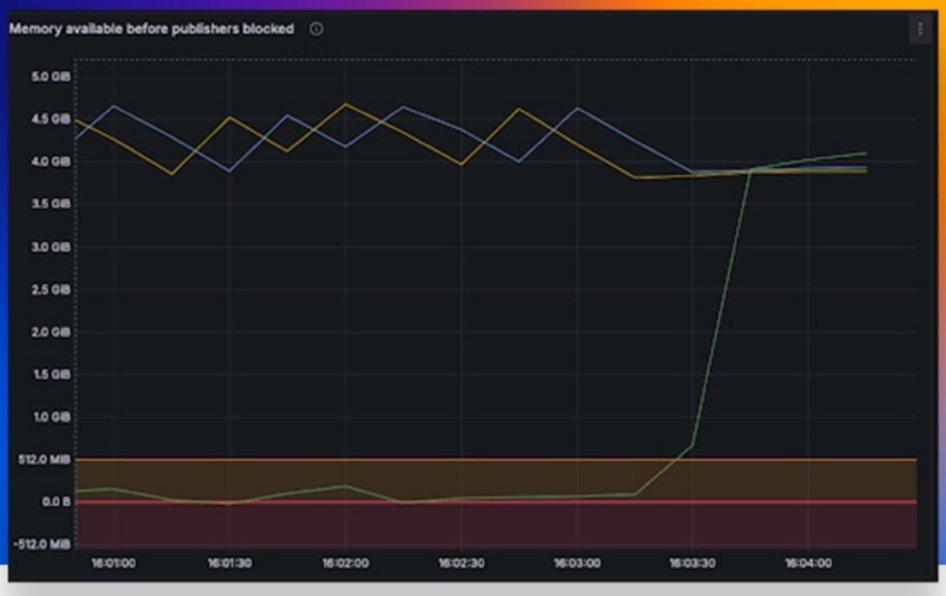


Process	Description	Type	Memory	Reductions / sec	Erlang mailbox	gen_server2 buffer	Status
<28654.284.0>	ra_log_wal	registered	2.2 GiB	272138	3448523	-1	runnable
<28654.29933.0>	172.31.36.224:33856 (1)	writer	1.7 MiB	140323	118	-1	running
<28654.29909.0>	172.31.36.224:33852 (1)	channel	1.3 MiB	543871	44	0	runnable
<28654.29920.0>	172.31.36.224:33854 (1)	writer	1.7 MiB	145537	43	-1	running
<28654.29894.0>	172.31.36.224:33850 (1)	writer	2.8 MiB	143681	16	-1	running

We can use the RabbitMQ Top plugin to investigate a bit deeper on what could be the cause of the issue. We see that the Write-Ahead Log (WAL) of the quorum queues is backed up and has high memory usage. The WAL process is also responsible for ETS memory usage.

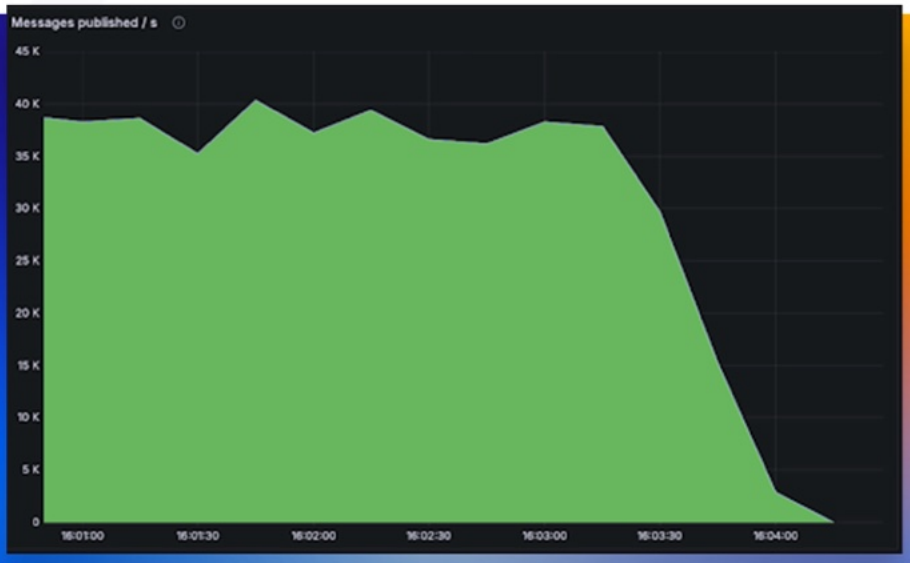
This points to the issue that RabbitMQ is overloaded and this particular node can not handle the load.

Memory alarm is not raised anymore after the load is reduced.



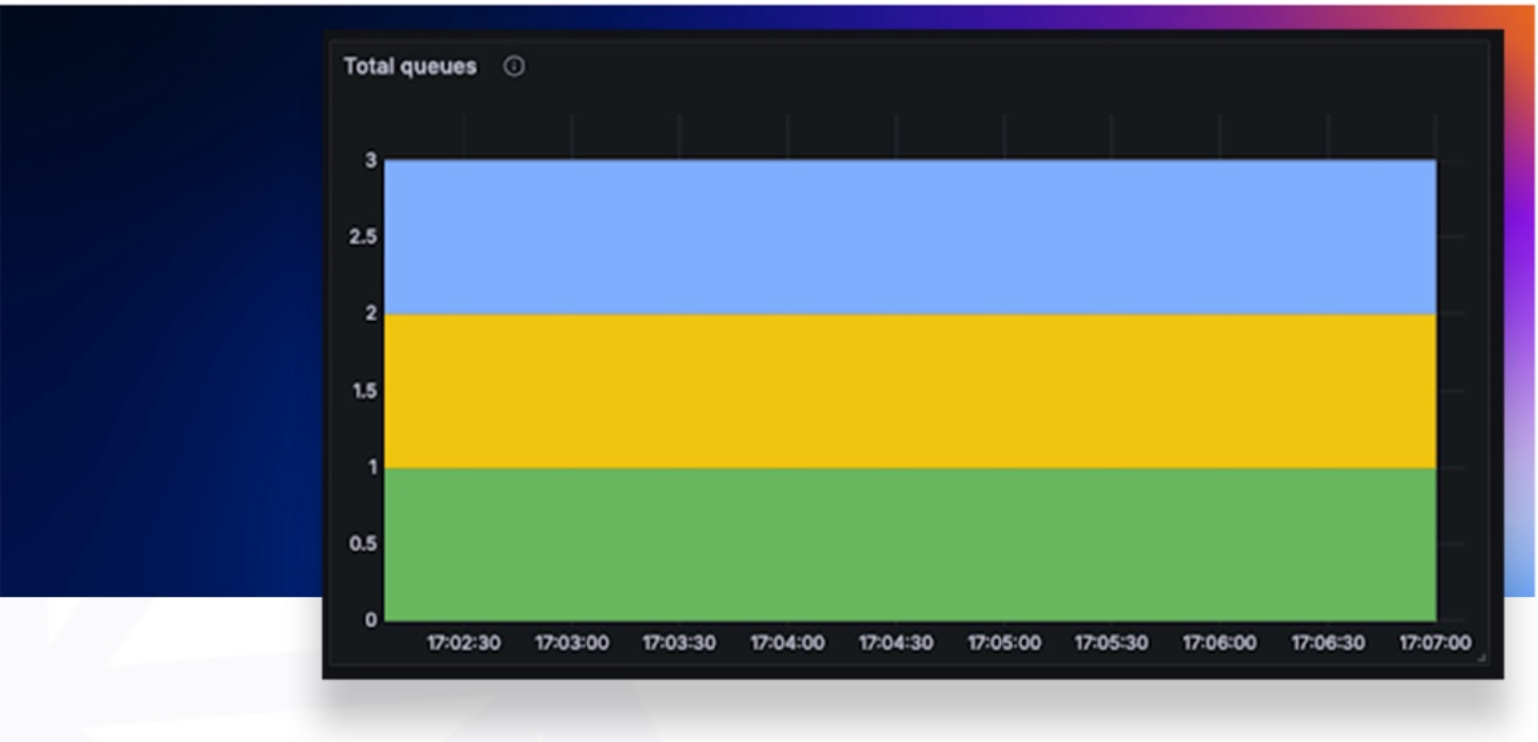
The write-ahead log on this node catches up with the messages.

After the load has stopped, the node returns to a normal level of memory usage and the memory alarm clears. Due to the non-usage of publish confirms, it takes some time for all nodes to catch up with incoming messages.



Case Study 3. Network Partition

Today RabbitMQ is mostly deployed in single datacenter installations. There can be many reasons for this but the main one is that RabbitMQ, especially Classic Mirrored Queues, do not handle failures on the network level very well. RabbitMQ needs to restart to resynchronise the internal databases, which even if the network issues were short, can create a few seconds or more of disruption in the traffic. Network partitions are not very common, but more common than we'd like, therefore it is recommended to introduce monitoring to ensure RabbitMQ recovers successfully.



In this trial, we've created three queues in the system. Initially the queue leaders are distributed equally on the cluster nodes, which we can verify using the RabbitMQ Overview dashboard.

Details

Features	arguments: x-queue-type: quorum durable: true
Policy	
Operator policy	
Effective policy definition	
Leader	rabbit@ip-172-31-25-196
Online	rabbit@ip-172-31-19-156 rabbit@ip-172-31-25-196 rabbit@ip-172-31-30-226
Members	rabbit@ip-172-31-19-156 rabbit@ip-172-31-25-196 rabbit@ip-172-31-30-226

On the Management Interface, we can drill into individual queues. At this point all members of the queues are online. For this queue pictured above, the leader is located on node `ip-172-31-25-196`.

We introduce an artificial network partition on the node, placing node `ip-172-31-25-196` into a full partition. In this case, the node loses all connectivity to the other nodes.

It takes some time for the nodes to realise that the network is down. The errors displayed are usually symmetrical, all nodes display similar messages, but this is not a given.

On node `ip-172-31-25-196` we can notice the following messages for both other nodes:

```
2024-04-12 15:18:21.311282+00:00 [error] <0.251.0> ** Node
'rabbit@ip-172-31-19-156' not responding **

2024-04-12 15:18:21.311282+00:00 [error] <0.251.0> ** Removing
(timedout) connection **
```

A timedout connection in this case means that the intra-cluster heartbeat timed out, and the TCP connection got forcefully disconnected by the runtime.

In certain error scenarios, we get an Mnesia partitioned error, which may require manual restarts of the nodes:

```
2024-04-12 15:10:15.532724+00:00 [error] <0.338.0>
Mnesia('rabbit@ip-172-31-25-196'): ** ERROR ** mnesia_event got
{inconsistent_database, running_partitioned_network,
'rabbit@ip-172-31-19-156'}
```

The node also displays that it has lost connection to the other RabbitMQ application, as well as the quorum queues stop serving traffic:

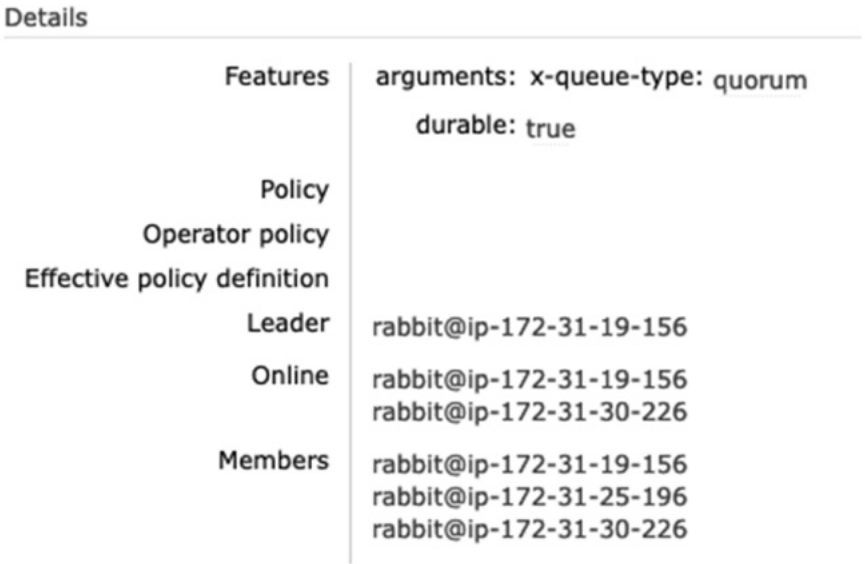
```
2024-04-12 15:18:21.311640+00:00 [info] <0.547.0> rabbit on node
'rabbit@ip-172-31-19-156' down

2024-04-12 15:18:21.311936+00:00 [info] <0.637.0> queue 'queue-3' in
vhost '/': Leader monitor down with noconnection, setting election
timeout
```

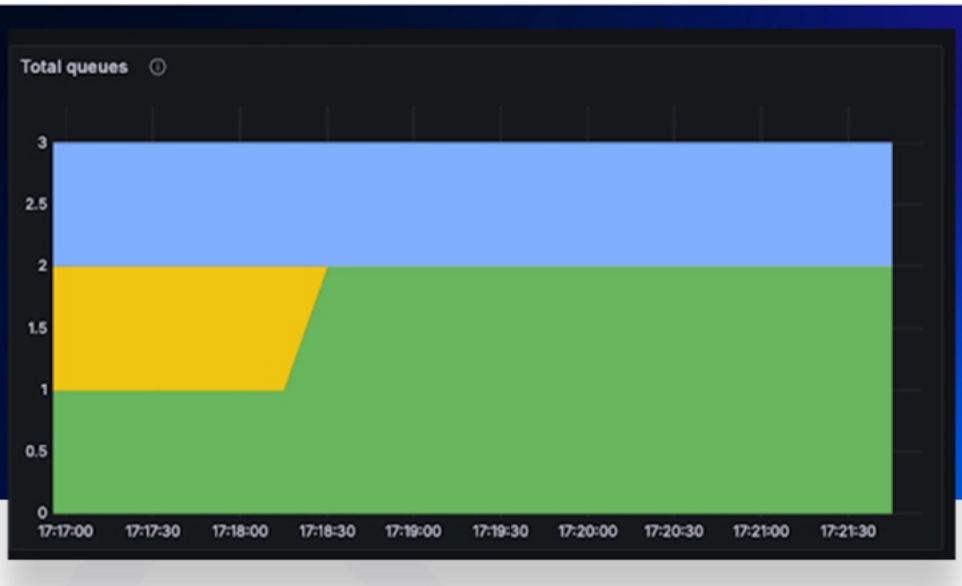
This is because quorum queues require the majority of the nodes to be online, but node `ip-172-31-25-196` is in a full partition, and has no connectivity to other replicas.

Because of how RabbitMQ works today, this node also needs to go into the paused state and will stop serving all traffic:

```
2024-04-12 15:18:22.822023+00:00 [warning] <0.547.0> Cluster
minority/secondary status detected - awaiting recovery
2024-04-12 15:18:22.822096+00:00 [info] <0.2442.0> RabbitMQ is asked
to stop…
```

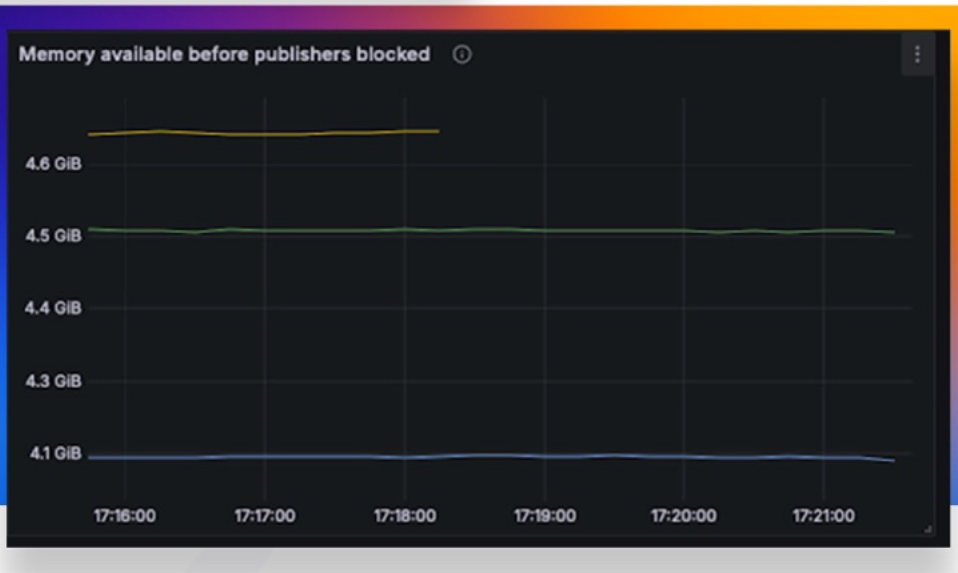
On other nodes in the cluster we can verify that the queue failed over, and now has the leader on a different node.



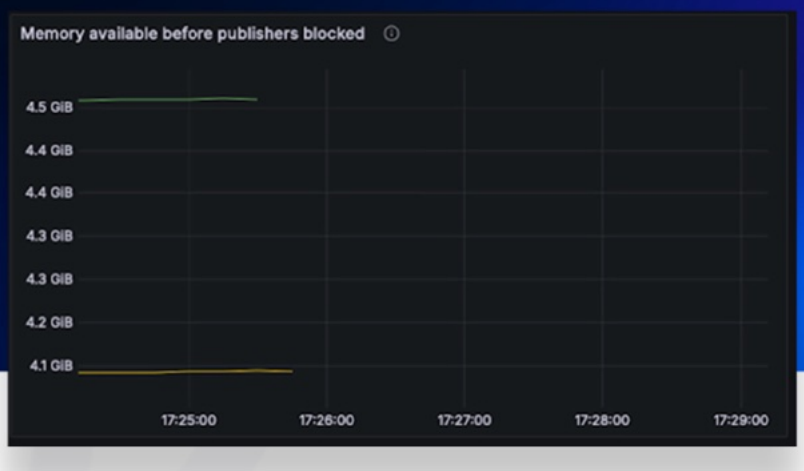


On the RabbitMQ Overview dashboard, we can observe that the paused node no longer reports metrics, and also does not host any queue leaders. The queue leader failed over to the green node.

Metrics are not received from paused nodes. The yellow line is the memory available metric of the paused node.



Let's introduce another network partition, on node `ip-172-31-19-156`. Due to how pause minority works in RabbitMQ, the cluster is fully unavailable, i.e. all nodes stop automatically:



No metrics are reported at all

After the network recovers, the nodes restart, traffic can flow again, and metrics will be reported again.

Conclusion

As we have seen in this paper, observability is a broad subject even if we restrict it to RabbitMQ. With more visibility, we can enable informed design decisions as well as shorten the reaction time to recover from incidents. We covered how the management UI can be used for monitoring purposes and how it will change in the future.

We discussed how to leverage the benefits of the industry standard ecosystem of Prometheus for monitoring and how RabbitMQ fits into this architecture. We also showed how the textual logging RabbitMQ augments the monitoring solution and what are the most important log messages that RabbitMQ generates. Throughout several examples we gave a brief overview of how to use all tools available to detect and recover from system failures.

SEVENTH STATE

THE RABBITMQ EXPERTS

SEVENTHSTATE.IO