Internet of Things

RabbitMQ plugins and management

AURIGA
the banking e-volution

THE # NEXTGENBANK

www.aurigaspa.com

Summary

- Plugins & Plugin Development
- Monitoring
  - Metric
  - Health checks
  - Monitoring tools
- Logging
- Firehose tracer

Plugins

Plugins

RabbitMQ supports plugins, which can be used to extend core broker functionalities (i.e. support for more protocols, system state monitoring, node federation, …).

A number of features are implemented as plugins shipped in the core distribution.

Plugins are activated when a node is started or at runtime when a CLI tool is used. RabbitMQ loads plugins from the local filesystem.

Plugins are distributed as archives (.ez files) with compiled code modules and metadata.

Plugins

The list of currently enabled plugins on a node is stored in a file, which is commonly known as the enabled plugins file.

Plugins that ship with RabbitMQ distributions are often referred to as Tier 1 plugins. Provided that a standard distribution package is used they do not need to be installed but do need to be enabled before they can be used.

A small partial list of core plugins shipped with RabbitMQ:

PLUGIN NAME	DESCRIPTION
rabbitmq_amqp1_0	AMQP 1.0 protocol support.
rabbitmq_auth_backend_http	Authentication and authorization plugin that uses an external HTTP API.
rabbitmq_auth_mechanism_ssl	Authentication mechanism plugin using SASL EXTERNAL to authenticate using TLS (x509) client certificates.
rabbitmq_federation	Scalable messaging across WANs and administrative domains.
rabbitmq_federation_management	Shows federation status in the management API and UI.
rabbitmq_mqtt	MQTT 3.1.1 support.

Plugin Development

Why develop a plugin?

- To enable access to internal RabbitMQ functionalities that are not exposed via the supported protocols.
- Running in the same Erlang VM as the broker to increase performance for certain workloads.
- To implement features that otherwise would have to be implemented by every application (service) in the system, creating duplication and increasing maintenance load.

Why NOT to develop a plugin

- Depending on internal RabbitMQ APIs can result in your application requiring changes when new RabbitMQ come out (including patch releases).
- A poorly written plugin can result in the entire node becoming unavailable or misbehaving.

Monitoring

Monitoring

Monitoring the behavior of RabbitMQ is essentially based on:

- **health checks**
- **metrics collection** (which is important for anomaly detection, root cause analysis, trend detection and capacity planning)

Monitoring systems typically integrate with external alerting systems and analytics tools in order to provide prompt anomaly notifications and useful insights the interested parties.

Metric Types

TYPE	DESCRIPTION	EXAMPLES
RabbitMQ Metrics	RabbitMQ specific metrics, since they are collected and reported by RabbitMQ nodes.	• # of socket descriptors used • total # of enqueued messages • inter-node communication traffic rates
System Metrics	Metrics collected and reported by the OS kernel and thus not specific to RabbitMQ.	• CPU utilization rate • Amount of memory used by processes • Network packet loss rate

RabbitMQ metrics

In a clustered environment every node can serve metric endpoint requests via the HTTP API.

- **Cluster-wide metrics** can be fetched from any node that can contact its peers. That node will collect and combine data from its peers as needed before producing a response.
- Every node also can serve requests to endpoints that provide **node-specific metrics** for itself as well as other cluster nodes. Like with infrastructure and OS metrics, node-specific metrics must be collected for each node.

Monitoring tools can execute HTTP API requests against any node.

RabbitMQ metrics (cluster)

Cluster-wide metrics provide a high-level perspective of the cluster state:

- ***Metrics on interactions between nodes***
  - i.e. cluster link traffic and detected network partitions
- ***Metrics aggregated across all cluster members***
  - i.e. the complete list of connections to all nodes

RabbitMQ metrics (cluster) - Examples

- Cluster-wide message rates
- Total # of connections
- Total # of channels
- Total # of queues
- Total # of consumers
- Total # of messages (ready plus unacknowledged)
- # of messages ready for delivery
- # of unacknowledged messages

- Messages published recently
- Message publish rate
- Messages delivered to consumers recently
- Message delivery rate

ENDPOINT	DESCRIPTION
GET /api/overview	Returns cluster-wide metrics.

RabbitMQ metrics (node)

There are two HTTP API endpoints that provide access to node specific metrics:

ENDPOINT	DESCRIPTION
GET /api/nodes/{node}	Returns stats for a single node
GET /api/nodes	Returns stats for all cluster members (returns an array of objects)

Whenever possible, monitoring tools should use the latter endpoint since it reduces the number of requests. Otherwise, the former endpoint can be used.

RabbitMQ metrics (node) - Examples

- Total amount of memory used
- Memory usage high watermark
- Is a memory alarm in effect?
- Free disk space low watermark
- Is a disk alarm in effect?
- File descriptors available
- File descriptors used
- File descriptor open attempts
- Sockets available
- Sockets used
- Message store disk reads
- Message store disk writes
- Inter-node communication links
- GC runs
- Bytes reclaimed by GC
- Erlang process limit
- Erlang processes used
- Runtime run queue

RabbitMQ metrics (queue)

Individual queue metrics are available through the HTTP API via the ***GET /api/queues/{vhost}/{qname}*** endpoint.

Examples:

- Memory
- Total # of messages (ready plus unacknowledged)
- # of messages ready for delivery
- # of unacknowledged messages
- Messages published recently
- Message publishing rate
- Messages delivered recently
- Message delivery rate
- Other message stats

Infrastructure & Kernel metrics

Infrastructure and/or Kernel metrics are typically collected on all hosts that run RabbitMQ nodes or applications.
Some of the most commonly used are:

Infrastructure & Kernel metrics

CATEGORY	EXAMPLES
CPU stats	user, system, wait & idle percentages
Memory usage	used, buffered, cached & free percentages
Virtual Memory stats	dirty page flushes, writeback volume
Disk I/O	operations & amount of data transferred per unit time, time to service operations
Disk	Free disk space on the mount used for the node data directory
TCP connections by state	ESTABILISHED, CLOSE_WAIT, TIME_WAIT
Network throughput	Bytes received/sent, maximum network throughput
Network latency	Between all RabbitMQ nodes in a cluster as well as to/from clients

Application-level metrics

Infrastructure-level and RabbitMQ metrics can show the presence of an unusual system behavior or issue but may not be enough to pinpoint the root cause.

For example, it is easy to tell that a node is running out of disk space but not always easy to tell why.

Application metrics can help identify these root causes, such as a run-away publisher, a repeatedly failing consumer, a consumer that cannot keep up with the rate, a downstream service experiencing a slowdown.

Some client libraries and frameworks provide means of registering metrics collectors or collect metrics out of the box.

Application-level metrics - Examples

Metrics tracked by applications can be system-specific, but some are relevant to most systems:

- Connection opening rate
- Channel opening rate
- Connection failure (recovery) rate
- Publishing rate
- Delivery rate
- Positive delivery acknowledgement rate
- Negative delivery acknowledgement rate
- Mean/95th percentile delivery processing latency

Health Checks

A Health check:

- Is the most basic aspect of monitoring
- Involves a command or set of commands collecting a few essential metrics of the monitored system over time and test them
- Is usually focused on a single aspect of the RabbitMQ service in order to verify whether it is operating as expected
- Is executed either periodically by machines or interactively by operators

Health Checks

Health checks can be used:

- To assess the state and liveness of a node (node health checks)
- To verify the state of an entire cluster (cluster health checks)
- As readiness probes by deployment automation and orchestration tools (including during upgrades)

Health checks may vary in complexity, starting with the most basic (which very rarely produce false positives) to increasingly more comprehensive ones (that have a higher probability of false positives).

In other words, the more comprehensive a health check is, the less conclusive the result will be.

For a more comprehensive assessment, it is advisable to collect more metrics over time. This allows to detect more types of anomalies as some can only be identified over longer periods of time. This is usually done through monitoring tools.

Management Plugin

The RabbitMQ management plugin provides an HTTP based API for management and monitoring of RabbitMQ nodes and clusters, along with a browser based UI and a command line tool, rabbitmqadmin.

In a multi node cluster, management plugin is most commonly enabled on every node. It periodically collects and aggregates data on a number of RabbitMQ metrics for nodes, connections, queues, message rates.

These metrics are exposed both to operators via the UI and monitoring systems for long term storage, alerting, visualization, analysis.

Management Plugin

The management plugin, however, has a number of limitations:

- It is intertwined with the system being monitored
- Additional overhead
- It only stores recent data (in terms of hours, not days or months)
- It has a basic user interface
- Its design emphasizes ease of use over best possible availability.
- Access is controlled via the RabbitMQ permission tags system (or a
- convention on JWT token scopes)

Furthermore, the Management plugin lacks certain features that external monitoring solutions (such as Prometheus and Grafana)

External tools

MONITORING TOOL	ONLINE RESOURCE
AppDynamics	AppDynamics, GitHub
AWS CloudWatch	GitHub
DataDog	DataDog RabbitMQ integration , GitHub
Dynatrace	Dynatrace RabbitMQ monitoring
Graphite	Tools that work with Graphite
Nagios	GitHub
Prometheus	Prometheus guide, GitHub
Sematext	SematextRabbitMQ monitoring integration, SematextRabbitMQ logs integration

Logging

Logging

Like metrics, logs can provide important clues to identify the root cause of a problem. It is always advisable to collect logs from all RabbitMQ nodes as well as any applications (whenever possible).

Default RabbitMQ logging configuration will direct log messages to a log file.
Standard output is another option available out of the box.

Several outputs can be used at the same time (in this case log entries will be copied to all of them).

Internal events

Operations performed by RabbitMQ nodes generate events which can be of interest for monitoring, audit and troubleshooting purposes and be consumed as JSON objects using a CLI command (rabbitmq diagnostics).

The internal events can also be exposed via plugin (rabbitmq event exchange) to applications for Events are published as messages with blank bodies. All event metadata is stored in message metadata (properties, headers).

Internal events

Queue, Exchange and Binding events:

- queue.deleted
- queue.created
- exchange.created
- exchange.deleted
- binding.created
- binding.deleted

Connection and Channel events:

- connection.created
- connection.closed
- channel.created
- channel.closed

Consumer events:

- consumer.created
- consumer.deleted

Internal events

Policy and Parameter events:

- policy.set
- policy.cleared
- parameter.set
- parameter.cleared

Permission events:

- permission.created
- permission.deleted
- topic.permission.created
- topic.permission.deleted

Firehose Tracer

Firehose Tracer

RabbitMQ has a "firehose" feature, where the administrator can enable

(on a per node, per-host basis) an exchange to which publish and delivery notifications should be copied

The Firehose tracer is especially useful during development or debugging,

since it allows to check every message that is published, and every

message that is delivered (even un ACKed messages).

Firehose Tracer

The Firehose publishes messages to the topic exchange **amq.rabbitmq.trace.**
Messages consumed and inspected via the Firehose mechanism are referred to as "traced messages".

Traced message routing key will be either:

- "publish.{exchangename}" (for messages entering the node)
- Or "deliver.{queuename}" (for messages that are delivered to consumers).

Firehose Tracer

Traced message headers contain metadata about the original message:

HEADER	DESCRIPTION
Exchange_name	Name of the exchange to which the message was published
Routing_keys	Routing key plus contents of CC and BCC headers
Properties	Content properties
Node	Erlang node on which the trace message was generated
Redelivered	Whether the message has its redelivered flag set (messages leaving the broker only)

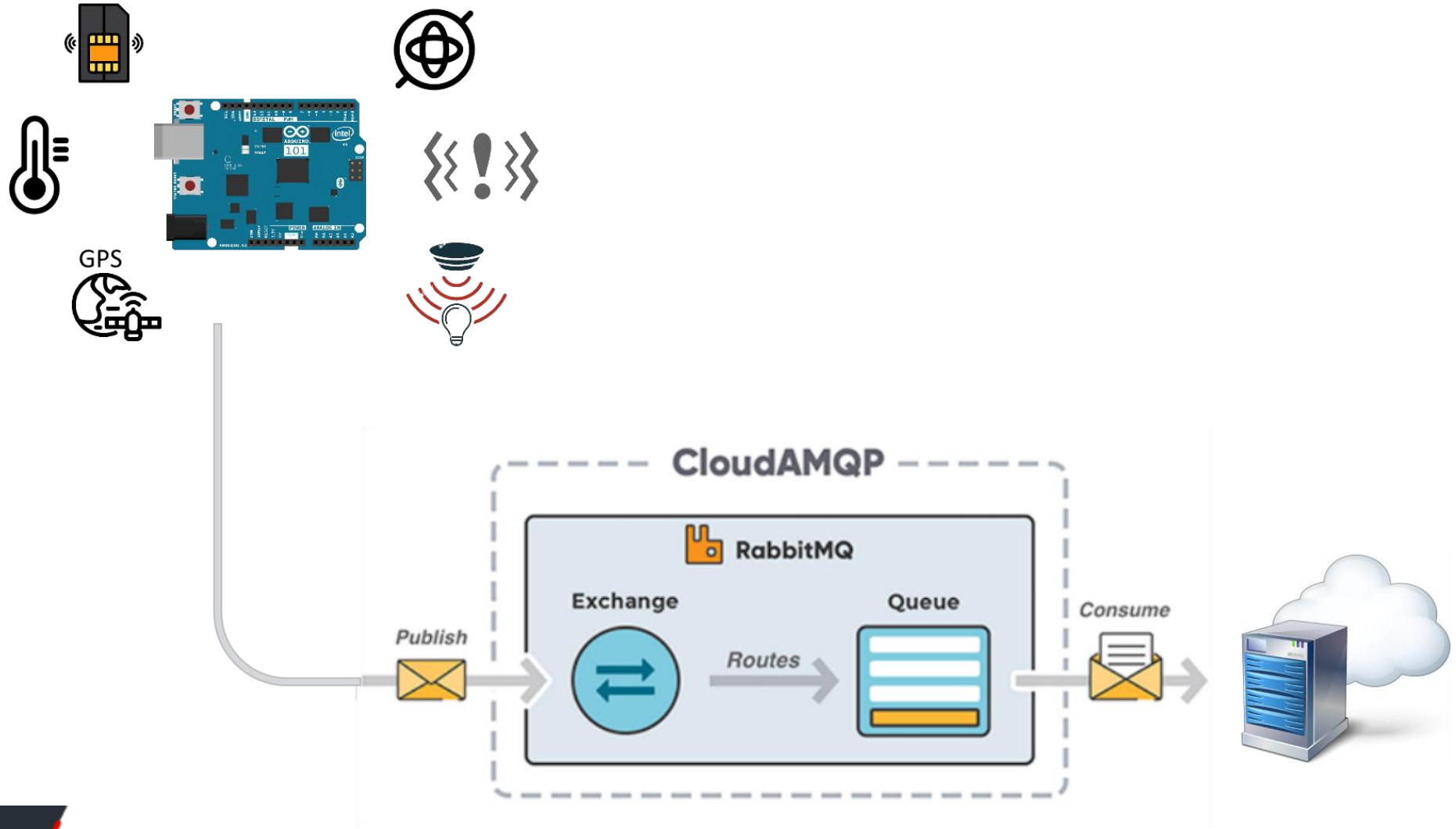
Firehose Tracer

Effects on system performances:

- When the Firehose tracer is switched off, it has no effect on performance
- when it is switched on, performance will drop due to additional messages being generated and routed

Before enabling the feature, it is advisable to decide which node, and
which vhost , should have it enabled.

Use case



COPYRIGHT

THANK YOU

Capocchiano Lorenzo

lorenzo.capocchiano@aurigaspa.com

AURIGA
the banking e-volution

THE # NEXTGENBANK

www.aurigaspa.com