ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA137

DISTRIBUTING NEAR REAL TIME MONITORING AND SCHEDULING DATA FOR INTEGRATION WITH OTHER SYSTEMS AT SCALE

F.J. Joubert*, M.J. Slabber,
SKA SA, Cape Town, South Africa

Abstract

The MeerKAT radio telescope control system generates monitoring and scheduling data that internal and external systems require to operate. Distributing this data in near real time requires a scalable messaging strategy to ensure optimal performance regardless of the number of systems connected. Internal systems include the MeerKAT Graphical User Interface (GUI), the MeerKAT Science Data Processing (SDP) subsystem and the MeerKAT Correlator and Beamformer (CBF) subsystem. External systems include Pulsar Timing User Supplied Equipment (PTUSE), MeerLICHT and the Search for Extraterrestrial Intelligence (SETI). Many more external systems are expected to join MeerKAT in the future. This paper describes the strategy adopted by the Control And Monitoring (CAM) team to distribute near real-time monitoring and scheduling data at scale. This strategy is implemented using standard web technologies and the publish-subscribe design pattern.

INTRODUCTION

MeerKAT [1] is a mid-frequency "pathfinder" radio telescope and precursor to building the world's largest and most sensitive radio telescope, the Square Kilometer Array (SKA). MeerKAT builds upon its own precursor namely KAT-7, a seven-dish array currently being used as an engineering and science prototype.

MeerKAT CAM [2] has a number of systems connected which require a constant stream of near real time sensor data updates in order to operate. There are internal systems including the GUI, the SDP subsystem and the CBF subsystem. There are also external systems including PTUSE, MeerLICHT and SETI. Many more systems are scheduled to be connected to CAM in the coming months and years. As the demand for live sensor data increases, CAM must be able to distribute sensor data to all of the interested systems without negatively impacting the performance of the running CAM system.

In the existing implementation, users would connect to CAM webservers using a Python client [3] and subscribe to sensor data that they were interested in. For each such connection, a Karoo Array Telescope Communication Protocol (KATCP) [4] connection is created to each component of interest between the CAM webservers and the CAM system. This design puts a high load on the CAM webservers as well as the CAM system.

In order to fix this bottleneck, the CAM software engineering team decided to combine our current monitoring messaging system with an existing high performance messaging system that allows for the use of a publish-subscribe pattern to enable a scalable distribution of near real time sensor data.

Several messaging platforms were evaluated. NATS [5] was found to be the most suitable to our needs. CAM is also using NATS as a messaging system to archive historical sensor data [6]. We will not be discussing the archiving of historical sensor data in this paper.

DATA DISTRIBUTION

The CAM system is distributed over multiple virtualized machines [7], referred to as CAM nodes. Each CAM node has a local NATS instance which is connected to all other NATS instances in the CAM system to form a messaging cluster. All the CAM components, which run on CAM nodes, publish messages to the local NATS instance and the NATS cluster routes the messages to the appropriate subscribers.

A group of webservers runs on one of the CAM nodes, this node is named the portal node and is collectively known as Katportal. Katportal acts as the main interface for all subscriptions by exposing websocket endpoints. Interested parties connect to the websocket interfaces and execute Remote Procedure Call (RPC) methods to subscribe and unsubscribe to subjects on the NATS messaging system. When the websocket closes all subscriptions are automatically pruned, therefore, connecting clients need to re-subscribe to subjects after a disconnect has occurred. Subjects can be individual sensor names, aggregated subjects that logically combine numerous individual subjects or application specific subjects.

Additionally, Katportal publishes to application specific subjects on NATS. These subjects include live observation scheduling data, current date and time (including local sidereal time and the current Julian date), alarms, user authentication and aggregated sensor subjects. Application specific subjects makes it more efficient for systems interested in a logical subsection of CAM to subscribe to certain types of data, which could be published as various different subjects. For example; the MeerKAT GUI [8], the operator control interface known as Katgui, will only subscribe to the observation scheduling subject in order to receive messages for all scheduling related updates. An example of an external system that requires observation scheduling data updates, is the MeerLICHT optical telescope. MeerLICHT aims to provide a simultaneous, real-time optical view of the radio (transient) sky as observed by MeerKAT.

Aggregated subjects are created specifically for use by Katgui. These subjects typically serve to update one specific

* fjoubert@ska.ac.za

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA137

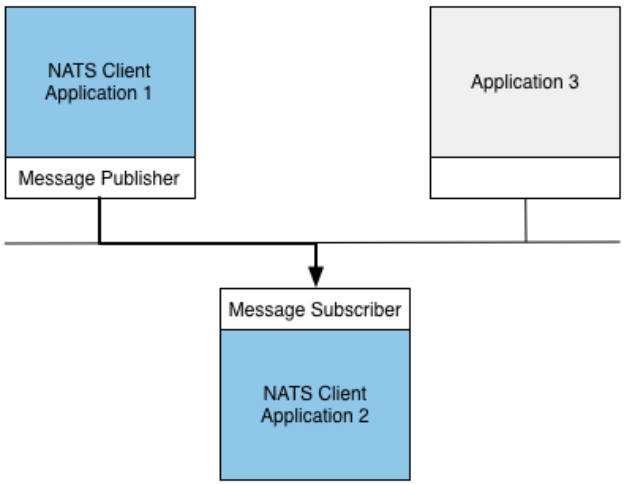

Figure 1: Message Bus.

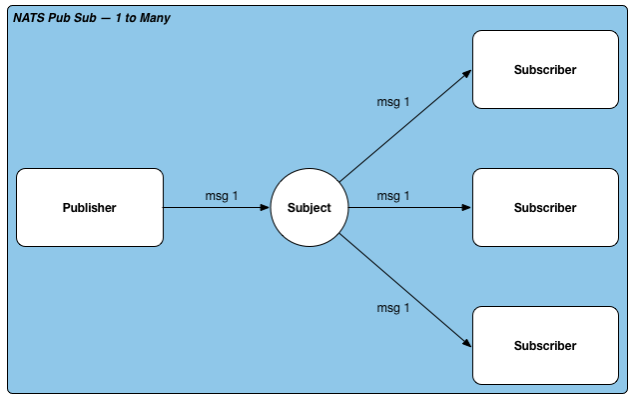

Figure 2: Publish-Subscribe Patter.

display. For example, the operator control interface will subscribe to a single aggregated subject for all receptor pointing sensors in order to update the pointing display.

Each CAM component exposes a RPC method to list sensor values and attributes by subscribing to a unique subject. The RPC method is called by a system via Katportal. Katportal executes the RPC method by publishing a message to a specific RPC subject. The message payload contains a filter string and a unique reply subject (that Katportal subscribed to before requesting the sensor list). When the component receives this message, the component compares the filter string to its own list of sensors and publishes the current values and attributes for all sensors that matches the filter string.

This allows for an extremely fast sensor value listing directly from CAM components, without the overhead of creating a direct connection to the component and requesting its current sensor values for each websocket connection to Katportal.

NATS Message Bus

NATS provides a lightweight server that is written in the Go programming language [9]. NATS server provides a publish-subscribe message distribution model, server clustering, scalability, auto-pruning of subscribers and Transmission Control Protocol (TCP) level reliability for message delivery. NATS server is a simple, high performance open source messaging system for cloud native applications, Internet of Things (IoT) messaging, and microservices architectures.

Figure 1 illustrate how NATS acts as message bus to transport messages between NATS client applications. Figure 2 illustrate how a publisher sends a message on a subject and all active subscribers listening on that subject receives the message.

The NATS messaging system allows for wildcard subscriptions, which makes it possible to subscribe to multiple subjects with a single subscription command. NATS is a fire-and-forget messaging system, if there are no subscribers listening to a subject, the message is not received.

NATS provides a lightweight Hypertext Transfer Protocol (HTTP) server on a dedicated port that allows for the monitoring of the NATS cluster. This HTTP server provides various general statistics, detailed information on current connections, information on active routes for the cluster and detailed information about the current subscriptions and the routing data structure.

NATS supports running each server in clustered mode. NATS clustered servers have a forwarding limit of one hop. Therefore a full mesh cluster is recommended for optimal performance.

To support resiliency and high availability, NATS provides built-in mechanisms to automatically prune the registered listener interest graph, which is used to keep track of subscribers. NATS automatically handles a slow consumer and lazy listeners. If a client is not processing messages quickly enough, the NATS server cuts it off (disconnected). To support scaling, NATS provides auto-pruning of client connections. If a subscriber does not respond to ping requests from the server within a specified ping-pong interval, the client is cut off. CAM NATS clients have reconnection logic built in.

NATS benchmarking has shown that a single publishing client and subscribing client can reach over 1 million messages per second. The completed MeerKAT CAM system is not expected to ever reach or exceed 50 000 messages per second.

Portal Webservers

The portal node hosts various webservers to facilitate the control and monitoring of the MeerKAT telescope via the operator control interface, known as Katgui. The various webservers are collectively known as Katportal. In this paper we will only discuss the monitoring webserver.

There is no direct interface to the NATS messaging system from outside the CAM nodes for internal and external systems. The monitor webserver exposes websocket endpoints that allows systems to subscribe, unsubscribe and to

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA137

Table 1: Example sensor sample

```
{
 "name": "wx_station_wind_speed",
 "time": 1506430493.4778199196,
 "status": "nominal",
 "value_ts": 1506430493.4774999619,
 "value": 10.3
}
```

list current sensor values and attributes of CAM components via RPC methods.

When a websocket connection is opened to the monitor webserver, a NATS client is created in the context of the websocket connection. The system that created the websocket then subscribes to subjects that it is interested in by executing RPC methods on the monitor webserver. When components publishes to these subjects, the NATS client receives the messages and sends them back to the subscriber over the websocket.

The system that created the websocket connection can request a listing of current sensor values and attributes. This is done by executing a RPC method over the websocket that is defined on the monitor webserver. This RPC method then publishes a message to NATS, which requests components to publish their sensor listings to a specified, unique subject.

The monitor webserver can be horizontally scaled over many nodes by creating multiple monitor webservers that are utilised in a round-robin fashion. This can be implemented by using a reverse HTTP proxy like NGINX [10].

Sensor Samples

Within CAM components, sensor values are continuously updating, the updates are dependent on the function and internal implementation of the components. These updates are handed to the message bus broker instance within the component. The message bus broker decides if the update should be further propagated (published) and to which subject. These decisions are based on the current system configuration. When a sensor update is to be published, additional information along with the value and timestamps are packed into a sample. The typical sample in MeerKAT CAM contains the attributes: *value*, *status*, *value_ts*, *time* [11].

Before the sample is published to the subject on the message bus, the normalised sensor name is added to the sample. This sample with the sensor name is then encoded into a JavaScript Object Notation (JSON) [12] object and used as the message. Refer to Table 1 for an example of a sensor sample message.

In the sample, the time attribute is a floating point representation of Unix time, the seconds since 1 January 1970 in Coordinated Universal Time (UTC).

Sensor Sample Subject

For each sensor two subjects are created on the message bus, one subject is for sending messages at the archive rate and a second subject to fill in samples for Katportal, which requires a faster rate of samples per second. NATS allows for wildcard subscriptions and Katportal always subscribes to both subjects using a wild card. Katstore [13], the archive system, only subscribes to the archive subjects and receives samples at the lower archive rate, while Katportal subscribes to both subjects per sensor and receives the samples from both subjects. The fill-in subject is called the normal subject.

The component that sends the sample onto the message bus will only send a sample to either the normal or archive subject. Care has been taken in the implementation to ensure that messages are not unnecessarily duplicated.

The composition of subjects takes advantage of the wild card capabilities of NATS. We found that wildcard support was lacking in many message bus implementations and even though the implementation in NATS is limited, it still provides sufficient flexibility and has a low-performance impact. NATS uses the `.` (dot) as token delimiter, `*` (asterisk) as the wildcard for one token at any level, and `>` (greater than) for any number of tokens at the end of the subject.

All sensor sample subjects have the word sensor as the first token, then normal or archive followed by the component and sensor name. To illustrate this we will use an example weather station, the component name is wx_station, with a wind speed (wind_speed) sensor. This component will publish samples to the subjects `sensor.normal.wx_station.wind_speed` and `sensor.archive.wx_station.wind_speed`. Katstore is interested in all the archive messages and subscribes to `sensor.archive.>`. Katportal has different displays and depending on the display the user is currently using it could subscribe to all the sensors on a component `sensor.*.wx_station.>` or to a single sensor `sensor.*.wx_station.wind_speed`.

Sensor Attributes

Associated with each sensor are several attributes. A sensor always has a description and a type with optional unit and parameters attributes. Katstore stores these sensor attributes in the database and long-term archive.

The message bus is also used to relay the sensor attribute information to Katstore. The attributes are static and only change with software updates, thus the attributes are sent infrequently and contribute very little to the overall traffic of the system.

Attributes are sent to a subject per component. The first token is always "attributes" and the second token is the component name. Using the weather station example again, the component (wx_station) will send the attributes of all its sensors onto the subject `attributes.wx_station`. Katstore subscribes to all the attribute subjects with the wildcard subscription `attributes.>`.

All the attributes of a sensor along with component name, the sensor's KATCP [4] name and the normalised sensor name are packed into a JSON object before it is sent out onto the message bus. Katstore is flexible in the storage of attributes and only require the name field (normalised sensor

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA137

Table 2: Example sensor attributes

```
{
 "name": "wx_station_wind_speed",
 "katcp_name": "wx_station.wind_speed",
 "component": "wx_station",
 "type": "float",
 "unit": "m/s",
 "description": "Runway anemometer"
}
```

name). Refer to Table 2 for an example of a sensor attributes message.

Since attributes are static it is not needed to send out the attributes at the same rate as that of sensor samples. Attributes are sent out the first time the component creates a sensor object and thereafter at 1 hour intervals.

Rate Limiting - Throttle

At present we have only implemented a throttling mechanism to ensure that samples are not sent out to the subjects faster that what Katstore or Katportal requires. This trivial implementation keeps an archive and normal backoff interval and will only send messages out on the respective subjects if the last message was sent more than the backoff interval seconds ago.

Future work will look at value change and only send out the sample if the value has changed and the throttle allows.

Internal and External Systems Use Case

Various internal and external systems require live updates from CAM in order to operate. Internal clients, like Katgui, requires live sensor updates in order to facilitate effective monitoring and controlling of the telescope.

CAM provides a Python client, named Katportalclient, which other systems use in order to operate. The Python client creates websockets to Katportal and subscribes to subjects in the same manner as Katgui.

CONCLUSION

A scalable data distribution system can be created with relative ease. CAM uses an existing high performance open source messaging system called NATS. With the messaging system in place, CAM components publish directly to the appropriate subjects without the need for an intermediate monitoring processes, thus greatly reducing the overall system load and improving the system responsiveness. CAM now scales quickly and efficiently to support many connected systems. NATS is also used to distribute data to the CAM sensor data archiving system.

REFERENCES

[1] R. S. Booth et al. 'MeerKAT Key Project Science, Specifications, and Proposals'. In: *ArXiv e-prints* (2009), pp. 1–16. arXiv: 0910.2935. http://arxiv.org/abs/0910.2935

[2] N. Marais. 'MeerKAT Control and Monitoring System Architecture'. In: *15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'15), Melbourne, Australia*. JACOW, Geneva, Switzerland. Oct. 2015, pp. 247–250.

[3] *Katportalclient*. Sept. 2017. https://github.com/ska-sa/katportalclient

[4] S. Cross et al. 'Guidelines for Communication with Devices'. In: *SKA SA, July* (2012).

[5] *NATS*. Sept. 2017. http://nats.io

[6] M.J. Slabber, F.J. Joubert and M.T. Ockards. 'Scalable Time Series Documents Store'. In: *16th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'17), Barcelona, Spain*. JACOW, Geneva, Switzerland. Oct. 2017.

[7] N. Marais et al. 'Virtualization and deployment management for the KAT-7/MeerKAT control and monitoring system'. In: *14th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'13), San Francisco, USA*. JACOW, Geneva, Switzerland. Oct. 2013.

[8] M. Alberts and F. Joubert. 'The MeerKAT Graphical User Interface Technology Stack'. In: *15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'15), Melbourne, Australia*. JACOW, Geneva, Switzerland. Oct. 2015, pp. 1134–1137.

[9] *Go*. Sept. 2017. http://golang.org/

[10] *Nginx*. Sept. 2017. http://nginx.org

[11] M.J. Slabber and M.T. Ockards. 'Illustrate the Flow of Monitoring Data through the MeerKAT Telescope Control Software'. In: *15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'15), Melbourne, Australia*. JACOW, Geneva, Switzerland. Oct. 2015, pp. 849–852.

[12] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Mar. 2014. DOI: 10.17487/RFC7159. https://www.rfc-editor.org/rfc/rfc7159.txt

[13] M.J. Slabber. 'Overview of the monitoring data archive used on MeerKAT'. In: *15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'15), Melbourne, Australia*. JACOW, Geneva, Switzerland. Oct. 2015, pp. 1155–1157.