RevMexAA (Serie de Conferencias), **51**, 15–23 (2019)

V Workshop on Robotic Autonomous Observatories (Mazagón, Huelva, Spain, October 16-20, 2017)
Editors: M. D. Caballero García, S. B. Pandey, & A. J. Castro-Tirado

USING OPEN SOURCE SOFTWARE AND OPEN STANDARDS FOR OPERATING ROBOTIC TELESCOPES

T.-O. Husser[1] and F. V. Hessman[1]

RESUMEN

La Universidad de Göttingen, el Observatorio McDonald de la Universidad de Texas en Austin, y el Observatorio Astronómico Sudafricano (SAAO) operan dos telescopios robóticos llamados *MONET* en el McDonald Observatory en Fort Davis, Texas (*MONET/North*), y en la SAAO en Sutherland, Sudáfrica (*MONET/South*). Después de problemas con nuestro sistema de control de observación original y algunas dificultades con otro, decidimos construir nuestro propio sistema, inicialmente proporcionando sólo la funcionalidad mínima requerida, pero permitiendo una fácil extensibilidad. Se tomó la decisión de construir sobre estándares abiertos y software de código abierto solamente, para que podamos usar tecnologías existentes y bien probadas. En este documento describiremos nuestros esfuerzos para implementar un sistema de control de observación abierto utilizando HTTP y XMPP. Además, discutiremos las posibilidades de conectar múltiples telescopios a través de VOEvents y RTML.

ABSTRACT

The University of Göttingen, the McDonald Observatory of the University of Texas at Austin, and the South African Astronomical Observatory (SAAO) operate two robotic telescopes called *MONET* at McDonald Observatory in Fort Davis, Texas (*MONET/North*), and at the SAAO in Sutherland, South Africa (*MONET/South*). After problems with our original observation control system and some difficulties with another one, we decided to build our own system, initially providing only the minimally required functionality, but allowing for easy extensibility. A decision was made to build on open standards and open source software only, so that we can use existing and well-tested technologies. In this paper we will describe our efforts to implement such an open observation control system using HTTP and XMPP. Furthermore, we will discuss possibilities for connecting multiple telescopes via VOEvents and RTML.

Key Words: instrumentation: miscellaneous — methods: miscellaneous — techniques: miscellaneous — telescopes

1. INTRODUCTION

The heart of a robotic telescope system is the Observatory Control Software (OCS) needed to operate, coordinate, and optimize the use of the various hardware and software systems. Unfortunately, many of the systems – including the OCS – often run on proprietary software, which makes changes or extensions to the system difficult or even impossible.

Our two robotic 1.2m *MONET* telescopes (Hessman & Beuermann 2002) originaly used a system that was incapable of running fully robotically and would have been difficult to extend. Thanks to our Potsdam colleagues, we have successfully been using the software designed to operate the two *STELLA* telescopes – telescopically almost identical twins of *MONET* — on Tenerife, Spain (Strassmeier et al. 2004; Granzer et al. 2012). Forgetting various hardware problems, we have encountered two long-term problems. First, we are using only a fraction of what the STELLA system offers, making the task of maintaining and extending the very complex software designed to do different scientific tasks more difficult. And second, where we have needed a functionality that the *STELLA* system does not provide, the choice of programming language (here: Java) caused some trouble for us, both due to our lack of experience with Java and due to Java's lack of good libraries for astronomy.

There are other (even free) observation control systems out there, for instance, *RTS2* (Kubánek et al. 2004), a widely used open-source system, written in C++. But, again, the choice of programming language puts some restraints on the system's extensibility due to the lack of readily available, easy-to-use astronomy and image manipulation libraries. Although the advantages of a language like Java and C++ (e.g. excellent compilers, widely used, hard-typed, fast, etc) are undeniable, both are not the first choice for today's astronomers. Instead, Python has become the lingua franca in astronomical software.

[1]Institut für Astrophysik, Universtität Göttingen, 37075 Göttingen, Germany.

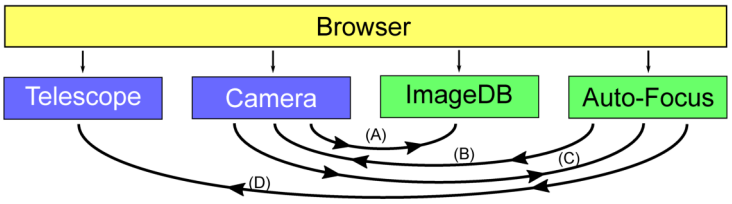

Fig. 1. A simple OCS based on HTTP with two main (Telescope, Camera) and two auxiliary (ImageDB, Auto-Focus) modules. Some possible communication between modules is indicated with arrows: the camera sends each new image to the image database (A), an auto-focus system takes an image (B), retrieves (C) and analyses it, and then sets a new focus at the telescope (D). Since all modules have an HTTP interface, communication with an internet browser is straight-forward.

V Workshop on Robotic Autonomous Observatories (Mazagón, Huelva, Spain, October 16-20, 2017)
Editors: M. D. Caballero García, S. B. Pandey, & A. J. Castro-Tirado

The simple but efficient syntax and the wide range of available open source packages like astropy, matplotlib, and pandas, makes any development fast and efficient.

Although widely used in astronomical research, there are some drawbacks when using Python for implementing an OCS. Especially its limitations on multi-threading, caused by the so-called Global Interpreter Lock (GIL), can cause some serious problems for systems that need to be able to respond in real time. The solution for this comes easily in the form of modularization: instead of having a monolithic application that operates in a single process, or splitting it up into physical components (e.g. telescope, CCD, ...), we can make the modules even smaller with each one providing only one functionality, e.g. for a CCD system this could be single modules for taking images, storing them, performing auto-focus, etc.

Using modularization, the communication between the different modules, like telescope and camera, becomes a major part of an OCS. Existing systems often use documented but still proprietary protocols for communication, most of them just sending simple text strings over a TCP connection. However, everything related to networking and internet has evolved massively over the decade, and open standards like HTTP are used uncountable times every single day and have proven themselves (mostly) robust and reliable. Therefore, a modern OCS should build on as many open standards as possible – which consequently reduces both the development time and also the chance of failure when using existing, well tested libraries.

After some experiments with HTTP, our communication protocol of choice became XMPP (Saint-Andre 2004), a protocol for instant messaging, used, for instance, by Google Talk and the Facebook Messenger. While this choice might seem peculiar at first, XMPP is widely used in realtime network applications and has direct support for remote procedure calls (RPCs) and other techniques that are useful in machine-to-machine communication. XMPP allows for a very modular way of building observation control systems, where, e.g., an auto-guiding system or a new instrument can easily be plugged in as needed. Above all, XMPP is an accepted industry standard and has libraries available in many programming languages, including Python.

In this paper, we will motivate the use of open source software and open standards in astronomical software and show that this is most often the better way than using proprietary software. We will demonstrate how standards like HTTP and XMPP, which were designed for a completely different purpose, can efficiently be used for operating a robotic telescope system. We will also describe our new OCS called *pytel* that is designed to be as simple as possible, builds on open standards only, and will eventually be released under an open source license. Furthermore, we will discuss the possibilities of VOEvents and RTML for telescope networking.

2. BUILDING AN OCS ON HTTP

The Hypertext Transfer Protocol (HTTP) is as old as the internet itself, and is mainly used for serving web pages to browsers. Nevertheless, it can be easily used as communication channel for various types of RPC systems and also has some other features that one can use when running an OCS.

HTTP is a text-based protocol that defines several commands. While HTTP GET is used for retrieving data (e.g. a web page or an image), HTTP POST can be used for sending data to the server. One of the most simple implementations of RPC is using this command together with a text-based RPC variant like JSON-RPC[2], which encodes both the request and the reply in the Javascript Object Notation (JSON, Bray 2017).

[2] `http://www.jsonrpc.org/specification`

V Workshop on Robotic Autonomous Observatories (Mazagón, Huelva, Spain, October 16-20, 2017)
Editors: M. D. Caballero García, S. B. Pandey, & A. J. Castro-Tirado

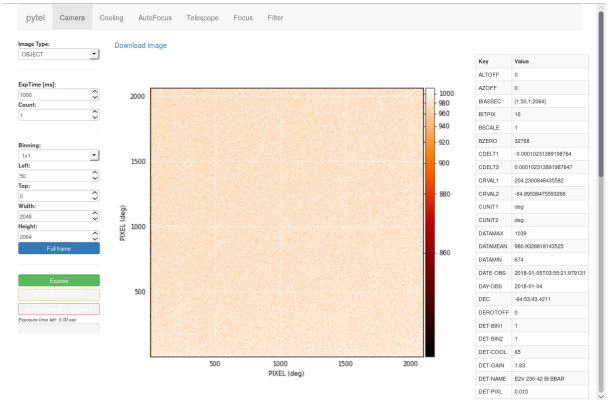

Fig. 2. The web interface for the camera as currently implemented in *pytel* showing the CCD controls on the left, a preview of the last image taken in the centre, and its FITS header on the right.

As a simple example, a request like

```
{"jsonrpc": "2.0", "method": "subtract",
 "params": [42, 23], "id": 1}
```

could return a reply like

```
{"jsonrpc": "2.0", "result": 19, "id": 1}.
```

The values of the `params` and `response` parameters can be single values, lists of values, or even dictionaries, which allows for named parameters. This way, a telescope module might be called like this:

```
{"jsonrpc": "2.0", "method": "moveAltAz",
 "params": {"Alt": 60, "Az": 30}, "id": 2}
```

Although this only works with simple datatypes like numbers and strings, the HTTP protocol itself can easily be used for sending and retrieving binary data.

Using only this simple way of communication already allows for setting up a basic OCS, as shown in Figure 1. In this example only four modules are shown, two of which operate actual hardware (Telescope, Camera). The camera module is configured to send each new image to all modules that process images, in this case only to an image database. Note that the camera module itself is only responsible for handling the CCD, it does not know about storing images, which is done by the image database. Another auxiliary module in this example is an autofocus system. It determines the optimal focus by taking a focus series, which requires setting a new focus at the telescope and taking images with the CCD. It is important to understand that in most cases the modules in the system do not know of each other: both the telescope and the camera module do not care about who controls them, whether it is a human observer or another automatic module.

Since HTTP is a widely used protocol, we can use many of its other features for implementing an OCS. For instance, password protection can easily be handled using *HTTP Basic Authentication*, or any other technique that is used a million times on websites all over the world. Furthermore, using HTTP for communication makes it easy to control the system from outside, e.g. from a web browser or even from the command line (e.g. with `cURL`). Figure 2 shows the *Angular*[3]-based web interface for the camera module as it is currently implemented in *pytel*.

However, using HTTP has some serious drawbacks. Some of them can be circumvented using modern web technology, usually resulting in a more complex code. Just to name a few issues:

- HTTP only allows for one-way communication, so a module has to continually poll (i.e. request in a regular interval) the status of another module, if interested in changes. There are ways for avoiding this like Comet[4] using long-held connections.
- HTTP is not ideal for long-lasting function calls, mainly due to undefined behaviour on terminated connections. One way of dealing with this is adding two-way communication as described above, but a more simple approach is the following: a long running RPC method immediately returns with the estimated time for the call to finish. The HTTP interface then either has a method for polling the execution status, or one for waiting for the call to finish. In both cases, a terminated connection does not end the method execution.
- HTTP has no built-in user identification or session management, although there are solutions available. This becomes important, when a module needs to know who called a method.
- HTTP has no means of broadcasting information. In the example of Fig. 1, the camera module needs to know who is interested in new images, so this needs to be pre-configured.

3. USING XMPP FOR OCS COMMUNICATION

The *Extensible Messaging and Presence Protocol (XMPP)*[5] is a technology for real-time communication based on XML. It was invented in 1999 as a

[3] `https://angular.io/`
[4] `https://en.wikipedia.org/wiki/Comet_(programming)`
[5] `https://xmpp.org/`

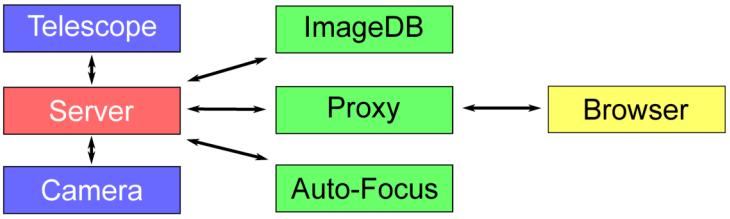

Fig. 3. The same OCS as in Fig. 1, but using XMPP for communication. There now is a central server component and an additional Proxy module that enables access to the system via HTTP, e.g. for internet browsers.

V Workshop on Robotic Autonomous Observatories (Mazagón, Huelva, Spain, October 16-20, 2017)
Editors: M. D. Caballero García, S. B. Pandey, & A. J. Castro-Tirado

protocol for instant messaging with a client called *Jabber*. Although originally designed for messaging only, due to its simple extensibility based on so-called *XMPP Extension Protocols (XEPs)*, it was quickly adapted for real-time machine communication, including, for instance, RPC. There are implementations available in all major programming languages – in Python we use *SleekXMPP*[6].

All communication in an XMPP network goes through a central server (which, e.g., can be the open source server *ejabberd*[7]), at which all clients have to log in with a username (here called a *JabberID*, having the form of an email address) and a password, before being able to send messages. This allows for unique identification of all clients in the network and also for basic security.

There are three types of messages, called *XML stanzas*, that clients and server can send:

- <message/> stanzas send human readable messages and are mainly used for instant messages (see Listing 1).
- <presence/> stanzas are the most important types of messages in XMPP. They indicate presence (i.e. availablility or online status) for a client (see Listing 2). These are the only kind of messages that are broadcasted to multiple clients at once.
- <iq/> stanzas are mostly used for machine communication. They can contain an arbitrary XML payload, e.g. for RPC. See Listing 3 for an example, which also shows the extensibility of XMPP, since the XML within the <iq/> stanza is completely arbitrary.

Listing 1. Example for a message stanza in XMPP.

```
<message from='spock@enterprise.com'
         to='kirk@enterprise.com'>
  <body>
    Live long and prosper.
  </body>
</message>
```

Listing 2. Example for a presence stanza.

```
<presence type="unavailable" />
```

Listing 3. Example of an iq stanza containing an RPC call.

```
<iq type='set'
    from='checkov@enterprise.com/rpc'
    to='warpdrive@enterprise.com/rpc'
    id='rpc1'>
  <query xmlns='jabber:iq:rpc'>
    <methodCall>
      <methodName>warpTo</methodName>
      <params>
        <param>
          <value>
           <string>Earth</string>
          </value>
        </param>
      </params>
    </methodCall>
  </query>
</iq>
```

The OCS as presented in Figure 1 can be converted to use XMPP as shown in Figure 3. Two new components are added: the server, which is third-party software and is therefore maintained externaly, and a *proxy* that allows HTTP access even for this XMPP system. The proxy is completely transparent, in a sense that it just receives JSON-RPC from the browser, translates it to XML, sends it to the corresponding client and finally returns the result, translated back into JSON-RPC.

There are some features in XMPP that help us implementing the OCS from the example. When taking an image, the camera module usually writes a lot of header keywords into the FITS file. While most of them are known by the camera, some must be requested from other modules, e.g. the current coordinates from the telescope module. In the HTTP OCS,

[6] http://sleekxmpp.com/
[7] https://www.ejabberd.im/

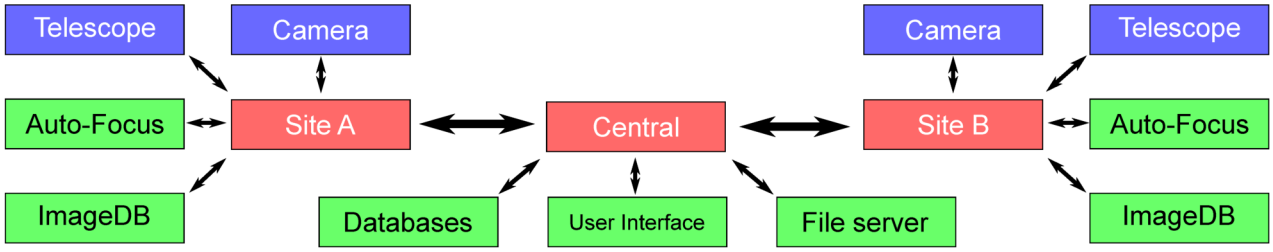

Fig. 4. Shortened example of the proposed XMPP infrastructure that we intend to use for *MONET* with two different telescope sites and a central one. All three XMPP servers are connected and each one hosts multiple modules. All user interactions takes place at the central node.

V Workshop on Robotic Autonomous Observatories (Mazagón, Huelva, Spain, October 16-20, 2017)
Editors: M. D. Caballero García, S. B. Pandey, & A. J. Castro-Tirado

the camera needs to know right from the beginning, which other modules provide those header keywords. In XMPP, we can use *XEP-0030: Service Discovery*[8]: every client can provide a list of supported features that all other clients can request. Now the camera module can just send a request into the network to get all the other clients that implement functionality for defining FITS header keywords. This way, an additional provider can be started after the camera module, without explicitly letting the camera module know about it. This becomes even easier with *XEP-0115: Entity Capabilities*[9], where the features are automatically broadcasted via <presence/> stanzas.

Another problem we had with the HTTP OCS, was figuring out who might be interested in new images from the camera module. In XMPP we can use *XEP-0060: Publish-Subscribe (PubSub)*[10]: one client can create a channel, to which others can subscribe. Whenever the owner sends a message into the channel, it is automatically broadcasted to all subscribers. So we can easily send the filename of a new image through this channel. Again, this becomes easier when using even another extension: *XEP-0163: Personal Eventing Protocol*[11], which allows each client to become its own PubSub node. Futhermore, using this extension a client can easily define, which type of channels it is interested in, and gets automatically subscribed, when a client, offering such a channel, appears. In *pytel*, we are pushing this even further by using the PubSub mechanism for event distribution. Using this method, a new image is converted into a *NewImage* event that is broadcasted into the network.

Using all these features, we end up with a *fully pluggable* system. For instance, we are running a module for science-frame auto-guiding, that works on the images from the science camera. The module subscribes to *NewImage* events and therefore gets notified whenever a new image is taken. Then it does a cross-correlation with a reference image and moves the telescope accordingly. Note that using Service Discovery, the telescope module can be discovered automatically. All it takes to activate or deactivate the auto-guiding now is starting or stopping the module, without a need for the remaining system to even know about it.

With all the advantages we get from using XMPP over HTTP, there is one drawback: XMPP has no efficient way of sending binary data. There are XEPs for sending binary data inline as Base64-encoded string[12] – but with the sizes of astronomical images this would result in massive XML files –, or setting up a direct SOCKS5 bytestream between the clients[13]. We chose a simpler approach: by adding another module to the network that also starts a HTTP server, other modules can push binary data to that server or retrieve it using a unique identifier. Using this, the camera module can just push an image to that server and retrieve an ID, which is broadcasted using a *NewImage* event. Other modules request the file with the given ID from the HTTP server and do their work.

4. OPERATING A TELESCOPE NETWORK WITH XMPP

For *MONET*, both telescope sites in South Africa and Texas are operating independently, so that they continue functioning even in the case of a network outage. The web interface for creating and editing tasks, and for downloading images, is hosted in Germany. The task database is synchronized in short intervals between there and the two sites. New images are copied automatically after they are finished.

[8] https://xmpp.org/extensions/xep-0030.html
[9] https://xmpp.org/extensions/xep-0115.html
[10] https://xmpp.org/extensions/xep-0060.html
[11] https://xmpp.org/extensions/xep-0163.html
[12] https://xmpp.org/extensions/xep-0047.html
[13] https://xmpp.org/extensions/xep-0065.html

Listing 4. Example XML for a VOEvent. Note that the schemaLocation has been shortened.

```
<VOEvent id="ivo://az.ca.saao.monet/VOEvent#123"
          role="observation" version="1.0"
          xsi:schemaLocation="http://www.ivoa.net/...">
 <Who>
  <PublisherID>ivo://az.ca.saao.monet</PublisherID>
  <Date>2018-01-01T12:00:00</Date>
 </Who>
 <What>
  <Param name="RA" ucd="pos.eq.ra" unit="deg" value="217.42895219"/>
  <Param name="Dec" ucd="pos.eq.dec" unit="deg" value="-62.67948975"/>
  <Param name="magnitude" ucd="phot.mag;em.opt.V" unit="mag" value="11.13"/>
 </What>
 <Why>
  <Concept>Transit event</Concept>
 </Why>
</VOEvent>
```

All this functionality can also be achieved using XMPP only, which is actually designed for a multi-server environment. In this case, the similarity to Email is not only due to *JabberIDs* looking like email adresses: a client always sends a message to its own server, which relays it to the recipient's server, that finally delivers the message to its destination. In case of *MONET*, we are planning to set up three inter-connected XMPP networks: monet.saao.ac.za in South Africa, monet.as.utexas.edu in Texas, and finally monet.uni-goettingen.de as central site in Germany (see Figure 4).

In this scenario, both telescope sites work independently, but can still be accessed directly via XMPP from the central site. This would mainly be used for distributing tasks, which can easily be achieved by adding new modules at both sites that implement a task database and adding RPC methods for creating and deleting tasks. For copying images, a module at the central site can explicitly subscribe to *NewImage* (or other) events from both sites, copy the associated files, create entries for them in the database, and, for instance, run a reduction pipeline.

These connected XMPP networks can also be of use for daily maintenance. After all, XMPP primarily is a protocol used for instant messaging, so we might as well use it that way. There are XMPP/Jabber clients for every major operating system, including those for smart phones and tablet computers. This allows an administrator to have a device connected to the network in his or her pocket at all times. By running a module at the central site that subscribes to certain events (especially those associated with errors in the system) and relays them as text message to the connected administrator, an immediate response is possible.

5. USING VOEVENTS FOR SIMPLE INTER-TELESCOPE COMMUNICATION

This networking of telescopes works fine as long as they are all controlled by the same organization. For a more heterogeneous network, some more level of authentication and access control would be necessary, given that the system, as described above, gives full control to everyone connected to the same XMPP network. Therefore, for these cases another approach is required, still keeping in mind the requirement for only using open standards.

VOEvents (Seaman et al. 2011) have been specified by the *IVOA Time Domain Interest Group* in order to have some means for publishing transient events in real-time, with the implied request for follow-up observations. So far they have widely been used within the GRB community, but there are also networks specialized on optical transients, like, for instance, the *Catalina Real-time Transient Survey* (Djorgovski et al. 2011). Libraries for handling VOEvents are available in many programming languages – for Python there is *voevent-parse*[14] for creating and parsing VOEvents, and *Comet*[15] for sending and retrieving them.

Listing 4 shows an example for the XML describing a simple event, including time, coordinates, magnitude, and type of event. While many more details

[14] https://github.com/timstaley/voevent-parse
[15] https://github.com/jdswinbank/Comet

V Workshop on Robotic Autonomous Observatories (Mazagón, Huelva, Spain, October 16-20, 2017)
Editors: M. D. Caballero García, S. B. Pandey, & A. J. Castro-Tirado

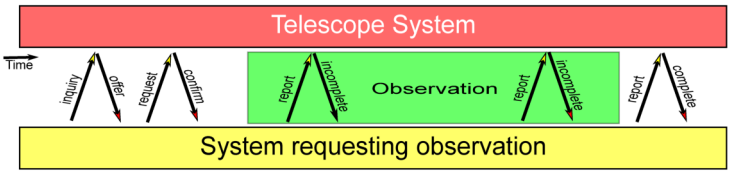

Fig. 5. Timeline for the negotation for an observation between requester and OCS using RTML.

V Workshop on Robotic Autonomous Observatories (Mazagón, Huelva, Spain, October 16-20, 2017)
Editors: M. D. Caballero García, S. B. Pandey, & A. J. Castro-Tirado

can be specified for an event, this example already satisfies the requirements for starting automatic observations. A telescope picking up this event can estimate the required exposure time (if more configuration is needed, maybe some kind of template observation based on the event type can be used) and take an image at the given coordinates. This is easily done for fully autonomous and queue based observatories, but there are possible solutions even for a manual observer, e.g. some form of user interface that shows incoming VOEvents.

Instead of using VOEvents for broadcasting events, it can also be used for targeting a single telescope and triggering an observation. Using this approach, requesting observations from a telescope is simple and easy to implement. Unfortunately, this also leads to some drawbacks, with the biggest one being the lack of any kind of return channel, i.e. the telescope receiving a request in the form of a VOEvent has no way of replying, it cannot send back the image taken, or even indicate that it is currently not operational. However, VOEvents are the perfect way to go, if a return channel is not required and observations are done on a best-effort basis.

6. THE REMOTE TELESCOPE MARKUP LANGUAGE

A possible way for the telescope to report back to the requester can be achieved using the *Remote Telescope Markup Language* (*RTML,* Hessman 2006). Like the other techniques presented in this paper, RTML is also based on XML. An RTML document defines a state than gets changed by both participants (requester and requestee) over the course of the negotiation for an observation.

As Figure 5 shows, this negotation always starts with an *inquiry*, which is a first informal request (like "Are you in principle willing to do this observation for me?"). The requested system can either *deny* the inquiry or return an *offer*, which, in the most simple case, is just the inquiry again. Next, the formal *request* is sent, which can either again be *denied* or *confirm*ed. From now on, the requester can always poll the system for a *report*. After the confirmation, at some point in the future the observations will start. The report will be answered depending on the current state of the observation (like *complete*, *incomplete*, *failed*), eventually describing a way for receiving the requested data (e.g. simple URLs), in case the observation has been successful.

Listing 5 shows an example for a request for an observation using RTML. Even with only using a small fraction of the possibilities offered by RTML, this already allows for defining a full configuration as required by many observation control systems: in addition to the coordinates that we already had defined in the VOEvents, we can also add binning and filter, time constraints, and even requests for calibration data.

If the requested telescope can work with this request as defined, it will return the same document, with the *mode* attribute in the root element changed to *confirm*, and start the observation. Polling the telescope will eventually result in a reply with a *complete* state, which, again, is just the same document as in the request, but with the *Observation* (see Listing 6) and *Calibration* elements actually filled with data about the finished observation.

Since RTML is transport-agnostic, it can be sent using any means of communication – a simple solution could be using HTTP as described earlier for the case of JSON-RPC. However, as described in Sect. 3, XMPP already allows for an arbitrary XML payload in its <iq/> stanzas, so one could easily use XMPP as communication channel for RTML. For security reasons, one would separate the internal XMPP network for the OCS from the public one that accepts RTML requests – for the latter one could even use a publicly available network, operated by a third party. As with VOEvents, while this will mainly be implemented for robotic telescopes, a simple user interface for human observers is easy to implement and allows to react to observation requests.

The idea of communicating with telescopes using RTML can be pushed even further, when adding an organisation unit, some kind of *mastermind*. Imagine an observatory site with multiple telescopes, including one or more larger ones, for which obser-

Listing 5. An example for a RTML request. Note that the schemaLocation has been shortened.

```
<?xml version="1.0" encoding="UTF-8"?>
<RTML mode="request" uid="rtml://de.uni-goettingen.monet/requests/12345678"
      version="3.2c"
      xmlns="http://www.rtml.org/v3.2c"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.rtml.org/v3.2c␣...">
  <History>
    <Entry timeStamp="2018-01-01T12:00:00" />
  </History>
  <Project>
    <Contact>
      <Username>GoeObserver</Username>
    </Contact>
  </Project>
  <Schedule>
    <DateTimeConstraint>
      <DateTimeStart value="2018-01-25T22:22:22"/>
      <DateTimeEnd value="2018-01-25T23:23:23"/>
    </DateTimeConstraint>
    <Camera>
      <Detector><Binning><X>2</X><Y>2</Y></Binning></Detector>
      <FilterWheel><Filter name="V"/></FilterWheel>
    </Camera>
    <Target name="USNO␣1234567">
      <Coordinates>
        <RightAscension><Value units="hours">23.45678</Value></RightAscension>
        <Declination><Value units="degrees">+12.34567</Value></Declination>
      </Coordinates>
    </Target>
    <Exposure count="1"><Value units="seconds">40.0</Value></Exposure>
    <Observation/>
    <Calibration>
      <BiasCorrection><Count>10</Count></BiasCorrection>
      <DarkCurrentCorrection>
        <Count>10</Count>
        <ExposureTime units="seconds">40.0</ExposureTime>
      </DarkCurrentCorrection>
    </Calibration>
  </Schedule>
</RTML>
```

vation time is expensive. Therefore, for some targets additional real-time information is required, e.g. about their current magnitude.

Combining all the techniques described above, this scenario might look like this: a VOEvent for a new Supernova comes in from a public event broadcaster. Someone at the large telescope (or some automatic system) wants to take a spectrum, but needs to know first, whether it is bright enough. An observation request is sent to the observatory's *mastermind*,

V Workshop on Robotic Autonomous Observatories (Mazagón, Huelva, Spain, October 16-20, 2017)
Editors: M. D. Caballero García, S. B. Pandey, & A. J. Castro-Tirado

Listing 6. Example of a partial *complete* reply in RTML from a telescope system, containing only the information about the science data that has been taken.

```
<Observation>
  <ImageData>
    <Uri>http://monet.uni-goettingen.de/path/to/image.fits.gz</Uri>
  </ImageData>
</Observation>
```

Fig. 6. Example for handling an observation request in a heterogenous telescope network with a mastermind.

V Workshop on Robotic Autonomous Observatories (Mazagón, Huelva, Spain, October 16-20, 2017)
Editors: M. D. Caballero García, S. B. Pandey, & A. J. Castro-Tirado

which relays it to all smaller telescopes. An actual observation is negotiated with one of them, and after some time a result is returned and forwarded to the large telescope. Now the astronomer can easily decide whether or not to observe the Supernova.

This setup is shown in Figure 6. As usual, the negotiation begins with the *inquiry* step, which is relayed to all the available telescopes in the system by the *mastermind*. If none gives a positive reply, a *deny* reply is sent back to the requester. Otherwise, the *mastermind* picks one telescope that returned an *offer*. All subsequent messages from the requester are now relayed to this single telescope – and so are the replies back to the requester. Therefore, the intelligence needed by the *mastermind* is limited to selecting one single telescope, if more than one is available.

7. CONCLUSIONS

For our *MONET* telescopes we are currently implementing a new observation control system, written in Python and building on open source software and open standards only. In this paper we showed that using existing technology for creating an OCS works well and results in fast development and a robust system. The choice of XMPP and HTTP (for binary data) for the communication between modules allows us to use mainly pre-existing code, which has been well tested. Using a third-party open source XMPP server like *ejabberd* frees us from maintenance duty on such a large and complex software.

Our new OCS, called *pytel*, is already in operation at *MONET/South*, strongly coupled with the *STELLA* system: telescope control, weather, roof, and scheduling is still done using the *STELLA* system, while CCD control and auxiliary modules like auto-focus and auto-guiding are fully implemented in *pytel*. Communication between both systems has been realized using JSON-RPC over HTTP. Following our open source directive, *pytel* will be released to the public as soon as it has reached a stable state.

Furthermore, we discussed the possibilities for connecting telescopes, especially those located at the same observatory. We strongly believe that the future of astronomical transient research, with its requirement for fast responses, lies in automatic inter-telescope communication. We are working on implementing a test case, showing the strength of this approach.

REFERENCES

Bray, T. 2017, The JavaScript Object Notation (JSON) Data Interchange Format, RFC 8259
Djorgovski, S. G., Drake, A. J., Mahabal, A. A., et al. 2011, arXiv:1102.5004
Granzer, T., Weber, M., & Strassmeier, K. G. 2012, ASInC, 7, 247
Hessman, F. V. & Beuermann, K. 2002, ASPC, 261, 674
Hessman, F. V. 2006, AN, 327, 751
Kubánek, P., Jelínek, M., Nekola, M., et al. 2004, AIPC, 727, 753
Saint-Andre, P. 2004, Extensible Messaging and Presence Protocol (XMPP): Core, RFC 3920
Seaman, R., Williams, R., Allan, A., et al. 2011, IVOA Recommendation 11 July 2011
Strassmeier, K. G., Granzer, T., Weber, M., et al. 2004, AN, 325, 527