Revista Mexicana de Astronomía y Astrofísica Serie de Conferencias (RMxAC), **59**, 209–215 (2025)

https://doi.org/10.22201/ia.14052059p.2025.59.34

VII Workshop on Robotic Autonomous Observatories (October 16-20, 2023)
Editors: Maria Griffsevich, Alberto J. Castro-Tirado, Petr Kubánek, Shashi B. Pandey, and David Hiriart - DOI: https://doi.org/10.22201/ia.14052059p.2025.59.34

SOFTWARE COMPONENTS OF THE INTELLIGENT OBSERVATORY

C. H. D. van Gend[1], S. B. Potter[1,2], N. Erasmus[1], S. Chandra[1], M. Hlakola[1], H. Worters[1], and R. Julie[3]

RESUMEN

El Observatorio Inteligente (OI) es un proyecto del Observatorio Astronómico de Sudáfrica (SAAO) para optimizar la flexibilidad y eficacia de los telescopios e instrumentos en el lugar de observación. Destacamos dos aspectos del proyecto OI: la actualización del software de control y la supervisión de instrumentos y telescopios, y permitir programar y ejecutar observaciones de forma dinámica.

En cuanto al primer aspecto, analizamos la arquitectura del software que hemos desarrollado para conectar los instrumentos y telescopios, así como su compatibilidad con las aplicaciones de la red telescopios y cómo esto soporta interfaces remotas y automatizadas con ellos. A partir de esta arquitectura, hemos desarrollado una Unidad de Control Local (UCL) para cada telescopio, responsable de la seguridad del mismo, interfaces web para cada instrumento y telescopio para permitir el control directo y la supervisión de los mismos, así como un componente de secuencia de comandos (*scripting*) que permite la operación automatizada.

Para la programación de solicitudes de observación utilizamos el componente denominado *Adaptive Scheduler del Observatory Control System* (OCS), desarrollado por el Observatorio Las Cumbres. Esto permite que los observadores envíen solicitudes de observación y produce un cronograma de observaciones que puede cambiar dinámicamente durante la noche.

El sistema de secuencia de comandos sirve de unión entre el programador, por un lado, y los instrumentos y el telescopio, por el otro. Un sondeador de horarios consulta periódicamente a la UCL para obtener permiso para observar y, si se le otorga, recupera la programación más reciente del programador OCS y pasa los elementos de programación individuales a un ejecutor de secuencias de comandos. El ejecutor de la secuencia de comandos configura los instrumentos y el telescopio de acuerdo con la solicitud de observación y a continuación, ejecuta las observaciones. Una vez completadas o fallidas, el estado se devuelve al OCS para que el programa pueda actualizarse.

ABSTRACT

The Intelligent Observatory (IO) is a project of the South African Astronomical Observatory (SAAO) to optimise the flexibility and efficiency of the telescopes and instruments at the observing site. We highlight two aspects of the IO project: updating the control and monitoring software for instruments and telescopes, and allowing observations to be scheduled and executed dynamically.

For the first, we discuss the architecture of the software we have developed to connect instruments and telescopes, and how this supports remote and automated interfaces to these. Using this architecture, we have developed a Local Control Unit (LCU) for each telescope responsible for the safety of that telescope, web interfaces for each instrument and telescope to allow direct control and monitoring of these, and a scripting component which allows automated operation.

For the scheduling of observation requests, we use the Adaptive Scheduler component of the Observatory Control System (OCS), developed by the Las Cumbres Observatory. This allows observation requests to be submitted by observers, and produces a schedule for observations which may change dynamically in the night.

The scripting system provides the glue between the scheduler on the one hand and the instruments and telescope on the other. A schedule poller regularly queries the LCU for permission to observe, and if this is given, retrieves the latest schedule from the OCS scheduler and passes the individual schedule items to a script runner. The script runner configures the instruments and telescope according to the observation request, then executes the observations. On completion or failure the status is returned to the OCS so that the schedule may be updated.

Key Words: Robotic observatory — Autonomous observing — Software architecture — Intelligent Observatory

[1]South African Astronomical Observatory, 1 Observatory Rd, Observatory, Cape Town 7925, South Africa (c.vangend@saao.nrf.ac.za).

[2]Department of Physics, University of Johannesburg, Auckland Park, Johannesburg, 2006 South Africa.

[3]South African Radio Astronomy Observatory, 2 Fir Street, Observatory, Cape Town 7925, South Africa.

VII Workshop on Robotic Autonomous Observatories (October 16-20, 2023)
Editors: Maria Gritsevich, Alberto J. Castro-Tirado, Petr Kubánek, Shashi B. Pandey, and David Hiriart - DOI: https://doi.org/10.22201/ia.14052059p.2025.59.34

1. INTRODUCTION

The South African Astronomical Observatory (SAAO) operates a diverse range of telescopes at its observing station near the town of Sutherland, in South Africa's arid Karoo region. These include the 10-metre class Southern African Large Telescope (SALT), the newly acquired 1.0-metre Lesedi telescope (Worters et al (2016)), as well as the two older 1.9- and 1.0-metre telescopes (the latter two commissioned in 1948 and 1964 respectively). Additionally the SAAO shares ownership or has access to observing time on several other telescopes on the observing plateau. The telescopes are equipped with a range of instruments, including high speed cameras, low and medium resolution spectrographs, and polarimeters.

The older telescopes and instruments each had their own specific control software, interfaces, data pipelines and operating procedures, and there was no common approach to interacting with these. As we developed control software for and interfaces to newer telescopes and instruments, we wanted to follow a more uniform approach and adopt common standards for development of control and access software. We also wanted to apply these to the older telescopes and instruments.

In addition, the requirements of observers and the observatory have changed over the years. It is no longer the best use of telescope time to grant a full week or more to an observer who then travels to the Sutherland observatory and spends each night (weather permitting) at the the allocated telescope. A target might only be visible for part of the evening, or an observer might equally well prefer to observe a given target over several months. With new telescopes such as the Legacy Survey of Space and Time (LSST) and the Square Kilometre Array (SKA) we expect many new targets meriting follow-up observations to be discovered each night, and we would like our systems and procedures to have the flexibility and capability of taking advantage of these.

The Intelligent Observatory (IO) project of the South African Astronomical Observatory was set up to address these issues (Väisänen et al. 2018; Potter 2021). The IO is an effort to systematically modernize and improve the efficiency of the SAAO's telescopes and instruments. It envisages a central control system (CCS) which has oversight of all telescopes and their associated instruments, and is able to send observation requests to each telescope, and receive status updates from these.

The project requires changes to hardware, software and procedures. Among the hardware updates are those allowing remote control and instrument selection by software alone. The software updates include changes to allow safe programmatic operability, following a uniform approach. This has allowed us to build a system to receive and execute remote observation requests, and a scheduler which can produce a dynamic queue of observations to be performed.

The IO is a project with many aspects, and in this paper we will concentrate here on control software, software providing auxiliary functions, and software required for autonomous operations.

2. SOFTWARE ARCHITECTURE

A fundamental need for the IO project is that instruments and telescopes be safely operable both remotely and autonomously.

To enable this, we need a software system which allows diverse hardware components and software services to communicate with one another. These should be accessible to other software which can receive observation requests and translate these into a set of operations which result in these requests being executed.

In the absence of human control, we need to ensure system safety. An independent system able to monitor weather should be able to take control of the telescope and issue a shutdown command, regardless of other applications which might be preparing for or taking observations. This implies that multiple applications should be able to access the hardware simultaneously.

The system should be modular and extensible, allowing new components to be added without requiring significant code changes in existing systems, and it should also be possible for application-level software to be developed without extensive knowledge of the lower-level software.

To accommodate these requirements we have chosen a modular, layered, and distributed architecture (Richards (2022)). In this context, modular means that the software consists of independently developed units which can be plugged together into larger aggregations. A layered architecture is one in which communication from hardware to control passes through a number of layers, each additional layer adding different functionality. Layers typically only communicate directly with the adjacent layers above and below. A distributed architecture, in this context, is one in which an entity can be built from several independent hardware components which don't necessarily need to be on the same host.

Figure 1 illustrates such an architecture.

The instruments and telescopes typically comprise several components. The required functionality

VII Workshop on Robotic Autonomous Observatories (October 16-20, 2023)
Editors: Maria Griffsevich, Alberto J. Castro-Tirado, Petr Kubánek, Shashi B. Pandey, and David Hiriart - DOI: https://doi.org/10.22201/ia.14052059p.2025.59.34

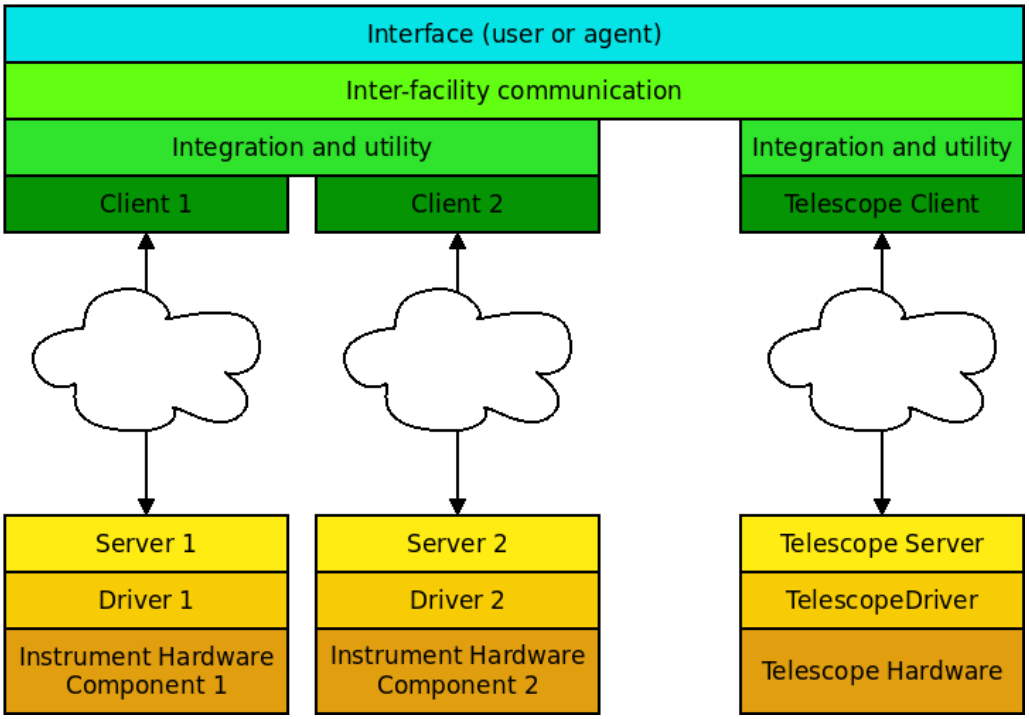

Fig. 1. A layered, distributed modular architecture.

of each hardware component is provided by a driver, embedded in server software allowing network access to these capabilities. A corresponding client for each component is able to access these capabilities. The client can be embedded in any software which requires access to the hardware item.

An integration and utility layer instantiates each of the clients representing the hardware components of the system. This layer also provides functions which depend on multiple components - for example, setting up an instrument to use frame transfer mode might require configuration of both a detector and a shutter.

Above the integration and utility layer, an inter-facility communication layer allows communication between facilities like telescopes and instruments. So for example if an instrument needs current pointing information from the telescope to populate FITS file headers, it could make a call to a `get_fits_info()` function in the telescope's integration and utility layer, via the top-level controller.

Finally, user or machine interfaces to the instrument can be built above the interfacility communication layer. Examples of these will be discussed later.

3. IMPLEMENTATION OF COMPONENTS

The architecture depicted above describes the pattern used for the interaction between hardware components and higher-level software, and the way in which we may assemble more complex software from more basic building blocks. When implementing the design, we need to make some concrete decisions.

A benefit of a distributed architecture is that the various components can be written in whichever programming language is most suitable for the task at hand. For hardware items like detectors, the driver software needs to be able to read data rapidly, and for these we use C++. Everywhere else, we have built our stack in Python.

For the client-server interaction, it would certainly be possible to develop the client and server code using low-level TCP/IP. However, there exist several packages which allow remote procedure calls (RPCs) and inter-process communication (IPC). We chose to use Apache Thrift[4] because it is a lightweight framework that makes such RPC/IPC easy. Thrift allows an interface consisting of functions and data structures to be specified in an intuitive interface definition language as shown in figure 2. The Thrift package includes a compiler which takes this interface definition and outputs server and client stub code in the languages of choice. In this case, the stub code consists of a set of empty function calls for each function defined in the interface, and

[4] `https://thrift.apache.org/`

VII Workshop on Robotic Autonomous Observatories (October 16-20, 2023)
Editors: Maria Gritsevich, Alberto J. Castro-Tirado, Petr Kubánek, Shashi B. Pandey, and David Hiriart - DOI: https://doi.org/10.22201/ia.14052059p.2025.59.34

the work to be done is mostly filling these in with working code. Thrift provides middleware modules (for Python) and libraries (for C++) which do the work of serializing and transporting function calls and data between client and server. Developers implementing clients and servers do not need to concern themselves with the details of the middleware code. Calls made on the client side are executed on the server side, and the results returned to the client.

Building the software components of the IO, following the architecture we have chosen, is a three-stage process: creating interfaces to hardware components, combining and extending individual hardware interfaces to form instrument or telescope interfaces, and finally, developing the end components.

The first stage, creating interfaces to each hardware item, means deciding which hardware functionality to make available, creating a Thrift interface definition reflecting this, compiling this to generate the stub code, and fleshing out the code to create working drivers. The drivers are wrapped in server code (also using a Thrift template) and then become stand-alone processes, listening for connections from their corresponding clients.

The second stage is to develop the integration and utility layer modules for the instruments or telescope. These modules include the client code for each associated hardware item but also functions which apply to more than one hardware item, and functions which contain business logic which is not specific to any particular hardware item. The functions of the integration and utility module, including those of the clients the module includes, form an application programming interface (API) to the particular instrument or telescope they describe.

As discussed in the architecture description, we use an interfacility communication layer above the integration and utility layer to allow telescopes, instruments and weather services to communicate with one another. This enables utility functions in the integration and utility layer to use more than one facility. Which of these get included is configured at run time by the software application.

The third and final stage is the development of these specific applications, using the interfaces to the various instrument, telescope and weather services.

It is important to note that there is a single instance of each hardware component server running at any time. The higher layers are software which provide access to the hardware, and multiple applications using these layers may be built, and may run simultaneously.

The applications we have built include:

- Command line tools to view and operate instruments and telescopes (used mostly for testing)
- Web interfaces for instruments and telescopes (providing a more user-friendly interface)
- Monitoring and safety tools
- Autonomous agents

We use the git (Chacon (2014)) version control system during development and deployment, with code stored in a central repository. Code to be deployed is tagged, and the tagged version is checked out onto the target machine. C++ code is compiled and installed using a Makefile. Where Python code is used, each subsystem has a dedicated Python virtual environment, and deployment entails installing the code into that environment. The Python setuptools package is used for overall controll of the build and deployment system.

Extensive logging is performed during runtime as an aid to monitoring, debugging and analysis of software and hardware. We also check for exceptional behaviour in code (operations timing out, system calls failing, etc) and propagate these exceptions upwards to where they can be best handled. A feature of Apache Thrift is that exceptions may be propagated across the interface; this enables exceptions arising in a driver process to be caught and displayed in a user interface. When this occurs a message can be displayed in the user interface and users may take corrective action before continuing with observations.

4. UTILITY FUNCTIONS

As part of the IO project, we have developed various functions which make life easier for astronomers observing manually, and which are necessary when observing in autonomous mode.

An example of this is an autofocus function. Previously, finding the best focus position required iterating through a list of focus positions using the telescope control, taking an image with the currently selected instrument at each position, and finally selecting the focus position which gave the best-focussed image.

If the temperature changes enough, if a filter is changed, or of course if a different instrument is selected, the best focus position needs to be determined again.

Since the software now allows instruments to communicate with and send commands to the telescope, we have been able to produce an autofocus() function in the instrument service. This iterates

VII Workshop on Robotic Autonomous Observatories (October 16-20, 2023)
Editors: Maria Griffsevich, Alberto J. Castro-Tirado, Petr Kubánek, Shashi B. Pandey, and David Hiriart - DOI: https://doi.org/10.22201/ia.14052059p.2025.59.34

```
service CameraService
{
    void set_binning(1: i8 xbin, 2: i8 ybin) throws (1: CameraException e);
    void set_readout_speed(1: ReadoutSpeed speed) throws (1: CameraException e);
    void set_gain(1: Gain gain_number) throws (1: CameraException e);
    void set_exposure_length(1: i32 exposure_length) throws (1: CameraException e);
    void start_expose() throws (1: CameraException e);
    void abort_exposure() throws (1: CameraException e);
    CameraState get_state() throws (1: CameraException e);
}
```

Fig. 2. An example of a basic Thrift interface definition

through the range of focus positions, obtaining an image at each. A source extraction is performed on each image, the full width at half maximum (FWHM) is then obtained for each source, and the average of these calculated. After iterating through the range a quadratic is fitted to the plot of mean FWHM vs focus position, and the best focus position is that which minimizes the focus measure.

For a given instrument and filter, the focus position depends linearly on the ambient temperature. Over time, using the autofocus function at a range of temperatures, we are able to acquire enough data to determine the parameters of this linear fit. This enables us to set the correct focus immediately, given the measured ambient temperature, the filter and the instrument.

A further utility function which relieves manual observers of some tedium but which is necessary for autonomous observations, is moving the telescope so that the light from the target falls on a specific pixel. This is particularly important in spectroscopy, where the light from the target must fall on the slit.

Our Mookodi instrument is both an imager and spectrograph (Erasmus et al. submitted). The spectrograph design is such that the beam is not deflected by the spectrograph optics grism and slit). These may be moved out of the beam, so that the CCD is directly illuminated.

The slit has a narrow and a wide section, the latter used when the seeing is high. The column of pixels corresponding to the slit position is known, and we have chosen two "magic pixels" corresponding to points under the narrow or wide slit sections. Pointing the telescope such that the target falls on either of these guarantees that when the spectrograph optical elements are placed in the beam, the light from the target will pass through the slit.

To achieve this, a `drop_star_on_slit()` function in the instrument service is used. This ensures that the optical elements are moved out of the beam. then makes a best effort to point the telescope in the right direction. Thereafter an image is obtained and the sources extracted. This list of sources is then sent to a WCS service - we use astrometry.net (Lang et al (2010)) - which returns the actual coordinates of the image centre. Comparing these with the coordinates of the best effort pointing, and knowing the offset of the magic pixel, we can calculate the direction and distance to nudge the telescope so that the light from the target falls directly on the magic pixel.

Again this is possible because the instrument and telescope can communicate with one another.

5. AUTONOMOUS OPERATIONS

Finally, our goal has been to prepare instruments and telescopes for autonomous observing. We have the building blocks in place, in the sense that we have interfaces to each of the instruments and the telescope, and we can do all of the tasks of pointing, focusing, acquiring a target, and taking exposures.

The general health and functioning of the IO ecosystem is made visible through a web-based dashboard, which shows the status of each service on each telescope. An alerting system sends text messages or emails to relevant parties in the event of critical processes failing. A "pre-flight" test is run in the late afternoon, ensuring that all relevant subsystems are functional, and the results mailed to the overseers of the IO.

Telescope and instrument safety is a primary concern when the telescope is operating autonomously. To address this we have a Local Control Unit (LCU) (van Gend et al (2022)) which monitors external conditions, and shuts the telescope down if required.

For the scheduling of observation requests, we use the Adaptive Scheduler component of the Observatory Control System (OCS) (Nation et al (2022)),

VII Workshop on Robotic Autonomous Observatories (October 16-20, 2023)
Editors: Maria Gritsevich, Alberto J. Castro-Tirado, Petr Kubánek, Shashi B. Pandey, and David Hiriart - DOI: https://doi.org/10.22201/ia.14052059p.2025.59.34

developed by the Las Cumbres Observatory[5]. This allows observation requests to be submitted by observers, and produces a schedule for observations which may change dynamically in the night

As observations are marked as completed, they are removed from the queue, while those marked as having failed may be rescheduled. The queue may be adjusted as new observation requests are added. Each observation request in the queue contains details about the telescope pointing, instrument configuration and observation requirements, and the time at which the observation is to be started.

We have developed a service which polls the scheduler, retrieving the queue at regular intervals (we use intervals of 1 minute). The queue is then examined to determine when the next observation is scheduled. If a previously scheduled observation is already running, the polling software does nothing and sleeps until the next interval elapses. If no observation is due in the next 20 minutes, and the telescope is open, a command is sent to shut down the telescope. If the next observation is due within the next 10 minutes, a command is sent to open the telescope. Finally, if the time for the next observation has been reached, the observation request is sent to the software which oversees its execution.

The first step in executing the observation is to translate the XML-format observation request, as sent by the OCS scheduler, to a form that our software (instrument and telescope) can act on. We produce two JSON format scripts: a setup script for the pointing of the telescope and target acquisition, and an execution script with the details of each observation to be made at the given telescope setup.

The telescope setup script contains the target right ascension and declination, the required tracking type (sidereal, non-sidereal or none) and for non-sidereal tracking, the parameters for this. For non-sidereal tracking, we currently support MPC[6] orbital elements to specify this, but we plan to extend this support to use JPL[7] orbital elements too (the latter is already supported by the OCS. If the telescope pointing needs to be finely adjusted (e.g. to put a star on a given pixel, as is required for spectroscopy, the details required for that are included.

The setup script is passed to a `run_script()` function, which passes the telescope and instrument specific parts of the script to the `stage_script()` functions in the telescope and instrument. These functions move the telescope or configure the instrument as required, but do not start any observations. When these functions have returned, the telescope is in position to begin observing .

The second script contains details of the exposures to be taken at that telescope position, including exposure time, number of observations, binning, gains, and filters. This script is passed to the instrument's `stage_script()` and after this a call is made to the instrument's `start_exposure()` function. There may be multiple such sets of exposures specified, in each case they are executed in succession.

At the beginning of each observation attempt, the OCS is notified and the observation is marked as attempted. After the observation is completed, the OCS is again notified and the observation is marked as complete. If instead the observation is unable to be completed, the OCS is notified and the observation is marked as failed, and possibly rescheduled.

6. CONCLUSIONS

We have developed a simple and flexible software architecture which we have applied to our systems, enabling remote and robotic operations

Various utility functions have been developed to replace formerly tedious and time consuming steps.

We have successfully applied this to the Lesedi telescope and the Mookodi instrument.

The telescope runs in queue scheduled mode, fully autonomously:

- The LCU enforces safe operation
- Biases are taken in the late afternoon
- Flats are taken immediately after sunset
- Items in the schedule queue are executed at the prescribed time
- We are able to observe sidereal and non-sidereal objects

We are currently commissioning Sibonise, a wide field imager on the same telescope, and will incorporate it into our autonomous observing plan when commissioning is finished.

We will then roll out fully autonomous observing to our 74-inch telescope, using the instrument selector currently being commissioned. Observations will use SHOC, a high speed camera, and SpUPNic, a medium resolution spectrograph. Following that we will turn our attention to the 40-inch telescope, and then to other telescopes.

Lastly, we have developed systems which listen for alerts of events meriting follow-up observations,

[5] `https://observatorycontrolsystem.github.io/`
[6] `https://www.minorplanetcenter.net/`
[7] `https://ssd.jpl.nasa.gov/orbits.html`

and when these are received the observation requests are added to the queue. We will be expanding these, and intend to be ready to react to alerts issued by the LSST when it comes online later in 2024.

REFERENCES

Chacon, S., Straub, B. 2014, Pro git, Apress
Erasmus, N., Steele, I. A., Piascik, A. S., et al. submitted
Lang, D., Hogg, D. W., Mierle, K., Blanton, M., & Roweis, S. 2010, AJ, 139, 1782
Nation, J., Bowman, M., Daily, M., et al. Proc. SPIE, 12186, 121860Q
Potter, S. B. 2021, Anais da Academia Brasileira de Cincias, 93, SciELO Brasil
Richards, M., 2022, Software Architecture Patterns, O'Reilly Media Inc.
Väisänen, P., Crause, L., Gilbank, D., et al. Proc. SPIE, 10704, 107040A
van Gend, C. H. D., Potter, S. B., Julie, R., et al. Proc. SPIE, 12189, 121890R
Worters, H. L., O'Connor, J. E., Carter, D. B., et al. Proc. SPIE, 9908, 99083Y

VII Workshop on Robotic Autonomous Observatories (October 16-20, 2023)
Editors: Maria Gritsevich, Alberto J. Castro-Tirado, Petr Kubánek, Shashi B. Pandey, and David Hiriart - DOI: https://doi.org/10.22201/ia.14052059p.2025.59.34