RTS2 JSON API documentation

Petr Kubánek

May 14, 2013

Contents

List of Figures

List of Tables

Chapter 1

RTS-2

This chapter provides overview of the RTS2 system. In-depth description of some of RTS2's features is beyond scope of this chapter, and is provided in the following chapters and published papers attached in appendix.

1.1 History

Remote Telescope System, 2^{nd} version (RTS2), as the 2 in the name suggests, evolved from Remote Telescope System (RTS). RTS was written by four computer science students of MFF UK, as part of compulsory team work required for completion of their university degrees.

The aim of RTS was to produce code to control a Meade LX200 mount, which was at that time available at Czech Astronomical Institute, equipped with two SBIG CCD cameras – one intended for 20 cm Meade telescope, the second for 8 cm widefield optics. Originally the group intended to write only the scheduling part of the control software and a package for processing acquired images – but due to the unavailability of control software, we decided to write the telescope and camera control part.

We decided to write the RTS control system in the Python language, as it seemed to be suitable for such a task. Image processing was written partly in Python, but mostly in Matlab. After half a year of designing, coding and testing, we were able to control the telescope and process the images which were acquired. The project was finished as expected in June 2000, and a successful final presentation was given.

The telescope was then put into routine operation, which revealed bottlenecks and bugs hidden in the control code. Petr Kubánek then decided to completely redesign the control code, taking into account the experience gained during RTS development.

The primary reason for this redesign was to make the code more portable, so it could be used on other observatories, with mounts and Charged Coupled Detector (CCD) detectors from other manufacturers. Python was abandoned

in favour of the C language. Python was only capable of throwing exceptions in run time, which is fine for some prototyping work but makes it completely unsuitable for a system which has to control a telescope, where most of such exceptions are encountered at 3 am. And it was quite hard to interface Python to low-level, mostly C, libraries used to control devices. The idea of creating a library for each kind of control (mount, CCD, ..) was introduced. That library was linked with a common executable to produce a server, which communicates with other devices using TCP/IP. From the beginning, RTS2 was designed to retrieve targets and log target observations to a PostgreSQL database, a major change from RTS which used text files for the same purpose.

RTS2 was initially designed in pure C, it means without use of Object Oriented Programming (OOP) techniques. After two years of development and active use of RTS2 on BART and BOOTES telescopes (and testing version for FRAM telescope), we decided to abandon C-purism in favour of OOP design in the C++ language. That decision has enabled us to produce code which is easily maintainable, and the progress achieved after this change shows that this decision has paid off. It may have been more appropriate to rename the new code RTS3, since there is not much remaining from the original C design, but the name RTS2 was retained.

RTS2 accumulated support for more and more devices, while its observation control part handles more and more tasks. The best demonstration of the progress achieved in RTS2 development is its growing use in the astronomical community.

Growing number of observatories running RTS2 is show in table 1.1.

Table 1.1: Observatories using RTS2

Observatory	D[1]	From	remarks
BART	25cm	12/2002	
BOOTES 1	30cm	6/2003	
BOOTES 2	30cm,60cm	7/2003	changed to 60cm in 2009
Watcher	16"	5/2005	the first overseas deployment
FRAM	25cm,30cm	7/2005	
Vermes Observatory	30cm	6/2008	the first use of RTS2 outside IAA-UCD-AU collabo
Bootes 3	60cm	3/2009	
CAHA 1.23m	1.23m	5/2009	the first LN2 cooled CCD
FLWO 1.2m	1.2m	10/2010	only scheduling and overall robot management
Aosta Valley	4x 20cm	4/2011	
RATIR 1.5m	1.5m	4/2012	

1.2 Architecture

RTS2 ecosystem consists of independent programs. Those programs call functions from various RTS2 specific libraries. The programs communicate through RTS2 protocol, acting both as a client and as server. The protocol includes commands for device discovery, authentification and authorization, as well as variable discovery, variable metadata transport and variable change and increment commands.

TCP/IP connection is created between any RTS2 program wishing to communicate with the other RTS2 program. Device discovery and authentification is handled by rts2-centrald daemon. Devices can connect to multiple rts2-centralds, so for example weather station data can be easily used by multiple telescopes operating in an observatory site.

Users programs communicate with RTS2 either directly, using the RTS2 protocol, or through some bridge inside RTS2 system, translating access to the other protocol. JSON and XML-RPC bridges are readibily available.

1.2.1 Layers

RTS2 programs can be split into layers. Bottom layer encapsulated hardware interfaces into daemon, and provides access to hardware functionalities through RTS2 protocol. The top layers includes programs for monitor and changing RTS2 environment, providing users means to control the observatory. To this layer belongs programs which communicates directly with the user, as well as deamons wrapping access to the observatory to some other protocol, such as JSON. Then there is scheduling and executing layer, which control autonomous operations of an observatory. Layers are described in figure 1.1.

1.3 Namespaces

RTS2 classes are distributed into namespaces. Classes belonging to a namespaces are performing similar operations. The following namespaces are present inside RTS2:

rts2core basic RTS2 components - block, application and connection classes

rts2fits image manipulation

rts2script scripting and queues

rts2db database access

rts2teld telescope drivers

rts2camd camera drivers

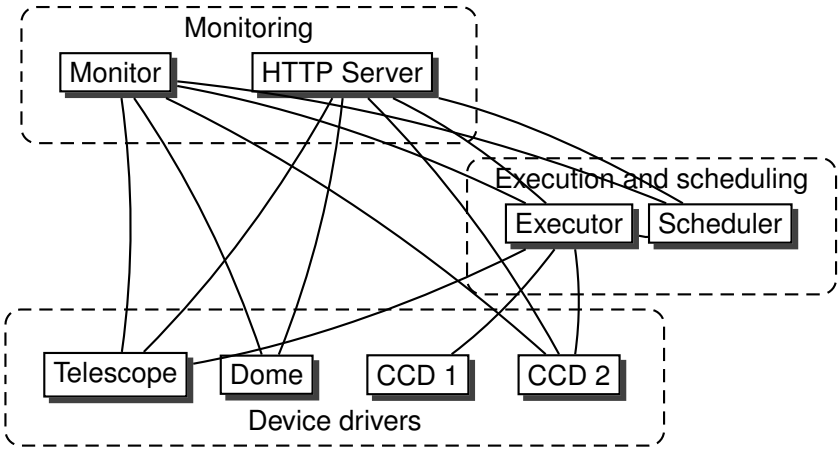

Figure 1.1: RTS2 layers

"Monitor" and "HTTP server" are compontents providing user access. "Telescope", "Dome", "CCD 1" and "CCD 2" are acting as interface between RTS2 and hardware. "Executor" and "Selector" are handing observation executing and scheduling.

rts2focusd focus drivers

rts2filterd filter wheels

rts2dome observatory enclosures

rts2sensord sensor drivers (devices which do not fit into any other category)

rts2plan execution and scheduling

1.3.1 Loosely or hard typed?

"Pořádek je pro blbce, inteligent zvládne chaos."

Czech saying; "Order is for idiots, smart guys can master chaos."

An important feature of the RTS2 and its protocol is that everything between RTS2 programs is loosely typed - there are not any standart interfaces as required by for example in an ASCOM environment. This solution comes from the idea that to control and display the state of a device, one need just three operations - read a value, write a value, and execute a command. RTS2 benefits from loose typed interfaces in multiple ways:

- Extension of an existing component is trivial and can be done fast, as it does not require change of an interface description.
- Advanced features of a component can be supported without need for defining a special interface file.

In a sense, the protocol provides for an introspective interface, with interface description send by the protocol. This allows the programmer to change the interface by typing the code without worring about interface description details. The interface is still visible from the code and can be extracted from the device using the RTS2 protocol.

1.3.2 Version systems

The evolution of both the late RTS and the whole RTS2 package can be tracked in Concurrent Versions System (CVS) or Subversion (SVN) repository, which we use for version control.

It is absolutely necessary to use CVS or similar tools for a project of this size, as maintenance and upgrading the code on all sites running RTS2 would be a nightmare without it.

1.4 Class hierarchy

RTS2 is designed using OOP. It uses its own classes, designed to make developer life easier. Thanks to an extensive set of classess allowing to communicate with devices over various interfaces, a new device driver can be developed in few minutes.

There is an abstract parent class, called Object. It provides support for sending events with its virtual method *postEvent*. Child classes overrides this method to handle custom events. Events are distributed only inside a RTS2 daemon, and are not propagated through RTS2 protocol to other system components. Events are extensively used for execution and synchronization control.

1.4.1 Object hierarchy

1.4.2 Device drivers hiearchy

1.4.3 Connection classes hierarchy

1.4.4 Value class hierarchy

Value classes are used to represet components variables. RTS2 provides

Class diagram depicting all classes is provided in figure 1.5.

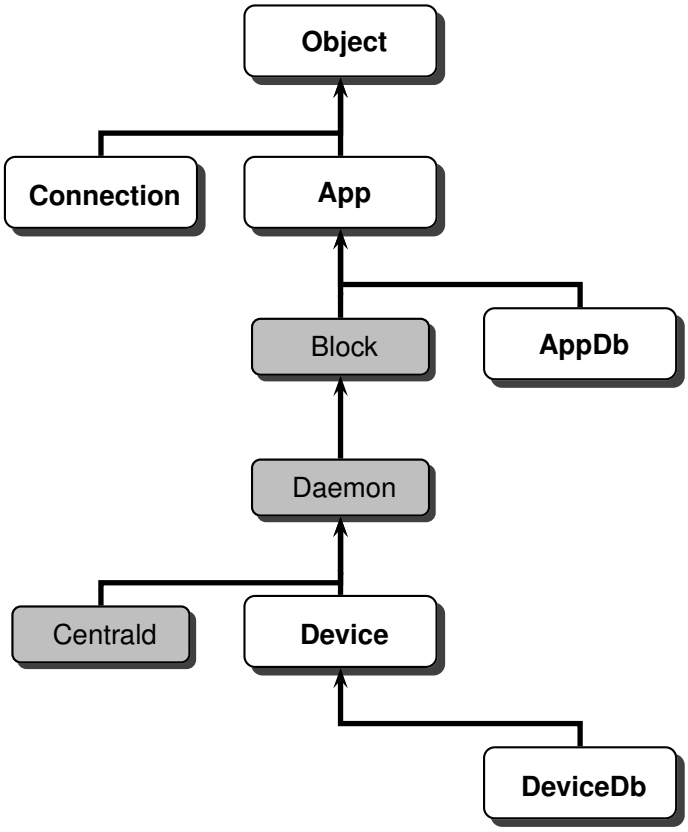

Figure 1.2: Basic classes
Core system classes. Every class capable to receive RTS2's *Event*s inherits from *Object* class, which declares a common *postEvent* method.

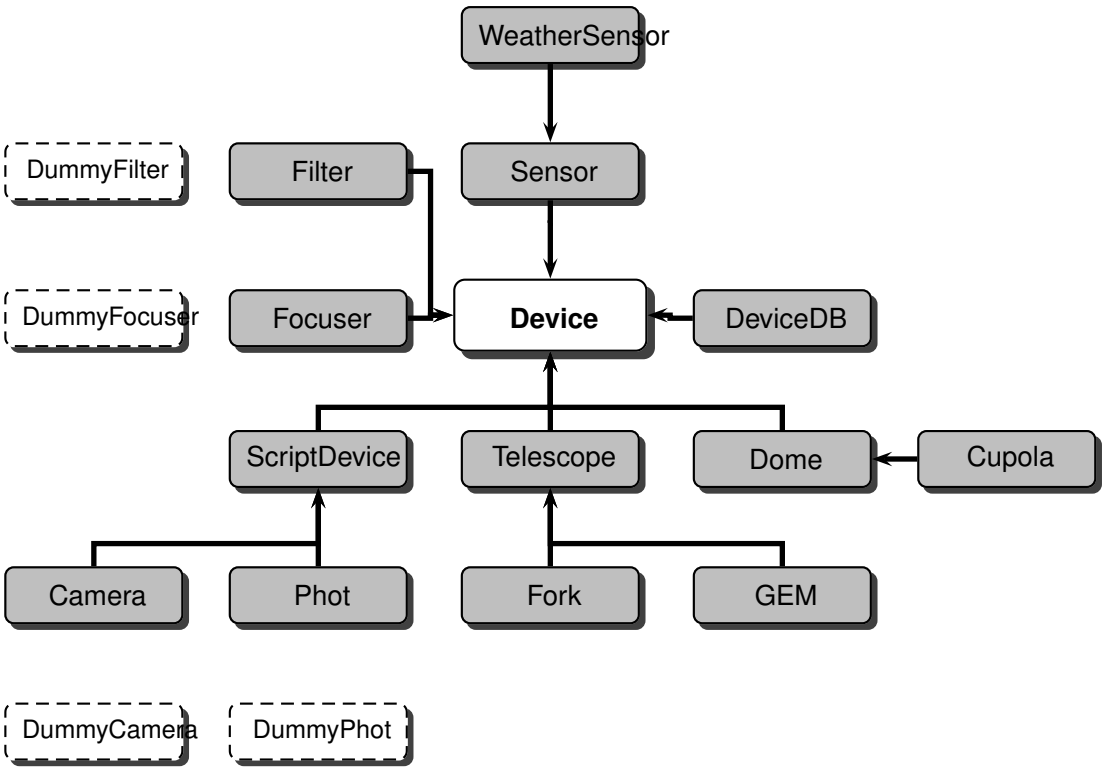

Figure 1.3: Device classes
Classes representing devices found in observatory. Beside obvious inheritance from *Device* class, notice how *Telescope* parenting *Fork* and *GEM* classes, which provides function for fork style and **G**erman **E**quatorial **M**ount.

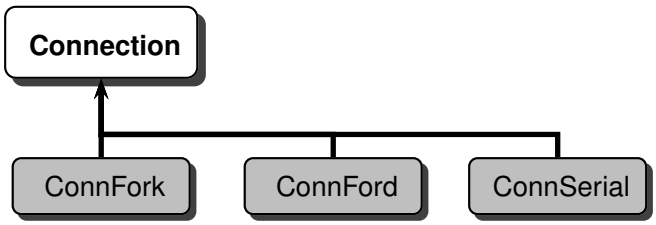

Figure 1.4: Connection classes
Relation between connection classes. As *Connection* inherits from *Object* class, *Event*s can be posted to connection classes. Connection classes are usually registered for idle calls using *addConnection* method.

Figure 1.5: Value Classes

1.4.5 Database classes

Database classes maps

1.5 RTS2 libraries

RTS2 core code is split into libraries. Those provide core functionality, which is used and extended by the programs using them. Detailed description of the libraries is provided in next sections.

1.5.1 librts2

This library contains core classes. Object, Block, Application and Daemon classes are provided by this library. It also contains abstract classes for various types of devices present in the RTS2 environment - mount, camera and filter wheel, to name just a few.

1.5.2 libxmlrpc++

1.5.3 librts2db

This is a database handling library. It provides functions to access and manipulate data stored in the SQL database used by the RTS2. Target and Observations classes are defined in this library.

1.5.4 librts2fits

Provides access to FITS images. It wraps CFITSIO[24] C functions into C++, and extends them with RTS2 specific functions - for example the method to fill FITS header with keywords defined in template. Image class is declared in this library.

1.5.5 librts2script

This library contains support for device scripting. Script class, which handles script parsing and execution, as well as rts2script::Element class, are provided there. The library also contains support classes for multiple metaqueues scheduling.

1.5.6 librts2scheduler

The library provides functions for NSGA-II observation scheduling.

1.5.7 Python libraries

1.6 RTS2 daemon design

RTS2 daemons are by design single thread program, multiplexing on open sockets by means of select system call. Sequence of calls called during device operation is described in figure 1.6.

Each RTS2 daemon has a state. State is a bitmasked integer, which holds various information about the module - if an error was detected during its operation, if module thinks weather is bad to prevent observatory operation and similar. There are generic states, which are common across modules, and module specific states, which are unique to a given module type. There are methods to change the device state, which propagates changes to all connections, as well as functions to decode the state into human readable string.

Apart from the state, each module has an assigned type. The type specify the "kind" of the module. Types are listed in table **??**.

1.7 Protocol

RTS2 uses simple, ASCII based protocol. The protocol is modelled after SMTP. Protocol commands are separated with newline. The protocol allow for transfer of binary data - a feature used during transfer of camera data to process responsible for processing the data, as for example writing the data to the FITS file. Protocol is described in detail in appendix A.

1.8 Device Drivers

RTS2 provides modules for device control. The code provides an abstract class, which contains all reasonably common code for device operation. The abstract class usually contain a few pure virtual methods, providing guidelines on which methods **must** be implemented by the device drivers.

1.9 Monitoring

Monitoring is provided primarly by rts2-mon. The monitor connect to all modules available in the system, and display changes of their states and variables to user.

rts2-mon is nice when user has direct access to RTS2 environment, and is usually run over a SSH connection to remote observatories. rts2-xmlrpcd provides JSON interface for RTS2, with publish-subscribe interface for rapid dissemination of updates to variables of user interface.

1.9.1 Executing

Normal flow of target execution is depicted in diagram 1.7. After target is created,

1.9.2 Scheduling

1.10 Simulated devices

Das ganze tschechische Volk ist eine Simulantenbande.

Dr. Bautze in "Good Soldier Švejk" by Jaroslav Hašek

Simulated[2] devices are primary used for testing. The code simulates device operation, without access to hardware. RTS2 contains dummy devices representing camera, telescope, cupola, filter wheel, focuser and sensor.

During software tests, real devices are emulated with simulated devices. Simulated devices correctly responds to command issued by other RTS2 modules, and changes states in similar fashion as real devices.

Simulated devices are particularly usefull to test changes to RTS2 protocol. The test environment can be setup without access to real hardware, protocol changes tested and when all the tests are done, new code can be released for use on observatories.

1.11 Synchronization

The part developed for BNL, but usefull for others as well.

1.12 Database

1.13 User interface

1.14 Image processing

1.15 Virtual Observatory

relation of VO to RTS-2.

[2] the devices are called "dummy" in RTS2 code. Dummy term was used, as the devices originally did not implement any logic.

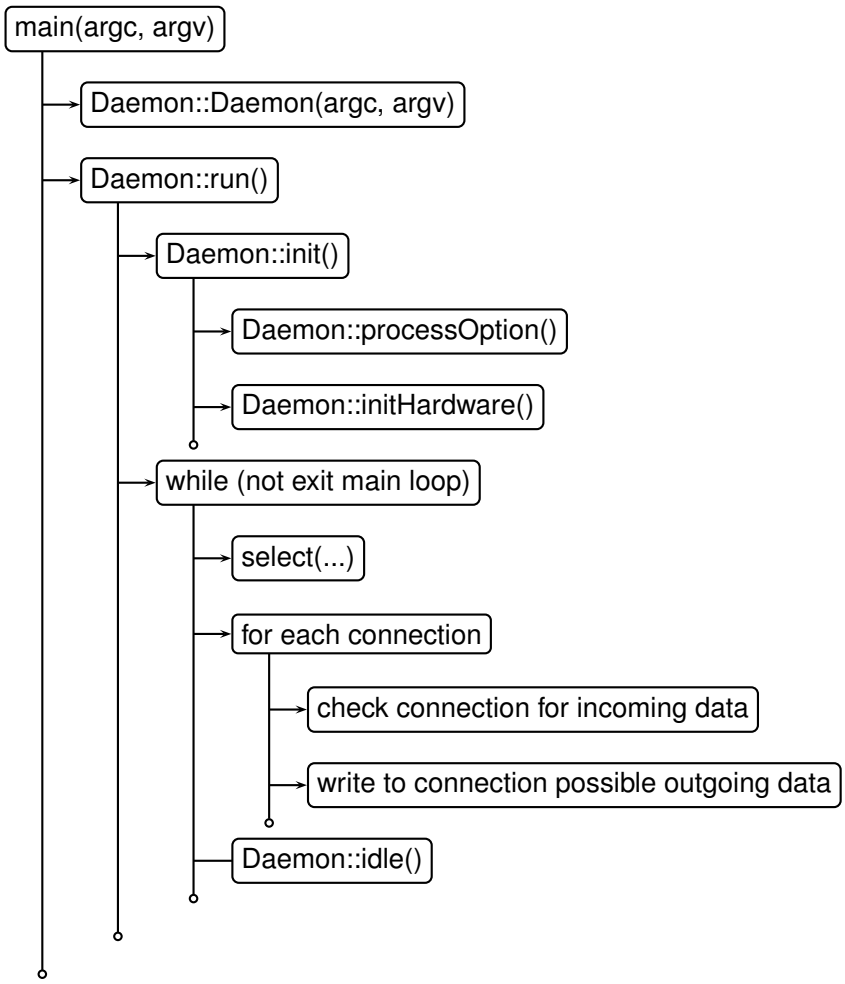

Figure 1.6: Daemon operation sequence diagram.
Diagram showing skeleton of daemon operation. From main class, daemon object is created, and its run method is called. Run method first calls routines to inialize the device. Then the code enter a loop processing requests from connected sockets, and calling the idle method after all pending requests were processed.

Figure 1.7: Target execution diagram
Diagram of target execution on a single observatory.

Chapter 2

RTS2 run observatories

The following observations and laboratories are benefiting from the work invested into RTS2. Here is presented their full list, along with technical details. This list shows the potential of the full RTS2 ecosystem, where independent modules can be employed to perform tasks demanded by both existing and new facilities.

2.1 BART

Bart is the original member of the RTS2 run observatories. This is the telescope where both RTS and RTS2 developements started. It was originally equipped with 20cm Meade telescope and a variety of wide field lenses. RTS2 was first run on this telescope in December 2002. The telescope is located at Ondřejov observatory of the Czech Republic.

2.2 D50

D50, named for its diamater of 50cm, is located at platform next to the BART telescope. It is a cutom made Newtonia, placed on equatorial fork mount. The mount movement is directly controlled by its RTS2 driver, which performs transformations from sky position into motor coordinates and vice versa.

2.3 BOOTES

BOOTES (Burst Optical Observing Transient Events System) is a major user of RTS2. The network capability was primary developed on this system. Bootes currently consists of four operational nodes.

2.3.1 Bootes 1

2.3.2 Bootes 2

2.3.3 Bootes 3

2.3.4 Bootes 4

2.4 Watcher

Watcher is a join Irish-Czech-Spanish-South Africa project. It uses 40cm optical tube, mounted on Paramount robotic mount. Dome control is custom made, through Zelio PLC.

2.5 FRAM

2.6 University of Columbia Lunar Transient Phenomenas monitor

2.7 Calar Alto Hispano-Aleman observatory 1.23m

2.8 LSST - BNL

Brookhaven National Laboratory (BNL) Large Synoptic Survey Telescope (LSST) CCD testing laboratory was the first place to see installation of RTS2 as laboratory controller. Lot of new features were developed primarly for this installation, among the most important was the replacement of model with fixed structure for system variables with device distributed variable lists. This laboratory installation has two major aims:to characterise CCDs coming from different vendors to choose the best one for LSST, and to characterise delivered CCDs before assembling them onto LSST camera.

The installation currently consists of the following devices: SAO CCD controller, Arizona Cryogenics dewar, Cryocooler cryogenic controller, 2 Keithely 6428 picoAmperimetes connected to photodiodes, one Keithley regulated power supply for back bias voltage,

2.9 LSST - LPNHE

The telescope without telescope.

2.10 FLWO 1.2m

FLWO 1.2m/48" telescope is equiped with "Keplercam". Its main use is in SN follow-up observations and transiting planet confirmations.

RTS2, modified as RTS2-F, is used only for scheduling and overall observatory management. RTS2 does not have direct control of the hardware. For each target, a XML file is produced, which is transformed into a script. The script is run by the current telescope control shell, named TelSH. Apart from this, RTS2 collect weather related data and make decisions to open or close the dome and to start or interrupt opservations.

FLWO 1.2m is a main telescope using RTS2 queue scheduling. After initial setbacks, when the software does not match all user expectations, the operation runs quit smoothly. In summer 2012 the 1.2m was last controlled directly from the observatory, and queue scheduling with RTS2-F becomes the standard way of observing during Fall 2012. Table 2.2 documents how was RTS2-F used during transition from remote control to fully autonomous system [21].

2.11 RATIR

Refurbished Johnson 1.5m telescope at SPM in Baja California, operated by Universidad Nacional Autóma de México (UNAM), hosts RATIR instrument. RATIR is 4 CCD, 6 band detector, aimed at studing early visible and infrared light evolution of transients objects, primarly GRBs. RTS2 provides interfaces for hardware control of RATIR's visible and infrared cameras and its associated electronics, as well as upper level control of observatory operations – queue scheduling, observation block execution, and observatory logs acquisition[11].

2.12 ESO La Silla 1.54m Danish telescope

2.13 Future telescopes

Use of the RTS2 is discussed for a wide variety for future projects. It is a good sign of the project maturity to see it being considered for growing number of medium class telescopes.

2.13.1 MDM 1.2m telescope

2.13.2 Another LSST testing laboratories

Month	Robot				Observer's log		Robot share	
	Nights	Sky time	Observ.	Images	Nights	Total time	Nights	Time
10/2010	4	4:23:10	17	58	28	195:30:00	14.3%	2.6%
11/2010	3	3:02:00	12	46	30	248:45:00	10.0%	1.4%
12/2010	4	19:56:30	44	247	28	219:30:00	14.3%	10.0%
01/2011	6	24:19:40	84	293	30	311:30:00	20.0%	8.8%
02/2011	4	12:29:31	47	179	26	224:00:00	15.4%	6.4%
03/2011	5	27:35:00	78	383	29	289:05:00	17.2%	10.7%
04/2011	4	11:32:38	80	215	29	247:45:00	13.8%	5.8%
05/2011	5	26:21:37	99	419	29	220:00:00	17.2%	13.8%
06/2011	1	2:42:00	11	36	29	214:00:00	3.4%	1.5%
07/2011	-	-	-	-	22	98:15:00	-	-
08/2011	-	-	-	-	1	8:00:00	-	-
09/2011	8	20:58:31	95	589	23	127:54:00	34.8%	20.0%
10/2011	16	80:16:42	306	2187	31	262:00:00	51.6%	36.8%
11/2011	11	57:53:07	215	2254	26	202:48:00	42.3%	35.5%
12/2011	19	99:42:55	430	4014	20	169:00:00	95.0%	74.6%
01/2012	25	114:05:29	416	5533	25	257:40:00	100.0%	56.7%
02/2012	22	97:46:28	449	3100	23	214:00:00	95.7%	56.6%
03/2012	18	94:58:26	362	4674	26	234:00:00	69.2%	52.2%
04/2012	29	153:58:31	579	7679	31	244:00:00	93.5%	81.3%
05/2012	31	157:40:38	571	6134	31	236:55:00	100.0%	82.6%

Table 2.2: FLWO 1.2m Robot statistics November 2010 - May 2012
Statistics of the autonomous observations of the 1.2m FLWO telescope. Nights, Sky time and number of observations and images are extracted from the robot database. Nights and Total time are counted from observing logs. Robot shares are computed as the percentage of nights and time the robot was active from the nights recorded in the observers' logs. To adjust for CCD readout and telescope slew, an estimated dead time is added to the sky time before the fraction of total time is calculated.

Chapter 3

RTS-2 network

The main goal of this work is to develop a software to control networked RTS2 observatories. The controll shall be distributed, enable a single shop access to explore resources available in RTS2 network. It must collect reasonable amount of data from all observatories, include functionality for network scheduling, and display users result of their requests.

3.1 Design

Overall design of the network infrastructure integrates existing RTS2 driven observatories with the central server, called BB. The whole network communicates only through Hyper Text Transfer Protocol (HTTP) protocol. This requirement was laid down to allow the network to operate at places with restricted Internet access, where HTTP/Web is the only way to directly communicate with servers on the Internet. The design is depicted in figure 3.2.

The network infrastructure leverages existing RTS2 design. BB contacts observatories JSON proxies, rts2-xmlrpcds. Both codes uses extensively existing database access classes to deal with the database as well as RTS2 Libnova integration.

Complex tasks are run by BB as a script, utilising existing standard output/input scripting interface developed for device scripting. They are coded in Python, and contact observatory servers with JSON API. This design allows for quick customization of the whole system.

Observatory rts2-xmlrpcd servers reports its state and progress of observations assigned to the observatory to a central server through HTTP calls. Should the call contain a big amount of fields, those are transmitted as JSON data attached to the call.

The next paragraphs describes various calls between BB and observatories rts2-xmlrpcds.

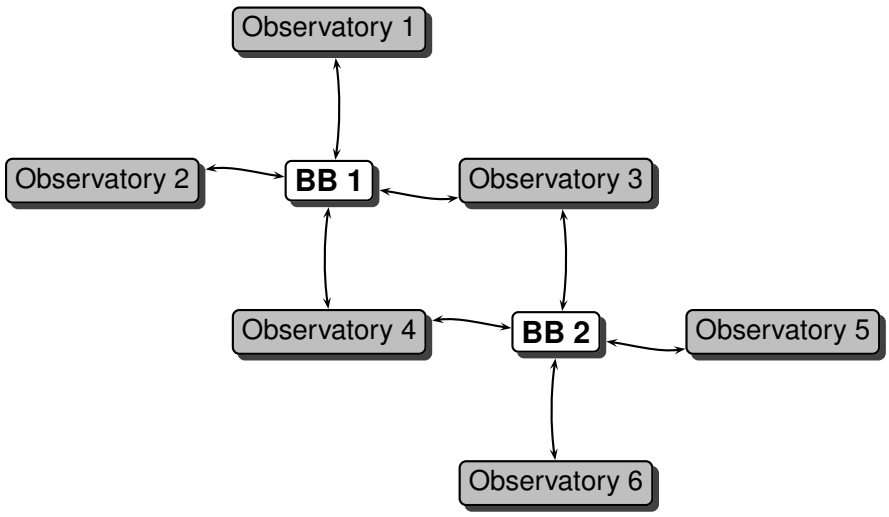

Figure 3.1: Overview of RTS2 network architecture
Four observatories, named *Observatory 1* to *Observatory 4*, are cooperating through *BB 1*. *Observatory 3* and *Observatory 4* are also cooperating in network mastered by *BB 2*.

3.1.1 Target creation

Before it is able to schedule an observation, BB must transfer target information to the observatory. Each observatory assign a local identifier to the target. BB keeps in E.3.3 a table mapping of the local observatory target numbers to the BB target numbres. Local ID, which is returned from the observatory, is mapped in BB database using "mapping" command.

3.1.2 Target scheduling

Each observatory node has available intervals when the network targets are allowed to schedule their observations. The local rts2-xmlrpcd uses those intervals searching for the time when an incoming target can be scheduled.

There are two API calls available to schedule a target. The first is "schedule", the second is "confirm". As central BB knows about mapping between its and local observatory target ID, every request to the local node is using the local observatory target ID. The local node offers the first time a target can be observed and falls inside free intervals. It reports this time as an answer to the "schedule" call.

After collecting scheduling from the local nodes, BB decides where the target will be observed. It confirms its selection through "confirm" API call. For the confirm call, the local observatory node runs same code as for the "schedule" call. It then puts target into local queue, and again indicates time the

request should be executed to the central BB. If this time does not meet BB expectations, or if somebody or something changes its mind and request the observation request to be canceled, BB can use "cancel" API call to do so.

Local rts2-xmlrpcd reports all changes to the scheduled observation request to the BB. BB can reschedule anytime the observation using combination of the "cancel" and "commit" calls. Also observation progress is reported to the BB, so BB can reschedule the target if it for example see that poor data are obtained.

3.1.3 Observation reporting

Observatory nodes reports to BB their states. The state is reported as JSON string, using same format as C.1 API call. Observatory node state is cached in BB, and is accessible to clients through same interface as on observatory nodes - C.1, C.1 and C.1 calls are supported. Cache is read-only, clients willing to set values need to contact the nodes directly.

Observatory reporting is configured in rts2-xmlrpcd configuration file. Connection is created from thread running inside rts2-xmlrpcd, so it does not block other rts2-xmlrpcd operations.

3.1.4 Observation state reporting

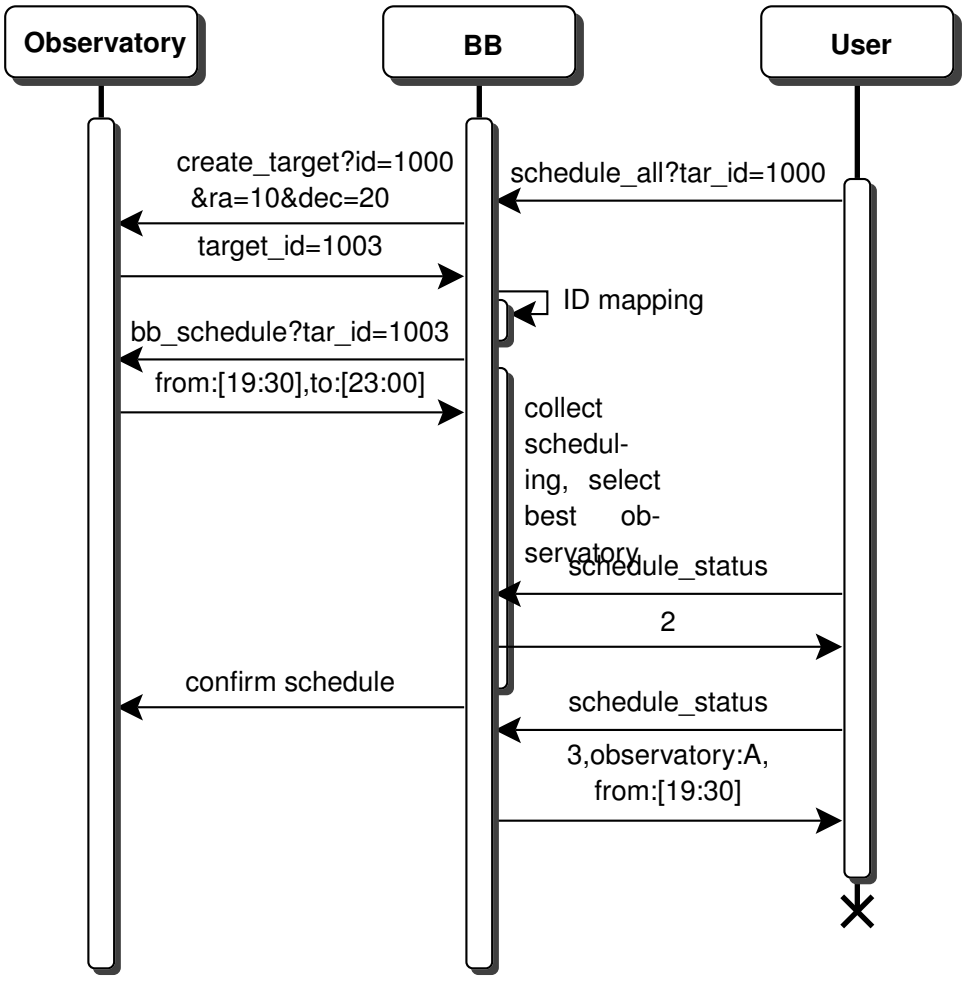

Figure 3.2: Processes handling distributed target creation
This diagram omits some unimportant calls. User process intiates sequence by issuing schedule_all request. User process can track scheduling progress using schedule_status call.

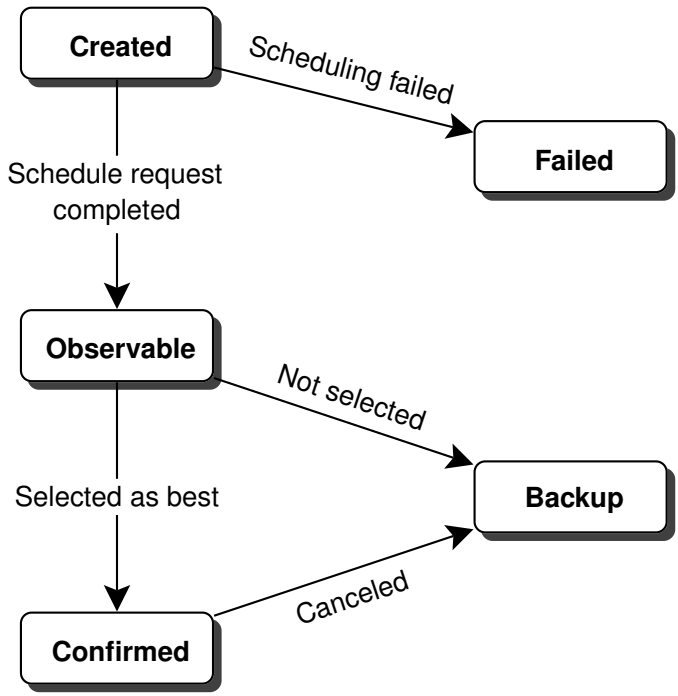

Figure 3.3: Observatory observation request state diagram
State diagram of observation request for observatory.

Chapter 4

Results

Results so far obtained with the network.

4.1 GRB observations

4.2 Other science

4.3 Scheduling

The development of software which operates an autonomous observatory is not an easy task. The development of software which allows observatory users with different scheduling requirements to schedule observations, obtain the observations, process the observations and perform other tasks associated with running an observatory is a task at least an order of magnitude harder.

It is of course not easy to design and code scheduling for a space-based observatory[18], but it is even more challenging to design and code scheduling for a ground-based observatory. The ground environment is more dynamic and less predictable than the space environment. Scheduling should be able to adjust night plans based on changing atmospheric conditions. It should allow observatory users to re-arrange observed objects so the observations they would like to get done while sleeping would be done in proper order.

Mid-size and large ground based observatories usually serve various user groups with various scheduling needs. Software which enables users to observe a single star for most of the night might be ideal for exoplanet researchers, but does not fit the needs of supernova observers.

Scheduling of observing time is a major factor contributing to the success of any autonomous observatory. Autonomous observatories with perfect scheduling succeed, whereas those without it fail. Multiple attempts were made throughout history to solve this problem – for example, see Stella [17] or Robonet[16] scheduling. Most of these rely on some kind of merit function, which evaluates target "usability". The scheduler always picks the target with the highest merit function value. Brief introduction to merit function scheduling is provided in section 4.3.1.

Although various systems based on merit-function scheduling may offer a programmer a perfect plaything, they are very difficult to explain to the investigators. Astronomers are not interested in hearing how they can make their observations by adjusting knobs which influence merit function. They would like to have their targets observed using a simple and elegant method to do so. That is where queues, used at most of the large observatories[27][13], come into action. Observers can place their targets into a queue and the queue is executed. For detailed discussion of queue scheduling see section 4.3.2.

Both approaches have advantages and disadvantages. Those are discussed in section 4.3.3. The next section, section 4.3.4, describes a design for a scheduling method that combines both approaches into a system satisfying both human and computer driven observatories.

4.3.1 Merit function scheduling

Merit function scheduling uses a function to calculate each target's immediate benefit for an observing program. The merit of an observation can be difficult to estimate, and this is one of the major problems of merit-function scheduling.

Let's assume observers would like to observe a target which is at the moment closest to the zenith. Then the merit function, $meritf$, is:

$$meritf(T, JD, observer) = altitude(T, JD, observer)$$

where:

T is a target

JD is the current Julian date

$observer$ is the observer position (longitude and latitude)

$altitude(T, JD, observer)$ is a function returning a target's T altitude as seen from the site on Earth at the $observer$'s coordinates at date JD

Merit functions are adapted to the needs of the individual observatories. Various terms and parameters can be added so the resulting schedule would match observer's wishes. The current RTS2 merit function is:

$$meritf(T, JD, observer, ha, Ld, lo, ldo) = priority(T) + bonus(T)$$

$$+altitude(T, JD, observer) * 2$$

$$+log((180 - ha)/15.0) \Leftrightarrow (ha < 165) + log((ha - 180)/15.0) \Leftrightarrow (ha > 195)$$

$$-log(61 - Ld) \Leftrightarrow (Ld < 60) - log(lo/3600) * 50 \Leftrightarrow (lo > 3600 \wedge lo < (3 * 3600)$$

$$-log(3) * 50 \Leftrightarrow (lo < 86400) - sin(ldo * \pi/4) * 5$$

Where, in addition to parameters described above and common mathematical symbols (log, sin and π):

$priority(T)$ is the current target T priority

$bonus(T)$ is the current target T bonus

ha is the hour angle for target T as observed from the site on Earth at the $observer$'s coordinates at date JD

Ld is the target lunar distance in degrees

lo is the difference in seconds between the current time (JD) and the last observation time for the target

ldo is the number of target T observations in 24 hours preceding date JD

$\Leftrightarrow$ is the logical iff expression. The value on the right will be used only if conditions on the left are satisfied. For example, $log((180 - ha)/15.0) \Leftrightarrow (ha < 165)$ evaluates to 0 if $ha >= 165$

There are other possible implementations of merit functions. The expression can include the current site seeing, moon phase, number of good images of the target or time left until the end of period during which the target can be observed. Expressions can include target-dependent multiplication factors, or weights. Thus, some targets may be configured with a high relative weight on one expression, while others can decide to ignore this expression.

Given a merit function, the algorithm for target selection is simple. It finds the target with maximum value of the merit function for given instant of all targets, and selects this target as the one to be observed.

Various methods can be used to optimise merit function scheduling. For example, the RTS2 merit function $merit f$ uses an expression parametrised with lo to force the selection of unobserved targets for observation. Besides this, the target properties can be set so the target will be removed from the set of targets evaluated for autonomous selection as soon as it is observed - either indefinitely[1] or for a given time[2].

The more complex the merit function becomes, the more complex it is to evaluate which targets will be selected and how the function will behave in the future. One can add multiple functions to the expression, but all those will just turn the selector behaviour even more obscure. The next paragraph discusses how the scheduling can be simplified by allowing multiple criteria to be considered.

Multiple objective global optimisation

As it was illustrated in the previous example, merit functions can become quite complex. This is partly due to the need to weight different factors leading to a single number representing a target's merit. As scheduling is in the NP-hard[7] class of problems, finding the best solution requires traversal of the full solution space – with a size that is exponential to the size of input target set. So the usual merit function implementations are short-sighted - only a handful of the successive targets can be considered. It is then hard to create a night plan using just a merit function. If a night plan must be created, some heuristic is usually adopted – for example targets are ordered in west–east direction, as targets near the west horizon should be observed as early as possible before they set, and targets to the east later in the night as Earth rotation places them closer to zenith.

Various strategies can be employed to simplify complex merit functions. Some expressions can be written as constraints, requiring targets to match certain conditions to be included in the target set considered for selection. One of a very good example of such constraint can be zenith distance limit – targets below pointing limits of the telescope shall not be considered for merit function

[1] targetdisable script command
[2] tempdisable script command

evaluation.

The Master's thesis "Genetic Algorithm for Robotic Telescope Scheduling" [22] discusses design and implementation of a multiple objectives scheduling algorithm which uses Non-dominated Sorting Genetic Algorithm II (NSGA II)[15]. This algorithm works by optimising a global observing schedule with NSGA II. Instead of relying on a single merit function, the algorithm works with multiple functions and tries to optimise all of them. This approach further reduces the complexity of the scheduling.

The genetic algorithm is inspired by natural selection process. The GA scheduling works by representing possible schedules as *chromosomes*. When algorithm starts, various possible schedules are generated for population zero. During virtual evolution, the *chromosomes* are *crossed*, *mutated* and *repaired*. The best are *selected* to form next generation. Instead of exploring full solution space, genetic algorithm explores only a small fraction of it. This allows it to effectively search solution for NP-hard, exponential size problems.

NSGA II scheduling can provide multiple possible schedules for the night. An experienced observer can then choose the one which best matches the expectations. This approach was recently chosen by CFHT for implementing the intelligent scheduler running on top of their queue system[23].

While NSGA II optimisation looks promising, its inputs are difficult to explain to the observers. Experienced observers prefer to simply provide a list of targets they would like to observe, combine it with targets proposed by other observers and let that schedule run. Queue scheduling, discussed in the next chapter, is much better for direct interaction of investigators with the scheduling.

4.3.2 Queue scheduling

"In computer science, a queue is a particular kind of abstract data type or collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure."[10]

Scheduling queue entities are targets which should be observed. When the next target should be observed, it is removed from the top of the queue. Targets are usually added to the end of the queue, but it is reasonable to assume that observers may like to change the order of the queue targets. It is better to think about a scheduling queue in a different manner than in the definition given above. The scheduling queue can be for our purposes defined as an ordered set, with the following operations: remove top entity, insert entity to arbitrary position, change set order, delete an entity, repeat as needed.

Queue scheduling can be depicted on a computer screen in a diagram, showing rectangles representing targets on the time axis (as can be seen in figure 4.1). Queue scheduling is mainly used in service mode observing at large observatories. Service mode means that the observations are done by observatory staff on behalf of investigators requesting the observation, who are not present during the observation. Trained observers spend each night at the telescope and work off a list of targets to observe together with required instrument setup. The targets are put into a queue, from which the observatory control system picks a target, sets up instruments for its observations, and commands exposures as required by the scheduling entry. The observer then just monitors the system and troubleshoots any problems.

4.3.3 Advantages and disadvantages of classical scheduling

Following table compares queue and merit function scheduling. As it was discussed in previous chapters, the main difference is that merit function scheduling is good for autonomous observatories, while queues are good for trained observers.

Table 4.1: Comparison of queue and merit function scheduling

Feature	Merit function	Queues
Investigator friendly	bad, as concept of merit functions is hard to explain and understand	moderate, queue combined with graphical user interfaces are relatively simple to explain
Flexibility	good, as merit function can be quickly recalculated with different input (seeing,..) parameters	bad, as it requires significant user interaction with the system
Automatic interruptions, ToOs	yes, new merit function can be calculated for next observation after ToO	yes, but can destroy end of the queue (targets becoming unobservable due to setting below pointing limits,..)
Autonomous operation	excellent, as merit function is a simple algorithm which can be robustly programmed	bad, queue requires human interaction – investigators must fill the queue

4.3.4 Combining queue and merit function scheduling

Merit function scheduling excels where queue scheduling fails and vice versa. Given this, the idea to integrate both scheduling approaches into a common framework seems very promising.

What if the queue system is to be used for obtaining user-requested observations, while merit function scheduling is to be used when there are no entries

in the queue system to obtain service-type observations of targets? This is a pseudocode representation of such an approach:

```
while (canObserve ())
{
   if (queueNotEmpty ())
      t = selectFromQueue ();
   else
      t = selectFromMeritFunction ();
   observe (t);
}
```

While this approach will work, it will not address the inflexibility of queue scheduling. There is a selection coming from a single queue, which in its raw base algorithm allows just for the top target to be removed by the **selectFromQueue** procedure. So either the system will pick a target from the queue if the queue is not empty, or it will select a target using merit function evaluation.

To resolve this, one can slightly modify a queue selection algorithm in the **selectFromQueue** method. The modified algorithm should select a target only if the target should be observed. There can be multiple criteria for when the target should be observed – starting with a flag indicating whether the queue scheduling is enabled at all, including the visibility of the target and checking whether the target meets additional constraints, such as a particular time during which it should be observed.

Given that there is an algorithm for a single queue with the properties given above, it is trivial to extend the system to multiple queues. The scheduling algorithm loops over all queues, requesting the next target from each queue. If a target is found, the loop terminates and the target is observed. If queue scheduling cannot find a target to observe, then merit function scheduling is asked to provide one. A pseudo-code representation of such an algorithm is:

```
while (canObserve ()) {
   for (q in queues) {
      t = selectFromQueue (q);
      if (t)
         break;
   }
   if not (t) {
      t = selectFromMeritFunction ();
      if not(t) {
         debug ("cannot find target for observation. 30 sec sleep");
         sleep (30);
         continue;
      }
```

```
   }
   observe (t);
}
```

This approach allows investigators to put their targets into their queue. Either a night observer or an autonomous system can then disable or enable queues, based on predefined constraints. Using such an approach, queues become flexible, allowing for simple change of the path the scheduling will take during the night.

We will call the structure holding multiple queues with special characteristics **meta-queues**. Use of meta-queues for scheduling is discussed in the next section.

4.3.5 Meta-queues scheduling

As discussed in section 4.3.2, the designation "queue scheduling" is somewhat misleading and might be renamed to "scheduling from an ordered set of targets."

RTS2 queues become even more complex than the ordered set. First, as discussed in the previous section, there are multiple queues – hence the plural in the name.

Second, a queue has an associated queue type. The queue type can change how each queue is ordered and what happens during and after a target is selected from the queue. For details please see discussion in section 4.3.5.

Third, the queue member is a structure, containing a pointer to the target and optional start and end times of required target observations. With this extra information, a member of the queue can be removed if its end time expires. A queue may also be put on hold when the start time of target on the top position is in the future.

The algorithm for selection from the queues can be written in a pseudo-code as:

```
proc selectFromQueue (q, &maxDuration) {
   filterExpired (q);
   filterUnobservable (q);
   proposed = selectNext (q, now, maxDuration);
   if (proposed) {
      markSelected (q);
      return proposed;
   }
   if (startTime (q) - now < maxDuration)
      maxDuration = startTime (q) - now
}
```

filterExpired removes from the queue all targets with end time in the past. This guarantees that the requests with target time expired will not be scheduled. It also removes targets which are in the queue before a target with start time in the past. This guarantees that if there is a target in the queue which should be observed now, it will be observed at the expense of targets ahead of it. This is important to allow for time-critical observations to start at a required user-specified time, even if the queue execution was delayed, for example by inclement weather.

filterUnobservable removes from the top of the queue targets which cannot be observed at the moment. Based on the queue parameters, those targets are either moved after the first target in the queue which can be observed at a given moment, or completely removed from the queue. This guarantees that the queue will not become blocked by the targets entered into the queue which due to any delay set before they can be observed.

selectNext returns the queue top target if and only if the queue top target start time is either not specified or in the past.

startTime returns the expected start time of the top target from the queue, or not-a-number if there is no start time attached to the top target. **now** is the current time.

maxDuration is used to constrain the maximal duration of execution of the selected target. If the expected execution duration of a target extends past the available time, the target is not considered for selection. **maxDuration** is set by a procedure that calculates the time available until dawn, and updated if it is lower than the start time of the first observation from the queue. The queue system guarantees that if there is a queue with the target which observations should be started later, targets from queues considered after this queue will finish before the given queue start time is reached.

Meta-queues Ordering and Selection Algorithms

Meta-queues can be configured to order and select targets in a different fashion to the standard first-in-first-out (FIFO) algorithm. The available variations are:

CIRCULAR similar to **FIFO**, but places selected targets at the end of the target queue. In this way, queues can be configured to continuously observe a set of targets throughout the night.

HIGHEST order targets by altitude.

WEST–EAST order targets from west to east. This allows for targets on the west to be observed first.

WEST–EAST–MERIDIAN order targets west of the meridian by priority and east from meridian the same way as the **WEST–EAST** queue. This is to allow for high priority targets west of the meridian, meaning those targets

that are currently setting, so the highest priority targets will be observed first.

OUT–OF–LIMITS order targets by remaining time until they set below observing constraints.

A user or program can change the queue type anytime. For example, one can change a queue from **FIFO** to **WEST–EAST–MERIDIAN** if bad weather is approaching. This allows for targets with the highest priority, which are setting, to be observed first.

Circular queue can be used to monitor multiple targets throughout a night. Assume we have targets A,B,C, which should vary on time scales long enough to allow for some gaps between target visits. Assuming A, B and C are visible, the scheduler will repeat sequence A,B,C. Targets that are not visible will be skipped.

Target of opportunity interruptions

While the previous section deals with making the queues flexible, it does not describe how this flexibility can be used for target of opportunity observations.

Targets of opportunity can be introduced into the RTS2 target database with custom clients, which are available for GCN[12] and Pierre-Auger high energy showers. The current procedure is to call the "now" command of the executor to immediately execute visible targets of opportunity, and raise the target priority so that the merit function will pick the target for the next observations. While the "now" part worked without problems and there is no need to change it, the follow-up planning with the merit function was difficult.

The preferred approach with the queue scheduling for the follow-up observations is to put targets of opportunity into a special queue. This queue can be disabled if the observers do not want to interrupt their observing plan. The target script can add targets to any queue if more observations are desired.

4.3.6 Planning of the night with meta-queues scheduling

Observers can interact with the scheduling subsystem either through a graphical user interface, or from a command line. Targets must be entered into the database before they are scheduled with various target management tools that RTS2 provides. Users can append targets into a queue, move targets to an arbitrary position, remove targets from the queue or clear the queue.

The scheduling subsystem includes a scheduling simulator. The simulator uses queue entries to produce a simulated night run, showing which targets will be executed during the night[3]. The simulator can be used by users to confirm that the entries they insert in the queues will be executed as desired.

[3] assuming ideal run conditions – i.e., a night without any interruptions caused by bad weather or instrument failure

Queue	Target	Start date and time	End date and time	Script duration
transit	TR 1	29^{th} December 2012, 21:30	29^{th} December 2012, 23:00	1m
(FIFO)	TR 2	30^{th} December 2012, 23:30	31^{st} December 2012, 09:00	2m
	TR 3		31^{st} December 2013, 23:00	1m
service	SN A			
(FIFO)	SN B			
	....			
	SN M			50m
	SN N	1^{st} January 2012, 00:20		
	Blazar A	1^{st} January 2012, 02:30		
	Blazar B			
	Blazar C			
backup	B 1			
(circular)	B 2			20m
	B 3			

Table 4.2: Example schedule combining transits, service and backup observations

Transit queue targets will not be remove before they *end time*, as the queue is marked as *Keep observed*. Service queue target will be considered only if there aren't targets in transit queue, backup queue targets only if there aren't targets in transit and service queues.

Different queues can be populated by different users. For example, occultation targets requiring execution at a given time can be put into one queue, and supernova follow-up targets into another. Observers, if awake during the night, can monitor the night run and disable the queue if conditions do not allow target observations.

Example schedule

Let's assume an investigator would like to observe three transits, named *TR 1*, *TR 2* and *TR 3*. The transits must be observed on nights from 29^{th} December 2012 to 1^{st} January 2013.

And let's assume another investigator would like to observe supernovae named *SN A*, *SN B* through *SN N*, and *Blazar A* through *Blazar C*. *SN A* through *SN M* can be observed anytime, but *SN N* observations must start nearest 1^{st} January 2013 at 00:20. If any of *SN A* through *SN M* are not observed before *SN N*, they should be removed from the schedule. Similarly, *Blazar A* observations must start nearest 1^{st} January 2013 at 02:30.

On top of that, there is "backup" queue containing targets to be observed while there is nothing else to do. This queue is of **CIRCULAR** type.

Table 4.2 shows the queue setup for such observations. Table 4.3 shows how those queues can turn into observations with a period of inclement weather. Figure 4.1 depict how the queues user interface would present an user with the queues setup.

Description	Start date and time	End date and time
day		29th Dec., 18:16
SN A	29th Dec., 18:16	19:06
SN B	19:06	19:56
SN C	19:56	20:46
B 1 *(as SN D is too long)*	20:46	21:06
B 2	21:06	21:26
merit scheduling or idle	21:26	21:30
TR 1	21:30	23:00
SN E *(SN D is not visible)*	23:00	23:50
SN F *(SN D is not visible)*	23:50	30th Dec., 00:20
bad weather – idle	30th Dec., 00:20	04:23
SN D	04:23	05:13
SN G	05:13	06:03
B 3 *(to fill time)*	06:03	06:23
merit scheduling or idle	06:23	06:34
(day)	06:34	18:17
bad weather	18:17	20:34
SN H	20:34	21:24
SN I	21:24	21:26
remote session	21:26	23:06
B 1	23:06	23:26
merit scheduling or idle	23:26	23:30
TR 2	23:30	31th Dec., 01:30
observer disables transit and service queues due to inclement weather		01:28
B 2	31th Dec., 01:30	01:50
B 3	01:50	02:10
B 1	02:10	02:15
bad weather	02:15	04:35
observer enables service queue		04:25
SN J	04:35	05:25
observer enables transit queue		05:15
TR 2	05:25	06:34
(day)	06:34	18:17
TR 3	18:17	23:00
SN K	23:00	23:50
B 2	23:50	1st Jan., 00:10
merit scheduling or idle	1st Jan., 00:10	00:20
SN N *SN L and SN M are removed from the queue*	00:20	01:10
B 3	01:10	01:30
B 1	01:30	01:50
B 2	01:50	02:10
B 3	02:10	02:30
Blazar A	02:30	03:20
Blazar B	03:20	04:10
Blazar C	04:10	05:00
B 1	05:00	05:20
B 2	05:20	05:40
B 3	05:40	06:00
B 1	06:00	06:20
merit schedule or idle	06:20	06:34
(day)	06:34	

Table 4.3: Simulation of observations resulted from the example schedule described in table 4.2.

This table shows important events happening between 29th December 2012 and 1st January 2013. This is assuming the SN and Blazar target observations take their allotted time. In a real run, the time will differ as targets will use different scripts and due to the different distances the telescope has to slew between targets.

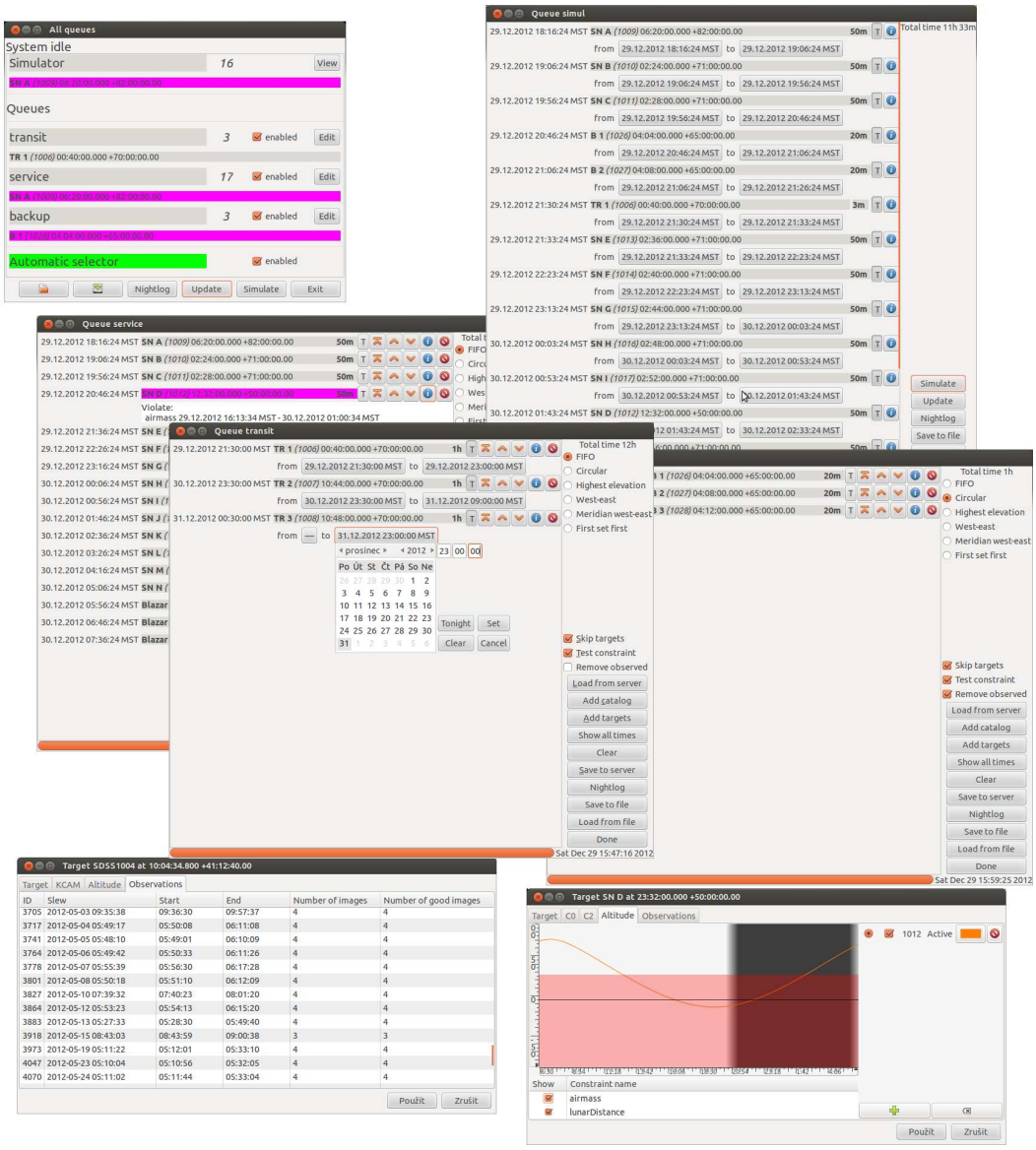

Figure 4.1: Queue GUI

User scheduling interface showing queues setup from the example described in section 4.3.6. On image top are windows showing queues available at the system and current selection from the queues. Right is simulation showing how the system thinks the observations will be executed. At middle of the image are shown service queue filled with supernova observations, transit queue demonstrating how user can enter queue times, and backup queue with **CIRCULAR** queue ordering. Windows showing observing logs of a target and altitude plot of another target are visible on image bottom.

Appendix A

The RTS2 protocol

From the early beginning RTS2 was designed as a plug–and–play system, capable to drive various instruments and to provide a common infrastructure for them. It is clear that this component system needs some communication layer. Initial investigation of the off–the–shelf communication libraries led to decision to develop an own layer on top of the TCP/IP protocol stack. After years of development this communication layer matured to a level that it easily supports requirements from new instruments and experiments. Due to its focus on observatory control it can be considered to be better observatory communication library then the off–the–shelf libraries available today.

A.1 RTS2 ecosystem

The usual RTS2 installation contains one central server, which works as a name resolver and a synchronisation coordinator. The network contains number of devices, each of which represents any observatory or experimental device. The devices are usually commanded by services or clients. The services are autonomous processes, which have goals (carry observation, listen for incoming events,..). Clients are usually interactive clients for monitoring device states and for sending commands to the devices.

The full description of the RTS2, including description of the system history and its current installations, is given in SPIE 2006 proceeding [20].

A.2 Protocol development

The protocol development started with the development of RTS2, a C/C++ replacement to RTS. The protocol was inspired by Simple Mail Transfer Protocol (SMTP)[26]. The first version allow only one way communication, with a client[1]

[1] The process which started the communication

sending the commands to a server[2]. The server responded to each command with some replies, and finish the response with a line starting either with - (the minus sign) or + (the plus sign), followed by the status code and status description string.

The original protocol design relied on the fact that the client sends only commands, and the server sends back only replies. Replies were variables and their values. This original protocol can be demonstrated on a following example session:

Client/**S**erver	message
C	info
S	ra 20.86
S	dec 20.6754
S	+000 OK
C	helpme
S	-005 unknown command "helpme"

The servers were usually devices, and the clients were either user interactive clients, or the automatic components.

This design took advantage of the fact, that only one command can run at a time on a single connection. The longer running commands, whose need some time to perform and finish operation, used device states to signal the progress of the command execution to the client. Originally, every server can define as many states as it needed. The commands, which the client was about to send over the connection, were aligned into a queue, and once their submission was completed with confirmation or rejection error, the client program was notified by call of an appropriate handler.

The information about state was passed through protocol using S prefix to inform receiving client that the message contains state message, not the usual variable message.

The protocol then naturally evolved into using prefixes for all informations passed through it. V was used to prefix the values, M for the system messages, and so on. This allowed client to quickly parse the received messages, and feed them to the appropriate handlers.

With prefixes for everything excepts commands, protocol gains ability to distinguish the commands. So it was possible to establish two-way communication, as it is shown in the following example:

[2]The process which held opened listening port and waits for connections

Client/**S**erver	message
C	move 20 30
S	info
C	V infotime 20080617015432.34566
C	S 0
S	V ra 20.86
S	V dec 20.6754
S	S 4 start moving
C	+000 OK
S	+000 OK
C	helpme
S	-005 unknown command "helpme"

A.3 Protocol building blocks

In the following subsections, development of the various building blocks of the protocol and rationale behind their design is described. For description of the current protocol, please see the section A.5.

A.3.1 Large binary data transfer

Large binary data were originally transported over separated TCP/IP connection as binary data. Because a separated channel was used for the data transfer, the system had the ability to send the control commands over ASCII channel at the same time. In times, when CCD readout over parallel port could took minutes, this looked as a necessity. In those times data were sent to TCP/IP connection right after readout of one line. The possibility to interrupt camera readout and start a new exposure as soon as the telescope reached GRB position looked very promising. Also separation of data and command channel looked as a right thing to do.

But this transfer mode created a big system overhead, associated with initialisation of the data TCP/IP connection. Original design requires one data connection for the data – for each image transfer, a new connection was started. If only a few images per minute could be obtained, that does not cause a significant problem. But when the cameras capable to produce few images per second were introduced to the system, it becomes clear that this design has to be changed.

There are two possible solutions for this problem. Either the opened data connection could be kept opened for the future data transfer, or the data can be transfered over commanding connection. The second solution is better when the data are transported from a closed network over a single opened TCP/IP port. This was the main reason why the second solution was chosen. Due to a redesign of the camera driver, the argument used to justify the use of the sep-

arate connections – that slow parallel port based cameras can be interrupted – was no more viable.

A.3.2 Variables

Initially, all variables names were coded in client and server. It was necessary to add a variable and its description at two places - to a server structure holding the variables description, and to a client which holds the variable list. It is clear that after few months, this mechanisms becomes unnecessary complicated, and some better design was needed.

Instead of turning to CORBA IDL–like design, with static description of the interfaces, dynamic variables were used. The server holds list of the variables, and distribute them to all clients which are connected to the server. The protocol contains commands to manage list of the variables, as well as commands to change directly value, or to perform some operation on it, like adding or subtracting a number.

The variable entry contains following informations:

variable name — string used to identify variable in scripting command, for a display and for a recording to a FITS header.

variable description — is displayed with the variable and written to the FITS comment fields

variable flags — specify a type of the variable (string, integer, double precision floating point,..), display conversion (degrees, ..), if and when value should be written to the FITS file and so on.

Variables system is coupled with FITS interface. So not only are the variables visible in monitoring interface. They are also autonomously, using generic routines, written to the output FITS files. Nice think about that design is that if new variable is added, without any extra coding its current value is recorded to the FITS file holding the acquired data.

Variable flags bit mask specify when the variable shall be recorded. The flags can specify if the value shall be recorded before, during or after the exposure.

Flags are also used to describe when the variable value can change. When conditions for value change are not met, the operation is put to a queue and waits until it is possible to carry it. The use of this feature can be demonstrated on an example. Suppose that camera has focusing element, whose value can be changed only when camera is not taking images. However, the command to change value of the focusing element can be sent anytime. But that command can be followed by a command to change other the camera value, for example some diachronic tilting mechanism. Both commands then ends in the queue. On each device state change, all operations in the queue are checked

if they can be executed. If all conditions for the operation execution are true, the operation is executed and removed from the queue.

Last but not least, variables can have default values. When an observation script finishes, variable value is reseted to the default value. That also happens when exposure is interrupted and new target is quickly followed. Use of this feature can be demonstrated on focuser driver. Focuser position can change during script execution. But when new script stars, the focuser should be back in default position. This and other similar problems are transparently handled by default values.

The dynamic variable definitions design pays off when a new driver for multiple devices controlled by same low–level driver is coded. Thanks to the dynamic list of variables, device driver can be extended without need to regenerate any interface files. And some variables can be created depending on the actual device connected. After device initialisation, the driver checks capabilities of the connected device, and creates only the variables which manipulates the settings presents on the device.

A.3.3 Messages

Messages are primary used to provide user with plain English informations about what the system is doing. At the beginning, the protocol did not include message transfer. All messages were passed to syslog facility and recorded to a disk file.

Latter, ability to pass messages was added. Information about what is the system doing could get to an end user, and was logged to the log file at the same time.

Originally, either XML-RPC bridge or SOAP bridge services were used to record the messages to the log files and a database. Currently centrald daemon is used to record the messages to the log file, and the XML-RCP bridge service is used to record the selected messages to the database.

A.4 Device states

Next chapters refers to the **device status** and **blocking states**. Here is a short description of those terms.

A.4.1 Device status

Device status is a bit mask. It represents what operation is device performing. In table A.1 is an overview of states of different devices.

As device state is a bit mask, more then one state can be set. For example, frame transfer camera can at one moment readout an image and expose a new image.

Table A.1: Device states overview

Device	State	Comment
camera	exposing	camera is exposing
	reading	image is read from the camera
photometer	integrating	photometer is performing measurement
	filter	photometer is changing filter position
focuser	focusing	focuser position is being changed
mount	moving	mount is moving
	parking	mount is moving to a park position
	parked	mount is properly parked
	wait_cop	mount is waiting for copula movement
	searching	mount is moving on search pattern
	correcting	mount is performing movement as a result of a correction operation
	guiding	mount is performing guiding movement
dome	closed	dome is closed
	opening	dome is being opened
	opened	dome is opened
	closing	dome is being closed
	copula	dome copula is rotating
filter	moving	active filter is being changed
centrald	daytime	centrald daytime state. The various values described observatory as being in day, evening, dusk, night, dawn or morning state
	off	observatory is in off state
	standby	observatory is in standby state
	on	observatory is in on state
executor	moving	executor is moving mount to a new position
	acquire	executor is acquiring images which confirms telescope pointing
	acquire_wait	executor is waiting for acquisition image processing
	observing	executor is taking core part of the observation
	lastread	last image in the script is read from camera
	end	executor should end the observation
imageproc	running	image processor is processing image

Table A.2: Blocking states overview

Blocking state	Blocked operations
exposure	operations which takes images
readout	operations which readout images
move	operations which changed something on light path

A.4.2 Blocking states

Blocking state is used for signalling that the observatory cannot perform some actions. Currently, blocking states described in table A.2 are defined.

System configuration file can specify which devices are blocked by which devices. For example, a camera is blocked only by the filter wheels which are on its light path. Other filter wheels did not contribute to the camera blocking state.

A.5 The protocol

The protocol is simple, ASCII based. After initial handshaking, both sides are made equal. The protocol then does not make any difference between who started connection (client) and who respond to request for connection (server).

The protocol consists of sentences, separated by either carriage return, new line, or both character. New lines and carriage returns inside strings passed through library are escaped with backslash notation, known from C.

The sentence consists of sentence type string and various number of parameters. Parameters are passed as strings. Parameters of the sentence are separated by at least one space or tab character. Strings with spaces are escapes with quotes, float point and integral numbers are converted to ASCII representation and back.

A.5.1 Sentence types

Sentence types overview is given in table A.3. This section contains description of the sentence – its parameters and when the sentence is used.

Authorisation request

response either authorisation_ok, authorisation_failed or registered_as.

id number which identifies the new connection

This sentence is send from central server, when authorisation request is finished.

Blocking state

blocking state

Table A.3: Sentence types

Character	Description
A	Authorisation request. Used during initial handshaking and authorisation.
B	Blocking state. Blocking state is a bit mask which is used to test which commands fro queues can be executed.
C	Binary data channel. Used to start binary data transfer.
D	Binary data. Following characters are binary data.
E	Variable description. This sentence is used to describe variables which are available o device.
F	Selection variable elements. The sentence contains description of one of the selectio variable options.
M	Message. The sentence contains message text.
P	Priority information.
Q	Priority information request.
S	Device status.
T	Housekeeping sentence.
V	Update variable value. It is send when a variable value changes.
X	Set variable value. Send when connection wants to set variable.
Y	Set variable value, update default value. Send when connection wants to set variabl and also change its default value.

Updates blocking state.

Binary data channel

data connection id
data size
data type

This sentence describes binary data connection. It is send before any data bytes are send, to start new binary data channel. Data connection id is used to track arriving data.

Binary data

data connection id
sentence data size
data of previous specified size

After data connection is established, data can flow through connection. The are identified by data connection id. The receiving part assembles them together. As data size is specified, data are send without any escaping.

Variable description

flags
name
description

Send description of new variable. Description contains variable flags, name and description used for FITS comment. Please see section on variables for detailed discussion.

Selection variable elements

selection variable name
selection string

This sentence specifies strings which will be in selection list for selection variable. The selection string is added to top of the selection variable list.

Message

timestamp
originator
message type
message text

This sentence sends message from one element to the other. Massage is send together with local time when it was generated, originator of the message and message type, which specifies message severity.

Priority information

priority client ID
priority timeout

This sentence is send from central daemon when new priority client is selected. Priority client can perform operations which might collide when they are executed from more then one component.

Priority information request

have priority 1 if receiving part currently has priority, 0 if it does not have priority.

This sentence is send when a connection receives or lost priority.

Device status

device status new device status

Sends new device status of the device. This sentence is used to distribute informations about new device state.

Housekeeping sentence

type either string "ready" or "OK"

This sentence is used to send housekeeping informations. As RTS2 block are single threaded, they can block in endless loop. When this occurs, the blocks will not call routine to check for incoming TCP/IP packets.

When connection was inactive for 2 minutes, the block send outs *"T ready"* sentence. The other block reply with *"T OK"*. If reply is not received within 2 minutes, connection is ended.

Update variable value

variable name name of the variable which value is changed

new value as string

This call is transmitted when the variable value of the transmitter was changed. It pass new value to the other part of the communication. Value is parsed by value parser, so it can consist of any string sequence.

Set variable value

variable name name of the variable which will be set

operation operation, which will be performed. Usual operations are +=, -=, =

operand string which describes operand value

This sentence sends variable update request. The device tries to perform variable change as requested. The response is either changed as requested without error, error during change or change was queued. When the change is queued, server informs user when new value takes effect by sending variable update sentence with new value.

Set variable value, update default value

variable name name of the variable which will be set

operation operation, which will be performed. Usual operations are +=, -=, =

operand string which describes operand value

This sentence sends variable update request. The current variable value and default variable value will be both updated. Response is similar to plain variable update sentence.

A.6 Basic commands

RTS2 has a limited set of commands which are supported by all connections. They are handled in blocks superclasses. The commands are described in following table:

command	comment
auth	command used for device authorisation
authorisation_key	transmits authorisation key
base_info	transmits device constants values. Value of device constant using the "V" sentence.
device	transmits informations about device.
device_status	query for device status. This command trigger a complex sequence of commands between devices and central server. The sequence results in device status update.
exit	ends the connection
info	transmits variables actual values. Values are transmitted with the "V" sentence.
killall	ends all commands. This is used in Rapid Reaction Mode overtake of the telescope.
ready	query device if it is ready.
script_ends	informs device that script has ended. Triggers setting variables back to default values.
this_device	sends name of the device which sends the command. Used during initial handshaking.

Devices can have own specific commands. But preference is given to commands linked to variable changes. Clear advantage of linking actions to value change is that value change is visible to other users, while command must send message in order to be visible.

That can be best demonstrated on simple telescope pointing. One approach is to have specific command to start telescope movement to target location. Other is to create variables which provides values of the telescope target location. Telescope then moves on change of this variable.

Target positions variable can be used to monitor system performance by comparing its value with a telescope actual position, which is displayed as another value. It is clear that this design is more transparent for user who would like to see how the system is behaving. Something is clearly wrong when there is a big difference between target position and actual telescope position.

A.7 Protocol Performance

The pure ASCII version of the protocol presents some additional overhead for communication protocol. It is without any doubts that large binary data, which are transmitted using the protocol and which usually represents images, must be transported in binary form. But given currently available high–speed detector, even relatively high rates of ASCII messages, which is about 2000 messages per seconds per component, is not sufficient. In order to allow such images to be processed properly, either library needs to be further optimised, or binary protocol must be used. Of course the best solution is the combination

of both approaches.

A.8 Synchronisation

Robotic observatory software system includes lots of various synchronisation mechanisms. One that probably comes first to mind is to not expose while mount is moving. But there are other synchronisation cases. Filter wheels and focuses usually does not move during exposure. And the opposite – perform some action, which affect the image, during exposure – is also required. For example calibration source must be swung while camera is exposing.

The paradigms, which governs how synchronisation is handled in RTS2, is: *"Try to do as much as possible, and leave it to commands to decide when to execute"*. This philosophy resulted in creation of different queues inside RTS2 building blocks, which holds commands while they wait for device to reach state when they execution can be performed. The used approach may look difficult, but it provides robust way how to handle synchronisation.

The other possible approach – pre–plan sequence in which commands shall be executed and the carry this execution – would be most probably simpler to debug and understand. But it most probably would results in system which will not be as robust to various devices failure as plain *"execute them and let them care of themself"* approach offers.

A.9 Command execution

Each command is queued for execution using *queCommand* method. Apart from the command, this method takes bit mask of states, which can block command execution. If this mask is zero, the command is send to target device without any extra operation performed.

Otherwise, following algorithm is executed:

1. Command is send to the device to report its current blocking status.
2. Device sends command to ask for its blocking state to central server.
3. The central server sends commands to all devices which are listed as possibly causing blocking of the device which asks for its blocking state
4. Once all blocking states are updated, final blocking state is send back to the device.
5. Device pass blocking state to the client.
6. Client check if the blocking state is acceptable for command execution. If it is, client send command to the device and this algorithm ends.

7. Client waits for updates of device blocking state.

8. Once the blocking state update is received, algorithm continues with step 6.

But system operation includes complex synchronisation scenarios, where this approach will not yield optimal result. The best example are telescope corrections. The images are processed as soon as possible, exact location of the image center is calculated, and correction between entered and real position are send to the telescope. The correction must be applied when none of the cameras attached to the telescope is exposing. It is quite easy to find an example how system with two cameras can remain in state when at least one of the camera is exposing forever, thus disabling possibility to execute any correction movement.

To deal with this, following simple extension to the algorithm is added:

1. Telescope driver receives correction from image processing process

2. Telescope driver sends to centrald daemon and all connected devices blocking mask, indicating that it is moving

3. Central daemon distributes moving mask to all connected devices. From now on the system will postpone all commands which required telescope to track the position.

4. Telescope driver sends query to central daemon, asking for its blocking state

5. Central daemon distribute query to all devices which might block telescope movement

6. Devices respond to central daemon. They previously received new blocking mask from telescope driver, so they will not start new exposure

7. Central daemon collect informations from the devices and send the resulting mask to telescope driver.

8. If telescope can move, algorithm continue with step 11.

9. Telescope waits for updates of device blocking state.

10. Once the blocking state update is received, algorithm continues with step 8.

11. Telescope starts moving.

12. If during moving any new exposure is tried, it is postponed, as system is not ready to receive it.

13. Telescope ends moving, sends blocking state without moving bit set to the central daemon.

14. Central daemon distributes blocking state updates to the device.

15. Devices receives blocking state update, If there is any queued exposure, the exposure is started.

This algorithm takes advantage of the blocking mechanism, developed for command execution. It works with telescope, filter wheels and any other possible devices which might interfere with camera exposures.

A.10 Example

Here is an example code, showing implementation of the simple sensor device, which has one integer and one selection variable. The device is connected by serial port, so it provides a command line option to specify which serial port is used, with "/dev/ttyS0" being the default serial connection.

```
#include "device.h"
#include "connection/serial.h"

class ASensor:public rts2core::Device
{
  private:
    rts2core::ValueInteger *testInt;
    rts2core::ValueDouble *testDouble;

    const char *serialDev;
    rts2core::ConnSerial *serialConn;

  protected:
    virtual int processOption (int opt)
    {
      switch (opt)
      {
        case 'f':
          serialDev = optarg;
          break;
        default:
          return rts2core::Device::processOption (opt);
      }
      return 0;
    }
```

```
    virtual int init ()
    {
      int ret;
      // first call init from parent class, as that will also parse command line optior
      ret = rts2core::Device::init ();
      if (ret)
        return ret;
      serialConn = new rts2core::ConnSerial (serialDev, this, rts2core::BS9600, rts2cor
      return serialConn->init ();
    }

    virtual int setValue (rts2core::Value * oldValue, rts2core::Value * newValue)
    {
      if (oldValue == testInt)
        return 0; // full version will write here to the device
      if (oldValue == testDouble)
        return 0; // same as above
      return rts2core::Device::setValue (oldValue, newValue);
    }

  public:
    ASensor (int argc, char **argv)
    :rts2core::Device (argc, argv, "S1")
    {
      createValue (testInt, "TEST_INT", "test integer value",
        true, RTS2_VWHEN_RECORD_CHANGE, 0, false);
      createValue (testDouble, "TEST_DOUBLE", "test double value", true);

      addOption ('f', NULL, 1, "serial port used for device communication");

      serialDev = "/dev/ttyS0";
      serialConn = NULL;
    }
};

int main (int argc, char **argv)
{
  ASensor device (argc, argv);
  return device.run ();
}
```

Appendix B

Constant values

B.1 Devices types

Numerical codes for RTS2 device types

Code	Device type
1	Central server
2	Telescope mount
3	CCD (Camera)
4	Dome
5	Weather sensor
6	Rotator
7	Photometer
8	Long-term planner
9	GRB daemon
10	Focuser
11	Moveable mirror
12	Cupola
13	Filter wheel
14	Auger shooter
15	Generic sensor
20	Executor
21	Image processor
22	Selector (short-term scheduling)
23	XML-RPC daemon (provides Hyper Text Markup Language (HTML) and JSON interfaces)
24	INDI bridge
25	Logger daemon
26	Scriptor (simple scripting)

B.2 System states

Numerical codes for RTS2 system states

Code	Meaning
0	Day
1	Evening
2	Dusk
3	Night
4	Dawn
5	Morning
11	Soft off
12	Hard off
0x10	Standby state (bit-masked)

B.3 Target types

Character code for target type

Type	Description
c	Calibration targets
d	Dark target
E	Elliptical targets, for asteroids and similar objects. Orbital parameters are taken from MPEC
f	Flat target
G	Gamma Ray Burst
H	HETE Field of View (FoV)
I	Integral FoV
l	Landolt calibrations fileds
L	Planets
m	Modeling target (for construction of pointing models)
o	fOcusing target
O	Opportunity targets
p	Plan target
P	Galatic Plate Scan
S	Sky survey
t	technical observation
T	terrestial (fixed ra+dec) target
W	Swift FoV

Appendix C

JSON API

The JSON is a lightweight data-interchange format. RTS2 provides JSON API which allows external programs to control the observatories and other systems running RTS2. Calls are realized over HTTP as GET/POST requests, with arguments used to parametrize call. Calls are handled by rts2-xmlrpcd component of the RTS2. Data returned through HTTP are JSON encoded data.

In order to be able to use JSON API, rts2-xmlrpcd must be installed and running on your system. Its manual page, available on most systems under *man rts2-xmlrpcd*, provides details needed for its configuration.

API calls are documented in the following paragraphs. Structure of those paragraphs is best to be depicted by providing an example of the description, which follows.

/api/example

Example API calls, which does not change the system. Please note that example above expect that % and / characters will be properly URI encoded before passing as paremeters to GET call.

Example	
http://localhost:8889/api/example?doit=f&expand=%N/%u	
Parameters	
doit	If operation should be performed. Required.
optional	*Optional parameter. Are optional parameters are typeset in italic.*
Return	

JSON encoded hash, with "ret" set to 1 and "error" idenfiing possible error source.

So the documentation consists of name of the call, Example in the section above. Name is followed by description of the call, its purpose and anything related to the call.

Then follows subsections. The one named Example containts example call, assuming that server is running on localhost, on standard port 8889, and you don't need username and login to access rts2-xmlrpcd server from the localhost.

Parameters are described in special section. Parameter name is typed in **bold**. Parameters that are optional has description written in *italic*, while required parameters has description in plain text.

Last is the return section, which describes format and content of (usually JSON) data returned from the call.

C.1 Hardware access

/api/devices

List name of available devices

Example http://localhost:8889/api/devices

Return

Array with names of all available devices.

/api/devbytype

Example http://localhost:8889/api/devbytype?t=2

Parameters

t Numerical code of device type. Values for different types are specified in table **??**

Return

Array with names of devices of given type.

/api/expose

Start exposure on given camera. Queue exposure if exposure is already in progress. Most of the image parameters (exposure length, dark vs. light image) can be set through variables present in the camera device (exposure, shutter,..).

Example http://localhost:8889/api/expose?ccd=C0&fe=%N_%05u.fits

Parameters

Name of CCD device. Required.	ccd
File expansion string. Can include expansion characters. Default to filename expansion string provided in rts2.ini configuration file. Please See man rts2.ini and man rts2 for details about configuration (xmlrpcd/images_name) and expansion characters.	fe

Return

Camera values in JSON format. Please see C.1 for details.

/api/exposedata

Start exposure, return with data from the device.

http://localhost:8889/api/exposedata?ccd=C0&chan=1 **Example**

Parameters

Name of CCD device. Required.	ccd
File expansion string. Can include expansion characters. Default to filename expansion string provided in rts2.ini configuration file. Please see man rts2.ini and man rts2 for details about configuration (xmlrpcd/images_name) and expansion characters.	fe
Data channel to send as return. Data channels are counted from 0, defaults to 0.	chan

Return

Image data. Please see C.1 for details on data header.

/api/lastimage

Send data (in binary) from the last image. Data are raw client-specific data, e.g. no endiannes conversion is performed.

http://localhost:8889/api/lastimage?ccd=C0 **Example**

Parameters

Name of CCD device. Required.	ccd
Data channel number. Channels are counted from 0. Default to 0, e.g. first channel.	chan

Return

JSON exception if data cannot be located. Otherwise binary dump of last image data (binary/data content type). Image data are prefixed with header.

Header structure, imghdr, is defined in imghdr.h. All numbers are transported in network order, e.g. in big endian.

/api/currentimage

Receive current camera image. Works similarly to C.1, but if there is an image currently being readout from the detector, client will receive this image. Image data are send as they are read from the detector.

Example http://localhost:8889/api/currentimage?ccd=C0&chan=2

Parameters

ccd Name of CCD device. Required.

chan *Data channel number. Channels are counted from 0. Default to 0, e.g. first channel.*

Return

JSON exception if data cannot be located. Otherwise binary dump of last image data (binary/data content type). Please see C.1 for details.

/api/expand

Expand string expression for FITS headers.

Example http://localhost:8889/api/expand?fn=/images/xx.fits&e=%H:%M @OBJECT

Parameters

fn Filename of the image to expand.

e Expression.

Return

Expanded value as JSON string in expanded member.

/api/get

Retrieve values from given device.

Example http://localhost:8889/api/get?d=C0&e=1&from=10000000

Parameters

d Device name. Can be *centrald* to retrieve data from rts2-centrald.

e *Extended format. If set to 1, returned structure will containt with values*

some meta-informations. Default to 0.

from *Updates will be taken from this time. If not specified, all values will be send back. If specified, only values updated from the given time (in ctime, e.g. seconds from 1/1/1970) will be send.*

Return

The returned structure is a complex JSON structure, with nested hashes and arrays. Format of the returned data depends on **e** parameter. Simple format, e.g. when e parameter is 0, is following:

"d":	
{ **"variable name"**:*value,...* },	device variables and their values
"minmax":	list of variables with minimal/maximal allowed value
{ **"variable name"**:*[min, max],...* },	
"idle":*0 or 1*,	idle state. 1 if device is idle
"stat":*device state*,	*full device state*
"f":*time*	actual time. Can be used in next get query as from parameter

If extended format is requested with **e=1**, then instead of returning values in d, array with those members is returned:

[	
flags,	value flags, describing its type,..
value,	actual value
isError,	1 if value has signalled error
isWarning,	1 if value has signalled warning
description	short description of the variable
]	

/api/getall

Get data from all devices connected to the system.

Example

http://localhost:8889/api/getall

Parameters

e *Extended format. If set to 1, returned structure will containt with values some meta-information. Default to 0.*

/api/push

Publish-subscribe interface. This interface allows for subscribing to value and state changes. At begining, server send data with values of all subscribed variables. The connection is kept open, and client receive updates of variables as soon as the change is detected by rts2-xmlrpcd.

Example http://localhost:8889/api/push?F0=FOC_POS&F0=FOC_TAR&F0=__S__

Parameters

DEVICE=value,.. *Subscribe to value from device. If device name equals to* **centrald**, *central daemon values are send.*

DEVICE=__S__ *Subscribe to state change.*

Return

Connection is kept open, values and state updates arrives as chunked HTTP response. Retuned data contain on each line record about value or state change. Each line contains JSON hash with following entries:

"d":*device name*,	device name
"t":*timestamp*.	date and time when data were send
"v":{*variable name*:*value*,..},	name of updated variables and their new values
"s":*device state*	new device state
"sf":*timestamp*	state from
"st":*timestamp*	state to

Here is an example for subscription to F0.FOC_TAR and C0.__S__:

```
> http://localhost:8889/api/push?F0=FOC_TAR
< {"d":"F0","t":1346680919.234,"v":{"FOC_TAR":[50332964,10,0,0, \
"focuser target position"]}}
< {"d":"C0":,"s":8,"sf":1346680912.234}
< {"d":"F0","t":1346680923.765,"v":{"FOC_TAR":[50332964,199,0,0, \
"focuser target position"]}}
```

/api/set

Set variable on server. Can set complex values, for example **camera ROI** (4 integers), through string containing new value as can be understand with **X** command from rts2-mon.

Example http://localhost:8889/api/set?d=C0&n=exposure&v=200

Parameters

Device name. If set to **centrald**, then values from rts2-centrald are set.	d
Variable name.	n
New value.	v
Asynchronous call. Asynchronous call return before value is confirmed set by the device driver.	async

Return

Return values in same format as C.1 call, augumented with return status. Return status is 0 if no error occured durring set call, and is obviously available only for non-asynchronous calls.

/api/inc

Increament variable. Works similarly to C.1, just instead of running = (assign) operation, runs += operation.

Example

http://localhost:8889/api/inc?d=C0&n=exposure&v=20

Parameters

Device name. Can be **centrald** to set value of rts2-centrald.	d
Variable name.	n
Increment.	v
Asynchronous call. Asynchronous call return before value is confirmed set by the device driver.	async

Return

Return values in same format as C.1 call, augumented with return status. Return status is 0 if no error occured durring set call, and is obviously available only for non-asynchronous calls.

/api/dec

Decrement variable. Works similarly to C.1, just instead of running = (assign) operation, runs -= operation.

Example

http://localhost:8889/api/dec?d=C0&n=exposure&v=20

Parameters

Device name. Can be **centrald** to set value of rts2-centrald	d
. Variable name.	n
New value.	v
Asynchronous call. Asynchronous call return before value is confirmed set by the device driver.	async

Return

Return values in same format as C.1 call, augumented with return status. Return status is 0 if no error occured durring set call, and is obviously available only for non-asynchronous calls.

/api/mset

Set multiple server variables. This call is similar to C.1 call.

Example http://localhost:8889/api/mset?C0.exposure=20

Parameters

VARIABLE=value List of variables and values to set.
async *Asynchronous call. Asynchronous call return before value is confirmed set by the device driver.*

Return

JSON hash. It contains "ret" value, with return code.

/api/statadd

Add value to statistical variable.

Example http://localhost:8889/api/statadd?d=WEATHER&n=windspeed&v=20

Parameters

d Device name. Can be **centrald** to set value of rts2-centrald
n . Variable name.
v New value.
async *Asynchronous call. Asynchronous call return before value is confirmed set by the device driver.*

Return

Return values in same format as C.1 call, augumented with return status. Return status is 0 if no error occured durring set call, and is obviously available only for non-asynchronous calls.

/api/cmd

Sends command to device.

Example http://localhost:8889/api/cmd?d=EXEC&c=queue 1234

Parameters	
Name of device where command will be send.	d
Device command.	c
Asynchronous call.	async
Return extended format.	e

Return

TODO

/api/selval

Return array with names of selection values. It is only possible to call this function on selection variables, otherwise the call return an error message. Variable type can be decoded from C.1 call with **e** parameter set to 1. The returned extended format contains flags. If (flags & 0x0f) is equal to 7 (RTS2_VALUE_SELECTION), then the selval call will succeed.

Example

http://localhost:8889/api/selval?d=C0&n=binning

Parameters	
Device name. Can be **centrald** to set value of rts2-centrald	d
. Variable name.	n

Return

JSON array with strings representing possible values of the selection.

C.2 Scripting

/api/runscript

Run script on device. Optionally kill previously running script, or don't call script end, which resets device environment.

Example

http://localhost:8889/api/runscript?d=C0&s=E 20&kill=1&fe=%N_%05u.fits

http://localhost:8889/api/runscript?d=C0&S=E 2&fe=%N_%05u.fits

Parameters	
Device name. Device must be CCD/Camera.	d
Script. Please bear in mind that you should URI encode any special characters in the script. Please see **??** for details.	s
Script, but call it without calling script ends (without reseting device state). See s parameter. Only one of the s or S parameters should be provided.	S
If 1, current script will be killed. Default to 0, which means finish current	kill

action on device, and then start new script.

fe File expansion string. Can include expansion characters. Default to filename expansion string provided in rts2.ini configuration file. Please See man rts2.ini and man rts2 for details about configuration (xmlrpcd/images_name) and expansion characters.

Return

TODO

/api/killscript

Kill script running on device. Force device to idle, set empty script for it.

Example http://localhost:8889/api/killscript?d=C0

Parameters

d Device name. Device must be CCD/camera.

C.3 Target database API

/api/lastobs

Return data of the last (possibly still running) observation.

Example http://localhost:8889/api/lastobs

Return

Structure describing both last observation and target associated with it. Observation ID, times of slew, observation start and end, number of images and number of good images are provided for the observation. Target ID, target name, description, Right Ascension (RA) and declination are returned for target associated to the observation.

/api/obytid

Return observation by its ID.

Example http://localhost:8889/api/obytid?id=100

Parameters

id Observation ID.

Return

Structure describing both observation and target associated with it. Observation ID, times of slew, observation start and end, number of images and number of good images are provided for the observation. Target ID, target name, description, RA and declination are returned for target associated to the observation.

/api/targets

List available targets.

/api/tbyname

Returns list (JSON table) with targets matching given name.

http://localhost:8889/api/tbyname?e=1&n=%	**Example**
	Parameters
Target name. Can include % (URL encoded to %25) as wildcard.	n
Ignore case. Ignore lower/upper case in target names.	i
Partial match allowed. Similar to specifing name as %name% - e.g. allow anything before and after name.	pm
Allow extended format. Result will include extra collumns.	e
Return target(s) proper motion.	propm
Datetime (as ctime, seconds from 1/1/1970) for which target(s) values will be calculated.	from

/api/tbystring

Resolve position from provided string.

/api/create_target

Create new target.

http://localhost:8889/api/create_target?tn=Test1&ra=20&dec=-30&type=E	**Example**
	Parameters
Target name.	tn
Target right asc.	ra
Target declination.	dec
Target type.	type
Additional target information. For elliptical targets, this field should contain	info

one line MPEC describing asteroid orbit.

Return

New target ID.

/api/update_target

Update target informations. Update various target informations - its position, name, description string and flag indicating whenever it might be included in autonomouse selection.

Example http://localhost:8889/api/update_target?id=1000&enabled=0&ra=359.34&dec=87.2

Parameters

id Target ID.

tn *New target name. If not specified, target name is not changed.*

ra *New target RA. In degrees, 0-360. If not specified (or DEC is not specified), then target position is not updated.*

dec *New target DEC. In degrees, -90 to +90.*

pm_ra *New target proper motion in RA, in arcdeg/year. Proper motion can be set only for certain target types.*

pm_dec *New target proper motion in DEC, in arcdeg/year.*

enabled *New value of target enabled bit. If set to 1, target will be considered by autonomouse selector.*

desc *Target description.*

/api/change_script

Update target script

Example http://localhost:8889/api/change_script?id=1000&c=C0&s=E 10 E 20 E 30

Parameters

id Target ID.

c Camera name.

s New script.

Return

0 on success.

/api/labels

Return label ID.

Example

http://localhost:8889/api/labels?l=DDT&t=1

Parameters

label string.	l
label type. See labels.h for types values.	t

Label types are currently those:

1 PI (Project Investigator)

2 PROGRAM (Program)

Return

Return label ID. Return error if label cannot be found.

/api/tlabs_add

Add label to target.

Example

http://localhost:8889/api/tlabs_add?id=1000&ltext=New Label&ltype=1

Parameters

target ID	id
label to add to the target	text
label type. Please see C.3 for type discussion	ttype

Return

List labels with specified type attached to the target.

/api/tlabs_delete

Remove label from target.

Example

http://localhost:8889/api/tlabs_delete?id=1000&li=2

Parameters

target ID	id
label ID	li

Return

List of labels attached to the target after removal of the specified label.

/api/consts

Retrieve target constraints. Constraints are stored in XML files specified in rts2.ini file (observatory/target_path).

Example http://localhost:8889/api/consts?id=1000

Parameters

id target ID

Return

Array of target constraints in JSON format.

/api/violated

Return constraints violated in given time interval.

Example http://localhost:8889/api/violated?cn=airmass&id=1000

Parameters

const Constraint name. Please see rts2db/constraints.h for list of allowed constraint names.

id Target ID.

from *Check constraints from this time (in ctime, seconds from 1/1/1970). Default to current time.*

to *Check to this time. Default to next day (from + 24 hours).*

step *Checking of most of the constraints is done by checking constraint in steps from "from" time. This parameter specifies step size (in seconds). Default to 60 seconds.*

Return

Array of time intervales when the target violates given constrain.

/api/satisfied

Return intervals when a target satisfies all constraints.

Example http://localhost:8889/api/satisfied?id=1000&length=1800

Parameters

id Target ID.

from *From time. Search for satisified intervals from this time (in ctime, seconds from 1/1/1970). Default to current time.*

To time. Default to from + 24 hours.	to
Return only intervals longer then specified lengths (in seconds). Default to 1800.	length
Step size (in seconds). Default to 60.	step

/api/cnst_alt

TODO

C.4 Big Brother interface API

/api/schedule

Schedule observation from BB

http://localhost:8889/bbapi/schedule?id=1003&from=20000003	**Example**
	Parameters
Local target ID.	id
Schedule from this time.	from
Schedule to this time.	to
	Return

0 if target cannot be scheduled, or time from which target can be scheduled.

/api/confirm

Confirm target schedule

http://localhost:8889/bbapi/confirm?id=1003&schedule_id=1	**Example**
	Parameters
Local target ID.	id

/api/cancel

Cancel BB scheduling request.

http://localhost:8889/bbapi/cancel?schedule_id=1	**Example**
	Parameters
Schedule ID.	schedule_id

Return

Nothing

C.5 Big Brother API

/api/schedule

Schedule observation on single observatory

Example http://localhost:8889/api/schedule?observatory_id=1&tar_id=1000

Parameters

observatory_id Observatory ID
tar_id Target ID

Return

TODO

/api/schedule_all

Schedule observation on all observatories

Example http://localhost:8889/api/schedule_all?tar_id=1000

Parameters

tar_id Target ID

Return

BB schedule ID. Status of scheduling call can be queried with [**?**] call.

/api/schedule_status

Retrieve status of queued scheduling requests

Example http://localhost:8889/api/schedule_status?id=2

Parameters

id Schedule ID

Return

JSON encoded string with status of scheduling call on different observatories.

Appendix D

Installing and configuring RTS2

Assume an observer has an observatory and would like to run it under RTS2. This appendix provides details on the steps necessary to install and configure the RTS2 package for the forementioned observer.

D.1 Computer and operating system choice

RTS2 can run on various operating systems and hardware platforms. The best choice for an easy and straightforward installation is some recent version of Ubuntu.

D.2 Installation

There are two methods how to install RTS2 on a host computer. It is probably easiest to install the package on Debian - based system, using provided packages.

D.2.1 Installing RTS2 from source on Ubuntu

The installation on Ubuntu is simplified by the existence of an instalation script. To install the latest SVN version, one need only to run two commands:

```
wget http://rts2.org/ubuntu-rts2-install
source ./ubuntu-rts2-install
```

The **ubuntu-rts2-install** script does the following:

1. Installs all packages needed for RTS2 compilation
2. Performs the checkout of the latest version from SVN repository

3. Runs GNU toolchain (autoconf, automake, ./configure, make) to compile RTS2
4. Installs RTS2 under /usr/local directory
5. Installs RTS2 configuration files under /etc (devices, services, centrald and rts2.ini) with values that set up environment only with a single dummy telescope and camera
6. Creates and configures PostgreSQL **stars** database for RTS2
7. Runs RTS2 backround processes (daemons) and rts2-mon

After those commands are sucessfully run, the computer will contain RTS2 environment setup with a dummy telescope and camera. Please see below for how to customize such environment to include devices found on the observatory.

D.2.2 Installing RTS2 from Ubuntu (Debian) packages

The installation from a Ubuntu packages is probably the most straigthforward. The packages are developed on Launchpad, under Petr Kubánek's account. They are available from https://launchpad.net/~petr-kubanek/+archive/rts2, the following commands shall work on Ubuntu 12.04:

```
sudo add-apt-repository ppa:user/ppa-name
sudo apt-get update
sudo apt-get install rts2
```

The packages containts configuration script, which asks user for basic configuration parameters - observatory longitude, latitude and altitude - and based on user's responses fill in a template configuration files. After package installation, the system shall be ready to run.

D.2.3 Installing RTS2 from source code

Installation from source is more difficult. But it has its benefits - the user learns structure of RTS2, its dependencies and how exactly is the whole software package configured, including its database.

> *Warning:* Please be aware that to execute the following, user must pose a basic knowledge of Linux operating system. This description does not deal with details of how to install required libraries, how to solve problems arising from unsufficient user rights and similar. Please consult the Internet or install the RTS2 system using the Ubuntu installation script, if you do not know solution of the problems arising during the installation.

Prerequsities

RTS2 is not a standalone project. It depends on number of external open source libraries. Before installing RTS2 from the source code, it is required to install libraries it depends on, including header files of these libraries.

Following table list name of all libraries on which RTS2 depends, if they are required for core functionality, and contains comments indicating which functions the library provides.

library	r	comment
archive	-	reading and writing archive files, used for compression in rts2-xmlrpcd provided web pages
comedi	-	Linux drivers for I/O cards
cfitsio	x	C FITS I/O library
crypt	-	used for password encription
ca	-	EPICS library, needed only for EPICS drivers (CAHA 1.23m)
ecpg	-	PostgreSQL C++ binding
graphicsmagic	-	writes JPEG images
json-glib	-	GNOME JSON library
m	x	standard C math library
ncurses	x	nCurses terminal graphics
nova	x	libnova - celestial mechanics
nsl	x	C DNS support for host resolving
pthread	x	POSIX threads
readline	-	needed for POSIX builds on Solaris
socket	x	TCP/IP socket
soup	-	GNOME HTTP communication library
wcs	-	WCS library

Obtaining source code

RTS2 source code can be obtained from Subversion repository at svn.code.sf.net/p/rts-2/code/. The following command will checkout actual master branch:

```
svn checkout http://svn.code.sf.net/p/rts-2/code/trunk/rts-2
```

Stable RTS2 releases can be obtained from the SourceForge project pages or from branches inside SVN repository. Please consult SVN manual for details. For a list of branches, browse the repository at http://svn.code.sf.net/p/rts-2/code/branches/rts-2/

Running GNU toolchain - automake, autoconf and friends

After obtaining the RTS2 sources, one must run the GNU toolchain to turn the source code into binaries. There is a *autogen.sh* script, which runs standard

automake and autoconf related tasks:

```
#!/bin/sh

rm -f aclocal.m4

#AC_VERSION=-1.7
AC_VERSION=""

aclocal$AC_VERSION && libtoolize && autoheader && automake$AC_VERSION --add-mis
```

Running this will generate the *configure* script and configure input files (*Makefile.in and others*). The user must then run *configure* script to turn the input files into files used in compilation. Please consult GNU toolchain manuals for details.

Compilation

RTS2 standard build uses GNU toolchain. To compile a vanila source tree, checked from SVN repository, with supporting files generated by running *autogen.sh* script, the folling sequence must be done:

```
~/rts2$ configure
~/rts2$ make
```

D.2.4 Binary installation

To install RTS2 binaries and libraries under a prefix defined while running *configure* script, the user must run **make install**.

```
~/rts2$ make install
```

D.3 System configuration

RTS2 is configured from files stored in predefined configuration directory, usually */etc/rts2*. Various RTS2 processes except to find there *rts2.ini* master configuration file. RTS2 startup script except to find there *devices*, *services* and *centrald* files used as a configuration describing processes which should be launched through it.

.ini files uses ; for comment. All other files are usually parsed by *bash* script, and thus uses # (hash) for comments.

D.3.1 rts2.ini

rts2.ini files contains system configuration. There user specifies things such as observatory latitude, longitude and altitude. Please consult *rts2.ini* manual pages, or the example rts2.ini file for details.

The example *rts2.ini* file, available in *conf/rts2.ini*, contains "xxxx" for all values which needs to be modified during installation. Please change those values to the one matching your installation. The complete lists of those values is below:

```
[observatory]
; You must provide your observatory altitude (above see level) in meters.
altitude = xxxx
; You must provide your observatory longitude in degrees. Please bear in mind,
; that negative is W from Greenwich, positive values are for E from Greenwich.
; Central Europe have positive longitude, all America have negative longitude.
longitude = xxxx
; You must provide your observatory latitude in degrees. Positive values are
; for north, negative for south. Europe have positive latitude,
; Australia negative.
latitude = xxxx
```

D.3.2 devices

Devices configuration file contains lines with device daemons the RTS2 start script should attempt to start.

Format of the file is quite simple. The file contains space separated devices types, device driver name and device options. So one row contains:

```
{device type} {device driver name} {device name} {device options}
```

See the following example, which configures a system with LX200 based telescope, MI CCD camera, dome controlled by customized Zelio Programmable Logic Controller (PLC) and NUT based UPS sensor.

```
#type   driver  name    parameters
teld    lx200   TELE    -f /dev/ttyS0
camd    miccd   CCD1    -p 2345
dome    zelio   DOME    -z 192.168.10.1
sensor  nut     UPS     -n myups
```

Device daemons are called rts2-type-driver. Please run device daemon with -h option to receive a list of all supported parameters. For example, there is output from *rts2-teld-lx200 -h*:

```
petr@bravo:~$ rts2-teld-lx200 -h
Usage:
LX200 compatible telescope driver. You probaly should provide -f options to specify seri
```

```
rts2-teld-lx200 -f /dev/ttyS3
Options:
  --conndebug           record debug log of messages on connections
  -f                    serial device file (default to /dev/ttyS0
  --wcs-multi           letter for multiple WCS (A-Z,-)
  --max-correction      correction limit (in arcsec)
  --horizon             telescope hard horizon
  -r                    telescope rotang
  --block-on-standby    block telescope movement when switching to standby
  -s                    park when switched to standby - 1 only at day, 2 even at
  -c                    minimal value for corrections. Corrections bellow that v
  -g                    minimal good separation. Correction above that number wi
  -l                    separation limit (corrections above that number in degre
  -m                    name of file holding model parameters, calculated by T-P
....
```

The following table list all know telescope drivers. Please see footnotes for important details regarding drivers availability – some require external libraries and some are closed source, available for a fee.

Daemon name	supported devices
rts2-camd-alta	Apogee Alta CCDs. http://ccd.com
rts2-camd-danish	Danish CCD controller [1]
rts2-camd-altanet	Apogee Alta TCP/IP CCDs. http://ccd.com
rts2-camd-andor	Andor CCCs. http://andor.com
rts2-camd-apogee	Apogee CCDs. http://ccd.com
rts2-camd-arc	ARC ("Leach") controllers
rts2-camd-edtsao	SAO controller
rts2-camd-dummy	Dummy camera driver
rts2-camd-fli	Finger Lake Instrumentation (FLI) CCDs. http://fli-cam.com
rts2-camd-gether	Gigabit ethernet CCDs
rts2-camd-miccd	MI CCDs. http://ccd.mii.cz
rts2-camd-miniccd	Starlight X-press CCDs
rts2-camd-reflex	STA Reflex driver
rts2-camd-sbigusb	SBIG CCDs. http://www.sbig.com
rts2-camd-sidecar	Teledyne Sidecar CCDs
rts2-camd-v4l	Video4Linux webcams
rts2-camd-urvc2	SBIG paraller CCDs
rts2-cupola-mark	Prague's observatory cupola
rts2-cupola-dummy	Dummy cupola driver
rts2-cupola-maxdomeii	MaxDome II cupola
rts2-cupola-vermes	Vermes observatory cupola

[1] closed source, available for a fee

Daemon name	supported devices
rts2-dome-123	Calar Alto Hispano-Aleman (CAHA) 1.23m dome
rts2-dome-ahe	Astrohaven Enterprise Domes
rts2-dome-bart	BART observatory dome
rts2-dome-bootes1a	BOOTES 1A observatory dome
rts2-dome-bootes1b	BOOTES 1B observatory dome
rts2-dome-bootes2	BOOTES 2 observatory dome
rts2-dome-door-vermes	Vermes observatory dome doors
rts2-dome-fram	FRAM observatory dome
rts2-dome-ieec	Institut d'Estudis Espacials de Catalunya (IEEC) observatory dome
rts2-dome-opentpl	Astelco's OpenTPL domes
rts2-dome-zelio	Custom Zelio PLC controlled domes
rts2-dome-watcher	Watcher custom dome
rts2-filterd-alta	Apogee filter wheel. http://ccd.com
rts2-filterd-dummy	Simulated filter wheel
rts2-filterd-ifw	Optec Intelligent Filter Wheel. http://optecinc.com
rts2-filterd-mdm	Michigan-Dartmouth-MIT Observatory (MDM) filter device
rts2-filterd-fli	FLI filter wheel. http://fli-cam.com
rts2-filterd-fw102c	Thorlabs FW120C. http://thorlabs.com
rts2-focusd-atc2	ATC2 focuser
rts2-focusd-dummy	Simulated focuser
rts2-focusd-fli	FLI focuser
rts2-focusd-mdm	MDM focuser
rts2-focusd-mdm_bait	MDM Berkeley Automated Imaging Telescopes (BAIT) focuser
rts2-focusd-microfocuser	Micro Focuser.
rts2-focusd-nstep	NStep stepper controller.
rts2-focusd-opentpl	Astelco focusers. http://astelco.de
rts2-focusd-optec	Optec focusers. http://optecinc.com
rts2-focusd-robofocus	Robofocusers.
rts2-mirror-dummy	Dummy mirror device
rts2-mirror-fram	FRAM mirror device
rts2-teld-123	CAHA 1.23m telescope driver. Requires EPICS library to compile
rts2-teld-aptgo	Astro-Physics Telescope (APT) LX200 based protocol. http://www.astro-physics.com
rts2-teld-ascol	ASCOL compatible telescopes
rts2-teld-bait	BAIT compatible telescopes
rts2-teld-d50	Ondřejov 50 cm telescope with custom control
rts2-teld-dummy	Simulated telescope driver

Daemon name	supported devices
rts2-teld-gemini	Losmandy mounts. http://losmandy.com
rts2-teld-hlohovec	Hlohovec custom telescope driver
rts2-teld-kolonica	Kolonica custom telescope driver
rts2-teld-lx200	LX200 compatible mounts. Works with Meade and a lot of other compatible products
rts2-teld-lx200gps	LX200 compatible telescope with Global Position System (GPS) receiver
rts2-teld-lx200test	Another version of the LX200 driver
rts2-teld-meade	Original Meade LX200 protocol
rts2-teld-mdm	MDM observatory telescopes
rts2-teld-mm2	MM2 LX200 mounts
rts2-teld-nexstar	Mounts supporting Celestron's Nexstar protocol. http://celestron.com
rts2-teld-indi	Bridge for INDI telescopes
rts2-teld-opentpl	Mounts supporting Astelco's OpenTPL protocol
rts2-teld-paramount	Software Bisque Paramount mounts
rts2-teld-trencin	Trenčín custom telescope driver
rts2-rotad-dummy	Dummy rotator device
rts2-rotad-pyxis	Optec's Pyxis rotor http://www.optecinc.com
rts2-sensor-a3200	Aerotech A3200 stage controller
rts2-sensor-aag	AAG cloud sensor http://lunatico.es/
rts2-sensor-agilent-e3631a	HP-Agilent power source
rts2-sensor-apcups	APC UPS client http://apc.com
rts2-sensor-arduino	Arduino test sensor
rts2-sensor-bart-rain	BART rain sensor
rts2-sensor-bigng	T-Balancer bigNG smart fan controller
rts2-sensor-bwcloudsensorii-weaher	Boltwood cloud sensor
rts2-sensor-bootes1	BOOTES 1 custom sensor
rts2-sensor-bootes2	BOOTES 2 custom sensor
rts2-sensor-brooks-356	Brooks 356 micro gauege
rts2-sensor-cloud4	CloudMeter 4 driver http://mlab.cz
rts2-sensor-colores	Colores integrated control board
rts2-sensor-cryocon	Cryocon cryognecis controller
rts2-sensor-davis	Davis weather sensor
rts2-sensor-ds21	DS21 steppers
rts2-sensor-dummy	Simulated sensor
rts2-sensor-dws-t5	DWS T5 custom sensor found on BOOTES 4
rts2-sensor-flwo-weather	FLWO weather
rts2-sensor-fram	FRAM custom electronics

Daemon name	supported devices
rts2-sensor-fram-weather	Pierre Auger Observatory - FRAM UDP weather data
rts2-sensor-hygrowin	Hygrowin humidity sensor
rts2-sensor-keithley-487	Keithley picoAmpermeter http://keithley.com
rts2-sensor-keithley-scpi	Keithley Standard Commands for Programmable Instruments (SCPI) picoAmpermeters http://keithley.com
rts2-sensor-lakeshore-325	Lakeshore 325 cryogenics sensor
rts2-sensor-lakeshore-330	Lakeshore 330 cryogenics sensor
rts2-sensor-lakeshore-340	Lakeshore 340 cryognecis sensor
rts2-sensor-lpnhe	Laboratoire de physique nucléaire et des hautes énergies (LPNHE) custom board
rts2-sensor-mm2500	Newport MM2500 multi axis controller
rts2-sensor-mdm-bait	BAIT based MDM weather sensor
rts2-sensor-micropirani-925	Micropirani pressure sensor
rts2-sensor-mrakomer	Original cloud meter (Mrakomer) by MLabs http://mlab.cz
rts2-sensor-ms257	Newport MS 257 monochromator http://www.newport.com
rts2-sensor-ms260	Newport MS 260 monochromator http://www.newport.com
rts2-sensor-newport-lamp	Newport lamp driver
rts2-sensor-niratir	RATIR National Instruments (NI) steppers control
rts2-sensor-nut	NUT UPS driver
rts2-sensor-osn	Observatório de Sierra Nevada (OSN) weather sensor
rts2-sensor-phytron	Phytron stepper motors
rts2-sensor-pixy	Pixy lighting sensor
rts2-sensor-system	System sensor, providing available disk space
rts2-sensor-thorlaser	Thorlabs lasers http://thorlabs.com
rts2-sensor-tps534	TPS custom board
rts2-sensor-triax	Triax monochromator

D.3.3 services

Services file is similar to *devices* file described above. It contains services the RTS2 start script should start. The following services are available in RTS2 source tree:

Daemon name	functionality
rts2-augershooter	Client for cosmic ray showers at the Auger Observatory
rts2-grbd	GCN client
rts2-executor	Executes observations in automatic mode
rts2-imgproc	Image processor
rts2-selector	Provides system with sheduling
rts2-xmlrpcd	XML-RPC, JSON/REST and web server

The following listing provides an example file for starting a GRB reaction service, executing service, image processor and selector with two queues, named q1 and q2:

```
# service       name     options
grbd            GRBD
executor        EXEC
imgproc         IMGP
selector        SEL      --add-queue q1 --add-queue q2
```

D.3.4 centrald

Centrald file configures **rts2-centrald** component startup. It should containts one or two line.

A line in the centrald configuration file with "centrald" specifies to the RTS2 starting script that the local instance of the **rts2-centrald** should be started on the machine.

Line starting with "-" specifies option(s) to add to all local devices and services started by the startup script. In case the **rts2-centrald** is run on another machine, the centrald should containt "–server" string followed by server name where **rts2-centrald** runs.

Please note that almost all devices and services can connect to multiple centralds. This can be particularly useful for shared meteo station or a shared dome. In such cases it is better to put extra centrald(s) into device or service line in *devices* or *services* file.

D.4 Database configuration

RTS2 can be build without database support by passing "–without-pgsql" option to the *./configure* call. This section applies only to the RTS2 build with the database support.

RTS2 services uses PostgreSQL database to store various data. The database must be created and configured. There is a **rts2-builddb** script in **rts2/src/sql** to build the database. The script must be run by user with database administrator priviledges, and requires at least one argument - database name. Run **rts2-builddb -h** for details.

The script performs the following actions:

1. create database functions
2. create database tables
3. run all database updates from the *rts2/src/sql/updates* directory
4. grant priviledges to the tables by running the *grant* script

> *Warning:* RTS2 services must be granted access to the database. Database user can be specified in *rts2.ini* configuration file or on service command line. The best way to make sure the database user has enough permissions to manipulate RTS2 database is to to add it to "observers" group.

D.5 XMLRPCd configuration

rts2-xmlrpcd is a process acting as a bridge between RTS2 and the following external world applications:

- GUI developed in different languages
- Web browser browsing RTS2 database and pages
- Scripts quering RTS2 devices status and database
- Scripts checking system performance
- Scripts triggered by system actions or state changes

The rts2-xmlrpcd configuration file(s) are passed to the daemon with *–event-file* parameter. If multiple configuration files are provided, their content is merged when possible, otherwise later configuration file on the command line overwrites the previous one.

The configuration file can have XML tags for various configuration options. The file is described in rts2-xmlrpcd manual page, which is available for more detailed information concerning this issue.

Below is example configuration file, which contains most of the available options.

```
<!--
  Configuration file for events actions.
  Please read man rts2-xmlrpcd to learn more.
  Petr Kubanek <petr@kubanek.net>
-->
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="file:events.xsd">
```

```
<http>
  <docroot>/home/observer</docroot>
  <public>/graph/SD/*</public>
  <dir path="/images/luthien" to="/luthien"/>
  <default-channel>0</default-channel>
  <defaultImageLabel>@OBJECT @EXPTIME @FILTER</defaultImageLabel>
</http>
<events>
  <device name="C0">
    <state state-mask="1" state="2">
      <command>/etc/rts2/event.d/c0_reading</command>
    </state>
    <value name="exposure">
      <command>/etc/rts2/event.d/exposure_state</command>
      <email>
        <to>example@example.com</to>
        <cc>name.surname@example.com</cc>
        <bcc>name_surname@sub_domain.example.com</bcc>
        <subject>Hey, you are exposing!</subject>
        <body>It's my pleasure to inform <device>D1</device>
          you, <device>D2</device> that the camera is exposing!
          Juhuu! Exposure is <value device="D1">V1</value>
          <value>device_value</value></body>
      </email>
    </value>
  </device>
  <device name="DOME">
    <state state="4">
      <command>/etc/rts2/event.d/dome_opened</command>
    </state>
    <state state="1">
      <command>/etc/rts2/event.d/dome_closed</command>
    </state>
    <value name="sw_open_left|sw_close_left|sw_open_right|sw_close_right">
      <record/>
    </value>
    <message type="CRITICAL">
      <email>
        <to>example@example.com</to>
        <subject>Critical error on dome occured.</subject>
        <body>There is critical error on Bootes dome. Please act
          immediatelly.</body>
      </email>
    </message>
  </device>
  <device name="PRESS">
    <value name="pressure" cadency="60">
      <record/>
    </value>
```

```
    </device>
    <device name="SD">
      <message type="CRITICAL">
        <email>
          <to>example@example.com</to>
          <subject>Critical error on dome occured.</subject>
          <body>There is critical error on Bootes dome. Please act
            immediatelly.</body>
        </email>
      </message>
      <value name="test_int" cadency="3" test="$SD.test_int &lt; 20">
        <record/>
        <command>/etc/rts2/event.d/sd</command>
      </value>
    </device>
  </events>
  <bb>
    <server>http://localhost:8890</server>
    <observatory>1</observatory>
    <password>secret Password</password>
    <!-- <cadency>300</cadency> -->
  </bb>
</config>
```

D.6 BB configuration

Appendix E

Database structure

This section describes various tables created in the RTS2 SQL database. Those are created by scripts stored in *src/sql* directory of the RTS2 source code.

E.1 Observatory database

This section depicts structure of tables used to kept a single observation.

E.2 Observation scheduling and log

Following tables are used to schedule observations, and keep track of what and when was observed. The main structure is along **targets** - **observations** - **images** relation, which associates images to target.

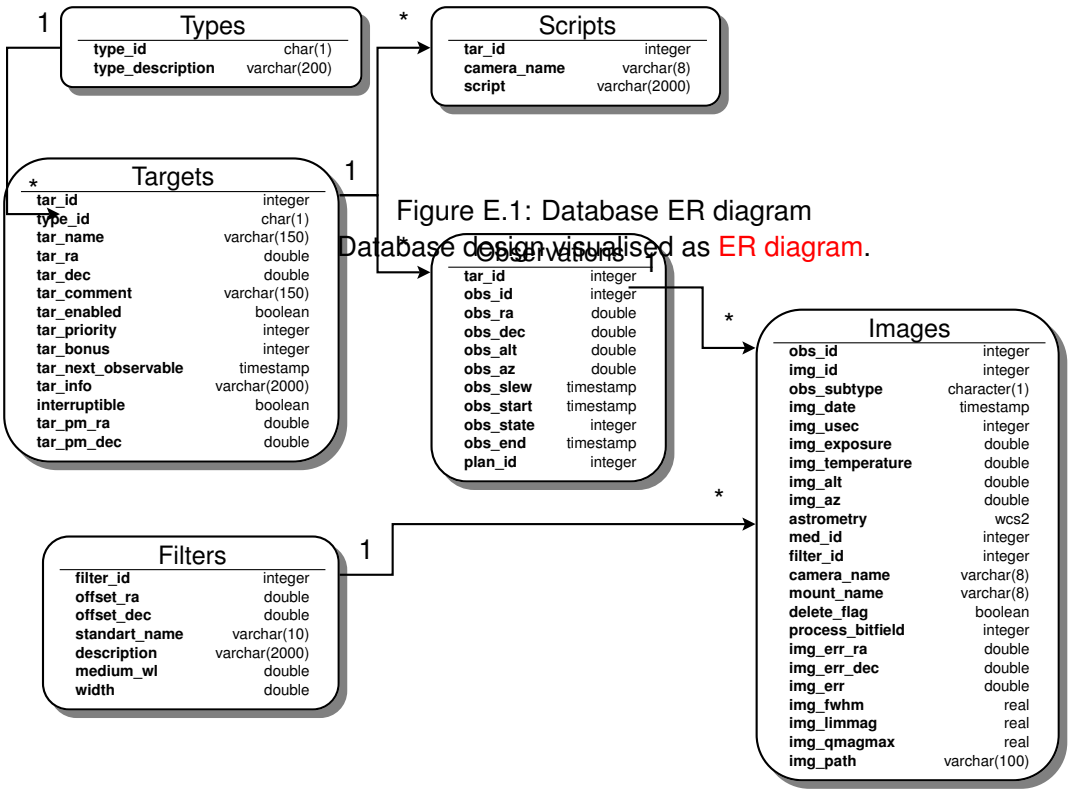

Figure E.1: Database ER diagram

Database design visualised as ER diagram.

E.2.1 targets

Targets, or objects entered in the database. Those object can be executed during autonomouse runs.

Column name	Column type	comments
tar_id	integer	target number
type_id	character(1)	target type, references to E.2.3 table
tar_name	varchar(150)	target name
tar_ra	double	RA of fixed target, in degrees (0-360)
tar_dec	double	Declination of fixed target, in degrees (-90 to +90)
tar_comment	text	free comment for the target
tar_enabled	boolean	if true, target is enabled for autonomouse selection
tar_priority	integer	target priority, used by merit function scheduler
tar_bonus	integer	target bonus
tar_bonus_time	timestamp	bonus validity time
tar_next_observable	timestamp	target cannot be observed until indicated time
tar_info	varchar(2000)	additional target data. Interpretation depends on target
interruptible	boolean	indicator if target can be interrupted
tar_pm_ra	double	target proper motion in RA
tar_pm_dec	double	target proper motion in Declination

E.2.2 Scripts

Target scripts. Text script describes actions undertaken by the camera for the given target.

Column name	Column type	comments
tar_id	integer	target number
camera_name	varchar(8)	camera name
script	varchar(2000)	script (as plain text)

E.2.3 Types

Types of targets one can encounter in the database.

Column name	Column type	comments
type_id	char(1)	character identification of target type
type_description	varchar(200)	type description

E.2.4 observations

This table holds list of observations acquired. Observation metadata, which includes fields needed to calculate time to slew to the target and time spend on the target, are present in this table.

Column name	Column type	comments
tar_id	integer	target number, references E.2.1
obs_id	integer	observation number
obs_ra	double	target RA at the beginning of the observation
obs_dec	double	target Declination at the beginning of the observation
obs_alt	double	target altitude at the beginning of the observation
obs_az	double	target azimuth at the beginning of the observation
obs_slew	timestamp	time when telescope started slew to target position
obs_start	timestamp	observation start time (usually first image)
obs_state	integer	observation state
obs_end	timestamp	observation end time (usually end of exposure of the last image)
plan_id	integer	plan ID (filled only in case observation was scheduled from plan)

E.2.5 images

This table holds images. Images are linked to an observation, which in turn is linked to target. There is not any direct connection between image and target. Auxiliary information, including filter and exposure time, are present in the table.

Abosolute image path is also recorded.

Column name	Column type	comments
obs_id	integer	observation number, reference to E.2.4
img_id	integer	image number (counting from 1 for every observation)
obs_subtype	character(1)	image subtype
img_date	timestamp	exposure start date
img_usec	integer	micro seconds part of start date
img_exposure	double	exposure length (in seconds)
img_temperature	double	CCD chip temperature
img_alt	double	altitude at the beginning of the exposure
img_az	double	azimuth at the beginning of the exposure
astrometry	wcs2	image WCS
med_id	integer	image medium, references E.2.7
camera_name	varchar(8)	camera name, references E.2.6
mount_name	varchar(8)	mount name, references E.2.8
delete_flag	boolean	indicating if image was deleted from the disk
process_bitfield	integer	image processing bitmask
img_err_ra	double	pointing error in RA
img_err_dec	double	pointing error in Declination
img_err	double	pointing error
img_fwhm	real	calculated image FWHM
img_limmag	real	calculated image limiting magnitude
img_qmagmax	real	
img_path	varchar(100)	absolute path to image data
filter_id	integer	image filter, references E.2.9

E.2.6 cameras

Table with list of cameras.

Column name	Column type	comments
camera_name	varchar(8)	camera name
camera_description	varchar(100)	camera description

E.2.7 medias

Column name	Column type	comments
med_id	integer	primary key
med_path	varchar(50)	path to medium
med_mounted	boolean	true if medium is available

E.2.8 mounts

Column name	Column type	comments
mount_name	varchar(8)	primary key
mount_long	double	mount location longitude
mount_lat	double	mount location latitude
mount_alt	double	mount location altitude
mount_desc	varchar(100)	mount description

E.2.9 filters

Column name	Column type	comments
filter_id	integer	filter number
offset_ra	double	
offset_dec	double	
standart_name	varchar(10)	
description	varchar(2000)	
medium_wl	double	
width	double	

E.3 Network management tables

E.3.1 observatories

Column name	Column type	comments
observatory_id	integer	observatory ID
longitude	float8	observatory longitude
latitude	float8	observatory latitude
altitude	float8	observatory altitude
apiurl	varchar(150)	Uniform Resource Locator (URL) to observatory API

E.3.2 targets_observatories

Column name	Column type	comments
observatory_id	integer	observatory number, refernces to E.3.1 table
tar_id	integer	target number, references to E.2.1 table
obs_tar_id	integer	target number on the given observatory

Figure E.2: Network scheduling ER diagram
Tables involved in network scheduling on BB side.

E.3.3 targets_observatories

E.4 Observation planning tables

E.4.1 plan

Column name	Column type	comments
plan_id	integer	plan ID
tar_id	integer	target ID
prop_id	integer	proposal ID
plan_start	timestamp	plan validity interval - start time
plan_status	integer	bit masked plan state
plan_end	timestamp	plan validity interval - end time
bb_schedule_id	varchar(50)	central node schedule ID

E.4.2 queues_targets

Column name	Column type	comments
qid	integer	queue entry ID
queue_id	integer	queue number
tar_id	integer	target ID
plan_id	integer	plan ID if the entry was created from plan
time_start	timestamp	entry start time
time_end	timestamp	entry stop time
queue_order	integer	entry order within the queue

Glossary

API An application programming interface (API) is a protocol intended to be used as an interface by software components to communicate with each other. An API may include specifications for routines, data structures, object classes, and variables.[1]. 27, 100, 107

APT Astro-Physics Telescope. 87

ASCOL ProjectSoft Telescope Control System (TCS) protocol. 87

ASCOM AStronomical Common Object Model, is an open initiative to provide a standard interface to a range of astronomy equipment including mounts, focusers and imaging devices in a Microsoft Windows environment. 12

BAIT Berkeley Automated Imaging Telescopes. 87, 89

BART Burst Alert Robotic Telescope, telescope at Astronomical Institute, Ondřejov, Czech Republic. 10, 87, 88

BAT Burst Alert Telescope, γ-ray telescope on board of the Swift mission.. 107

BB Big Brother, central server of the RTS2 network infrastructure. 27–29, 93, 101

BNL Brookhaven National Laboratory. 24

BOOTES Burst Optical Observation TElescope System, Spanish network of robotic telescopes for fast GRB observations. 10, 87, 88

CAHA Calar Alto Hispano-Aleman. 87

CCD Charged Coupled Detector. 9, 10, 24

chunked HTTP Variant of HTTP transfer. Lenght of the data is unknow to the server, connection stays open as long as new data be available. Chunked response can be used to send to client updates of server data, thus allowing for mechanism to bypass usuall HTTP request-response model. 70

COTS Commerical Off-The Shelf. 105

CVS Concurrent Versions System. 13

Debian Linux distribution. It uses own package system, which allows easy distribution of binary packages and integrates package installation with its configuration.. 81, 107

Declination One of the two direction coordinates of a point on the celestial sphere in the equatorial coordinate system, the other being either right ascension or hour angle. Declination's angular distance is measured north or south of the celestial equator, along the hour circle passing through the point in question.[2]. 97–99

EPICS **E**xperimental **P**hysics **I**nterface **C**ontrol **S**ystem, library developed for experiment control.[14]. 83, 87

ER diagram Entity-relation diagram provide visual description of database structure.. 96

ESA European Space Agency. 105

FITS Flexible Image Transport System (FITS) is a digital file format used to store, transmit, and manipulate scientific and other images[25].. 17, 18, 83, 108

FLI Finger Lake Instrumentation. 86, 87

FLWO Fred Lawrence Whipple Observatory, astronomical observatory located in Green Valley, Arizona, USA. 25, 88

FoV Field of View. 64

FRAM Fast Robotic Atmospheric Monitor, robotic telescope at Pierre-Auger Observatory, Argentina, owned and operated by Institute of Physics, Czech Republic. 10, 87–89

FWHM "Full width at half maximum" is an expression of the extent of a function, given by the difference between the two extreme values of the independent variable at which the dependent variable is equal to half of its maximum value[4]. It is used in image processing to measure extend of Gaussian shape sources.. 99

GCN Gamma Ray Bursts Coordinate Network. 90

GNOME Linux desktop environment, comes with a rich set of libraries. 83

GNU **G**nu is **N**ot **U**nix, Unix like open source operation system and its environment.. 104

GNU toolchain GNU software build system. 82–84

GPS Global Position System. 88

GRB GRB, gamma ray burst. 25, 90, 105, 107

HETE High Energy Transient Explorer, National Air and Space Administration (NASA)'s small experimental satellite able to detect GRBs, precursor to Swift mission. 64

HTML Hyper Text Markup Language. 63

HTTP Hyper Text Transfer Protocol. 27, 65, 83, 103

IEEC Institut d'Estudis Espacials de Catalunya. 87

INDI INDI stands for the Instrument-Neutral-Distributed-Interface. It is a protocol designed to support control, automation, data aquisition, and exchange among hardware devices and software frontends. 63, 88

Integral INTErnational Gamma RAy Laboratory, European Space Agency (ESA)'s mission to observe γ rays.. 64

introspective Introspection is the automatic process of analyzing a component design patterns to reveal the component properties, events, and methods. This process controls the publishing and discovery of component operations and properties.. 13

IP Internet Protocol. 107

JSON Java Object Script Notation, http://json.org, is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language. 11, 18, 27, 63, 65, 70, 72, 83

Launchpad Internet site for development of the Ubuntu packages. http://www.launchpad.net.. 82

Libnova Celestial Mechanics, Astrometry and Astrodynamics Library [5]. 27

Linux Unix based operating system, distributed with open source.. 82, 104, 107

LPNHE Laboratoire de physique nucléaire et des hautes énergies. 89

LSST Large Synoptic Survey Telescope. 24

LX200 Meade serial protocol. Common for small Commerical Off-The Shelf (COTS) telescopes.. 87, 88

magnitude Magnitude is the logarithmic measure of the brightness of an object, in astronomy, measured in a specific wavelength or passband, usually in optical or near-infrared wavelengths.[6]. 99

MDM Michigan-Dartmouth-MIT Observatory. 87–89

MFF UK Matematicko-fyzikální fakulta University Karlovy (Faculty of Mathematics and Physics, Charles University). 9

MPEC Minor Planet Ephemeride Center, center collecting small solar system observations and calculating and providing its orbits.. 76

NASA National Air and Space Administration. 105, 107

NI National Instruments. 89

NUT Network UPS Tools, collection of open-sourced UPS drivers and management programmes. 85, 89

OOP Object Oriented Programming. 10, 13

OSN Observatório de Sierra Nevada. 89

PLC Programmable Logic Controller. 85, 87

POSIX An acronym for "Portable Operating System Interface", is a family of standards specified by the IEEE for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.[8]. 83

PostgreSQL Open source relation SQL database. 82, 83, 90

ProjectSoft Czech Republic based company working in industry automation, including milk factories, breweries and telescopes. http://projectsoft.cz. 103

publish-subscribe In software architecture, publishâĂŞsubscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers. Instead, published messages are characterized into classes, without knowledge of what, if any, subscribers there may be. Similarly, subscribers express interest in one or more classes, and only receive messages that are of interest, without knowledge of what, if any, publishers there are.[9]. 18

pure virtual A pure virtual function or pure virtual method is a virtual function that is required to be implemented by a derived class, if that class is not abstract. Classes containing pure virtual methods are termed "abstract"; they cannot be instantiated directly. A subclass of an abstract class can only be instantiated directly if all inherited pure virtual methods have been implemented by that class or a parent class. Pure virtual methods typically have a declaration (signature) and no definition (implementation).[3]. 18

RA Right Ascension. 74, 75, 97–99

RATIR The Reonization And Transients InfraRed project, a 4 camera 6 channel instrument attached to San Pedro Martir 1.5m telescope [19]. 25, 89

RTS Remote Telescope System. 9, 13

RTS2 Remote Telescope System, 2^{nd} version. 9–13, 18, 19, 23, 65

rts2-centrald Central component of the RTS2. Act as register, name resolver and synchronization broker in the RTS2 environment. 11, 71–73

rts2-mon RTS2 monitoring program. It can display state and values of all devices and services present in the RTS2 environment. 18, 70, 82

rts2-xmlrpcd RTS2 daemon handling HTTP trafic, and providing XML-RPC and JSO APIs. 18, 27–29, 65, 66, 70, 83, 91

SCPI Standard Commands for Programmable Instruments. 89

select select system call. Used on Unix operation system for multiplexing wait on a set of file descriptor. Please see select manual page for details. 18

SMTP Simple Mail Transfer Protocol, protocol used to send e-mails[26]. 18

SPM San Pedro Martir observatory, Baja California, México. 25

SQL Structured Query Language, term used also to describe relation database. 95

SSH Secure SHell. 18

SVN Subversion. 13, 81, 83, 84

Swift NASA's mission to detect and observer GRBs. Satellite equipped with γ-ray imagining telescope BAT, X-ray telescope XRT and optical-UV UVOT. Launched in late 2004, current workhorse for γ-ray astronomy.. 64, 105

TCP/IP Transmission Control Protocol/Internet protocol, a common low level network protocol. 10, 11, 83

TCS Telescope Control System. 103

Ubuntu Linux distribution based on Debian distribution. Focused on desktop users.. 81, 82, 105

UDP Universal Datagram Protocol, non-guarantee Internet Protocol (IP) protocol. 89

UNAM Universidad Nacional Autóma de México. 25

UPS Uninterrupted Power Source, usually battery powered to provide power for safe shutdown of electric devices in case of powerfailure. 85, 89

URL Uniform Resource Locator. 100

UVOT Ultra Violet and Optical telescope, optical telescope on board of the Swift mission.. 107

WCS World Coordinate System, FITS standard for storing transformations from image to astronomical coordinates. 83, 99

XML-RPC eXtended Markup Language - Remote Procedure Call, http://xml-rpc.org, is a HTTP based protocol using XML to perform remote procedure calls. 11

XRT X Ray Telescope, X ray detector on board of the Swift mission.. 107

Bibliography

[1] Api. http://en.wikipedia.org/wiki/API.

[2] Declination. http://en.wikipedia.org/wiki/Declination.

[3] Fits. http://en.wikipedia.org/wiki/Virtual_function.

[4] Fwhm. http://en.wikipedia.org/wiki/Full_width_at_half_maximum.

[5] Libnova. http://libnova.sf.net.

[6] magnitude. http://en.wikipedia.org/wiki/Magnitude_(astronomy).

[7] Np–hard. http://en.wikipedia.org/wiki/NP-hard.

[8] Posix. http://en.wikipedia.org/wiki/POSIX.

[9] Publish-subscribe. http://en.wikipedia.org/wiki/Publish-subscribe_pattern.

[10] Queue. http://en.wikipedia.org/wiki/Queue_(data_structure).

[11] Ratir. http://ratir.org.

[12] S. D. Barthelmy. GRB Coordinates Network (GCN): A Status Report. *American Astronomical Society Meeting Abstracts*, 202, May 2003.

[13] A. M. Chavan, G. Giannone, D. Silva, T. Krueger, and G. Miller. Nightly Scheduling of ESO's Very Large Telescope. In R. Albrecht, R. N. Hook, and H. A. Bushouse, editors, *Astronomical Data Analysis Software and Systems VII*, volume 145 of *Astronomical Society of the Pacific Conference Series*, page 255, 1998.

[14] L.R Dalesio, M.R. Kraimer, and A.J. Kozubal. Epics architecture. In *International conference on accelerator and large experimental physics control systems, Tsukuba (Japan), 11-15 Nov 1991*, January 1991.

[15] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6:182–197, 2002.

[16] S. N. Fraser. Scheduling for Robonet-1 homogenous telescope network. *Astronomische Nachrichten*, 327:779, September 2006.

[17] T. Granzer, M. Weber, and K. G. Strassmeier. Three Years of Experience with the STELLA Robotic Observatory. *Advances in Astronomy*, 2010, 2010.

[18] Mark D. Johnston and Glenn E. Miller. Spike: Intelligent scheduling of hubble space telescope observations. In *Intelligent Scheduling*, pages 391–422. Morgan Kaufmann Publishers, 1994.

[19] C. R. Klein, P. Kubánek, N. R. Butler, O. D. Fox, A. S. Kutyrev, D. A. Rapchun, J. S. Bloom, A. Farah, N. Gehrels, L. Georgiev, J. J. González, W. H. Lee, G. N. Lotkin, S. H. Moseley, J. X. Prochaska, E. Ramirez-Ruiz, M. G. Richer, F. D. Robinson, C. Román-Zúñiga, M. V. Samuel, L. M. Sparr, C. Tucker, and A. M. Watson. Software solution for autonomous observations with H2RG detectors and SIDECAR ASICs for the RATIR camera. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 8453 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, July 2012.

[20] P. Kubánek et al. Rts2: a powerful robotic observatory manager. In *Advanced Software and Control for Astronomy. Edited by Lewis, Hilton; Bridger, Alan. Proceedings of the SPIE, Volume 6274, pp. (2006)*., July 2006.

[21] P. Kubánek, E. Falco, M. Jelínek, M. Prouza, J. Å trobl, M. Fuchs, and J. Gorosabel. RTS2: meta-queues scheduling and its realisation for FLWO 1.2m telescope. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 8448 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, September 2012.

[22] Petr Kubánek. Genetic algorithm for robotic telescope scheduling. Granada, Spain, 2008. Universidad de Granada.

[23] W. Mahoney and C. Veillet. The use of a genetic algorithm for ground-based telescope observation scheduling. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 8448 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, in press.

[24] W. Pence. CFITSIO, v2.0: A New Full-Featured Data Interface. In D. M. Mehringer, R. L. Plante, and D. A. Roberts, editors, *Astronomical Data Analysis Software and Systems VIII*, volume 172 of *Astronomical Society of the Pacific Conference Series*, page 487, 1999.

[25] W. D. Pence, L. Chiappetti, C. G. Page, R. A. Shaw, and E. Stobie. Definition of the Flexible Image Transport System (FITS), version 3.0. *Astronomy & Astrophysics*, 524:A42, December 2010.

[26] J.B. Postel. Rfc821: Simple mail transfer protocol. August 1982.

[27] P. Puxley and I. Jørgensen. Five years of queue observing at the Gemini telescopes. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 6270 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, July 2006.