The RTS2 protocol

Petr Kubánek[ab], Martin Jelínek[b], John French[c], Michal Prouza[d], Stanislav Vítek[b], Alberto J.Castro–Tirado[b] and Victor Reglero[a]

[a]GACE Valencia, Spain; [b]Instituto de Astrofísica de Andalucía (IAA-CSIC), Granada, Spain;
[c]University College Dublin, Dublin, Ireland;
[d]Fyzikální ústav Akademie Věd, Praha, Czech Republic

ABSTRACT

Remote Telescope System 2nd version (RTS2) is an open source project aimed at developing a software environment to control a fully robotic observatory. RTS2 consists of various components, which communicate via an ASCII based protocol. As the protocol was from the beginning designed as an observatory control system, it provides some unique features, which are hard to find in the other communication systems. These features include advanced synchronisation mechanisms and strategies for setting variables. This presentation describes the protocol and its unique features. It also assesses protocol performance, and provides examples how the RTS2 library can be used to quickly build an observatory control system.

1. INTRODUCTION

Remote Telescope System development started in 2001 as a student project at the Charles University in Prague. The goal was to develop a system capable of operating a telescope devoted to follow-up observations of γ – ray bursts (GRBs). This task was later extended to a development of a system for a full control of the robotic observatory, with one of the observations targets being a quick follow–ups of the γ – ray burst optical transients.

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.

Articles describing a complete observatory software environment are very rare. This paper tries to fill the gap by providing the description of a software environment for the complete control of the autonomous observatory operations.

Further author information: Send correspondence to Petr Kubánek : E–mail: petr@iaa.es

2. 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.[1] The RTS2 source codes, as well as the documentation, the bug and feature tracking system, are available at the SourceForge site at the URL http://rts-2.sf.net.

3. 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).[2] The first version allow only one way communication, with a client* sending the commands to a server†. 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, which 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:

*The process which started the communication

†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"

4. 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 6.

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

4.2 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.

4.3 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.

5. DEVICE STATES

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

5.1 Device status

Device status is a bit mask. It represents what operation is device performing. In table 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 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 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

Table 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 from 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 on device.
F	Selection variable elements. The sentence contains description of one of the selection 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 variable and also change its default value.

5.2 Blocking states

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

6. 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.

6.1 Sentence types

Sentence types overview is given in table 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

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.

7. 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.

8. 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.

9. 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.

10. 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.

11. 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 "../utils/rts2device.h"
#include "../utils/rts2connserial.h"

class Rts2DevSensorDummy:public Rts2Device
{
  private:
     Rts2ValueInteger *testInt;
     Rts2ValueDouble *testDouble;

     const char *serialDev;
     Rts2ConnSerial *serialConn;

  protected:
     virtual int processOption (int in_opt)
     {
       switch (in_opt)
       {
          case 'f':
            serialDev = optarg;
            break;
          default:
            return Rts2Device::processOption (in_opt);
       }
       return 0;
     }

     virtual int init ()
     {
       int ret;
       // first call init from parent class, as that will also parse command line option
       ret = Rts2Device::init ();
       if (ret)
         return ret;
```

```
      serialConn = new Rts2ConnSerial (serialDev, this, BS9600, C8, NONE, 40);
      return serialConn->init ();
    }

    virtual int setValue (Rts2Value * old_value, Rts2Value * newValue)
    {
      if (old_value == testInt)
        return 0; // full version will write here to the device
      if (old_value == testDouble)
        return 0; // same as above
      return Rts2DevSensor::setValue (old_value, newValue);
    }

  public:
    Rts2DevSensorDummy (int in_argc, char **in_argv)
    :Rts2Device (in_argc, in_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)
{
  Rts2DevSensorDummy device = Rts2DevSensorDummy (argc, argv);
  return device.run ();
}
```

12. CONCLUSION

This article presents core part of the Remote Telescope System – the protocol which is used to communicate between devices. Major protocol features are described, with aim to explain features designed to easy a developer live during developing control environment for complex observatory system. Article also contains section on protocol development history is described, a discuss a bit protocol performance issues, and small example which demonstrates protocol simplicity.

RTS2 is still being developed, with new features added almost daily. With its astronomical–observatory centered design, RTS2 provides an interesting alternative to complex communication libraries. The system provides communication interface together with facilities for observation management, image processing and other related task of an autonomouse observatory. The ultimate goal of the future development is to create package, which will be able to reasonably replace human operator on all possible observation instruments setups.

RTS2 is ready to enable others groups or individuals to switch from remotely controlled observatory to full observatory environment. We will be more then happy assisting customisation of RTS2 to other observatory environments.

REFERENCES

[1] Kubánek, P. 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).

[2] Postel, J., "Rfc821: Simple mail transfer protocol," (Aug. 1982).