Hindawi Publishing Corporation
Advances in Astronomy
Volume 2010, Article ID 902484, 9 pages
doi:10.1155/2010/902484

Research Article

RTS2—The Remote Telescope System

Petr Kubánek[1,2]

[1] *Image Processing Laboratory (IPL), Universidad de Valencia, 46010 Valencia, Spain*
[2] *Instituto de Astrofísica de Andalucia-Instituto del Consejo Superior de Investigaciones Cientificas (IAA-CSIC), 18008 Granada, Spain*

Correspondence should be addressed to Petr Kubánek, petr@iaa.es

Received 30 June 2009; Revised 25 November 2009; Accepted 29 January 2010

Academic Editor: Lorraine Hanlon

RTS2 is an open source observatory manager. It was written from scratch in the C++ language, with portability and modularity in mind. Its driving requirements originated from quick follow-ups of *Gamma Ray Bursts*. After some years of development it is now used to carry tasks it was originally not intended to carry. This article presents the current development status of the *RTS2* code. It focuses on describing strategies which worked as well as things which failed to deliver expected results.

1. Introduction

RTS2 originates from *RTS*, *the Remote Telescope System*. *RTS* was written during years 1999 and 2000 as a team project for Computer Science courses [1] by students of Mathematics and Physics faculty of Charles University in Prague.

RTS2 is primarily developed on the *Ubuntu Linux* distribution. It is known to run on a wide variety of Linux distributions, the *Solaris* operating system and partially on *Microsoft Windows* and *Mac OS X*.

Project source code can be obtained from the Web site http://rts2.org/. The project Wiki pages on http://rts2.org/wiki/ list group experiences with various fully autonomous observatory projects.

The outline of this paper is as follows: the first section is this introduction. The next section describes goals and history of the project development. The third section focus is a description of how various tasks inside *RTS2* are divided. The fourth summarises communication protocol. The next section describes approaches used during development. The sixth section forms the core of the article—it lists currently added features. The seventh section informs readers about currently running *RTS2* systems. The eighth is a reminder about our experiences with restarting failed devices. The contribution concludes with a section about expected developments.

2. Project Goals and Its Development

The original goal of the *RTS2* development was to produce software for a small robotic telescope, devoted to study Gamma Ray Burst (*GRB*) transients. The requirements of the project were

(i) must be fully autonomous,

(ii) must react as fast as possible to incoming GRB alerts,

(iii) must perform regular observations during periods with no GRB observation to verify system readiness,

(iv) must be modular to enable easy switching of the instruments.

Those requirements were laid in early 2000. They share similarities with other projects, notably *AudeLA* [2] and the *Liverpool Telescope System* [3]. At the time, open-source projects of similar scale did not exist. Available commercial, closed-source solutions were either associated with a single instrument, with questionable portability to others telescopes, or did not fulfill the requirement for fast switching between targets. It is worth noting that even today it is not easy to implement this requirement. It forms a strong entry barrier for those who would like to become *GRB* observers.

As the project advanced, additional requirements were added. They reflected experiences gained during development, and particularly pains and problems associated with early porting of the system to the other observatories:

(i) system must be robust enough to continue operations even when non-critical part(s) fail;

(ii) observer must have possibility to remotely interact with observations;

(iii) system must be fully configurable using configuration files;

(iv) system must provide clear description of what its components are performing;

(v) observer must be presented with a list of devices which failed;

(vi) the code should include dummy device drivers for testing the software without hardware.

Creation of dummy environments brings great benefits during system development. They enable developers to debug the system before it is deployed. There are still cases when errors and bugs are detected only during night runs. But as the project matures, the number of those cases significantly decreases—and there are recorded cases when new software, with significant new features, was deployed and it just worked, without any debugging.

As the number of observatories running *RTS2* grows, their management, fixing various small glitches, as well as their scheduling and data processing started to saturate staff time. We are aware of this, and we are in progress of creating tools which will allow us to manage network operations using a smaller amount of operators' time.

3. RTS2 Quick Overview

RTS2 is based on the "*plug and play*" philosophy. The parts which constitute the system can be started, restarted or stopped anytime, without affecting system performance. Special care is taken of resolving all possible blocking states, so the system will always respond to requests in reasonable time.

The whole system is user-space-based and except for drivers provided either by hardware manufacturers or by our group, it does not require any kernel-space-based components.

Code is designed around a central *select* system call, which picks any incoming messages. If there is no incoming message, class *idle* method is called. The code uses extensive hierarchy of its own C++ classes.

The *RTS2* system consists of the processes summarised in Table 1. Figure 1 shows processes and connections used on an example observatory, which includes three CCDs, two domes, and a single weather station serving both domes. For a detailed description please see [4].

Table 1: RTS2 processes.

Process	Description
Centrald	Central component of the *RTS2* system. It provides three main services—a list of devices and services present in the system, system state changes and synchronisation among devices.
Devices	Corresponds to hardware attached to the observatory system. Different classes of devices are provided. They represents hardware coming from different manufacturers.
Services	Represents execution logic of the system. They provide functions for the end-user-carry observations, receive GCN and other alerts, enables XML-RPC (XML-RPC web site: http://www.xmlrpc.com/) based access to the system.
Clients	Provides information to end-user. They are usually run in interactive mode, with end-user interacting with the programm. They include an interactive monitor and simple tools to execute observation scripts.

4. Communication Protocol

RTS2 employs its own communication protocol. A detailed protocol description can be found in [5]. The protocol is based on sending ASCII strings over TCP/IP sockets. It is fast, simple and robust. Among its important features are

(i) sending of "*I am alive traffic*," disconnecting connection if reply is not received—this removes connections to dead components,

(ii) ability to switch to binary mode for data transfer, to transfer images and other large data items,

(iii) traffic speed-up values which were not changed are not transported.

The protocol supporting libraries provides developers with a flexible and simple way to use common parts of the system. This results in a robust, transparent code, which can be easily extended. This greatly enhances software reuse. There are reasonable hopes and claims that the interfaces are simple enough to be understand by developers unfamiliar with the project.

5. Coding Style and Philosophy

It is widely rumoured that there are as many approaches to coding as there are software developers. This section aims to list rules and customs used during *RTS2* development. Although most of them are widely known we hope they are worth mentioning.

The best documentation for the code is the code itself. Readers of this article are encouraged to check out the system and study it. Code with complexity of the *RTS2* project cannot be made completely transparent. It is not expected that the developer will understood code on the first encounter. It is important that he/she find interfaces being

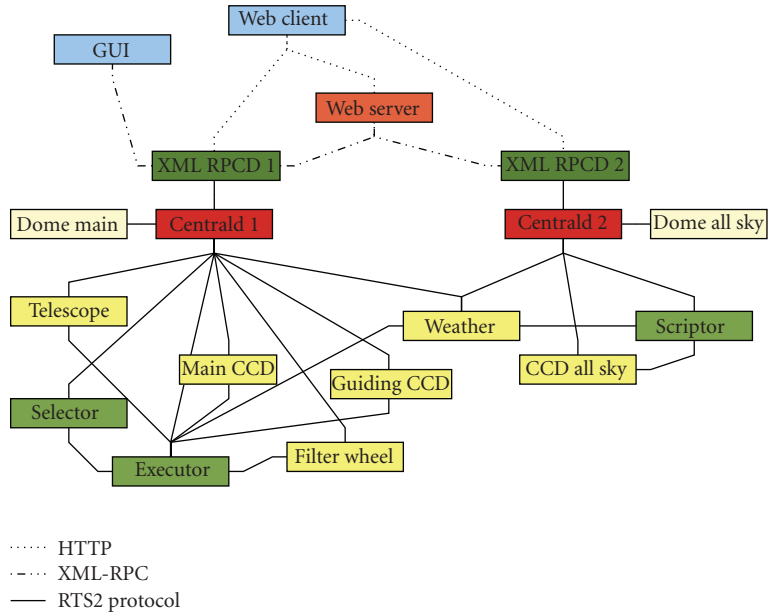

Figure 1: Example RTS2 environment. Two basic setups are present on the site—a telescope with a guiding CCD and a main CCD equipped with a filter wheel, and an all sky camera. There are two domes (one for telescope, second for all sky camera), three CCD detectors, *executor* and *selector* services controlling telescope setup and *scriptor*, a simplified executor service, controlling the all sky camera. Both observatories offers external access through XML-RPC protocol, provided by *XMLRPCD*. This is used by the Graphical User Interface and the Web server. *XMLRPCD* also provides a Web browser with direct access to some functions. Access to XML-RPC and Web functions can be protected with a password.

used. Later he/she can continue progressing towards system core classes.

5.1. Track Software Changes in Version Control System. The project is tracked in Subversion (Subversion web site—http://subversion.tigris.org/) tracking system. Previously version tracking relied on Concurrent Version System (CVS). (CVS web site—http://www.nongnu.org/cvs/). We have to admit that switch of the version control system to Subversion provides significant improvements, and we were really surprised with Subversion capabilities. Subversion really provides simply resolution for situations which were difficult or impossible to handle with CVS.

5.2. Code Releases. As the project is still under active development, a release formalism is yet not well established. There are usually two big releases during a year. All observatories are running Subversion code—code is regularly updated from Subversion and put to use. This way version management is also used for software distribution. We expect to calm this pace and establish a formal release mechanism.

5.3. Development Cycle-Release Early. Changes are usually committed to Subversion as soon as they compile without any errors. Before each commit, difference between new code and code in repository is reviewed. Sense and purpose of committed lines is examined once again.

This approach tracks vast majority of bugs right before they make it to the version control repository. After all changes are committed, and at least some documentation which explains new features is provided, behavior of the code is tested once again. It is up to developer, where tests will be performed. Usually the system is tested on dummy devices, before moving to a real observatory. But small changes, which should introduce predictable results, are tested directly on observatories.

5.4. Follow Common Practices. The common practices can be summarized by a few sentences, such as: Think twice, code once. Discuss with others, inform users about changes. Try to design a generic solution instead of a simple additions for new problems. Add new features slowly and test them before implementing another. Divide a complex problem to simpler subproblems, implement and test the solution for them first, and then integrate the code to solve the complex problem. Keep in mind that the best developers are able to design, code, test, and release no more then 100 lines per working day—try to keep number of lines small by reusing what is already available either in your code or in C++ libraries.

6. Current Developments

This section deals with features which were recently added to *RTS2*. The list provided bellow is not complete—please see project change-logs for a more in-depth description.

6.1. State Machines. RTS2 uses state machines. The states represents various states of the hardware, for example camera can have idle, exposing, and readout state.

The states were originally used for coding purposes, to distinguish various states of the code. They were displayed in monitoring applications, so the user was informed what the device should do.

Later state use was expanded toward synchronisations. States prevent the camera from taking exposures during telescope movement, and unwanted telescope movements during camera exposures. They are displayed on users displays, allowing observes to identify which device is blocking the next exposure or the next telescope movement. The use of states for synchronisation is depicted in Figure 2.

6.2. Default Configuration. Originally *RTS2* did not change configuration of devices prior to observations. That leaves the user responsible for device configuration. He/She can do that using either a monitor application or a script for observation.

This handling was later extended by *RTS2* remembering default values. The command *script_ends* was added to the protocol. It loads back all changed values.

This code was successfully used for focusing. The system was able to change back to the last known good focuser position even when a focusing run was interrupted. But it starts to show its bottlenecks, notably

(i) unpredictable behavior for the end user—he/she changes something, and suddenly another value appears in acquired images;

(ii) unknown default values—system unfortunately does not show which values were changed;

(iii) the system was quite hard to code, and as a consequence, quite hard to maintain.

Various solutions were considered to resolve those bottlenecks. The following was selected as the best one:

(i) when appropriate, the system offers default value and various offsets. The default value can be changed only on user request, and is never changed by the system. Offsets can be changed by user and/or system. Offsets are zeroed before beginning of the next observation;

(ii) or default values are provided only for values written in devices default configuration file.

Both cases can be best understood from an example. The first case can be best demonstrated on the new focuser interface and the second on handling mode settings on CCD.

6.2.1. Offsets, Default and Target Values. The interface provides end user with the following values:

(i) *current value,* which is read from focuser;

(ii) *target value,* equal to sum of values listed below;

(iii) *default value,* set by the user;

(iv) *focusing offset value,* settable by the user;

(v) *temporary focusing offset value,* settable by the user, zeroed at the end of the observing script.

Changes of the last three values are used for user focuser interaction. The default value is changed when the user would like to significantly change focuser position. For example the offset can be set for different optical thickness of the filters. And temporal focusing offset is used in the focusing script to change the focuser position during focusing runs.

If a focusing run is interrupted, the focusing script finished, or the focusing program is disconnected, the temporal focusing offset is set by focuser driver to zero. Then a focuser new target value is calculated and used.

Similar control is used for the telescope. There offsets are used to command dithering from the target position.

6.2.2. Camera Acquisition Modes. New modern cameras provide various settings. Some of the combinations of settings are very good for astronomy, while some are pretty bad. A desired behaviour of the system is to allow the user to set only good settings, avoiding the bad ones, while still leaving an option to set all variables manually.

A desired behaviour of the system will also switch the camera at the end of the script to a default mode so that the next script will be able to use camera, without changing any variable.

This functionality is provided by setting the camera mode back to default at the end of the script. A camera mode file provides various modes. The first is the default one, which is used to set camera. An example configuration file is included in the software distribution.

6.3. Observatory Scheduling. It is widely known that observatory scheduling is not a trivial problem. Scheduling is known to belong to NP-hard class of problems, which does not make the problem easier. On top of that, it is not always clear what exactly should be observed in order to maximise the scientific output of the observatory.

RTS2 currently provides users with three scheduling modes: dispatch scheduling, queue scheduling and preprogrammed night plan. The advanced, genetic algorithm scheduling, which should be able to schedule full night runs, is in the late integration phase. The scheduling modes are described in the following sections.

6.3.1. Dispatch Scheduling. Dispatch scheduling is the kind of scheduling used on most, if not all fully autonomous observatories. Each observable target is assigned a merit function, its expected benefits. Dispatch scheduling then calculates merits for all targets, picks the one with the best merit, and observes it. After this target is finished, the scheduler recalculates merit functions, and picks a new target. This approach is discussed by multiple authors, notably [6, 7]. *RTS2* provides this mode in the *selector* component.

6.3.2. Queue Scheduling. Queue scheduling schedules targets with the human-night observer in the loop. The night observer is presented with a list of possible targets, which are worth observing. Based on the current conditions,

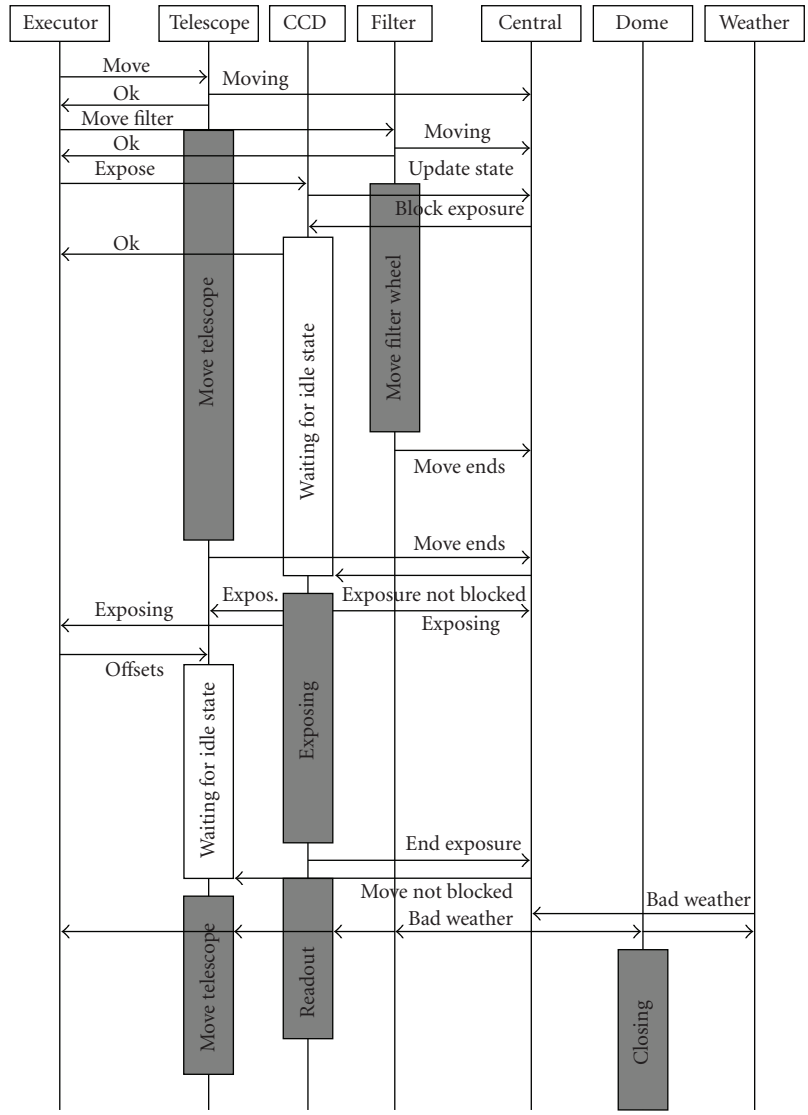

Figure 2: Interactions of devices using states for synchronisation. Note how various block are placed to keep telescope from moving and camera from exposing. The observation is interrupted by bad weather reported from a weather sensor, which closes the roof.

instrument setup and astronomer preferences, he/she picks a queue and executes its observation. After the target is finished, a new target is picked from the queue, or the queue is changed by the observer.

This scheduling is used by current big observatories, for example, by ESO VLT [8], CAHA, and IAC telescopes.

The observer can fill a queue of targets in the *RTS2 executor* component. There are plans to provide automatic selection from those targets, so the observer enters them in the evening, and the system will observe them during the night at optimal weather and time.

6.3.3. Night Plan. Another option is to provide a detailed night plan, which will list sequence of observations, their time and how images should be handled. Although *RTS2* does provide support for night plan, it does not provide any interfaces for plan creation and its management. The observer must fill a text file and load it into a database. Because of the complexity of this operation and lack of support tools this option is not widely used.

6.3.4. Genetic Algorithm-Based Scheduling. None of the scheduling modes discussed above is optimal. Dispatch scheduling lacks predictability and is short-sighted—it produces schedules which search for a local maxima, instead of focusing on long-term gains. Queue scheduling and night planning requires nontrivial involvement of the observer.

We proposed and implemented scheduling based on genetics algorithms (GA). The problem is described in full depth in [9]. Here is outline of this approach.

Observation targets can list various constraints and merits. The algorithm then searches for the Pareto optimal front [10]—a set of schedules which does not violate any constrains and their respective merit functions are at least

comparable to the other best schedules. The search for this front is performed with NSGA-II [11].

Experimental implementation of GA scheduling is part of the current *RTS2* developer branch. The main benefits of GA scheduling are a simple addition of new constraints and merits. Ease of its reuse is demonstrated in [12].

The plan is either to provide the observer with Pareto front schedules and let him/her pick the one he/she likes, or to have system select autonomously during night the path which will be followed.

6.4. Weather Blocking. It is very important to close the observatory roof when conditions are hostile for its normal operations and keeping it closed as long as those conditions prevail. Our records clearly indicate that having the roof open with bad weather over it produces significant problems as soon as the wrong roof state is detected.

The following sections deal with this problem. First the original approach is described. This is followed by a detailed description of current hardware and software setup, which is protecting equipment against the elements.

6.4.1. Original Weather Reporting. Keeping track of observatory conditions was originally a job of the dome module. It was regarded as a single point of failure. Dome code was handled with extreme care, and dome software was carefully tested after each software upgrade. Each device which might indicate bad weather sends information about weather state directly to the dome module. The dome module puts them together, decides if weather is favourable for the observation, and reacts accordingly—if all conditions were satisfied, it switched the system to "on" mode. If a single condition was not met, the dome control module tried to close the roof and switch the system to "standby" mode.

This solutions has the following problems:

(i) dome software testing is time consuming;

(ii) in case of a serious computer hardware problem the dome can be left open;

(iii) every new weather sensor requires modifications to dome control software and extensive testing before being put in operation.

6.4.2. Design of the State-Based Weather Handling. As different new sensors were added to the observatory setup, it becomes clear that weather reporting deserves special attention. The following requirements were put for the new algorithm:

(i) it must be easy to add new sensors for weather reporting;

(ii) it must be possible to configure the system to include various sensors;

(iii) the system must report bad weather in the event of a sensor failure.

Apart from those software requirements, additional requirements were put on the hardware responsible for roof operations:

Figure 3: Roof control box switches. Switching roof to manual or stop also stops autonomous observations. Unlike (graphical) user interfaces running on computers, it is very unlikely that this panel will fail. Watcher telescope, Boyden Observatory, Republic of South Africa.

(i) it must run independently from controlling computer,

(ii) it will be as reliable as possible,

(iii) it will be as simple as possible,

(iv) it must close the roof on its own when it loses connection to its controlling computer,

(v) it must report to the computer the states of all sensors connected to it.

The software and hardware implementation which fulfills all those requiremens is described in the next section.

6.4.3. Current Weather Blocking. *RTS2* centrald and all devices have a state. When changed, the state is propagated to all connections.

One of the bits in the state represents bad weather. If that bit is set, it means that the component (in case of device) or whole system (in case of centrald) reports violation of observing conditions, and therefore asks for controlled system shutdown.

Centrald also holds a list of devices which are mandatory for observation. If any of the mandatory devices is not present, or do not reply to centrald requests, the system is switched to bad weather state. This presents developers with a very simple way to add a new weather sensor to the system. They can use methods for weather state manipulation provided in *SensorWeather* class.

The ultimate weather protection element is on most systems a programmable logic controller (PLC). The inputs of this hardware are connected to the various sensors carring information about the roof state. The outputs of the relay are connected to the motors responsible for roof operations. The PLC is conservatively programmed. It is actually harder to open the roof then to close it.

PLC can be controlled manually with switches. For an image of actual roof control panel please see Figure 3.

If the computer would like to open the roof it must send signals every few seconds to the PLC, to check that it is alive.

Table 2: RTS2 installations.

Name	Location	Year	D
BART	Ondřejov, Czech Republic	2001	25 cm
BOOTES 1	INTA El Arenossilo, Spain	2002	30 cm
BOOTES 2	La Majora, Spain	2003	60 cm
FRAM	Pierre-Auger South, Argentina	2004	30 cm
Watcher	Bloemfotein, South Africa	2005	40 cm
BOOTES IR	OSN, Sierra Nevada, Spain	2005	60 cm
LSST CCD testing	BNL, New York, USA	2007	
Columbia	UC, New York, USA	2007	
D50	Ondřejov, Czech Republic	2008	50 cm
BOOTES 3	Blenheim, New Zealand	2009	60 cm
CAHA 1.23 m	CAHA, Spain	2009	1.23 m
LSST CCD testing	LPNHE Paris, France	2009	

The PLC is programed to close the roof if this "*I am alive*" signal is not received.

The critical failure points of this setup are motor failure, failure of power wire to motors and signals to PLC, PLC failure or power failure.

We do not have capability to make motors redundant. PLCs are designed for harsh industrial conditions. The PLC program is relatively simple and is very carefully tested. As its installation requires the presence of the person installing it on site, it is not sensitive to usual bugs introduced by system upgrades. Power backup is provided by UPS capable of closing the roof. The UPS state is monitored by *RTS2* and the roof is commanded to close if the remaining UPS uptime or battery level drops below a predefined values.

PLCs are also commodity value. In the unlikely event that the PLC is damaged or destroyed, a new can be purchased without significant problems on almost any place of the planet and installed within a few days from time the failure was detected.

Source code of the PLC program is available from *RTS2* subversion repository. As was shown by installing it on multiple sites, this setup is fully replicable and quite modular.

7. RTS2 Installations

Table 2 provides list of the current *RTS2* installations together with the year they started using *RTS2* and current diameter of the primary optic. The full listing of *RTS2* installations is provided on project web pages.

The system is currently being considered for various new and refurbished telescopes. Please note that 1.23 m at CAHA is currently operated in semiautomatic mode, and *RTS2* is not on controls only during a few nights. More details about this can be found in [13].

8. Coding Strategy—To Restart or to Code Properly?

At the beginning of the project, we sometimes employed a "last chance" strategy where if something fails (which in our case usually means "produces core dump"), the system will notice it, wait for a while and restart the failed part. Although this looked as very wise approach, it turned out that it hides potential serious problems and can affect system performance. Moreover, with errors in certain places it can produce situations when devices will keep restarting, produce a few lines in the system log, and exit.

Our experience is that the *if it fails, let system restart it and hope for the best* strategy is contraproductive and should not be employed. After fixing the most important bugs in the code, the programmes run on some setups for months without need for a single restart. If a problem is encountered, its root cause is found and fixed. A fixed driver is then installed and started with the remote ssh access.

The following things are vital for this strategy to work: compile all code with debugging option, change ulimit for core dumps to unlimited and knowledge of how to use *gdb* (GDB web site—http://www.gdb.org/) to find cause of the core dump. If all of the previous fails, have *valgrind* (Valgrind web site—http://www.valgrind.org/) or a similar memory profiler installed, with knowledge of how to use it to find memory allocation problems.

Once the coder has managed all those issues, not only will the system operate smoothly, but he/she also will not be temped to switch to some language with a garbage collector (which promises to free developers from memory allocation problems—the most probable reason for failure of the C++ code). That is not to say that high-level languages are not fine for some jobs—we do use Python for GUI and Java/PHP for Web pages. But in our experience, for low-level, hardware control algorithms, nothing beats properly designed and coded C/C++ code.

9. Expected Project Changes

Although *RTS2* is able to control autonomous observatories, there is still a lot which deserves more attention, some solutions and some coding. These items are presented in the following list.

9.1. Image Quality Monitoring. The system currently provides some basic quality checks. Those includes results of a plate solving routine provided by Jibaro [14] and/or Astrometry.net (Astrometry.net web site: http://www.astrometry.net/) packages, so observers knows if telescope is pointing towards the expected position.

The system lacks real-time display of various image quality parameters-average and median values, minimal and maximal values, number of detected objects (stars) in the image, and so on. Those are calculated and saved to the image header. Our experience shows that those should be easily accessible in real-time displays, as well as in tool for navigation through the image archive.

9.2. Image Processing. Image processing can be configured as an external script, providing relative photometry and other services. We would like to better integrate it with the system by

(i) adding calibration image database, and providing tools to maintain and use this database;

(ii) providing observations-to-paper solutions for predefined well understood problems, among others observatorion of planetary transit occultations and microlensing events.

9.3. Better User Interfaces. The system is primarily controlled through an ncurses based interface. While this is sufficient for experienced users, the interface is not well suited for occasional users.

Development of both Web and X-Windows Graphical User Interfaces is currently in initial phases. Plans also call for addition of applets for GNOME and other desktops environments, so we will be able to track telescope operations world-wide from a single desktop.

9.4. Network Management. The System currently lacks a central management console. Observatories are usually managed and monitored through ssh connections. While this approach is feasible, it quickly grows beyond the capabilities of a single maintainer.

There is a need for application, which will

(i) show states of individual observatories in the network,

(ii) list unresolved problems of individual telescopes, keeping track of actions to fix the problem, and the results of those actions,

(iii) enable scheduling of the whole network,

(iv) synthesise results obtained by network members.

It would be nice to have those features backed by simple, low-level, Internet interface. There is work in progress on an XML-RPC interface which will do just this.

9.5. Archive Access. *RTS2* keeps all important information in a database. Information about executed observations, and images acquired, together with basic image characteristics (which includes WCS coordinates, fitted by Jibaro and/or Astrometry.net) is recorded for quick retrieval.

Current archive access is provided primarily through console-based utilities. Without doubt in this is a bit too old fashioned in the graphical user interfaces age.

There were PHP scripts for Web-based archive access, which even included cut-out service. Unfortunately they were not very well designed. As the project matures and introduces changes to the database, those scripts cease to function, to the point when it was ruled to be too expansive to make them work again, as is the usual case with simple small fast PHP and other scripts.

Up to now multiple attempts to provide better solutions were made. Yet till now each of them has failed to deliver usable results.

Current attempts include a Google Web Toolkit XML-RPC-backed application. We have reasonable hopes that this will deliver promised results, although not in a short timescale.

10. Conclusions

This article presented an open-source system for robotic observatory control. It provides an overview of rationales for its development, its composition and its main features. It then focused on a list of recent improvements. The article concludes with lists of development items still left on the agenda.

Acknowledgments

The author would like to acknowledge generous financial support provided by Spanish *Programa de Ayudas FPI del Ministerio de Ciencia e Innovaci (Subprograma FPI-MICINN)* and European *Fondo Social Europeo*. Work on *RTS2* was supported, influenced and encouraged by numerous people, whose list would be too large for this article. Persons from this list which according to author deserve to be mention explicitly are Martin Jelínek, Alberto Castro-Tirado, Antonio de Ugarte Postigo, Ronan Cunniffe, Michael Prouza, René Hudec, Victor Reglero and Beatriz Sánchez Félix. The article was carefully reviewed by two anonymous referees, whose suggestions significantly improved it. Stephen Bailey made final gramatical improvements to the article.

References

[1] T. Jílek, et al., "Detekce astronomických objektů s proměnnou intenzitou za pomoci robotického teleskopu," Team project, Praha, UK, 2000.

[2] A. Klotz, F. Vachier, and M. Boër, "TAROT: robotic observatories for gamma-ray bursts and other sources," *Astronomische Nachrichten*, vol. 329, no. 3, pp. 275–277, 2008.

[3] S. N. Fraser and I. A. Steel, "Object oriented design of the Liverpool Telescope Robotic Control System," in *Advanced Telescope and Instrumentation Control Software II*, vol. 4848 of *Proceedings of SPIE*, pp. 443–454, Waikoloa, Hawaii, USA, August 2002.

[4] P. Kubánek, M. Jelínek, S. Vítek, A. De Ugarte Postigo, M. Nekola, and J. French, "RTS2: a powerful robotic observatory manager," in *Advanced Software and Control for Astronomy*, vol. 6274 of *Proceedings of SPIE*, Orlando, Fla, USA, May 2006.

[5] P. Kubánek, M. Jelínek, J. French, et al., "The RTS2 protocol," in *Advanced Software and Control for Astronomy II*, vol. 7019 of *Proceedings of SPIE*, Marseille, France, June 2008.

[6] S. N. Fraser, "Scheduling for Robonet-1 homogenous telescope network," *Astronomische Nachrichten*, vol. 327, no. 8, pp. 779–782, 2006.

[7] Y. Tsapras, R. Street, K. Home, et al., "RoboNet-II: follow-up obervations of microlensing events with a robotic network of telescopes," *Astronomische Nachrichten*, vol. 330, no. 1, pp. 4–11, 2009.

[8] A. M. Chavan, G. Giannone, D. Silva, T. Krueger, and G. Miller, "Nightly scheduling of ESO's very large telescope," in *Astronomical Data Analysis Software and Systems VII*, vol. 145 of *ASP Conference Series*, 1998.

[9] P. Kubánek, *Genetic Algorihm for Robotic Telescope Scheduling*, Máster en Soft Computing y Sistemas Inteligentes, Universidad de Granada, Granada, Spain, 2008.

[10] V. Pareto, *Manual di Economia Politica*, MacMillan, New York, NY, USA, 1906.

[11] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[12] F. Förster, N. López, J. Maza, P. Kubánek, and G. Pignata, "Scheduling in targeted transient surveys and a new telescope for CHASE," *Advances in Astronomy*, vol. 2010, Article ID 107569, 8 pages, 2010.

[13] J. Gorosabel, P. Kubanek, M. Jelinek, et al., "Recent GRBs observed with the 1.23m CAHA telescope and the status of its upgrade," *Advances in Astronomy*. In press.

[14] A. de Ugarte Postigo, M. Jelínek, J. M. G. Urquía, et al., *JIBARO: un conjunto de utilidades para la reducción y análisis automatizado de imágenes*, Astrofísica Robótica en España, Ed. Sirius, Madrid, Spain, 2005.