Hindawi Publishing Corporation
Advances in Astronomy
Volume 2010, Article ID 620424, 6 pages
doi:10.1155/2010/620424

Research Article

Robotic and Non-Robotic Control of Astrophysical Instruments

Gerd Küveler,[1] Van Dung Dao,[1] Axel Zuber,[1] and Renzo Ramelli[2]

[1] *Hochschule RheinMain, Institut für Automatisierungsinformatik, Am Brückweg 26, 65428 Rüsselsheim, Germany*
[2] *Istituto Ricerche Solari Locarno, Via Patocchi, 6605 Locarno, Switzerland*

Correspondence should be addressed to Gerd Küveler, gerdkueveler@hs-rm.de

Received 2 June 2009; Revised 20 October 2009; Accepted 24 December 2009

Academic Editor: Taro Kotani

We present a system of device control programs, developed for the solar observatory at Locarno/Switzerland (IRSOL). Because these programs are implemented as servers—clearly separated from the higher levels—scientific instruments, for example, telescopes, can be operated both in a user-controlled mode (GUI, *telnet*) and in a fully automated mode by use of a script. Astronomical instruments such as telescopes or spectrographs will be responded to by ASCII command strings, which are the same for all clients. In case a device control software does not support multiclient operation or in case it is used together with other devices in a complex measuring procedure, it is worthwhile implementing an intermediate layer that relieves the individual device control servers of routine tasks and provides for a safer operational sequence. In addition, the system may make use of an easy-to-learn script language specialised for controlling fully automated processes.

1. Introduction

An automation of technical procedures generally leads to

(i) a discharge of routine tasks,

(ii) a higher operating speed,

(iii) more reliability and precision,

(iv) easier handling,

(v) avoidance of costs and damage.

At large automation effects a higher productivity.

For scientific laboratories, here particularly in astrophysical observatories, some of these reasons may be specified.

(i) If the different levels of automation are clearly separated, it is possible to operate the various instruments (telescope(s), spectrograph(s), cameras(s), etc.) via graphical user interfaces (GUI) and text-based interfaces such as telnet or netcat at the same time.

(ii) The observer is able to combine several instruments of an observatory (or even several observatories) in order to perform a complex measurement procedure by making use of scripts.

(iii) An automatic execution of calibration procedures and of routine jobs (e.g., telescope and dome to target or home position) saves time and avoids maloperation and possible damage (e.g., moving the telescope while the shelter is not sufficiently opened).

The following features are not primary intentions of automation but desirable properties:

(i) It is advantageous to separate the various levels of automation in order to design a system more clearly arranged and in order to facilitate the exchange of defect or out-of-date components (hardware and software) without affecting the overall system.

(ii) If possible, industrial hardware standards should be used in order to avoid costs and to preserve a good chance of external support.

(iii) The same is true for using standard and easily portable programming languages (like C, C++, Java) and software tools (like LabVIEW). Furthermore, it will be easier to find the programmer's successor.

(iv) The higher levels of automation software (GUI, scripts, etc.) should run under the standard operating systems Windows and UNIX/Linux.

(v) Standard information transfer and data communication protocols should be used.

(vi) It is possible to create detailed logs (files, databases, etc.) that enable keeping track of what happened during the data acquisition in a very reliable way.

In this paper, we describe an automation system, developed for the Istituto Ricerche Solari Locarno (IRSOL), a solar observatory which meets the demands mentioned above. It consists of device control programs that operate the various instruments at their low level, of the *command server* which coordinates the interaction between several instruments and the user interfaces. Moreover, the *command server* is able to absorb numerous routine tasks, such as authorisation and syntax check. A special script language (AMI) was developed to allow for an easy assembly of complex measuring procedures. Our new telescope control software will be used as an example for a device control program.

2. Robotic and Non-Robotic Control

Ideally, automated devices will work in both a robotic and a non-robotic modes. This means that an instrument may be handled by a human user (non-robotic mode) as well as by a script in order to run a measurement procedure, optionally also in cooperation with other instruments (robotic mode). This can be achieved by means of a complete separation of the actual control hardware and software from the higher levels. If there are no special real-time requirements given, the communication with user interfaces or scripts of all kinds should follow the standard data communication protocol TCP/IP. Any control software should be designed as an automat with a definite number of functions. All of these functions can be activated through certain commands transferred via Ethernet from a remote computer. Due to modern programming languages and today's powerful hardware, any control function may be performed in a parallel mode, provided that there are no conflicts. Commands may be binary based or consist of ASCII-strings. Statistically, binary numbers are more compact than ASCII numbers but binaries have some important disadvantages with respect to data exchange: not all programming languages and tools use the IEEE format for the internal representation of floating point numbers. Misunderstandings whether a date is float (real, 32 bit) or double (64 bit) may arise and even integer numbers which are longer than one byte can lead to confusion because Intel processors—as opposed to most others—use byte swapping. In contrast, ASCII-strings are unique, independent of software and hardware. Moreover, ASCII-strings allow easy testing of commands by using a standard text-based interface such as *telnet*. Consequently, by using ASCII-strings, the access from all kinds of user interfaces and scripts is the same.

In this paradigm, any device control software is a server. Ideally, the server should be capable of handling multiple clients. This means that more than one high-level interface may simultaneously communicate with the control device, for example, a script and a GUI and *telnet*.

3. Command Server

Control programs for different instruments have many functions in common. Moreover, problems may arise if several clients communicate with the same device at the same time or if a device is involved in a complex procedure, jointly with other devices. For this reason, we developed a program system which generates an intermediate layer between the device control layer and the high-level layer (user interface, client). This program system we call *command server (cs)*. It is written in Java and thus executable on each computer on that the Java Runtime Environment (JRE) is installed. The *cs* is divided into the modules *command* (which exists only once) and *dispatcher* (which will be created for any application that connects to the *cs*). Between these modules a system of buffers will be established, one for each application.

The *cs* communicates with all applications currently active via TCP/IP. The application is typically a *device control* software. Every application first needs to register at the *cs* in order to get an application number or name as a link to its socket (IP address and port number). Subsequently, the *cs* creates a command buffer (*queue*) and a *dispatcher* for this application in a separate thread.

The server-module *command* of the *cs* looks for commands dispatched by a user interface or by a script. Each command will be checked for formal validity and, if the check succeeded, will be distributed to the addressed application. If a valid status query or an abort commands occurs it will be directed immediately to the *device control* program. Commands effecting an action (e.g., "telescope go to set point") will be buffered and delayed upon completion of the previous command. Queued commands are prioritised (all priority 0 commands will be executed prior to the first priority 1 commands). Many commands are management commands. These are merely dedicated to the *cs*, not to an application (e.g., "add a new application," "remove a command from the queue," "ask for a completion report of a previously given command"). Each command will be confirmed immediately. Syntactically wrong or illegal commands will be sent back to the client with an error byte.

The *dispatcher* sends the next command from the *queue* to its application and waits for the completion report. A command to an application must adhere to the format

application no. or name | *priority* | *command name* | *arbitrary string (may be lacked)* | "*<CR>*"

"|" represents a blank.

All commands usually consist of ASCII-characters found on any computer keyboard. In exceptional cases, binary parameters are possible as well. The different components of a command are always separated by at least one blank. Any command string and receipt should be terminated by a carriage return. Other terminators can be declared in a setup-file.

The *dispatcher* only sends the command name and the following string to the application, for example, *klambda 08 630.2123* (spectrograph go to wavelength 630.2123 nm, grid order 8). The report, returned by the application upon execution of the command, will also be stored in the queue

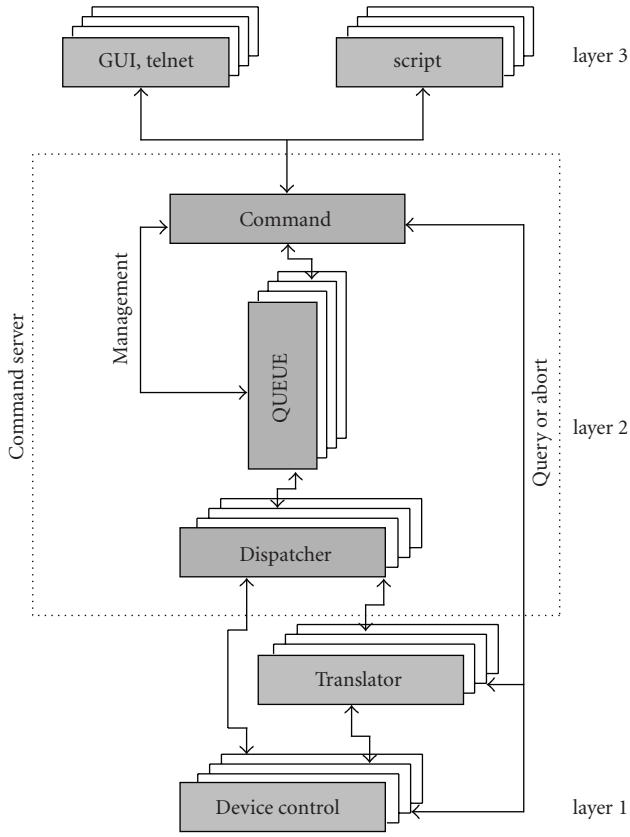

Figure 1: Three-layer-model of automation.

and may be requested by any client at any time. If one cannot exert influence on the kind of communication of the *device control*, for example, because it is a commercial product, a *translator* may be necessary. This will be the case, for example, if the device communicates via an industrial field bus or if a digital output bit is set instead of sending a completion report. Therefore, a *translator* is always a very individual software, sometimes supplemented by a special hardware, and thus not part of the *command server*.

Figure 1 shows our three layer automation model and illustrates the functioning of the *command server*.

The major properties of the *command server* are the following.

(i) Clear separation of the different levels of automation.

(ii) A central server coordinates and supervises all distributed activities of a scientific measurement procedure, with various instruments participating.

(iii) Uniform communication structure between command level (clients) and device control level.

(iv) Communication via TCP/IP.

(v) Standard mechanisms such as command priorities and queues, execution reports, premature aborts, and status commands do not need to be invented anew for each application.

(vi) Multiple clients may communicate with the same application at the same time.

(vii) Simple ASCII command structure which is accessible by each conceivable interface (consoles such as *telnet* or PuTTY (Windows), script, GUI).

(viii) A crash of one of the applications currently controlled will be noticed immediately.

(ix) A password protection distinguishes between normal users with restricted rights and system managers being allowed to administrate users and to change setup settings.

(x) Parameter check of a command if a parameter range was specified before.

(xi) Status broadcasts every few seconds (adjustable) keep all clients informed that are connected with the same application.

(xii) Instances (e.g., different observatories scattered over the world, which accomplish a joint observation program) share a uniform communication interface for all tasks of automation.

(xiii) Possibility of chatting with other active users.

(xiv) Written in Java and thus largely independent of the operating system (Windows, Linux/UNIX, ...).

The *cs* commands are not intended to be given directly in ASCII format by a human user, aside from testing purposes. Such commands typically are hidden behind a GUI or they are elements of a script.

4. Script Language

Whereas the *command server* is still being tested and improved, the development of a script language called AMI convenient to run full automatic measurement procedures and permanent robotic applications is already completed [1]. Of course it is possible to use other script languages like Perl in order to communicate with the *cs* but AMI is much easier to learn for technicians or astronomers who are less familiar with programming. Furthermore, it has some features of interprocess communication and functions for easy conversion of all data types, especially to and from strings, which make AMI appropriate for its main purpose.

AMI means Automation Macro Interpreter and is a "C"-like script language for the operating systems Windows and UNIX/Linux. The language contains statements, functions, and control structures which are explained in a help file. A variable must be declared with its data type (see below) before its first usage. The language supports commands to the shell in wait and no-wait modes with parameter passing. A result string can be returned via an unnamed pipe allocated by AMI, but pipes work only if the remote process runs on the same computer. The AMI-function *tcpclient()* is of much more importance, because it is able to send messages also to a distributed remote process via TCP/IP. The syntax for sending such commands is

tcpclient < *IP-address* >, < *port* >,
< *message string* >, < *return variable* >;

The *message string*, for example a command, will be sent to the remote process, for example, the telescope control. Afterwards *tcpclient* waits for the report and stores it within the string variable *return variable*. Thus AMI is able to keep contact with all distributed processes in an easy way and to coordinate them.

AMI needs no special installation. Only the executable program and the help file are required. One may download these files together with some short examples of AMI scripts (*.ami) from our web page [2].

The most important properties of AMI are the following

(i) An AMI script <macro.ami> will be started by input of *ami* < *macroname* >< *parameters* >.

(ii) Input of *ami -?* shows a short but complete description of AMI.

(iii) $[1-10] mark the external parameters which are passed to the AMI script from the shell.

(iv) It is possible to start an AMI-script via internet browser. The transfer of parameters via CGI-interface will be supported.

(v) Comments have to be initialised and terminated by the character '#'.

(vi) AMI-variables must be declared initially. Valid data types are *string*, *int*, *float*, *complex*, *date,* and *time*.

(vii) AMI provides functions to read and write (console and files).

(viii) There are many functions to change data types.

(ix) AMI has C-like loop- and-if-structures which may be nested.

(x) AMI has the arithmetic operators +, −, *, /, % and ^ (power).

(xi) AMI has the usual mathematical functions.

(xii) AMI has a UNIX-like test-function in order to test properties of files and strings.

(xiii) AMI is able to start programs via shell in wait and no-wait (parallel) modes with parameter passing.

(xiv) AMI supports pipes in order to return result-strings from a remote process.

(xv) AMI provides a client (-command) in order to send commands via TCP/IP to a remote process and to receive confirmation of operation (report) as a string.

(xvi) AMI returns a message in case of a syntax error.

(xvii) AMI versions exist for UNIX/Linux and Windows.

5. Primary Image Guider as an Example of a Device Control

The control system of the main solar telescope of the Istituto Ricerche Solari Locarno (IRSOL) [3] is over ten years old and was in need of renovation for some reasons. Whereas the sensor-system [4] has remained unchanged due to its continually satisfying performance [5], the revision was prompted by outdated computer hardware and by increased demands regarding communication with other hardware and software. Ten years ago the GUI of the old control system called "Primary Image Guider (PIG)" was the most important user interface [6], whereas today the remote access via TCP/IP from different clients—occasionally at the same time—becomes ever more important.

In its former version, the telescope control part and the GUI of PIG were integrated in the same LabVIEW program. The digital outputs to send signals to the telescope control hardware were part of an ISA bus plug-in card within the PIG PC. This outdated combination of components gave reason to establish a new version of PIG with more functions and complete separation of the different levels of automation.

The new hardware we used to perform our telescope control program is a Compact Real-Time PowerPC Controller (cRIO-9012) from National Instruments. Bundled with the I/O-Modules needed for this project, it is integrated in a small chassis of $18 \times 9 \times 9\,\text{cm}^3$. The new telescope control software now running on this module is also written in LabVIEW. All software for this module has to be developed on a Windows PC and accordingly downloaded via TCP/IP in conjunction with a small real-time operating system. Thus the postulation of using industry standards is satisfied.

The control software is split up into a number of functions which all run parallel in separate threads. They can be called by the ASCII-commands. We do not like to describe the various functions of a solar telescope control system here in detail and refer to [4, 7]. An actual listing of all present PIG commands is specified in Table 1. The syntax has been chosen in order to be compatible with the software used by the Zurich Imaging Polarimeter (ZIMPOL) system, which is largely used at IRSOL [8] and which is also intended to be used with the new GREGOR solar telescope at Tenerife.

Since the functions have been designed independently, subsequent programmers will find it easy to create new ones. Higher level functions such as "position on the solar disk in other coordinates than (x, y)" or "store object position in a table" are requirements of a client. A GUI for Windows and UNIX/Linux which provides such higher level functions has also been written in LabVIEW. Since the PIG control software has been designed as a multiserver, it may be accessed by multiple clients simultaneously. If a command prompts physical access to the telescope motors, these are temporarily locked for other clients. Simple applications may be executed without the assistance of the *command server*. In cooperation with other device controllers PIG may be integrated in fully automatic measurement procedures guided by a script, which may be written in AMI. Especially if the procedure is complex, possibly with other observatories involved via the internet, the employment of the *command server* is very

Table 1: Commands and functions of the telescope control system.

pigoff	Turn off guiding, release motor
pigxr?	Query for actual x position [1/10 arcsec]
pigyr?	Query for actual y position [1/10 arcsec]
pigx=%d	Select x position (set point) [1/10 arcsec]
pigy=%d	Select y position (set point) [1/10 arcsec]
pigactu	Let actual position be set position
pigenc=%d	Set guiding mode (0 = sensor, 1 = encoder guiding)
pigenc?	Query for guiding mode (0 = sensor, 1 = encoder guiding)
pigxs?	Query for selected x position
pigys?	Query for selected y position
piggo	Guide to selected position
piggf	Guide to selected position and follow solar rotation
pigrunxy	Go to selected position but do not guide
pigabort	Abort going to selected position (Stop guiding mode)
pigmco=$m$nn	Manual control commands [m = speed, nn = direction]
piggmode?	Query for motor status (possible answer 0–6) 0 = motor is free 1 = manual control active 2 = guiding without following solar rotation 3 = guiding and following solar rotation 4 = flatfield mode active 5 = go to sun active 6 = go to home position active
pigi?	Query for intensity
pigthimin=%d	Set threshold for minimum intensity
pigimin?	Query for minimum intensity
pigthdelta=%d	Set threshold for guiding accuracy [1/10 arcsec]
pigthdelta?	Query for threshold of guiding accuracy [1/10 arcsec]
pigloops=%d	Set number of averagings for position sensor (default is 500)
pigloops?	Query for number of averagings of position sensor
pigffmx=%d	X-length of flatfield area [1/10 arcsec]
pigffmy=%d	Y-length of flatfield area [1/10 arcsec]
pigffms=%d	Flatfield motor speed (1 = slow, 2 = medium)
pigffm?	Query for all flatfield parameters (e.g., 3000,4000,2)
pigstartffm	Start flat field mode
piggosun	Telescope go to sun
piggohome	Telescope go to home position
pigsb?	Query for status

The prefix "pig" may be omitted. Formats of replies (examples):
Command: pigoff
Reply: pigoff:done
Command: pigxr?
Reply: pigxr = 2342.

helpful. Figure 2 illustrates the integration of the telescope control system into the observatory environment.

6. Conclusions

In most cases, the ubiquitous Ethernet with TCP/IP protocol is sufficient for the networking of automated astronomical devices. On this level, real-time problems rarely arise. The actual device control (sometimes real-time) software should be designed as (multi)server(s). Accordingly, the user interfaces are clients. Between these two layers, we implemented an additional layer which relieves the lower layer and provides for additional security. Besides, this layer (*command server*) is capable of joining different institutions, such as observatories, to combined measuring programs. Safety precautions must be met, even though they are not as extensive as in commercial applications because the transmitted contents are not sensitive.

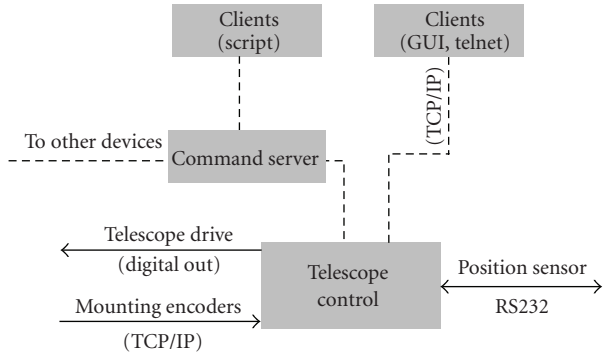

Figure 2: Integration of the telescope control system into the observatory environment.

Finally we would like to suggest that programs doing calculations of potential interest to many persons, such as conversion of coordinates or positions of objects, should be arranged as servers accessible via Internet sockets (a sort of astronomical "cloud computing") or via download. Unfortunately, many interesting web applications are currently only available in an interactive mode.

References

[1] G. Küveler and A. Zuber, "Eine Prozedursprache zur Steuerung vollautomatischer Prozesse," *Automatisierungstechnische Praxis*, vol. 46, no. 8, pp. 20–23, 2004.

[2] http://www.iai.fh-wiesbaden.de/ami/.

[3] M. Bianda, R. Ramelli, A. Feller, J. O. Stenflo, and G. Küveler, "Instrumental developments at the Gregory-Coudé telescope at IRSOL," in *Proceedings of the Workshop on Modern Solar Facilities—Advanced Solar Science*, F. Kneer, K. G. Puschmann, and A. D. Wittmann, Eds., Göttingen, Germany, 2007.

[4] G. Küveler, E. Wiehr, D. Thomas, et al., "Automatic guiding of the primary image of solar Gregory telescopes," *Solar Physics*, vol. 182, no. 1, pp. 247–255, 1998.

[5] G. Küveler, E. Wiehr, and M. Bianda, "Automatic guiding of solar Gregory telescope," in *From the Gregory-Coudé. Telescope to GREGOR: A Development from Past to Future*, F. Kneer, E. Wiehr, and A. D. Wittmann, Eds., vol. 324 of *Astronomische Nachrichten, no. 4*, p. 308, 2003.

[6] G. Küveler, E. Wiehr, and M. Bianda, "Eine sensorgestützte Computersteuerung für Sonnenteleskope," *Automatisierungstechnische Praxis*, vol. 42, no. 7, pp. 50–54, 2000.

[7] http://www.mnd-umwelttechnik.fh-wiesbaden.de/pig/.

[8] A. M. Gandorfer, H. P. Povel, P. Steiner, et al., "Solar polarimetry in the near UV with the Zurich Imaging Polarimeter ZIMPOL II," *Astronomy and Astrophysics*, vol. 422, no. 2, pp. 703–708, 2004.