ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOBPP06

20 YEARS OF WORLD CLASS TELESCOPE CONTROL SYSTEMS EVOLUTION

T. D. Gaggstatter*, I. Arriagada, P. E. Gigoux, R. Rojas, Gemini Observatory, La Serena, Chile
J. Molgo, GMTO Corporation, Pasadena, California, USA
F. Ramos, Grantecan S.A., Breña Baja, La Palma, Spain

Abstract

This paper analyzes the evolution of control systems for astronomical telescopes. We look through the lens of three world class telescopes: Gemini, GTC and GMT. The first two are in operation for twenty and ten years respectively, whilst the latter is currently under construction. These facilities have a planned lifetime of 50+ years, therefore obsolescence management is a key issue to deal with. For the telescopes currently under operation, their real-time distributed control systems were engineered using state-of-the-art software and hardware available at the time of their design and construction. GMT and newer telescopes are no different in this regard, but are aiming to capitalize on the experiences of the previous generations so they can be better prepared to support their operations. We will compare and contrast software and hardware infrastructure choices including operating systems, middleware and user interfaces with a particular focus on obsolescence management.

INTRODUCTION

Every facility in the world, be it an industrial manufacturing plant or scientific installation, relies fundamentally on a control system to maintain optimal levels of performance. In addition to maintaining the control system as it evolves over time, support engineers must also remain informed of new technology as it becomes available to make careful adoption decisions balancing performance and stability. Telescopes are no different in this matter, and thus we present two decades of telescope control system evolution with examples from three telescopes at different stages in their life-cycle.

A telescope environment can be divided into three main control systems: the Telescope Control System, the Enclosure Control System and Support Systems. The Telescope Control System manages the main optics and its support structure. The Enclosure Control System controls the dome, bearing systems and safety infrastructure. Finally, the Support Systems manage climate control, wave front sensors and remote observations infrastructure.

The analysis is limited to three observatories which we consider a fair representation of the evolution of telescope control systems. We know we are not covering the full *observatory universe* and thus we try to compensate for this fact by presenting our conclusions in a technology-agnostic way.

In the next sections we will provide a short description of each observatory to provide a proper context. Later the most meaningful comparison points will be discussed with a closing paragraph in each section summarizing the lessons learned on that topic.

* tgaggstatter@gemini.edu

CONTROL SYSTEMS OVERVIEW

The next three sections show an introduction to each of the observatories under analysis.

Gemini Control System

The Gemini Observatory consists of twin 8.1-meter diameter optical/infrared telescopes located on two sites, Maunakea, Hawai'i, and Cerro Pachón, Chile. Having an installation on both hemispheres allows the observatory to cover the whole night sky, and the longitude separation between telescopes allows for longer tracking of events occurring in the shared zones of the sky. Gemini began its operations in Hawai'i in 1999 and in Chile in 2000. It operates mostly in queue observing mode [1] and started operating remotely from its base facilities in 2015 [2].

The Gemini telescopes are amongst the largest single mirror telescopes in the world. They were designed to be multi instrument telescopes, with a center located Cassegrain unit where the instruments are installed. [3]

Infrastructure and geographical location alone cannot guarantee performance, which is why the control system is a key part of the Gemini Observatory to ensure that the best data will be acquired and provided to scientists. With this objective in mind, Gemini uses the Experimental Physics and Industrial Control System framework (EPICS) [4]. This system was chosen because of its widespread adoption at large facilities such as particle accelerators, strong open source community support, and its high adaptability and ease of customization. It was adopted as the standard control framework in which to run the real-time telescope subsystems.

EPICS uses Client/Server and Publish/Subscribe techniques for the communication between the control computers (also called Input/Output Controllers, or IOCs). IOCs talk to each other, and other types of clients, using the EPICS Channel Access network protocol [5]. This protocol is also used by the high level software that sequences science observations. Channel Access is used at Gemini for soft real-time networking applications. Dedicated communication protocols and channels are used in places where faster communication rates are required.

The heart of an EPICS application is the database, a collection of function-block objects called records. Most EPICS records have a predefined functionality; others can be customized linking C code to them. Gemini developed a set of custom records to support the Action Command Model. In this model, actions are driven from changes to

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOBPP06

attributes to the database records. The modified set of attributes defines the new configuration of a system which is applied in a single step [6-8].

GTC Control System

The Gran Telescopio Canarias (GTC) is currently the largest and one of the most advanced optical/infrared telescopes in the world, it is located at the Roque de Los Muchachos on the Canary Islands in Spain. Its primary mirror consists of 36 individual hexagonal segments that together act as a single mirror. The surface area of the primary mirror at GTC is equivalent to that of a telescope with a 10.4m diameter single monolithic mirror. Its First Light Ceremony was celebrated in the early morning of 14th July, 2007. Scientific operations of the telescope started in March of 2009 [9.]

The Grantecan Control System (GCS) was created to manage the telescope subsystems. The GCS was built and designed with the following requirements: ease to extend or adapt new changes and specifications, ease of use, reuse the code using design patterns, portability, robustness, efficiency, etc. The system is based on several services called "common services" and distributed components called "Devices" [10].

The common services are responsible for managing the telemetry data, logs, alarms and configurations of all Devices. They are composed of the following managers: MonitorManager [11], LogManager, AlarmManager and ConfigManager. The Devices can be an abstraction that provide a logical representation of the physical elements controlled by the system. They can also carry out different tasks such as coordination between Devices, analysis of data quality, etc.

The Device interface is defined by commands, monitors, properties and alarms. The commands are methods associated with an action, whereas a monitor represents a physical or logical value, such as a temperature sensor value. Conversely, a "Property" is a read-write attribute which is used to configure the Device at startup and it can be changed at run-time, such as a conversion factor of a calculation. The Device behaviour is modeled using a Finite State Machine (FSM) where each state defines a behavior.

All GCS Devices are integrated with the common services thanks to LogAgent, AlarmAgent, MonitorAgent and ConfigAgent which send logs, alarms, monitors and configured values to each manager respectively.

The GCS uses a real-time implementation based on the Common Object Request Broker Architecture (CORBA) which is responsible for providing connections and interoperability among distributed objects or Devices in the system. The main benefit of using CORBA is that a client can transparently invoke a method on a server object, which can be on the same machine or across networks. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. To achieve this location-transparent access, it is necessary to use a set of distributed services which are required to guarantee a level of service (QoS) such as NamingService, InterfaceRepository, etc.

GMT Control System

The Giant Magellan Telescope (GMT) will be one member of the next generation of giant ground-based astronomical telescopes. It is a segmented telescope, with seven segments of 8.5 meters in diameter each, forming a single optical surface equivalent to an aperture of 24.5 m. diameter. The chosen site for the GMT is Las Campanas peak, in Chile [12].

The GMT Observatory Control System (OCS) follows both a component-based and a model-based approach. The basic building block of the GMT software architecture is the Component, which is the root of the class hierarchy tree (Controller, Pipeline, Server, etc.). The Component interface is defined by a set of "Component Features", which can be classified into Inputs, Outputs, State Variables, Properties, Faults and Alarms. The State Variables define the observed and desired state of the system, by providing an output for the current value of the magnitude that they represent and also an input for the desired value (goal) for this magnitude. The GMT Components are reactive, i.e., their behavior is not triggered externally by commands, but they react to the differences between the value and the goal of their State Variables.

The information flow between Components is achieved by means of a set of Connectors, which connect a feature of one Component to a reciprocal one in another Component (see Fig. 1). For example, an Output of a higher level Controller can be connected to an Input, or an Alarm or Fault of a low level Controller can be connected to another Component higher in the hierarchy, so errors are easily propagated upstream. The chosen middleware for inter-Component data sharing is *nanomsg* [13], which provides robust and sophisticated communication patterns that are abstracted out in the Frameworks to limit the coupling of the OCS to any external library. The preferred protocol for the communication to the field is *Ethercat* [14].

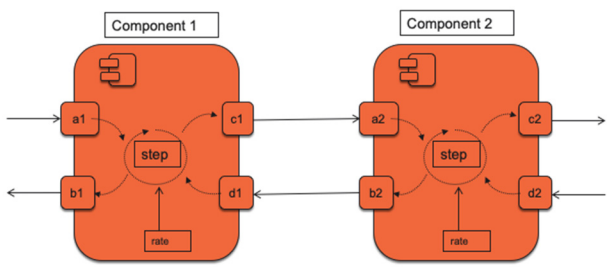

Figure 1: GMT Component model.

The set of all the Components of a module, their "Component Features" and the "Component Connector Map" (either internal to the module or external) are formalized in the OCS model. These are used for code generation during the development process and also in run-time as a mecha-

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOBPP06

nism for the user interface (and the whole system in general) to dynamically discover the location and interfaces offered by the different Components present in the system. See [15] for more information.

COMPARATIVE ANALYSIS

The following subsections present a comparative analysis between several key aspects of the three Telescope Control Systems and a discussion about their evolutionary trends.

Framework

The EPICS framework used by Gemini is based on a collection of "Records" of different types grouped into "databases". In this context, a Record is an object with a unique name, with properties (fields), a single C/C++ routine and optionally associated hardware input/output drivers. Each record field is mapped to a channel by the communication layer and all its values are published to the network for read/write access, making each stage easier to control, simulate and monitor. Databases are created using a graphical editor that allows dragging and dropping graphical representations of the records. The editor provides the developer with a graph of the data and action flow (see Fig. 2 below).

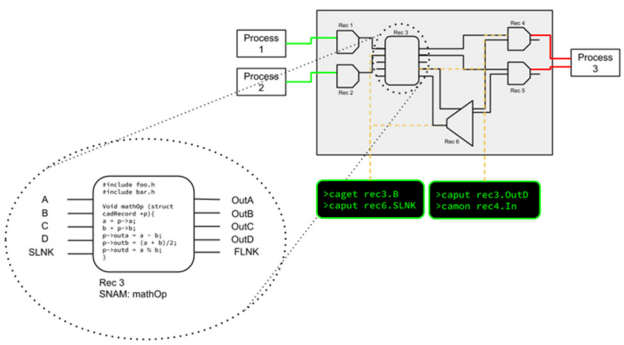

Figure 2: Example of the interaction between process stages of an EPICS IOC. The black blocks represent terminals monitoring inner variables of the specific process.

GMT also uses a channel oriented architecture but with less granularity: the building block is the component and not the routine. Channel oriented architectures are very flexible, as every channel can be connected to any other one with a minimum impact to the code. This provides a larger flexibility on the deployment and unit testing, but it can also lead to a complicated implementation and many errors being detected at runtime rather than at compilation time. In addition, the channel-oriented architecture of the GMT frameworks provide an added layer of abstraction between the software controllers and the hardware. This allows it to separate communication problems from the implementation of the control logic, and prevents a tight coupling between the source code and the chosen hardware solution.

GTC uses an object oriented architecture with a strong coupling between components. This makes it less prone to errors and provides a stronger cohesion, which facilitates a cleaner implementation. This coupling can be minimized using generic interfaces. In Fig. 3 we can see a conceptual representation of the coupling between components for each observatory.

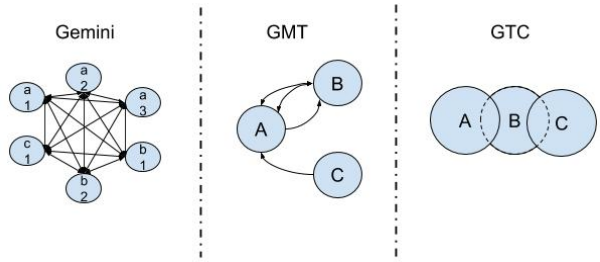

Figure 3: Conceptual representation of coupling between software components.

GMT and GTC use template files for their code construction. When an additional attribute needs to be added, the template needs to be reparsed. This is useful to keep a clean interface for each subsystem. On GMT the templates that generate the initial source code skeleton stem directly from the requirements models. These are the same models used to calculate budgets in project, which means it will have a strong consistency throughout the whole development process.

To sum up, it is very useful to keep one interface model that needs to be synchronized to the source code. Moreover, GMT defines the connections between the different subsystems, which gives a clear overview of the overall system architecture. Regarding modularity, it is a powerful feature that needs to be handled with care to avoid worse performance.

Middleware

Gemini uses EPICS Channel Access, a Middleware supported by the community. The main advantage is you benefit from bug fixes and new releases that are made available. The community is very active, comprising over 30 Astronomy and Physics Particle institutions worldwide. There is a very active email list where people exchange information and can ask for help in case of problems.

GTC is using CORBA, a distributed object middleware standardized by the Object Management Group (OMG) group. However, GTC developed an abstraction layer embedded in its framework to isolate the CORBA-specific code from the domain code. This had two major goals: protect the developer against the complexities of CORBA development and eventually facilitate the migration to another middleware if needed. This was partially achieved, as there are some parts of the code where the CORBA system is exposed, but the problem is minimized as most of these parts are auto-generated.

The main benefits of using CORBA were:

- Transparency in the object location: using the CORBA naming service it is possible to decouple completely the code from the remote objects location. An application can be changed to another physical machine, and the remote clients will dynamically detect it. This adds a lot of flexibility in the system deployment.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOBPP06

- Interoperability: CORBA has Application Programming Interfaces (APIs) for several programming languages and there are Object Request Broker implementations available in multiple Operating Systems.
- Object-oriented semantics: as CORBA provides a method invocation on objects (that can be either local or remote) with the same syntax as the programming language, then the integration with an object-oriented design is straight forward.

Some of the drawbacks that we have identified with the use of CORBA include interconnection issues, debugging challenges and obsolescence. The interconnection problems that arise when a CORBA system is deployed in a network that has firewalls. There is an increased cost in the process of debugging a failure (either when some part of the CORBA system fails, or when the configuration of the system is incorrect), because the broker part of the system is typically a black box. Finally, when the GTC telescope was designed, CORBA was considered the cutting edge of middleware, but emerging technologies contributed to its downfall.

GMT is using Nanomsg, which is a socket library that provides several communication patterns, like pair (one-to-one), pub/sub (one-to-many), pipeline, etc.. It works in a wide range of operating systems and there are implementations available in multiple programming languages. Nanomsg can be thought of as a "smart sockets" implementation that handles all the typical complexity about handling connections: message fragmentation, connection failures and multi-element communications. In contrast to EPICS and CORBA, Nanomsg only provides communication capabilities, but it does not provide any higher level abstraction like remote method invocation or object access. These higher level features are provided by the GMT frameworks which use Nanomsg as the underlying messaging library. The coupling to Nanomsg is contained within the frameworks and the domain code is not exposed to it.

The main benefits of Nanomsg are:

- Powerful communication patterns for data exchange, that are provided off the shelf
- Enhanced system robustness, as most of the connection complexities and their related failures are gracefully handled
- Very high message throughput
- Very good interoperability

The fact that Nanomsg is only a light-weight messaging library is a two-edged sword: it has a very good performance and it allows a lot of flexibility, but at the same time it requires a high effort for the framework developers to wrap it with a component model that provides all the needed functionality. In addition, there are some concerns about the projection of Nanomsg to the future, because it is a relatively new technology and its community is not very wide.

The main difference in this aspect is that Gemini sticks to a specific middleware, while the other two have written an intermediate abstraction layer that had a higher up-front cost but gives the flexibility of switching to other middleware having only to rewrite the adapter/interface layer. Not being attached to a specific technology in long term projects might be an advantage at the end.

Programming Languages

The adopted programming languages for real-time execution are very homogeneous in the three observatories. Gemini started using C, which was the most common option for systems programming, and then they gradually adopted C++ for real-time programming. C++ has proven to also be a valid language for real-time and systems programming, while offering a higher level of abstraction, more readability and a large set of available libraries. GTC and GMT, on their part, have standardized the use of C++ as the real-time language since the beginning of their projects.

For scripting, data analysis and image processing pipelines the three observatories started with different solutions (IRAF, IDL, raw C++, shell scripts) but now they have adopted the same programming language, Python. This language features a very readable syntax, with a relatively shallow learning curve. In the latest years, the engineering and scientific communities have shown a tendency towards this language which, in turn, has led to an increased availability of data processing and scientific-oriented libraries.

On the other side, there is more diversity in the choice for the user interfaces and observatory services. Gemini uses DM, EDM (both EPICS-centric) and TCL/TK. GTC initially adopted Java for the user interface, along with the Swing library for the graphical part, and some parts of the jSky system for the astronomical data display. In addition, GTC is also currently transitioning to Java for the observatory services (telemetry, etc.), as in the latest decades the enterprise software community has been providing very good solutions for data persistence and clustered services execution. Finally, the standard language in GMT for the user interface and services is Coffeescript, a language that can be transpiled into Javascript and then executed in a Nodejs environment. This allows GMT to join the current trends of the industry (specifically, the web-based solutions industry) for front-end and back-end development.

In summary, real-time environment keeps its path on C/C++ there has been no major breakthrough on this kind of languages in the last decades. For data analysis Python has taken over, especially in the last decade and is now mainstream. While on GUIs there are constant changes and several choices, the choice usually lies on the mainstream of that era, currently this is a web-based environment, all three observatories are slowly tending to it.

Operating Systems

Initially on Gemini and GTC the real-time operating system (RTOS) as well as the general purpose operating system (OS) were the same, VxWorks [16] for the RTOS and Solaris Sparc for general purpose OS. Due to obsolescence both were forced to search for alternatives when upgrading their OSs. In the past few years Gemini was engaged in a

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOBPP06

major effort to upgrade its RTOS to RTEMS [17]. Meanwhile, GTC is working on a new standard for its real-time platform, building its first prototype under Linux Preempt-RT patch.

The Gemini Real Time Upgrade [18] included an update to a newer version of EPICS that implements an Operating System Independent (OSI) layer, which means that the source code can run on different operating systems without modifications.

Conversely, GMT has opted for Linux Preempt-RT, profiting from the usage of the Linux standard tools and libraries. This has freed development from the need for specific real-time APIs such as VxWorks and RTEMS. Also, Linux has a strongly supported global community that keeps the OS updated with new technologies and features. One of its main features is its top of the line debugging facilities.

The key takeaway is that there are usually very few OS dependant calls in a Control System, it is worth handling these with an abstraction layer which makes porting to other OS seamless. All three projects have adopted this approach.

Hardware

Gemini has a strong coupling to its hardware peripherals, while GMT is implementing a loose coupling as a proactive obsolescence strategy. Currently, GTC hardware coupling is closer to that of Gemini, but it is adopting a more flexible/obsolescence-aware design in its latest systems. In this example (Fig. 4) we see a representation of the standard control systems. On the older telescopes the hardware interface code is very entangled with the Controller source code, while on GMT and GTC (2019) the controller code interacts with a single fieldbus interface. This achieves an efficient decoupling between the Controller and its hardware components.

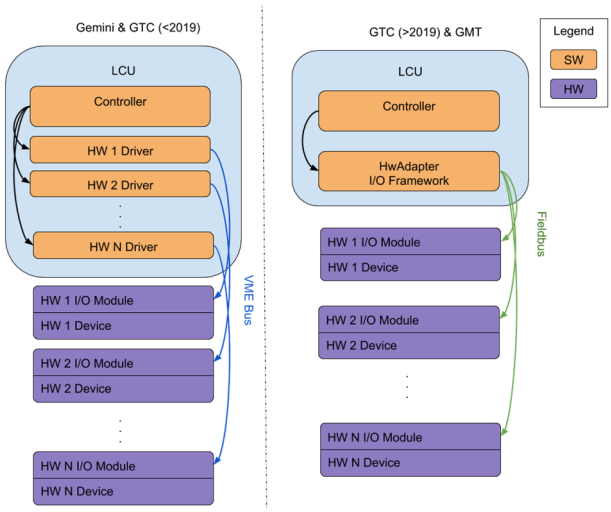

Figure 4: Hardware component coupling seen on a standard subsystem.

Regarding the core hardware platform, we are observing a shift from VME based systems to more general purpose platforms, e.g. PCs running LinuxRT. This is driven on the software side by the huge utilities/support that environments like Linux provide and on the hardware side by vendors continually developing interfaces and support for standard PCs.

Telescopes are expected to last 50+ years in that timespan the hardware will change dramatically. No vendor guarantees its products for a decade. It could happen that some part of the hardware stops being produced which means the hardware might not even be subject to repair. Worst case scenario, vendors can go out of business with no direct alternatives available.

Summarizing, it is good practice to modularize components at hardware level, in a way that allows for the domain code to remain as stable as possible. As an example, a fieldbus gives you much more flexibility than a hardware concentrated system (e.g. VME, cPCI). In the case of the VME bus you are forced into hardware that fits your physical system, this means you need to support all the drivers for the different hardware while when relying on a fieldbus you just need to support its communication protocol and standards.

Philosophically, the goal is to move most of the interfaces between the control system and the hardware as close to the final control devices and as far away from the CPU as possible.

Engineering Databases

At the beginning, databases used to store telemetry data did not play an important role in a telescope control environment. As an example of this, both Gemini and GTC started operations without process values databases. The Telescope telemetry was stored in binary files and this was used to diagnose errors after they occurred. With the passing of the years, the evolution of Systems Engineering and the emergence of Big Data, engineers have realized that there is a very big potential in all the telemetry data gathered by the telescopes. This collected data would allow us to predict possible faults. In that sense GTC has constructed models based on that data which have helped to prevent errors and improve the system. GMT is actively planning, since its design stage, to make a profit from that data.

In this aspect the underlying technology has not played a key role, this is shown as GTC is using a relational database (MySQL) and GMT is using a non-relational database (MongoDB). Even using different technologies, the most important thing is the data analysis and the concept of using this valuable data.

Graphical User Interfaces

The graphical user interface, GUI, is one of the components of a control environment that has seen considerable transformations over the years. Initially a GUI displayed very basic information since it was meant as a peripheral tool for the user. A simple command would require the user to manually calculate values, manage several windows to control the individual components and often interact with the electro-mechanical hardware directly.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOBPP06

In the last twenty years new technologies as Java Swing and later Web GUI toolkits have emerged. These kits allowed for the development of more sophisticated GUIs, which were able to provide a more realistic representation of the real system. Thanks to technology advancement and good engineering criteria, GUIs have become more user friendly. This has allowed the user to interact with the system in an easier and more intuitive way, improving overall performance. Smarter implementation has led to higher level actions. In these cases, the users configure the system for a 'goal' they want and the system knows the 'steps' to configure every single component according to user needs [19].

On Fig. 5 we show one of the many screens that are used to control each of the telescopes, it is one of the key screens that shows a general overview of the telescope status.

Figure 5: GUIs from different telescopes.

In this context, it is possible to see a clear evolution in the technologies, Gemini uses EDM and Tcl/TK, GTC uses Java and GMT uses ElectronJS to develop their interfaces. Currently, there is a trend to use web frameworks to develop GUIs and GMT has opted for it, benefiting from a strong community and tools to make agile development.

Clearly GUIs have gained importance during the years and are currently a key element to efficiently run any telescope. For this reason, GMT included GUI development in their initial design stage. It is worthwhile to mention that in the early days, understanding the system through the numbers shown on the screen was very challenging. Nowadays it is the other way round, it is easier to understand a system from its GUI than from the real physical system.

CONCLUSIONS

The most significant conclusion that we draw from our analysis is the importance of obsolescence management as a challenge for astronomical telescope control systems. Normally, the lifetime of such facilities goes well over 50 years. In this timespan, the available hardware will evolve several leaps, with each leap driving the control software. Other software, e.g. operating systems or external libraries, can evolve much more rapidly, with several generations of a product in a few decades.

Earlier telescopes such as Gemini, did not consider obsolescence, and have had to spend considerable effort in the operations phase to catch up with change. In contrast, as designers have become more aware of this fact, more telescope control systems are including obsolescence control in their project design requirements.

As a result, the newer generations have incorporated the obsolescence control deeply in the system architecture. This is achieved mainly by in-house developed frameworks that allow for a separation between the domain code and the underlying technologies. These frameworks minimize strong coupling between the system elements and the hardware, by adding an additional software abstraction layer.

When it comes to programming languages and operating systems, the differences between the three observatories have a different source. While the obsolescence control can be considered as a "lesson learned" evolution, the changes in the programming languages in use at the different institutions reflect which were the mainstream adopted solutions when each system was designed. For real-time the three observatories have a similar solution and evolution (C and C++ for language, VxWorks first and then RTEMS or Linux-RT for OS), as it has also been the standard used by the industry in the latest decades. For UI programming the choices are more diverse, but it is clear that each observatory adopted the most common option at their time: TCL/TK and Epics-centric solutions for Gemini, Java for GTC and Nodejs for GMT.

The key takeaways from each section could be summarized as the following:

- Framework: Channel oriented architectures are very flexible, although this opens many possibilities which can lead to an entangled implementation, thus it is advised to have an interface source file that defines its external interface and a connection map.
- Middleware: An intermediate abstraction layer is a useful safeguard in the event of a chosen technology getting obsolete.
- Programming languages: RT programming languages are C and C++, these show slow to no evolution, sticking with industry standards. On general purpose languages many functions programmed in a variety of languages are merging into Python. On the GUI development we can see a wilder language panorama with a diverse and quickly evolving landscape.
- Operating Systems: The small set of OS specific calls should be handled under an abstraction layer to enable a seamless transition to a different OS.
- Hardware: In this case a well placed abstraction layer will play a key role to ease hardware replacement, having a hardware abstraction layer like a fieldbus seems to be a good strategy to further leverage loose coupling to a specific piece of hardware
- Databases: Collecting and storing data cannot only be used for postmortem analysis, but to predict system degradation and faults, and schedule preventive maintenance work to avoid these.
- GUIs: A well designed GUI will enhance the operational efficiency as well as the user experience, nowadays technology enables us to develop an excellent human machine interface.

An important observation from this work is how the control systems software landscape has changed since 20 years ago. Back in the early 2000s, a lot of the well established control environments/frameworks were not flexible enough to accommodate for the demands of Observatories. This meant that an "open source" system like EPICS, was

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOBPP06

the better alternative to handle the low level interaction requirements. Therefore, if a required driver was not readily available, the solution could be developed in house, without the time and overhead costs of going through a vendor.

Fast forward to today, and the increase in the number of telescopes worldwide has meant that established vendors see observatories as potential clients. Current automation and control devices comply perfectly with the performance and flexibility requirements for telescope control. These same vendors are tapping into the human know-how of the past two decades. As a result, control devices, hardware, firmware and software are being commercially developed with built-in support suited to Observatories standards.

A trend that can be observed in the newer projects is that the low-level motion control is outsourced to external vendors, which also usually provide the hardware itself. However, most of the higher level software is still developed in-house. This is due to two reasons: requirements that evolve continuously, and the need for dynamic upgrades and efficient maintenance.

REFERENCES

[1] B. W. Miller and R. Norris, "Gemini queue planning, in *Proc. SPIE, Observatory Operations: Strategies, Processes, and Systems II*, vol. 7016, pp. 260-271, Jul. 2008. doi:10.1117/12.790169

[2] A. Serio *et al.*, "Gemini Observatory base facility operations: systems engineering process and lessons learned", in *Proc. SPIE 9911, Modeling, Systems Engineering, and Project Management for Astronomy VI*, vol. 9911, pp. 338-355, Aug. 2016. doi:10.1117/12.2231831: 338 -- 355

[3] Gemini website, http://www.gemini.edu

[4] Experimental Physics and Industrial Control System, http://www.aps.anl.gov/epics/index.php/

[5] EPICS Channel Access Protocol, https://epics.anl.gov/docs/CAproto.html

[6] P. M. McGehee; S. Wampler; and K. K. Gillies, "Command completion within an EPICS database", in *Proc. SPIE Proceedings, Telescope Control Systems*; Jun. 1995, vol. 2479, pp. 193-203. doi:10.1117/12.211463

[7] K.K. Gillies, "lCD la - The System Command Interface", Controls Group, Gemini 8-rn Telescopes Project, Mar. 1995.

[8] K.K. Gillies *et al.*, "lCD lb - The Baseline Attribute/Value Interface", Controls Group, Gemini 8-m Telescopes Project, Feb. 1995.

[9] GTC, http://www.gtc.iac.es

[10] J. M. Filgueira, M. Pi i Puig, P. Gomez-Cambronero, M. Gonzalez, and R. Penataro. "Architectural design of the GTC control system", in *Proc. SPIE Advanced Telescope and Instrumentation Control Software,* Munich, Germany, Mar. 2000, vol. 4009, pp. 35-45. doi:10.1117/12.388408

[11] M. Huertas, J. Molgo, R. Macías, and F. Ramos. "Monitoring service for the Gran Telescopio Canarias control system", in *Proc. SPIE Software and Cyberinfrastructure for Astronomy IV* , Edinburgh, UK, Jun. 2016, vol. 9913, pp. 1319-1327. doi:10.1117/12.2231442

[12] GMT website, https://www.gmto.org

[13] Nanomsg socket library, http://nanomsg.org

[14] EtherCAT fieldbus, https://www.ethercat.org/

[15] M. Pi *et al.*, "Status of the Observatory Control System for the GMT", in *Proc. SPIE Software and Cyberinfrastructure for Astronomy V,* Jul. 2018, vol. 10707, pp. 18-26. doi:10.1117/12.2315850

[16] Wind River VxWorks RTOS, http://www.windriver.com/products/vxworks/

[17] RTEMS RTOS, https://www.rtems.org

[18] A. J. Nunez *et al.*, "Experience upgrading control systems at the Gemini Telescopes", in *Proc. 16th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'17)*, Barcelona, Spain, Oct. 2017, pp. 99-106. doi:10.18429/JACoW-ICALEPCS2017-MODPL03

[19] J. Molgo *et al.*, "Automatization of the guiding process in the GTC", in *Proc. SPIE Software and Cyberinfrastructure for Astronomy IV,* Edinburgh, UK, Jul. 2016, vol. 9913, pp. 35-46. doi:10.1117/12.2231278