Status report of the SRT radiotelescope control software: the DISCOS project

Orlati, A., Bartolini, M., Buttu, M., Fara, A., Migoni, C., et al.

A. Orlati, M. Bartolini, M. Buttu, A. Fara, C. Migoni, S. Poppi, S. Righini, "Status report of the SRT radiotelescope control software: the DISCOS project," Proc. SPIE 9913, Software and Cyberinfrastructure for Astronomy IV, 991310 (8 August 2016); doi: 10.1117/12.2232581

SPIE.

Event: SPIE Astronomical Telescopes + Instrumentation, 2016, Edinburgh, United Kingdom

Status report of the SRT radiotelescope control software: the DISCOS project

Orlati A.[a,*], Bartolini M.[a], Buttu M.[b], Fara A.[b], Migoni C.[b], Poppi S.[b], and Righini S.[a]

[a]INAF - Istituto di Radio Astronomia, Bologna, Italy
[b]INAF - Osservatorio di Cagliari, Cagliari, Italy

ABSTRACT

The Sardinia Radio Telescope (SRT) is a 64-m fully-steerable radio telescope. It is provided with an active surface to correct for gravitational deformations, allowing observations from 300 MHz to 100 GHz. At present, three receivers are available: a coaxial LP-band receiver (305-410 MHz and 1.5-1.8 GHz), a C-band receiver (5.7-7.7 GHz) and a 7-feed K-band receiver (18-26.5 GHz). Several back-ends are also available in order to perform the different data acquisition and analysis procedures requested by scientific projects. The design and development of the SRT control software started in 2004, and now belongs to a wider project called DISCOS (Development of the Italian Single-dish COntrol System), which provides a common infrastructure to the three Italian radio telescopes (Medicina, Noto and SRT dishes). DISCOS is based on the Alma Common Software (ACS) framework, and currently consists of more than 500k lines of code. It is organized in a common core and three specific product lines, one for each telescope. Recent developments, carried out after the conclusion of the technical commissioning of the instrument (October 2013), consisted in the addition of several new features in many parts of the observing pipeline, spanning from the motion control to the digital back-ends for data acquisition and data formatting; we briefly describe such improvements. More importantly, in the last two years we have supported the astronomical validation of the SRT radio telescope, leading to the opening of the first public call for proposals in late 2015. During this period, while assisting both the engineering and the scientific staff, we massively employed the control software and were able to test all of its features: in this process we received our first feedback from the users and we could verify how the system performed in a real-life scenario, drawing the first conclusions about the overall system stability and performance. We examine how the system behaves in terms of network load and system load, how it reacts to failures and errors, and what components and services seem to be the most critical parts of our architecture, showing how the ACS framework impacts on these aspects. Moreover, the exposure to public utilization has highlighted the major flaws in our development and software management process, which had to be tuned and improved in order to achieve faster release cycles in response to user feedback, and safer deploy operations. In this regard we show how the introduction of testing practices, along with continuous integration, helped us to meet higher quality standards. Having identified the most critical aspects of our software, we conclude showing our intentions for the future development of DISCOS, both in terms of software features and software infrastructures.

Keywords: SRT, ACS, Control Software, Astronomical instrumentation, Radiotelescope

1. INTRODUCTION

The Sardinia Radio Telescope (SRT) is a new Italian facility, located 35 km North of Cagliari, on the island of Sardinia. The SRT is a general-purpose radio telescope aimed at operating with high aperture efficiency in the frequency range from 300 MHz to 100 GHz. After its inauguration (October 2013) SRT completed the set of antennas devoted to radio astronomical science in Italy. The other two telescopes are 32-m dishes located in Medicina[1] and Noto.[2] Since the early stages of the SRT project, staff from all the three telescopes has been involved in the development of its control software. This forced the team to cope with both the development of a brand-new system and the maintenance of the already existing telescopes, compelling the involved personnel to learn and enhance their competence in distinct and heterogeneous systems, based on completely different technologies. The natural approach to this issue was to focus on a single system to be developed and installed

Further author information: Andrea Orlati, E-mail: a.orlati@ira.inaf.it, Telephone: +39 051 6965831

Software and Cyberinfrastructure for Astronomy IV, edited by Gianluca Chiozzi, Juan C. Guzman, Proc. of SPIE Vol. 9913, 991310 · © 2016 SPIE
CCC code: 0277-786X/16/$18 · doi: 10.1117/12.2232581

at all the telescopes. In recent years this idea was formalized with the creation of the *Development of the Italian Single-dish COntrol System* (DISCOS) project. It aims at developing, optimizing and maintaining the software required to fully exploit the capabilities of the Italian facilities. In this paper we include a description of the Sardinian telescope, the project from which DISCOS originated. Chapter 2 focuses on SRT capabilities and provides a snapshot of the present status. A description of the control software and its architecture is reported in chapter 3, some recent developments and improvements are also described. The extensive usage along many years of activity, including astronomical validation and early science observations, allows us to assess the achieved goals and to discuss the open issues. These considerations, drawn from our experience, are also discussed. Finally in chapter 4 we describe the strategies for improved testing and maintenance practices we put in place in order to overcome some of these issues and to improve the quality of our control software.

Figure 1. View of the Sardinia Radio Telescope

2. THE SARDINIA RADIO TELESCOPE

The SRT design is based on a wheel-and-track configuration. The antenna can be steered around the Azimuth and Elevation axes, managed by a servo system composed by absolute encoders and 12 brushless asynchronous motors. The 64-m main mirror is a shaped parabola-ellipse surface. It consists of a back-structure that supports 1008 aluminum panels divided into 14 rows. The panels are tightened to the structure through 1116 electromechanical actuators that allow to correct the systematic deformation of the mirror. A quadrupod connected to the back-structure sustains a 7.9 m subreflector, while a set of 6 actuators defines its motion with 5 degrees of freedom in the primary focus. Two servo systems and a mechanical arm permit to place and remove the receiver boxes in this focal position. A turret mounted on the focal plane is rotated by a proper servo system allowing to position up to 8 receivers in Gregorian focus. Further mirrors beneath the Gregorian focus (Beam Wave Guide) make 4 more focal positions available, to be exploited for an intermediate frequency range. In this case the desired BWG layout is selected by proper rotations of a portion of an ellipsoid mirror. The remote control of the described servo systems makes the telescope operate in frequency agility, permitting to switch between receivers in seconds or minutes, according to the operations involved. At present the telescope has one microwave receiver installed

in each of the three focal positions. Table 1 lists the observable sky frequency bandwidth, the focal position and the number of available pixels (beams) for these receivers. The K-band receiver is the only multi-beam apparatus; it is equipped with a built-in mechanical rotator that allows to compensate for field rotation while tracking or scanning. The output lines of the receivers are connected to the back-ends in the control room via RF over fiber technology. Table 2 lists the main characteristics of the presently available back-ends and their main employment within specific or general-purpose scientific use cases.

Table 1. Installed receivers at the SRT

Receiver	Freq. Range [GHz]	Focal Position	Beams
L- and P- band coaxial feed	0.305-0.410 1.300-1.800	Primary	1
C-band single feed	5.7-7.7	BWG	1
K-band multi-feed	18-26.5	Gregorian	7

Table 2. Available back-ends at the SRT

Back-end	Employment	Inputs	Bandwidth (MHz)	Spectral channels	Integration
Total Power	Continuum	14	300 - 2000	-	1-1000 ms
XARCOS	Spectro polarimetry	14	0.5 - 62.5	2048	10 s
DBBC	VLBI, RFI monitoring	4	0.512	4096	10-1000 ms
DFB	Pulsar, Spectro polarimetry	4	0 - 1024	8192	0.1 ms - 4 s
SARDARA	Pulsar, spectro polarimetry	2	300, 1500	1024, 16384	5 ms - 1s

Some salient facts regarding the telescope performance, as measured with the commissioned receivers, are reported in Table 3. The parameters are the telescope beam size, the system temperature at zenith position and the peak gain; these refer to a fully operational telescope, with the active surface and all the corrections enabled. A full description of the telescope performance, including the measurements obtained during the technical commissioning phase, is reported in Bolli et al. 2015 [3].

Table 3. Summary of the SRT performance as measured during the telescope commissioning

Rec.	Center Freq. [GHz]	Beam size [arcmin]	Tsys, El=90 [K]	Peak Gain [K/Jy] (±5%)
L	1.705	11.12 ± 0.11	34 (LCP)	0.52 (LCP)
C	7.35	2.58 ± 0.11 4	27 (LCP), 25 (RCP)	0.61 (LCP), 0.60 (RCP)
K	23	0.805 ± 0.004	74 (LCP), 77 (RCP)	0.66 (LCP), 0.65 (RCP)

3. THE DISCOS CONTROL SOFTWARE

3.1 Architecture

The DISCOS control software was designed on top of the Alma Common Software (ACS), so the overall outcome is highly influenced by the framework architecture and development patterns: in particular ACS is based on a Component/Container model, where each Component represents either a device to be controlled or a computational task.

The architecture is composed by several packages or subsystems. Packages are sub-sets of the whole system: they group all the elements (components, configuration schemas, codes and interfaces) that logically or function-

ally stay together. Packages are themselves hierarchically grouped into two layers: the User and the Telescope Layer. The User Layer runs on top of the Telescope Layer and hides its complexity to the upper one.

- *Telescope Layer*. It contains the packages that directly deal with the telescope hardware (Antenna, Receivers, back-ends, servo systems, etc.). These subsystems work independently from each other. A special component (Boss) leads its own package by checking and publishing its overall status and by managing the life-cycle of other components. The Boss components expose common and specific interfaces that define the operations on each package.
- *User Layer*. It is essentially composed by a Management package, the user interfaces and other tools used for science operations. The Management subsystem is responsible for schedule execution, instrument setup, supervision, remote access, data file output, log handling, etc. These high-level capabilities are implemented on top of the lower-layer Boss components. The Boss component of this package is the system scheduler, which also takes care of executing the user commands.

This strictly hierarchical scheme, based on Boss components communications, is only broken to transmit system-scope information ("telescope is tracking", for example). In order to avoid an excessive increase of complexity, this kind of events are notified to all the packages through a n-to-m asynchronous communication channel: the ACS Notification Channel (NC) [4].
The packages of the Telescope Layer must cope with the different characteristics of each telescope, thus leading to a more complex design. Generally speaking, we identified two categories of problems:

- the devices or the hardware are peculiar, they could differ by vendor, philosophy, communication protocol or logic;
- the devices or the hardware are missing or do not apply to the telescope.

In order to deal with the first group, we define a generic interface for each device type we need to control. This interface exposes all the operations and capabilities that are fundamental for the package to properly work. The Boss component drives the device through this interface, independently from the underlying hardware. A second-level interface shows all the device features that could be employed by other applications (e.g. monitoring panels used by the technical staff). The peculiar implementation of the device that has to be installed at the telescope is performed transparently, at deployment time, by the installation scripts. The User Layer package stores all the available parts and components of the system into configuration tables. The Management package also knows the package dependencies of each of the high-level operations. In that way, if an operation on a non-existent package is issued, it is blocked and a proper error message is notified. This simple mechanism proved to be very effective to deal with the missing or non-applicable hardware.

3.2 Recent development

The technical commissioning of the SRT [3] ended in October 2013, with the obvious goal of making the radio telescope reach an operative status. By that time the software control system was thus implementing all the necessary elements for telescope movement, receivers setup, subreflector control, active surface control, and basic scheduling capabilities. A total power back-end was integrated into the control system as well, *de facto* enabling the first complete observations and the collection of astronomical data.

Since then, our efforts were dedicated to the astronomical validation of the radio telescope and, more recently, to the early science observations. This led to the development of new features and to the integration of new instruments in the DISCOS control software, enabling a broader set of science cases.

In most cases, the addition of features to the control software was realized via the development of a new component in our Component/Container ecosystem, often re-implementing an already existing interface for the hardware integration and adding generic functionality by developing new libraries available to all the components. A broad set of tools was also implemented in order to support users in preparing and performing observations at the telescope.

3.2.1 Spectroscopy observation support

Enabling spectral-line observations needs a specific infrastructure. For example, it is necessary to compute the velocity of the observed target with respect to the observer, as this turns into a frequency offset to be applied to the line rest frequency, when we calculate the actual frequency to be observed.

The target velocity, as an input value, must be specified using these parameters:

- **Reference Frame** in which the velocity is defined. Supported frames are: topocentric, barycentric, kinematic LSR, dynamical LSR, galactocentric and Local Group
- **Velocity Definition** can be specified as radio, optical or redshift
- **Radial Velocity Value** which is expressed in km/s - apart from the redshift case, when it is a pure number.

The instant topocentric radial velocity must then be achieved, in order to compute the actual observed line frequency, taking into account the specified frame and definition. The computation consists in the following steps:

1. $F_1 = F(V_rad)$
2. The vector velocity V_b of the Solar System barycenter with respect to the observer is computed. (the Earth's rotation is taken into account)
3. the vector velocity V_f of the specified target velocity frame V_ref with respect to the Solar System barycenter is computed
4. the resulting vector $V_t = V_b + V_f$ is obtained
5. the topocentric frequency (i.e. the observed line frequency) is then calculated with the relativistic formula: $F_top = F_1 * \frac{\sqrt{1-(|Vt|/C)^2}}{1+\frac{V_t \cdot S}{C}}$, where S is the unit vector in the direction of the source.

This algorithm is implemented in the DISCOS system in the form of a standalone tool called **FTrack**, or as a library used by the Scheduler component. The Scheduler is in charge of properly tune the system in order to center the specified frequency in the available bandwidth(s). It can obtain this in two different ways:

- **ftrack=LO**: the local oscillator of the receiver is tuned so as to center the line in the sub-band presently sampled by the back-end
- **ftrack=ALL**: the system exploits both the local oscillator of the receiver and the ones of the back-end in order to center the specified frequencies; this is particularly useful when trying to center different lines in the various sub-bands allowed by the back-end.

3.2.2 XARCOS back-end integration

As described in [5] XARCOS is a spectro-polarimeter developed by the Astrophysical Observatory of Arcetri. The peculiar features of the back-end consist in the capability of processing up to 16 IFs simultaneously, thus fitting the 7-feed full-polarization K-band receiver the SRT is equipped with. A similar but limited version, managing 8 IFs only, is available at the 32-m Medicina radiotelescope; no significant differences, in terms of software code, are present between the two versions.

XARCOS was integrated into DISCOS by implementing the GenericBackend interface, which had previously been conceived and produced in order to ease the addition of new back-ends. A new component called XBackend was written; it makes use of an XARCOS library that cleverly exposes its functionality via a single class. Such class acts as a wrapper for every other complex mechanism directly dealing with the hardware layer. Even though users are ideally enabled to configure XARCOS with vast freedom, we defined a preset of fixed configurations

that are accessible via the Configuration Data Base (CDB) [6]. They are conceived to cover the most common use cases and the XARCOS setups that map to the available receivers and observing modes.

XARCOS was also tested in complex observation setups, exploiting all of its features in combination with the FTrack library - e.g. performing the tuning and tracking of different frequencies in different back-end sub-bands - thus enabling further observing strategies.

3.2.3 SARDARA

SARDARA is a new digital back-end installed at the SRT radio telescope, the name standing for "SArdinia Roach2-based Digital Architecture for Radio Astronomy", whose replication at the Medicina radio telescope is planned to take place in mid 2016. The Roach2-based architecture[7] implements a spectro-polarimeter analyzing 2 input signals with a maximum bandwidth of 2.1 GHz each, performing FFT channelization into 1024 or 16384 frequency bins. Data is thus integrated for a minimum of 5 ms and output via 10Gbps Ethernet links.

For this back-end we choose not to implement a new component aimed at directly interacting with the hardware. Instead, we defined a TCP-based protocol that should be valid for every external back-end we might need to integrate into the system (see 3.2.5). The ROACH component is just a client for a server implementing this particular protocol and it responds to the GenericBackend interface, delegating most of its operations to the server it connects to. In this scenario, meta-data is acquired by the total power back-end (integrated into our system), and a daemon running on the data nodes is in charge of reading the spectra produced by SARDARA and fill in our FITS files, relying on timestamps for synchronization.

3.2.4 Multi-Feed Derotator

The 7 feeds of the K-Band receiver are arranged in a hexagon around a central feed, as showed in Figure 2. The receiver is equipped with a built-in mechanical rotator, whose main aim is to prevent field rotation while the telescope is tracking. The rotation angle, required to keep the receiver position fixed with respect to the Equatorial celestial frame, is the parallactic angle p. Let Az and El be the Azimuth and the Elevation of the pointing direction, and ϕ the antenna latitude, the parallactic angle is:

$$\tan p = -\frac{\sin Az}{\tan\phi \cos El - \sin El \cos Az} \tag{1}$$

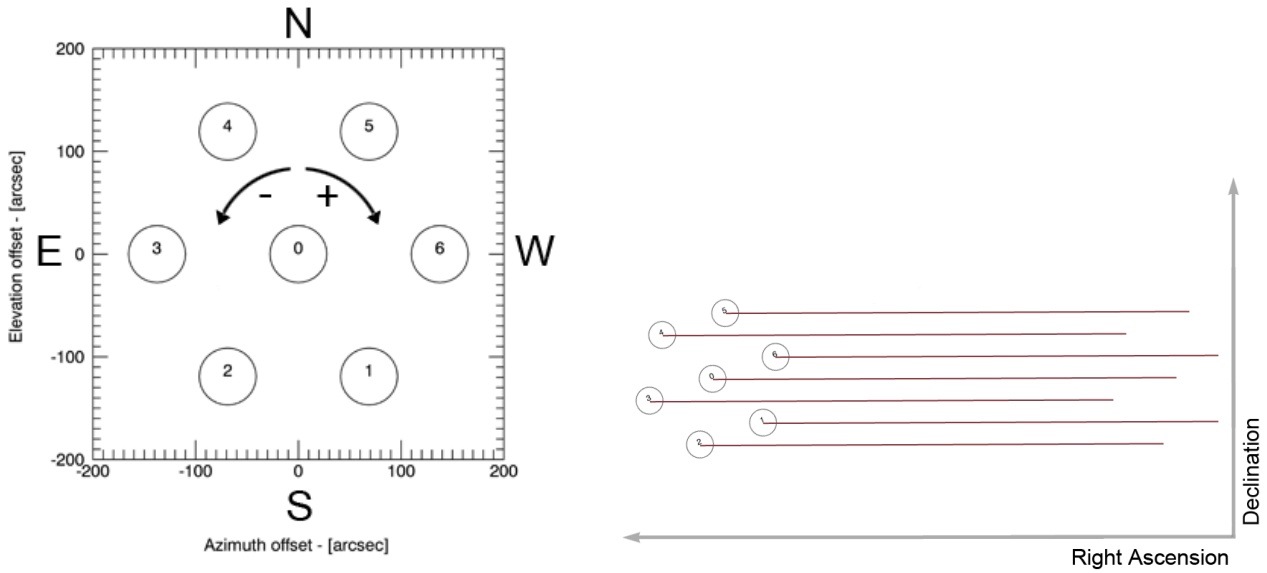

Figure 2. Multi-feed rest position

Users can choose one of the following configurations: *fixed*, *custom*, and *best space coverage* (BSC):

- In *fixed configuration* the receiver maintains a fixed position with respect to the horizon: no derotation is performed;
- *Custom* configuration allows the user to choose which angle the dewar axis must form with the y-axis of the scanning frame; the correction for the parallactic angle is applied;
- *BSC* allows the optimization of the sky coverage when exploiting observing strategies such as on-the-fly mapping or raster scanning; the receiver is positioned so as to obtain equispaced beams along the axis that is orthogonal to the scanning direction.

During the scanning, it might be necessary to periodically rewind the derotator, as it reaches its range limits. The rotator, in fact, can be positioned in the -85.77° to +125.23° range. The hexagonal layout of the lateral feeds allows to reproduce a given pointing by swapping the feeds via rotations of $N \cdot 60°$, where $N = 1, 2, 3$.

3.2.5 External support tools

Some software development activity is performed outside the DISCOS core, with no interface with the ACS subsystems, to offer standalone tools aimed at easing the user experience. This activity is carried out as an open-source software development, creating the DISCOS organization on the platform offered by github.com, [8] and is structured in different repositories, one for each standalone product:

- **basie** is the software tool used for the automatic generation of schedule files used to perform observations with the radio telescope. The software is implemented in python and largely based on the astropy [9] package. It is highly covered by tests and continuously integrated using the TRAVIS CI [10] service. The software can be used to easily configure a new observation by specifying a source list and a configuration of the kind of scans the astronomer is going to perform on the source, which get translated into the telescope language in a more complex schedule structure;
- **quicklook** are the software tools used by on-site astronomers in order to display an almost real-time preview of the data acquired by the system. At present, the collection consists in two different procedures written in IDL (by Exelis Visual Information Solutions, Inc.): one is devoted to the inspection of raw total-power or spectral data, the other performs a pre-reduction of the spectra and plots the "(on-off)/off" results;
- **discos-backend** is a reference implementation of the protocol defined for integrating external digital back-ends into a DISCOS-controlled system. The package consists in a server that defines the protocol, the grammar (with complete consistency checks) and the event loop. This server is developed in python and is entirely based on the Twisted [11] framework; it is complete with the implementation of a simulated back-end. This can serve as a template for the developers willing to integrate future back-ends.
- **discos-doc** is the repository where we maintain all the user and developer documentation of the DISCOS project. Efforts were made to port all the previously written documentation into Restructured Text format, as it eases the collaboration via standard software tools - such as git - and the versioning of different releases. All the documentation is now automatically compiled and published online via readthedocs website [12].

3.3 System review and lessons learned

The experience accumulated during these years of development and user feedback allows us to infer some mature considerations about our project. The adopted technologies and design choices can be evaluated; of course, the ACS framework played a relevant role in both aspects.
The idea to provide a common software and infrastructure to all the three Italian telescopes is the foundation of DISCOS. This goal implies not negligible costs in terms of design complexity and development efforts. These efforts are, from our point of view, very well counter-balanced by the following considerations:

- Operations: staff can now maneuver the telescopes facing almost no differences. Reference and user manuals are written with a common effort, sharing most of the information;

- Maintenance/Development: regular telescope activities require to minimize the time devoted to maintenance; the common development workflow and the common infrastructure allow to test the software releases at the telescope where adequate allocated time is available;
- Users: astronomers are presented with the same observing modes, available tools and interfaces. The output data format is also the same: there is no dependence on which telescope or which back-end is being used, as long as the back-end is fully integrated in the system. This common data format allowed us to develop and use shared analysis tools and, most importantly, to adopt a unique archiving system.

A very valuable help came from the choice to adopt a framework. More specifically, the use of ACS guaranteed, at no cost, some indispensable features like: component decoupling, service distribution mechanisms, automatic discovery of distributed components, centralized logging system and configuration database. This last item, in conjunction with some smart design choices, allowed us to cope with all the three sites. In particular, we can handle and configure very easily different policies on users and data management, as well as completely different workstation and network environments. Also, having chosen the ALMA framework we could, at least in the first stages of the project, benefit from a developers' community quite active in sharing experiences, solutions and code.
The development and maintenance of such a sophisticated system, whose roots lie in a temporally distant initial design, are now carried out while providing support to its users; all of this suffers from the poor decisions made when the system was conceived and first implemented. Many of these decisions were consequences of the risks endemic of most software projects: incomplete initial requirements and lack of manpower lead to take shortcuts, adopting time-saving and non optimal solutions. Our control software, as a consequence of the manpower shortage, completed the technical commissioning of the SRT with very limited test coverage and virtually no integration infrastructure. After this stage, the astronomical validation has been characterized by a huge amount of bug fixes and by the collection of numerous requests for new features and enhancements. Addressing all these requests turned out to be a very critical task, without the support of an adequate test infrastructure to guarantee software quality. In order to address this issue we started a review of our workflow, introducing some modern concepts for testing, continuous integration and provisioning. More details are given in section 4 and in a paper in this Conference [13].

One of the most demanding aspects of our system is the need to handle very complex subsystems such as the telescope active surface, which counts 1116 actuators that are to be simultaneously moved to operate a proper gravity compensation. The design of the control reflects the ACS paradigma of one-device-one-component. That results into a federation of more than 1000 components that are running at the same time, only for this subsystem. Even if the active surface turns out to be very stable and reliable, the big issue with this configuration is that the system takes several minutes to complete the boot-up sequence. Neither the network/memory/threading optimization nor a strategy oriented to a balanced load distribution improved the situation: the ACS Manager turned out to be overloaded and unable to fulfill all the requests at startup. With the ACS Manager being a bottleneck, we can conclude that our system does not scale well with the number of running components.
Handling the data flow also proved to be challenging. The output file format for DISCOS-integrated back-ends is a standard FITS file, containing a sequence of consecutive binary tables that increase in size with the total observation time. This design eased the initial implementation but is now showing its weaknesses: the standard CCFits library [14] does not optimally deal with the binary tables and forces lots of I/O operations on the disks, due to continuous space reallocations. A hierarchical approach would have allowed to avoid some of the latencies when the system closes the output files.

Most of the technologies employed since the early stages of the project are derived from ACS. ACS is CORBA-based [15] and it generally comes with conservative choices related to mature technologies which, by definition, have been in use long enough for most of its initial faults to be removed - or at least reduced by further developments. On the other hand, these technologies do not have any new breakthrough advances that permit to overcome some of the inherent problems. CORBA, for example, is very complex and has opaque references that force the applications to rely on an external naming service. This not only complicates the system development

and deployment, but creates also a single point of failures. ACS masks most of CORBA APIs and complexity, yet the point of failure remains an unresolved issue.
Many implementations of CORBA provide a A/V streaming service which defines an architecture conceived for distributed multimedia applications. On top of this service, the ACS team developed a Bulk Data Transfer [16] (BDT) mechanism suited for transferring (one-to-one or one-to-many) high volumes of data. In DISCOS this service has been used to send raw data from integrated back-ends to the file-creation components or real-time processors. When we designed this part of our control software, our focus was on the maximization of the available bandwidth. The performance tests were quite satisfactory, as we were able to transfer 800 Mbit/s out of a 1 Gbit/s network. Unfortunately, we discovered that some other aspects should have been considered, in particular the TAO implementation of the CORBA A/V service [17] and consequently the ACS BDT suffers from a problematic error handling, the lack of proper resource cleanup and adequate re-initialization of the connection and instabilities generated if the receiver-end of the channel is slower than the sender. Considering all these issues, tuning the data acquisition software of DISCOS turned out to be a very challenging task. Presently a significant part of the system failures we are experiencing are due to the BDT. ALMA decided the migration to Data Distribution Service (DDS) some years ago, and we are planning to move to a more widespread and reliable technology as well.
During the years we also ran into a rigidity of the ACS framework, limiting the component interface flexibility. ACS has a built-in `sequence` type that is very useful when dealing with multiple attributes (a `position`, for example, made of several coordinates). On the other side, the built-in alarm system does not handle properly `sequence` attributes, as it allows to specify just a single threshold to trigger the alarm. In order to bypass some of these limitations and constraints (legacy environment and operating system), when on the verge of developing new DISCOS tools (User Interfaces and middlewares) we started to experiment third-parties tools and technologies. These tools are mostly frameworks and libraries for Python programming* because of its simplicity. Unfortunately, we encountered insurmountable difficulties with the ACS Python support, which is not completely reliable because of some bugs, not implemented features and outdated documentation.

4. DEVELOPMENT AND MAINTENANCE

The DISCOS project has grown in size up to 500k lines of code and configuration. The development activity is split into three product lines: one for each currently involved telescope. Along with the development of a common trunk, the single telescope branches are tagged with individual releases whenever a significant upgrade or milestone is reached, preceding the *in-situ* installation. The coding activity is completely carried on by INAF personnel: the core developers' team is composed by 6 persons: three software engineers, two support astronomers and one person dedicated to SRT dev-ops. Developers are also involved in telescope operations and user support, and none of them fully dedicated to the DISCOS project itself.

Another source of fragmentation is the heterogeneous environment between the development workstations and the production servers: even though the utilization of the ACS framework is imposing a specific operative system (namely RHEL5) there is a plethora of dependencies from other third-party libraries, hardware drivers and OS configuration, which slowly led to inconsistencies between the development and production environments: this results in non-repeatable behaviors and uncertainties in the outcomes of the development process.

Moreover, as the SRT telescope has officially entered its operational phase, it is highly scheduled and there is very little time for software maintenance and tests on production machines. The lack of hardware simulators and the poor test coverage of our project make it difficult to develop and deploy new features in a timely manner.

We thus decided to move the project life-cycle towards a Continuous Integration approach in order to reduce the time to develop and deploy new features and bug fixes, trying to reach a higher stability through the standardization of the development environments and a new deploy process.

4.1 Towards Continuous Integration

The key to a successful Continuous Integration strategy adoption is the automation of repetitive tasks and the standardization of procedures among environments and among developers. We developed the AZDORA

*ACS is supposed to support three programming languages: Python, C++ and Java

[18] project with the goal of building and configuring our complex environment in a consistent way on test and production machines. The project is based on virtualization technologies such as Vagrant and ORACLE VirtualBox, and it enables the creation of a brand-new Virtual Machine with a simple command line invocation: the newly-created Virtual Machine will then be automatically configured using provisioning scripts in order to create users, set environments variables, install the ACS framework and every third-party dependency Figure 3. The first effect of such a step forward is that a developer can create a new machine in a matter of one hour or less, avoiding to forget simple configuration steps that could cause incorrect behaviors of the system (which are difficult to diagnose afterwards). Previously, the same operation used to take more or less one full day of work, thus the new approach results in a great time optimization, making it inexpensive to create a clean environment from scratch. This can be useful whenever there is a doubt about some corrupt global status. While we cannot use the virtual machines in production for performance and hardware bindings reasons, we still can use the provisioning scripts to reach exactly the same configuration on a newly installed OS.

4.1.1 Automated builds

Once the standardization of the environments was obtained, we automated the build process for each of our product lines; in fact we were facing these issues:

- the common components were typically built only against the product line in use at the moment;
- each code module used to be compiled independently, without a real verification of compatibility with other code units, not even at compile time;
- the build of the whole system used to prosecute on failure of single units of code making it difficult to understand which version of each code unit was installed at the end of a new build operation.

We thus introduced the Jenkins [19] server in our technology stack in order to perform nightly builds of each of our product lines, in each of its supported versions. The first step towards nightly builds had to be a rethinking of our build process, making it less robust to errors and letting it fail explicitly in case of issues with the single code units. The second step involved the ability of the ACS framework to let the user-defined code live under a single directory, called the INTROOT: we decided a naming scheme, making it possible to install different versions of the DISCOS software using different INTROOT directories.

We could have spun a brand-new virtual machine for each build job, and this would have been a more robust approach, but it is too resource-intensive, so we decided to execute every job on the same machine by simply changing the INTROOT for every possible configuration. Maybe moving the virtualization stack towards a container-based technology (such as docker) would make it possible to perform each job on top of a brand-new environment with a lower impact in terms of resources.

We then configured Jenkins in order to build the complete DISCOS project for each possible combination of:

- Product line: each telescope in fact has a different environment
- Product tag of each actively maintained DISCOS version
- Common core and telescope-specific code

This results in 12 build jobs from the very first moment, ranging from 20 minutes to one hour of CPU time each. Complete builds are executed every night and are triggered by changes in the code detected by Jenkins, which keeps monitoring our subversion repository. Taking confidence with the Jenkins server commands, we used it to perform further checks on our code base: for example trying to enforce the integrity of the configuration database. Now we also have jobs that, upon successful builds of entire product lines, archive the result creating a package, comprising configuration and versioning information: these packages are ready-to-install binary versions of the DISCOS project that can safely be used in production servers.

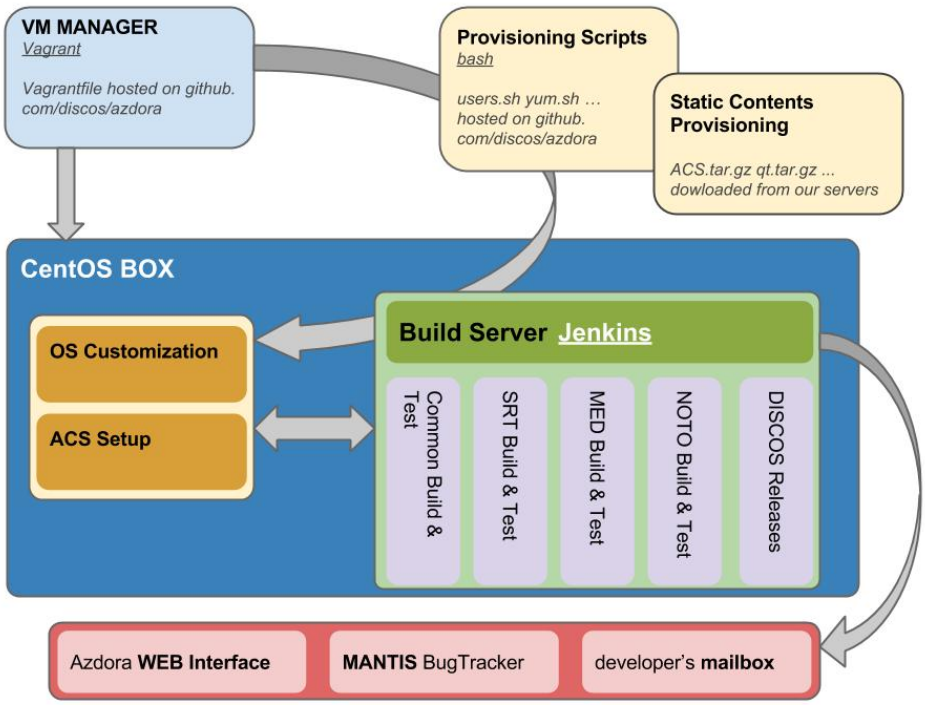

Figure 3. Azdora project block diagram

4.1.2 Testing Infrastructure

As the DISCOS project is growing in size and complexity, it is more and more important for developers to be able to test whether new features and bug fixes affect the behavior of the whole system, or if they break compatibility with other code modules and product lines. As described in another paper at this conference [13] we introduced best practices and infrastructures to write unit tests and functional tests, both for Python and C++ code (the only two programming languages we are using), by adding skeleton projects for different use cases, so that developers can write tests based on solid and simple examples.

4.2 First results and possible improvements

As a first result we achieved the stability of the development branch for each product line, just after some month from the introduction of the Jenkins mail alert for failed builds. Now the development process is more accurate and, once the stability is obtained, it is immediately corrected upon new failures, so that even the trunk branch is often in a safe status and can be deployed and tested with better confidence. Also, the introduction of testing examples, guidelines and infrastructures is slowly penetrating into the project as a testing culture, giving raise to the adoption of the first unit tests and integration tests. Unfortunately our code base is very large and the percentage of test coverage is still very low. Some tests depend on external resources and would require hardware simulators or mockers in order to be executed: this is something we are working on, in view of long-term maintainability.

Another major improvement could be the adoption of container-based technologies for automated build execution, leading to the creation of completely isolated environments for each job. Contrarily to virtual machines, containers could be safely used in production, greatly easing the deploy operations and dramatically reducing the time needed to deploy bug fixes and new features.

5. CONCLUSIONS

DISCOS is serving the scientific community by enabling observations at the Italian INAF radio telescopes, and, while potentially subject to major changes, it fulfills the main expectations of both astronomical validation and early science activities. Future changes and improvements to the system depend mostly on the availability of funds, that are required to retain the highly-skilled manpower needed to pursue our goals. Some of these foreseen changes are planned to address issues depending on design flaws or on the adoption of the ACS framework. On top of our list there are: the re-engineering of the output file generation and of the data storage organization, the insulation of the Active Surface module in order to workaround the bottleneck posed by the ACS Manager, the migration to new technologies for the high-rate, asynchronous data streams (DDS) and the migration to 64-bits architecture.

The growth of the project and its dissemination among different sites led to the adoption of best practices borrowed from the software industry, such as Behavior Driven (BD) and Test Driven (TD) Development and strategies of integration and provisioning, in order to assess software quality and stability. This approach is still evolving, thus our efforts will still be focused on improving the continuous integration and provisioning procedures and on increasing the test coverage of our code. We also intend to develop and add new observing modes, and to improve the usability of DISCOS by introducing new user tools and web-based user interfaces.

REFERENCES

[1] http://www.med.ira.inaf.it/.

[2] http://www.noto.ira.inaf.it/.

[3] Bolli, P., Orlati, A., and Stringhetti, L., "Sardinia radio telescope: General description, technical commissioning and first light," *Journal of Astronomical Instrumentation* **4**, 20 (2015).

[4] Pisano, J., Fugate, D., and Lucero, S., "Acs notification channel module software design and tutorial." Atacama Large Millimiter Array documentation 2009-04-30 http://www.eso.org/~almamgr/AlmaAcs/OnlineDocs/Notification_Channel_Module_Software_Design.pdf (2009).

[5] Melis, A., Migoni, C., Comoretto, G., Castangia, P., Casu, S., Tarchi, A., Orlati, A., Poppi, S., and the SRT Astrophysical Validation Team, "Integration of the digital full stokes spectrometer xarcos into the control software of sardinia and medicina radio-telescopes." SRT technical report http://www.oa-cagliari.inaf.it/download.php?id_file=gxMdbWdHMHAX5N5LfD%2FPka6WkQVs2YieMDrJ%2BFxcCQ6%2B3K3sJRym3RoKWrLY2E3OOW8PVPuD5tNkwmBWf2VHMA%3D%3D (2015).

[6] Caproni, A., "Caonfiguration database cdb." Atacama Large Millimiter Array documentation 2005-105-17 http://www.eso.org/~almamgr/AlmaAcs/OnlineDocs/ConfigurationDatabase.pdf (2005).

[7] https://casper.berkeley.edu/wiki/ROACH2.

[8] Orlati, A., Righini, S., Bartolini, M., Buttu, M., Poppi, S., Migoni, C., and Fara, A., "Discos organization on github." https://github.com/discos/ (2015).

[9] Robitaille, T. P., "Astropy: A community python package for astronomy," (2013).

[10] https://https://travis-ci.org/discos.

[11] https://twistedmatrix.com.

[12] Righini, S., Orlati, A., Bartolini, M., Buttu, M., Poppi, S., Migoni, C., and Fara, A., "Discos control software documentation." http://discos.readthedocs.org/ (2015).

[13] Orlati, M. B. A., Bartolini, M., Fara, A., Migoni, C., and Poppi, S., "Rules of thumb to increase the software quality through testing," *Proc. SPIE* (2016).

[14] http://heasarc.gsfc.nasa.gov/fitsio/ccfits/.

[15] http://www.corba.org.

[16] Cirami, R., Marcantonio, P. D., Chiozzi, G., and Jeram, B., "Bulk data transfer distributer: a high performance multicast model in alma acs," *Proc. SPIE* (2006).

[17] Surendram, N., "The design and performance of corba audio/video streaming service," in [*Proceedings of HICSS-32*], **8**, 8043 (1999).

[18] https://github.com/discos/azdora/.

[19] http://jenkins.io.